Enhancing Application Servers with Semantics

Marta Sabou
Dep. of Artificial Intelligence
Vrije Universiteit Amsterdam
The Netherlands
marta@cs.vu.nl

Daniel Oberle
Institute AIFB
University of Karlsruhe
Germany
oberle@aifb.uni-karlsruhe.de

Debbie Richards
Computing Department
Macquarie University Sydney
Australia
richards@ics.mq.edu.au

Abstract

We report on using semantic technology to enhance application servers. In particular, the conclusion of our analysis is that OWL-S, an emerging effort for semantic web-service descriptions, is a good starting point for supporting many frequent tasks within a concrete application server which facilitates reusing and combining Semantic Web software modules (e.g. ontology stores, reasoners, etc.). The focus of this paper is on detailing the design of the supporting ontology: (1) identifying the aspects of application servers that benefit from semantic technology and (2) analyzing and extending OWL-S for this purpose. We also report on the integration of this ontology within the server.

1. Introduction

Context *Middleware* is a broadly used term nowadays and comes into play as soon as one divides an application into several tiers. Generally speaking, it facilitates multi-tier application development in distributed information systems.

Service oriented architectures (SOAs) are a new kind of middleware that are essentially a collection of services that communicate with each other. The communication can involve either simple data passing or it could involve two or more services coordinating some activity. Some means of connecting services to each other is needed. At the moment the technology of web-services is the most likely connection technology of SOAs.[1] It relies on a set of syntactic standards (XML, SOAP and WSDL [3]).

Current research promises to automate discovery, composition and execution of web-services using Semantic Web techniques. Namely, they propose augmenting web-services with formal, ontological descriptions which can be processed by reasoning engines in order to automate the aforementioned tasks. The OWL-S[2] [4] ontology is a major initiative in this direction. This approach differs from traditional reuse techniques by blending syntactic (WSDL) and formal (Description Logics based) descriptions.

Another, more established kind of middleware are *Application Servers* — component-based products that provide functionality for security and state maintenance, along with data access and persistence for the development of web applications. Despite the comprehensive functionality of application servers, realizing a complex distributed system remains all but an easy task. For instance, managing component dependencies, versions, and licenses is a typical problem. In Microsoft environments, this is often referred to as "DLL Hell". Developers are also confronted with an ever-increasing repository of programming libraries for standard tasks such as IO, networking, database access, or the handling of XML. It would be desirable to assist the developer in using these resources. We propose the addition of semantics to assist with these problems.

Research questions and goals To address this issue, and also, encouraged by the promises of semantic web-services, we strive to semantically enhance application servers. In particular, we wish to answer several research questions:

1. *Can we embed semantic technology in such a server?* — this core question is sub-divided in the following more concrete questions.
2. *What are the requirements for ontology enhanced application servers?* — we investigate the aspects of application servers that can benefit from semantic technology and distill some concrete requirements for a supporting ontology.
3. *Can we reuse similar work from the web-services field?* — namely OWL-S.

We investigate these questions in the context of a concrete application server: the *Application Server for the Semantic Web (ASSW)* [13]. Based on the results of our anal-

1 `http://www.service-architecture.com`

2 We worked with the OWL version of DAML-S v0.9.

ysis, as a proof of concept, we build an ontology to be embedded in the server.

In what follows, section 2 introduces our application server (2.1), presents a set of scenarios that would benefit from semantic descriptions (2.2) and states a set of concrete requirements (2.3), as an answer to our second research question. Section 3 briefly introduces OWL-S and analyzes the degree to which it corresponds to our goals, in line with the third research question. Section 4 presents the ontology we built based on previous analyses in sections 2 and 3. We report on the status of embedding the ontology into the server in section 5. We present related work, conclusion and future work in sections 6 and 7.

2. Motivation

2.1. Application Server for the Semantic Web

Integration of existing software modules is an important issue for the Semantic Web since complex applications require 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 *Application Server for the Semantic Web (ASSW)* [13] addresses this issue by facilitating reuse of existing modules, e.g. ontology stores, editors, and inference engines and, thus, the development and maintenance of comprehensive Semantic Web applications.

The architecture of the application server relies on the Microkernel and component approach. The Microkernel offers a minimal functionality of managing, i.e. starting, stopping and initializing components. Existing software modules have to be made deployable[3] in order to be managed by the Microkernel. This process adds a wrapper around a software module and transforms it into a *Component*. At run-time, the application server may host several components of the same API.

