Developing and Managing Software Components in an Ontology-based Application Server

Daniel Oberle, Andreas Eberhart, Steffen Staab, Raphael Volz
Institute AIFB, University of Karlsruhe, Germany
lastname@aifb.uni-karlsruhe.de

Abstract

Application servers provide many functionalities commonly needed in the development of a complex distributed application. So far, the functionalities have mostly been developed and managed with the help of administration tools and corresponding configuration files, recently in XML. Though this constitutes a very flexible way of developing and administrating a distributed application, e.g. an application server with its components, the disadvantage is that the conceptual model lying behind the different configurations is only implicit. *Hence, its bits and pieces are difficult to retrieve, survey, check for validity and maintain. To remedy such problems, we here present an ontology-based approach to support the development and administration of software components in an application server. The ontology captures properties of, relationships between and behaviors of the components that are required for development and administration purposes. The ontology is an* explicit *conceptual model with formal logic-based semantics. Therefore its descriptions of components may be queried, may foresight required actions, e.g. preloading of indirectly required components, or may be checked to avoid inconsistent system configurations — during development as well as during run time. Thus, the ontology-based approach retains the original flexibility in configuring and running the application server, but it adds new capabilities for the developer and user of the system. The proposed scheme has been prototypically implemented in KAON SERVER, an application server running Semantic Web components.*

1. Introduction

Application Servers are component-based middleware platforms that offer an environment in which users can deploy components developed by themselves or by third-party providers [1]. As a sophisticated middleware, application servers provide functionality such as dynamic loading, naming services, load balancing, security, connection pooling, transactions, or persistence.

Despite the bundled functionality, realizing a complex distributed application remains all but an easy task. For instance, managing component dependencies, versions, and licenses is a typical problem in an ever-growing repository of programming libraries. In Microsoft environments, this is often referred to as "DLL Hell". Configuration files, even if they are more or less human-readable XML, do not provide an abstraction mechanism to tame the complexity issues arising in such systems.

As a way to abstract from many such low-level and often platform-specific problems, the paradigm of *Model-Driven Architectures (MDA)* has gained wide-spread influence. The principal idea of MDA is to separate *conceptual concerns*, such as which component is using which other component, from *implementation-specific concerns*, such as which version of an application interface requires which versions of windows libraries. MDA achieves this separation by factorizing the two concerns, specifying them separately and compiling them into an executable.

Notwithstanding that MDA already provides conceptual modelling in order to improve management of complex systems, MDA is disadvantaged in two ways. First, MDA requires a compilation step preventing changes at run-time which are characteristic for application server software. Second, an MDA itself cannot be queried or reasoned about. Hence, there is no way to ask the system whether some configuration is valid or whether further components are needed.

Therefore, the logical next step in developing and managing complex applications is the use of an *explicit conceptual model that is executable, too*, meaning it may be queried and reasoned with, i.e. an ontology and corresponding semantic metadata (descriptions of components in terms of the ontology). The goal of this paper is to show how ontologies may be defined that support the developer in creating new software or in running new components in the complex environment of an application server. For example, the ontology allows for finding APIs that come with certain capabilities (development time support) or for preloading components that are required by other components (run time support).

The reader may note that though an application server is not the only software that may be supported in that way, it is a very worthy challenge. The reasons are that the needs are huge in this area and at the same time ontologies may fruitfully contribute to this complex, but nevertheless reasonably restricted domain of application.

We present some motivating use cases for our approach of an explicit conceptual model (Section 2). We embed our approach into a generic architecture for an ontology-based application server and briefly refer to its prototypical implementation that supports building Semantic Web applications [14] (Section 3). As a major contribution of this paper, we describe the ontologies that have been developed since the inception of the architecture implementation (Section 4) and highlight some of the new management capabilities provided to developers of the application server (Section 5). Finally, related work and conclusions are discussed in sections 6 and 7.

2. Motivation

The first subsection considers use cases for semantic metadata of components and APIs that target development time support. Subsection 2.2 focusses on the run time use cases.