Client software hard-codes the use of a certain API. In order to facilitate the working with a deployed component, a client software can use so-called surrogates which are client-side objects that reveal the same API like a particular component and relay communication to them[4]. Thus, the client is relieved from handling network protocols and middleware idiosyncrasies. At run-time, the application server allows the client to chose which component the surrogate should relay communication to.

3 We use the word deployment as the process of registering, possibly initializing and starting a component to the Microkernel.

4 Similar to stubs in CORBA.

The Application Server for the Semantic Web is based on the design and development of existing application servers, applying and augmenting their underlying concepts for use in the Semantic Web. Even more, we wish to use semantic technology within the server itself in several scenarios as discussed in the next subsection.

2.2. Scenarios

In order to tackle the problems of managing and locating components, we propose to introduce a formal conceptualization of these software related issues. Using this formalism allows representing common knowledge about the domain, such as the fact that component dependencies are transitive. A multitude of software tools can leverage the knowledge specified by means of a reasoning engine. Semantic descriptions of software modules can improve many of the frequently occurring scenarios within an application server. The scenarios listed below apply to any application server but we detail them in the ASSW setting.

Implementation details Libraries often depend on other libraries and a certain archive can contain several libraries at once. Given this information, a system could assist the developer in locating all the required libraries necessary. Furthermore, the user might be notified when two libraries require different versions of a certain third component. For instance, the variously early versions of XML parsers cause a lot of trouble. The system only runs if the libraries are included in the path in a certain order in order to make sure that the class loader picks up the later version. We envision to reason with this kind of data in order to make an educated suggestion in these situations.

Component Discovery At run-time, a client can dynamically decide to which component its surrogate should relay communication. At this moment information other than functionality is important, most prominently certain properties of a component, e.g. if an RDF store component is capable of transactions.

API 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 of a certain taxonomy entry. For example, a developer of client software might need an API that provides ontology *storing* and *retrieving* functionality. We envision that this will be a semantic search, i.e. in terms of concepts describing certain types of functionalities.

Classification of APIs A developer might want to determine the type of a new API based on the type of its offered functionality (i.e. its methods). For example an API offering ontology storage and inferencing capabilities will be of both "StoreAPI" and "InferenceAPI" types.

Publishing web-services Development toolkits usually provide functionality for creating stubs and skeletons or for automatically generating interface descriptions à la java2wsdl. With representation languages like OWL-S, 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 measures. However, having a semantic description of the API, it will be easier to generate the corresponding OWL-S description.

2.3. Requirements

The scenarios discussed above lead us to a set of requirements which served as design principles for the ontology presented in section 4.

R1 Module Implementation and Functionality Syntax The ontology should contain means to describe the implementation and syntax details of software modules which will be used by the application server for implementation tasks.

R2 Module Characteristics and Functionality Semantics The ontology should contain means to give high level descriptions of software modules, e.g. their types such as ontology stores and reasoners, as well as their characteristics, providers etc. This description supports component discovery at run-time. R1 and R2 ensure that software modules' APIs are to be described syntactically and semantically to ensure an easy coupling between both and thus to support reuse of descriptions (see R3).

R3 Reusability and Sharing Semantic descriptions of software modules should be reusable. Easy coupling of syntactic and semantic description is related to that. As an ontology is a shared conceptualization, it should incorporate existing efforts (such as standards and technologies) that are already used by communities.

R4 Domain Independence The ontology should be reusable over a wider range of domains (not just in our Semantic Web domain), therefore we should separate generic and domain specific concepts.

3. Extracting design principles from OWL-S

In line with our desire to support reuse and sharing, and also motivated by requirement R3 of incorporating existing efforts, we have taken OWL-S as a starting point for our ontology.

OWL-S [4] is an OWL ontology conceptually divided into three sub-ontologies for specifying *what a service does* (Profile), *how the service works* (Process) and *how the service is implemented* (Grounding)[5]. The existing grounding allows aligning the semantic specification with implementation details described using WSDL [3], the industry standard for web-service description. There are several interesting design principles underlying OWL-S which inspired us in our work:

1. Semantic vs. Syntactic descriptions OWL-S differentiates between the semantic and syntactic aspects of the described entity. In OWL-S the *Profile* and *Process* ontologies allow for a semantic description of the web-service while the WSDL description simply encodes the syntactic aspects of the service (such as the names of the operations and their parameters). The *Grounding* ontology provides a mapping between the semantic and the syntactic parts of a description facilitating flexible associations between them. For example a certain semantic description can be mapped to several syntactic descriptions if the same semantic functionality is accessible in different ways. The other way around, a certain syntactic description can be mapped to different conceptual interpretations offering different views of the same service. This modelling satisfies our requirements R1 and R2 enforcing separation while, maintaining easy coupling between semantic and syntactic descriptions.

2. Generic vs. Domain knowledge The second principle which underlies the design of OWL-S is the separation between generic and domain knowledge. OWL-S offers a core set of primitives to specify any type of web-service. These descriptions can be enriched with domain knowledge specified in a separate domain ontology. This modelling choice allows using the core set of primitives across several domains just by varying the domain knowledge, as envisaged by our requirement R4.

3. Modularity Another feature of OWL-S is the partitioning of the description over several concepts. The best demonstration for this is the way the different aspects of a description are partitioned in three concepts. As a result a *Service* instance will relate to three instances each of them containing a particular aspect of the service. These are, *ServiceProfile*, *ServiceModel* and *ServiceGrounding*.

5 The OWL-S *Service* ontology contains four concepts. *Service*, *ServiceProfile*, *ServiceModel* and *ServiceGrounding*. The last three are specialized in three sub-ontologies called *Profile*, *Process* and *Grounding* and augmented by additional concepts and properties (cf. also Figure 1).

There are several advantages of this modular modelling. First, since the description is split up over several instances it is easy to reuse certain parts. For example one can reuse the *Profile* description of a certain service. Second, service specification becomes very flexible as it is possible to specify only the part that is relevant for the service (e.g. if it has no implementation one does not need *ServiceModel* and *ServiceGrounding*). Finally, any OWL-S description is easy to extend. If a concept is not appropriate for a certain application domain one can subclass it to a more specialized concept.

Besides all these attractive characteristics of OWL-S, in previous work [16, 17], where we used OWL-S for its original purpose, i.e. the description of web-services, we found a number of shortcomings including the inability to handle overloading, ambiguity between the definition of inputs, outputs, preconditions and effects in the *Process* and *Profile* ontologies and limited ability to express complex internal structures. Both our positive and negative conclusions guided us in designing the ontology we present next.

4. Ontology Design

This section presents our ontology (graphically sketched in the appendix) and shows how our requirements are met. The conclusion of the previous section is that OWL-S can be a good starting point for our own ontology. The main difficulty was in the type of software entities to be described. While OWL-S describes software entities accessible via a web interface, known as web-services, our goal is to describe components and their APIs. As a result some of the parts of OWL-S were not reusable, however many of its underlying ideas proved to be useful in our modelling effort. We provide a comparative overview of our ontology and OWL-S in 4.1. Subsection 4.2 presents each sub-ontology in detail and shows how it implements the specified requirements. Readers interested in a concrete example of a module description showing how all these ontologies are used at instantiation level are referred to [11].

4.1. Overview

The design principles of OWL-S identified in the previous section underpin our work, as comparatively depicted in Figure 1. These principles influenced the kinds of sub-ontologies and their relationships. The following discussion gives the rationale of our design decisions.

1. Semantic vs. Syntactic descriptions We have adopted the separation between semantic and syntactic descriptions in order to achieve a flexible mapping, therefore complying with requirements R1 and R2. A number of our ontologies allow semantic description and others are used for syntactic descriptions. A mapping exists between the description of both aspects. However, given the different type of entities we want to describe, we modified some of the OWL-S ontologies as follows:

- we have kept the OWL-S *Profile* ontology for specifying semantic information about the described components. Also we have extended it with a few concepts for describing the functionality of APIs (and their methods) at the conceptual level. This was necessary because the *Profile* ontology's constructs for specifying functional descriptions were too shallow (see also section 4.2.2). These extensions are grouped in a small ontology called *API Description* which is described in section 4.2.3.
- we did not use the *Process* ontology because our previous analysis [16] yielded that Semantic Web tools usually offer a set of simpler functionalities, but there is no predefined way of invoking them that could be captured in a certain dataflow. Should the type of described components change, our modularly-designed ontology can easily be extended with the Process ontology.
- we defined our own language for describing APIs syntactically since WSDL is designed for specifying network endpoints. For this purpose, we formalized a subset of IDL (Interface Description Language [9]) terms in the *IDL* ontology.
- as a consequence of the changes above, we could not reuse the existing OWL-S *Grounding*, rather we wrote our own grounding ontology (*IDLGrounding* — see 4.2.6) which allows mappings between the conceptual description of the APIs (in the *Profile*) and their syntactic specification (*IDL*).