2.1. Use Cases of Semantic Metadata for Development Time Support

Today, the introspection feature of object-oriented languages provide syntactic metadata of classes, fields, and methods along with their parameters, return types, and possible exceptions. Similar information is available from WSDL Web Service metadata, whereas component libraries such as dynamic linked libraries or Java archives only carry very little information. We believe that rich, semantic metadata can provide added value to the user. Consider the following use cases:

Component dependencies and versioning. Libraries often depend on other libraries and a certain archive can contain several libraries at once. Given this information, a system can assist the developer in locating all the required libraries[1]. Furthermore, the user might be notified when two libraries require different versions of a certain third component. For instance, the multitude of versions of XML parsers causes a lot of trouble. We envision a system, which reasons with this kind of information in order to make an educated suggestion or to display inconsistencies.

Licensing. Similar to the component dependencies, we can describe licensing, trustworthiness and quality. Using an external module in one's software has effects on the licensing. Using external GPL licensed code prohibits distributing the bundle under a LGPL license. Along the same lines, ISO software certification or a security guideline of a government agency might prohibit certain external components to be used in software to be deployed. In all of these cases, it would be useful to model development constraints and reason with these and semantic metadata to avoid licensing problems.

Capability descriptions. Database interfaces typically offer some method to execute an SQL command. However, the behavior of specific database implementations can vary dramatically. Earlier versions of MySQL do not support transactions or subqueries. Component capabilities adhering to standard interfaces can be made explicit to the developer.

Service classification and discovery. Given APIs with similar functionality, one will find different methods and services with essentially the same functionality. We suggest associating these implementations with a common service taxonomy. This will allow the user to discover implementations for a certain taxonomy entry and to classify a given service.

Semantics of parameters. Parameters and return types of methods and services are often implicitly encoded in the respective names. Providing meaningful names is considered to be an important practice when developing software systems. However, it will be desirable to associate the names with concepts and relations of a common agreed-upon domain ontology. This will allow more powerful searches over a large unfamiliar API. These descriptions can even be used to generate a sequence of method invocations in order to achieve a goal specified [6].

Automatic generation of component and service metadata. Development toolkits usually provide functionality for creating stubs and skeletons or for automatically generating interface metadata *à* la java2wsdl. With an entire set of new markup languages like BPEL4WS[2] or OWL-S[3] emerging, tool support for these new languages is needed. Whereas WSDL tools can obtain almost all of the required input directly from the source code, more powerful languages require additional semantic metadata as the source of the more complex metadata. If the respective metadata are already available within the system, automatically generated BPEL4WS

1 This idea is already realized in the RPM package manager: http://www.rpm.org/. Our goal is to generalize this approach and integrate it with other tools and services for the end user.

2 `http://www.ibm.com/developerworks/library/ws-bpel/`

3 `http://www.daml.org/services/owl-s/1.0/`

or OWL-S metadata can be a side product of a unified framework.

2.2. Use Cases of Semantic Metadata for Run Time Support

Application servers handle issues like load balancing, distributed transactions, session management, user rights or access controls. All of these tasks are orthogonal to specific application issues in that they reappear in just about any scenario. Consequently, it makes sense for an application server to manage these issues in an application independent way. This means that the responsibility is shifted from the coding to the deployment process.

While it is always a good idea to reduce the amount of source code that has to be written, the deployment process can be quite tricky in itself. Consider the J2EE platform as an example. The specification describes the structure of XML deployment metadata. J2EE implementations like JBoss (cf. [7]) provide a set of tools, which help the user to generate such metadata. However, the tools merely act as an input mask, which generates the specific XML syntax for the user. This is definitely a nice feature, however, she or he must fully understand the quite complicated concepts that lie behind the options for the transactional behavior for instance. The current deployment tools do not help to avoid or even actively repair configurations that may cause harmful system behavior. Consider the following situations:

Access Rights. The access control mechanisms of application servers are based on users and roles to whom access can be granted for certain resources and services. In addition, services can be run using the credentials of the caller or those of another user that runs the service on behalf of the caller. This is often referred to as the authentication problem [8]. It is quite evident, that access rights within a large business process can be very complex. A system should be able to assist the user in suggesting suitable settings and in determining potential flaws in the security design. We believe that formal reasoning over group memberships or resources being accessed by processes running on behalf of other users will prove to be valuable here.