2. Generic vs. Domain knowledge Currently our core ontology allows specifying semantic and syntactic knowledge about APIs in a generic way facilitating its combination with domain knowledge in line with our desiderata expressed by R4. For our specific goals we have built two domain ontologies in the area of the Semantic Web. The first one specifies the type of existent Semantic Web software modules at a very coarse level. The second one describes the functionality of semantic web specific APIs at a more fine grained level (i.e. in terms of methods and their parameters). Naturally, these ontologies can be easily replaced depending on the application domain, for example bio-informatics.

Our approach can be described in terms of the ONIONS [7] ontology development methodology which advises grouping knowledge with different generality in three separate ontologies. *Generic* theories contain general truths and their concepts are used in defining *Intermediate* knowledge which can be specialized in corresponding *Domain*

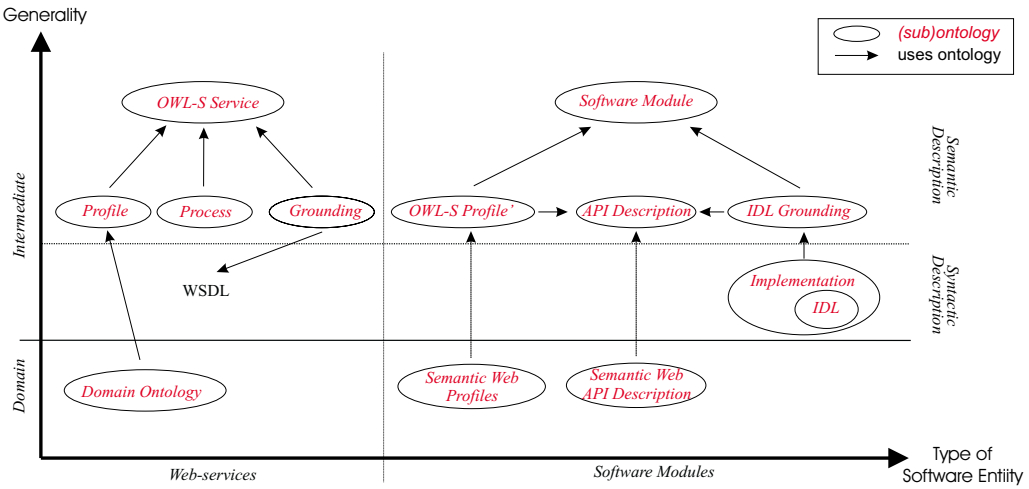


Figure 1. Overview of ontologies

ontologies. From this point of view the OWL-S ontology (and WSDL) is considered to be at the *Intermediate* knowledge level. The same is true for our extensions of OWL-S, as shown in Figure 1. The intermediate knowledge can be specialized in *Domain* ontologies as illustrated by OWL-S and our approach. We are currently aligning our intermediate level to the DOLCE [14] generic ontology.

3. Modularity Modularity enables easy reuse of specifications and extensibility of the ontology. An important issue is the size of the reusable parts. For example, because a *Profile* instance contains a lot of information, which is often very specific such as the contact information of the providers, it is less likely that this instance will be reused by any other description (except if it is provided by the same company). Therefore a coarser granularity (less information per concept) increases the chance of reusability.

We have reused this principle by identifying related content, relating it to a central concept and grouping everything in small ontologies which can be re-used as sub-ontologies. We will describe the process of isolating reusable knowledge in the following section, where we present a short overview of each ontology.

4.2. The sub-ontologies

A second source of inspiration for our design were the requirements put forward in section 2.3. In this section we briefly describe each of our sub-ontologies indicating the scenarios (2.2) that they support as well as the requirements (2.3) that they fulfill. Table 1 shows the relationship between our requirements and sub-ontologies, confirming the major influence that these requirements had on our design.

Requirement \ Sub-ontology	Software Module	OWL-S Profile'	API Description	Implementation	IDL	IDL Grounding	Domain Ontologies
R1 Implementation/Syntax				×	×		
R2 Conceptual/Semantics		×	×			×	
R3 Reuse and Sharing	×	×			×	×	
R4 Domain Independence							×

Table 1. Dependencies between requirements and sub-ontologies.

A more detailed description of the ontologies is available in [11]. Finally, we advise the reader to check the graphical sketch in the appendix for further clarification.