Error handling. Modern programming languages make heavy use of exceptions. Exceptions are raised and propagated along the calling stack in order to be handled at the appropriate level. In order to avoid the embarrassing situation that an exception is not handled at all and simply passed to the user interface or business partner, a consistency check can be put in place. Similar to the argument made in the previous example, rules describing how exceptions are thrown, passed across the calling stack, and being caught or not can be applied in this scenario.

Transactional settings. Resources such as databases and message queues offer transactional recovery. This notion is extended to regular software components, which access transactional resources. Methods can be declared to not support transactions, to initiate a new transaction, or to participate in the caller's transaction. Again, a chain of calls across many components can contain inconsistent settings such as a component which requires a transaction calling one that does not support transactions. A formalization of invocations and the possible transactional settings can be applied here.

Secure communication. Confidential data might be made accessible to business partners only. Settings on the application server typically determine that a digital signature has to be checked before the request is passed along and that a service can only be bound to a secure communication line or protocol. Similar to the arguments made above, a system should be able to detect, that a confidential resource is accidentally made accessible via a non-encrypted communication channel.

The contribution of this paper is to show how some of these use cases of semantic metadata work. The claim that we make is that all of them — and many more — can be handled in a generic way using an ontology and corresponding semantic metadata.

3. An Ontology-based Application Server

The various examples make it evident that there is a need for a conceptual model. The advantage of such a model is twofold. The model abstracts from the specifics of implementations such as J2EE with their proprietary configuration and deployment formats. Instead it focuses on and reflects the underlying, well defined and agreed-upon concepts such as users, access rights or transaction settings. Furthermore, general domain knowledge about these concepts can be formalized and reasoned with. This is a valuable basis for a variety of value-added services.

We use ontologies as a representation of the conceptual model. Like schemata, ontologies define concepts and concept relationships (associations). Ontologies differ from schemata mainly in two ways. First, ontologies aim at capturing the *shared* understanding, which often includes linguistic issues as well. Issues such as performance, compact storage, and other application specific features do not play a role. Second, ontologies contain axioms, which further define domain concepts and allow to reason with them. Components are described in terms of the ontology what results in semantic metadata.

Ontologies play a key role in area of the Semantic Web [2] and based on this work, several representation languages have been standardized within the World Wide Web Consortium (W3C). Consequently, we have chosen these languages as the basis for our work.

The following subsections survey both the overall system architecture of an ontology-based application server and our implementation called KAON SERVER.

3.1. Architecture

Figure 1 shows the overall system architecture. The left side outlines potential sources, which provide input for the framework. This includes web and application server configuration files, annotated source code, or metadata files. This information is parsed and converted into semantic metadata, i.e. metadata in terms of the ontology. The semantic metadata and the ontology are fed into the inference engine which is embedded in the application server itself. The reasoning capability is used by an array of tools at development and at run time. The tools either expose a graphical user interface (e.g. security management) or provide core functionality (e.g. the dynamic component loader).

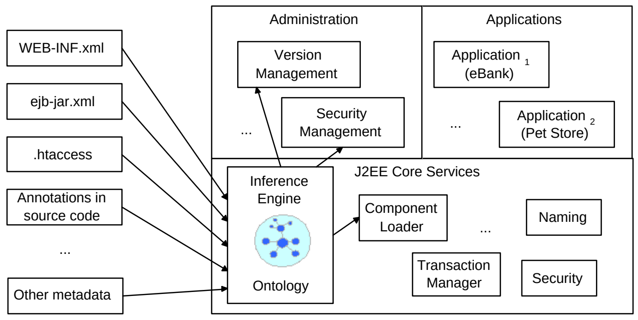


Figure 1. System architecture.

3.2. Implementation

This section gives a concrete example of how the overall system architecture can be realized. For a detailed discussion the reader is referred to [14].

The aforementioned architecture is implemented in a system called KAON SERVER which is part of the KArlsruhe ONtology and Semantic Web Toolsuite (KAON). We made use of the Java Management Extensions (JMX [9]) — an open technology for component management. With JMX it becomes possible to configure, manage and monitor Java applications at run time, as well as break applications into components that can be exchanged. Basically, JMX defines interfaces of managed beans (*MBeans*) which are JavaBeans that represent JMX manageable resources.

JMX only provides an API specification with several available implementations. We have chosen *JBossMX* which is the core of the open-source JBoss application server [7] that augments J2EE by dynamic component deployment. This choice allows us to inherit all the functionality provided by JBoss in the form of its MBeans (Servlet Containers, EJB Containers etc.). We deploy our inference engine as an additional MBean and augment the existing component loader and dependency management to exploit the inferencing. A version and security management tool allows to browse and query the ontology at run time. Thus, it is possible to use the KAON SERVER as a "semantically enhanced JBoss".[4]

4. The KAON SERVER Ontology

This section details the ontology used in the KAON SERVER. It is subdivided in an ontology dealing with development use cases and run time use cases. Both are interrelated and split into several modules. They are further discussed in subsections 4.1 and 4.2.[5]

4.1. Ontology Modules for Development Use Cases

In this section we present our ontology, which allows to conceptualize the development use cases introduced in section 2.1 by semantic metadata of components and their APIs. Note that we only give a short overview due to the lack of space. The interested reader is referred to [13]. The ontology is divided into several modules (cf. Figure 2) which are explained in the following[6]:

API Description The *API Description* module offers a framework for taxonomically describing the functionality offered by methods of APIs (e.g. setter methods of an EntityBean would be instances of a method "AddData") and accordingly several types of APIs (e.g. StoreAPI). It also allows to express the semantics of parameters. This kind of information is used to perform service classification and discovery as well as for automatic generation of component and service metadata. Semantically enriched information, in this case specifying that the argument is information like a Person or an Address, facilitates discovery and can also be used to enrich automatically generated Web Service metadata.

Component Simply consists of one concept that groups together *Profile* and *Grounding* information (explained below) about every kind of component.

Profile We use the *Profile* module to express capability descriptions of a component. For example, a database

4 The KAON SERVER can be obtained at `http://kaon.semanticweb.org`.

5 Note that the ontologies are expressed in the KAON language [12] that is equivalent to Datalog. For the sake of readability, we will express axioms in First Order Logic syntax throughout the paper.

6 The uses cases' names introduced in 2.1 are written in sans serif.

adapter component would have an attribute specifying the SQL dialect used. Information of this type might be used for service classification and discovery.

Grounding Basically, the *Grounding* module allows to express the mapping between the existing syntactic metadata (e.g. J2EE descriptor files like ejb-jar.xml) and the semantic metadata. In order to express the mapping between the *API Description* and the source code we came up with a conceptualization of IDL terms which is grouped together in the *Implementation* module (see below).

Implementation This module contains implementation level details of a component responding to the use case of component dependencies and versioning. Code details are described, like the class name or required archives. Besides, each component has a certain version and potentially depends on others. Along the same lines, some components will not work properly, if a conflicting component is loaded at the same time. These relationships are modelled by *dependsOn* and *conflictsWith* which are transitive and symmetric associations, respectively[7]. Components can be in different states (active, available, serialized, etc.) that are captured by an attribute of the same name. We also model the signature whose methods and parameters are expressed according to the *IDL* module (see below).

IDL We have conceptualized a small subset of the IDL (Interface Description Language [15]) specification into an ontology-module that allows describing signatures of interfaces. The ontology module features concepts like *Interface*, *Operation*, *Parameter* and so on.

Domain Ontology Modules While the ontology modules presented so far formalize generic knowledge only, domain ontology modules grasp knowledge specific to a certain application. For a new domain it can be easily replaced. First, **Domain Profiles** may distinguish component types in a particular application server. While most J2EE servers Servlets and EJBs, newer ones also introduce MBeans and Microsoft's .NET world introduces further idiosyncracies. Second, *Domain API Descriptions* contain sets of APIs and functionality types (methods) that are typically offered by components in a certain domain. An analysis of an online store results in a domain ontology conceptualizing Orders, Suppliers and Products, for instance.

Like discussed in [11, 13] we used OWL-S as a starting point for building the ontology. OWL-S is an ontology expressed in the Ontology Web Language (OWL). Its aims are to enable automatic Web Service discovery, invocation, composition and execution monitoring. Several interesting design principles are realized by OWL-S that inspired our work: separation of semantic and syntactic metadata, separation of generic and domain knowledge and modularization. Our current state of work comprises a full analysis, yet our formalization still lacks the individual groundings. The *Component*, *Profile*, *Implementation* and *Domain* modules are already used within the KAON SERVER.[8]

7 That means $\forall c1, c2, c3 : dependsOn(c1, c3) \leftarrow dependsOn(c1, c2) \wedge dependsOn(c2, c3)$ and $\forall c1, c2 : conflictsWith(c1, c2) \leftrightarrow conflictsWith(c2, c1)$.

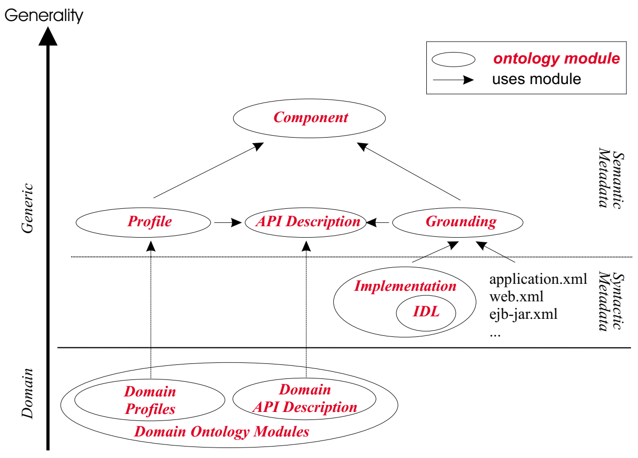


Figure 2. Ontology Modules for use cases supporting development

4.2. Ontology Modules for Run Time Use Cases

In this section we describe part of our ontology for run time use cases. It conceptualizes interceptors, libraries, archives, security aspects and their interrelationships. We have modelled most of the modules. However, they are not yet used within the KAON SERVER.

Due to space limitations, we focus on the security use case. The J2EE specification distinguishes the key concepts of realm, user, group, and role. We extend these basic notions by introducing additional concepts and some important associations and axioms in the following subsections.

4.2.1. Concepts

Resource The J2EE specification distinguishes security issues on the web, application, and persistence tiers. Even though the physical ways of accessing these resources differ a lot, the notions of access control and security are the same at all three levels. In our ontology, resources can be web resources, components or databases (tables or SQL views). These are identified by URLs, class names and database URIs (usually server URI augmented by table or view name), respectively.

8 The ontologies ship with the KAON SERVER distribution and are also separately available at `http://kaon.semanticweb.org`. The site features a screenshot of browsing the ontologies with a graphical user interface.

Method Resources have methods, which constitute the most fine grained level for access control. Methods are identified in combination with the resource. Web methods are identified by the resource URL and the protocol's method such as HTTP GET. Methods of components are identified by their class name and the method identifier consisting of name and signature[9]. In the case of database resources, methods correspond to operations such as delete, update, select, send, or receive, which can be granted individually by the database management.

ResourceGroup Systems usually allow declaring security settings for an entire set of resources. A web container allows URL patterns such as `<url-pattern> /secure/*</url-pattern>`. A similar wildcard notation can encompass all methods of a class.

ACL The right to access a ResourceGroup, Resource or Method is formalized as a concept *AccessRight*. This was necessary to circumvent ternary associations that cannot be modelled in KAON and other Semantic Web languages like RDF and OWL. Subsumption reasoning capabilities allow us to specialize AccessRight in Read, Write, Modify and Execute. An Access Control List (ACL) is comprised of one or more AccessRights.

Invocation and RequestContext The definitions so far captured the static aspects of security. At run time, any kind of resource is accessed by an incoming request. The request is associated with context information, e.g. on whose behalf the request is carried out. The context is propagated from tier to tier, unless an explicit context change takes place. We model this situation by *Invocation* and *RequestContext* concepts which are interrelated (cf. *associatedWith* below).

4.2.2. Associations

definedOn*x* and grantedFor*y* Like mentioned before, AccessRights might be defined on ResourceGroups, Resources or even Methods, and they might be granted for Roles, UserGroups or even Users.

executes and accesses During processing, resources can use other resources. This might be "cart.jsp" invoking the shopping cart EJB, the product entity bean accessing the respective database table, or also an SQL view reading other tables and views. In our ontology, Resources execute Invocations and Invocations in turn access Resources.

9 Methods can be overloaded such that the same method name is used with different parameter lists. Consequently, the signature needs to be included in the method identification.

associatedWith Any kind of resource is accessed by an incoming request. The request is associated with context information, e.g. on whose behalf the request is carried out. The context is propagated from tier to tier, unless an explicit context change takes place. We associated an Invocation with a RequestContext, where each RequestContext carries information about authentication or transactions.

4.2.3. Axioms

invokes For convenience we defined a transitive association *invokes* by axioms. It abbreviates *executes* and *accesses* in the following way:

$$\forall r1, r2, i : invokes(r1, r2) \leftarrow executes(r1, i) \wedge accesses(i, r2)$$

$$\forall r1, r2, r3 : invokes(r1, r3) \leftarrow invokes(r1, r2) \wedge invokes(r2, r3)$$

Roles, users, and groups Further axioms are necessary to fully model the domain described so far. As we mentioned before, users can be associated to groups and access is granted via the role indirection. The effect of the resulting relationships can be captured by the following rules:

$$\forall ar, r, ug : grantedForUserGroup(ar, ug) \leftarrow AccessRight(ar) \wedge Role(r) \wedge UserGroup(ug) \wedge inRole(ug, r) \wedge grantedForRole(ar, r)$$

$$\forall ar, r, u : grantedForUser(ar, u) \leftarrow AccessRight(ar) \wedge Role(r) \wedge User(u) \wedge inRole(u, r) \wedge grantedForRole(ar, r)$$

$$\forall ar, u, ug : grantedForUser(ar, u) \leftarrow AccessRight(ar) \wedge User(u) \wedge UserGroup(ug) \wedge member(u, ug) \wedge grantedForUserGroup(ar, ug)$$

A similar rule can be defined for permissions on groups of resources:

$$\forall ar, r, rg : definedOnResource(ar, r) \leftarrow AccessRight(ar) \wedge Resource(r) \wedge ResourceGroup(rg) \wedge partOf(r, rg) \wedge definedOnResourceGroup(ar, rg)$$

As we may recognize with this small example, it is preferable to specify complex interactions with a few logical rules rather than with extensive coding.

5. Example

This section introduces an example for a run time use case. Since section 4.2 focused on security, we also take an example from this use case.

Obtaining the semantic metadata There are several ways of extracting the required information from configuration files, source code, or registries. Our goal is not to provide a complete set of parsers and extraction tools at this point. Instead, we aim at demonstrating the feasibility of our approach in providing a proof of concept in extracting a reasonable amount of information.

Information about the available resources is obtained by reading the file system of the web container and the application server. Database management systems often make metadata on tables, users, and rights available via SQL. In J2EE, the access control lists are specified in the ejb-jar.xml and WEB-INF.xml deployment descriptors for the web and the application server tiers. Various realms can manage the user, group, and role information. We worked with the JDBC realm, where the data is read from tables in a database, making it also easily available to our tool. Arguably the most complicated step is determining the invocations from one resource to others. We performed a shallow analysis of the source code and SQL statements and manage to pick up the commonly used patterns such as resolving a JNDI home interface reference. Furthermore, the ejb-ref tag in the beans' deployment descriptors provides hints as to which other beans are used. Obviously, we are restricted to static code analysis, which is also used by development environments. Note that this is not really a problem if the complete invocation graph is not extracted automatically, since the system does not actively intervene in the deployment process. It merely helps the user to assess the situation before making an educated decision.

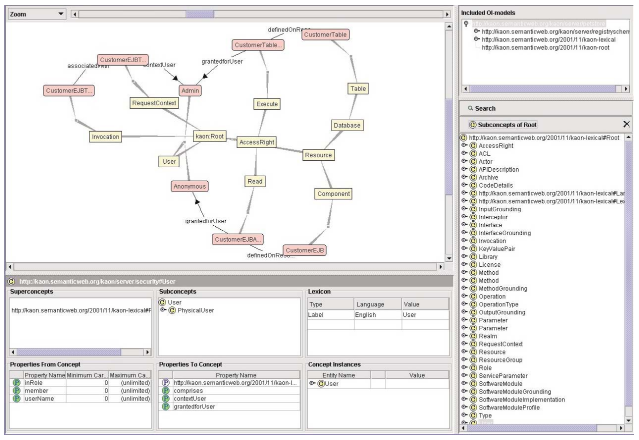
Applying the inference engine An interesting example for reasoning over security settings is to see which resources a user gets indirect access to. For instance, a customer table, accessible only by the database admin, may be indirectly readable to other users via a customer entity bean, since this bean performs a context switch. Thus, the call is carried out using admin rights on behalf of the user. This case is definitely not a bug; however, it might be useful to assess the combined effect of various security settings by analyzing the result of such a query. First, the axiom below introduces a convenience predicate *permission* that is true when an AccessRight is granted for a User *u* on a Resource *r*. Note that it captures permissions also when access is granted for a UserGroup only but the User is a member (because of the axioms introduced in 4.2).

$$\forall ar, u, r : permission(u, r) \leftarrow User(u) \wedge Resource(r) \wedge AccessRight(ar) \wedge grantedForUser(ar, u) \wedge definedOnResource(ar, r)$$

Second, the following axiom recursively extends the definition above that any User having permissions to a Resource, implicitly has indirect access to all resources, which are indirectly invoked by it. The same can be expressed for ResourceGroups, Methods and Roles, UserGroups or combinations.

$$\forall u1, u2, r1, r2, i, ct : permission(u1, r2) \leftarrow permission(u1, r1) \wedge Resource(r2) \wedge invokes(r1, r2) \wedge accesses(i, r2) \wedge Invocation(i) \wedge associatedWith(i, ct) \wedge RequestContext(ct) \wedge contextUser(ct, u2) \wedge permission(u2, r2)$$

Security management tool Our tool allows the user to query the inference engine about any concept in the ontology. For instance, one can retrieve the users who are able to indirectly read a database table. As schematically introduced in Figure 1, Figure 3 shows our version and security management tool that allows to browse the ontology and execute queries at run time. The users who are able to indirectly access the customer table are retrieved by a query.



http://kaon.semanticweb.org/kaon/server/petstore
http://kaon.semanticweb.org/kaon/server/registryschem
http://kaon.semanticweb.org/2001/11/kaon-lexical
http://kaon.semanticweb.org/2001/11/kaon-root

http://kaon.semanticweb.org/2001/11/kaon-lexical#Root
AccessRight
ACL
Actor
APIDescription
Archive
CodeDetails
http://kaon.semanticweb.org/2001/11/kaon-lexical#Lar
http://kaon.semanticweb.org/2001/11/kaon-lexical#Lex
InputGrounding
Interceptor
Interface
InterfaceGrounding
Invocation
KeyValuePair
Library
License
Method
Method
MethodGrounding
Operation
OperationType
OutputGrounding
Parameter
Parameter
Realm
RequestContext
Resource
ResourceGroup
Role
ServiceParameter
SoftwareModule
SoftwareModuleGrounding
SoftwareModuleImplementation
SoftwareModuleProfile
Type

http://kaon.semanticweb.org/kaon/server/security#User

Superconcepts
http://kaon.semanticweb.org/2001/11/kaon-lexical#R

Subconcepts
User
PhysicalUser

Lexicon

Type	Language	Value
Label	English	User

Properties From Concept

Property Name	Minimum Car...	Maximum Ca...
inRole	0	(unlimited)
member	0	(unlimited)
userName	0	(unlimited)

Properties To Concept

Property Name
http://kaon.semanticweb.org/2001/11/kaon-l...
comprises
contextUser
grantedforUser

Concept Instances

Entity Name		Value
User		

Figure 3. The security management tool

We plan to realize a more convenient user interface in order to hide the base query language from the user. Furthermore, we would like to develop a watchdog that actively checks for inconsistencies such as a role with no assigned users or groups, empty groups, or, as shown in section 4.2, clashes in the security settings for inter-resource invocations.

6. Related Work

The current generation of application servers uses XML-based configuration files, some of which follow fixed XML schemata. The individual schemata represent static conceptualizations of fragments of the complete configuration. Our ontology-based approach aggregates individual aspects in a platform-independent and extensible way.

Classical Software Reuse Systems also describe software modules for efficient and precise retrieval. However, techniques like the faceted classification [5] are limited to the representation of the provider's features. In analogy, software reuse shares a representation of modules that is based on functionalities achieved by the software, roles and conditions [10]. [4] introduce a software repository system that uses an ontological representation language for describing information about requirements, designs and implementations of software. However, none of these approaches take into account the run time use cases that occur in application server settings. These efforts either describe different kinds of components or concentrate solely on syntactic or semantic metadata without blending them together like in our approach.

Microsoft's System Definition Model (SDM)[10] takes a similar approach to ours in including information about software, hardware, and network in a unified system model. SDM targets design, deployment, and operation. The first actual software tool implementing this strategy will be the next version of the Visual Studio development environment. Unfortunately, not much detailed information is available at this point. Nevertheless, SDM illustrates the trend of representing different system aspects in a common framework.

[3] shows how description logics can be used to augment CORBA IDL specifications such that *Compatibility testing of IDL specifications*, *Local consistency checking*, *More thorough treatment of exceptions* is possible. However, this approach just augments the syntactic part of an API's description. It does not deal with semantic information about method functionality and does not describe component configurations.

7. Conclusions and Future Work

In this paper we have presented our approach, an ontology as an explicit, executable conceptual model for administrating an application server. Though we are still dealing with a preliminary version of the ontologies, we could demonstrate small, sophisticated and yet very practical examples of how to improve the development and maintenance of software components for and in an application server.

Doing so, our intention was to substantiate a twofold message: First, ontologies and corresponding semantic technology provide huge, practical benefits for handling middleware environments — which we think are *heavily underestimated*. Second, the bread and butter issues of developing and administrating services and components will outlive the utmost fancy issues like automatic composition of services — which we think are *grossly overestimated in feasibility as well as with regard to practical benefits.*

10 http://www.microsoft.com/windowsserversystem/dsi/sdm.mspx

Acknowledgements We are indebted to Marta Sabou, VU Amsterdam, as well as Debbie Richards, MacQuarie University Sydney for their fruitful work on the ontology presented in section 4.1. This work is financed by WonderWeb, an EU Information Society Technologies (IST) funded project (http://wonderweb.semanticweb.-org) IST-2001-33052.

References

[1] G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services*. Springer, Sep 2003.

[2] T. Berners-Lee, J. Hendler, and O. Lassila. The Semantic Web. *Scientific American*, pages 28–37, May 2001.

[3] A. Borgida and P. Devanbu. Adding more DL to IDL: towards more knowledgeable component inter-operability. In *Proceedings of the 21st international conference on Software engineering*. IEEE Computer Society Press, 1999.

[4] P. Constantopoulos, M. Jarke, J. Mylopoulos, and Y. Vassiliou. The software information base: A server for reuse. *VLDB Journal*, 4(1):1–43, 1995.

[5] R. P. Diaz. Implementing faceted classification for software reuse. *Communications of the ACM*, 34(5):88–97, May 1991.

[6] A. Eberhart. Ad-hoc invocation of semantic web services. In *IEEE International Conference on Web Services, July 6-9, 2004, San Diego, California, USA*, 2004.

[7] M. Fleury and F. Reverbel. The jboss extensible server. In *ACM/IFIP/USENIX International Middleware Conference, 2003, Proceedings*, volume 2672 of *LNCS*, pages 344–373. Springer, 2003.

[8] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

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

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

[11] P. Mika, D. Oberle, A. Gangemi, and M. Sabou. Foundations for service ontologies: Aligning owl-s to dolce. In *Proceedings of the 13th International World Wide Web Conference*. ACM, 2004.

[12] B. Motik, A. Maedche, and R. Volz. A conceptual modeling approach for building semantics-driven enterprise applications. In *DOA/CoopIS/ODBASE 2002 Proceedings*, volume 2519 of *LNCS*. Springer, 2002.

[13] D. Oberle, M. Sabou, and D. Richards. An ontology for semantic middleware: extending daml-s beyond web-services. Technical Report 426, University of Karlsruhe, Institute AIFB, 76128 Karlsruhe, Germany, 2003.

[14] D. Oberle, S. Staab, R. Studer, and R. Volz. Supporting application development in the semantic web. *ACM Transactions on Internet Technology (TOIT)*, 4(4), Nov 2004. to appear.

[15] Object Modelling Group. Idl / language mapping specification - java to idl, Aug 2002. 1.2.