4.2.1. Software Module ontology This ontology is similar to the OWL-S *Service* ontology and thus responds to our requirement R3 of sharing and reuse of existing standards. It contains the main concept and the top concept for each type of description, ensuring a coarse-grained modularity for the whole description. We performed some changes:

- we have renamed the *Service* concept to *SoftwareModule*, as such entities are the focus of our descriptions. Accordingly we have renamed the *ServiceProfile* and *ServiceGrounding* concepts.
- we have excluded the *ServiceModel* concept, since, as stated in 4.1, we are not interested in the internal working of the modules.

- we have added a *SoftwareModuleImplementation* concept that groups together implementation details described in the *Implementation* ontology.

The three concepts that describe a *SoftwareModule* can be specified using the corresponding ontologies as described next.

4.2.2. OWL-S Profile (extension) We use the OWL-S *Profile* ontology to specify the particular characteristics of a *SoftwareModule* such as the contact information of the providers and certain parameters. For example an ontology store would have a service parameter specifying the representation language used. Therefore, our *Profile* describes the component as a whole. Information of this type might be used during component discovery at run-time and corresponds to our requirement R2 of providing generic, high level characteristics of the described modules. Some examples of such parameters are provided in 4.2.7.

We found that the current functional description specification of OWL-S is focused on expressing a single functionality, while we want to describe several functionalities offered by a software module, which correspond to (a set of) methods in the API. Because of that, we have added a new property to *Profile*, namely *hasAPIDescription*, which ranges over the *APIDescription* concept that groups the information used to describe an API and is separated in a small ontology (*API Description*). We separated this content in a small ontology because we expect that many modules will be able to reuse such functionality descriptions (much more than the contact information of the providers). The *Profile* was kept also to support sharing and reuse of existing standards (requirement R3).

4.2.3. API Description The *API Description* ontology offers a framework for semantically describing the functionality offered by methods of APIs (*AddData, RemoveData*) and accordingly several types of APIs (*StoreAPI, InferenceAPI*). As such, it complements the OWL-S *Profile* for our purposes.

The ontology's central concept, called *APIDescription*, can have multiple *hasMethod* properties for instances of type *Method*. Furthermore, each instance of *Method* has a set of parameters such as inputs, outputs, preconditions and effects. Each parameter features a *hasType* property which points to a concept in a domain ontology. Types of *Methods* and *APIDescriptions* are specialized in terms of domain ontology concepts as exemplified in 4.2.7.

This kind of information is used to perform the task of discovering available APIs according to their offered functionality (methods), to classify new APIs (and methods) and to derive OWL-S descriptions for the corresponding webservices. Requirement R2 motivates such semantic functionality descriptions.

4.2.4. Implementation This ontology contains implementation level details of a module and thus responds to requirement R1. There are two aspects of the implementation:

- *CodeDetails* describe characteristics of the code, such as the class that implements the code, the required archives or the version of the code. All these aspects are modelled as properties of the *CodeDetails* concept. Note that these characteristics are specific for a certain implementation and therefore not reusable. They are used during automatic deployment of the components.
- the signature of the interface. The name of the methods and their parameters are modelled using the ontology presented next (*IDL*).

The main concept, *Component* (which is a subclass of *SoftwareModuleImplementation*) bundles an instance of *CodeDetails* and an instance of *Interface* (the class which describes the signature of the API).

4.2.5. IDL We have formalized a small subset of the IDL (Interface Description Language [9]) specification into an ontology that allows describing signatures of interfaces. The *Interface* concept corresponds to a described interface. It features a property *hasOperation* which points to an *Operation* instance. Each *Operation* can have a set of (input) parameters of a certain type. Also each *Operation* returns an *OperationType* (which can also be void). *Interfaces*, *Operations* and *Parameters* have identifiers (which correspond to the names by which they are used in the code). This allows us to specify all the syntactic details needed for automatic invocation of the methods (see requirement R1) using a widely used industry standard (therefore complying to R3). It also addresses requirement R1 of describing implementation details.

4.2.6. IDL Grounding The *IDL Grounding* ontology provides a mapping between the *APIDescription* and the *Interface* description. The mapping is straightforward: concepts *InterfaceGrounding*, *MethodGrounding*, *InputGrounding* and *OutputGrounding* map between respective concepts from the *API Description* and *Implementation* sub-ontologies.

We acknowledge the possibility of redundancy in our approach (given that both the *IDL* and the *API Description* ontologies look similar) but easy reuse and flexible coupling (see R2 and R3) were a higher design goal in this work. Namely, a certain concept level description can be grounded to many different interfaces that may look technically different, i.e. there might be other signatures.

4.2.7. Domain Ontologies We have built two domain ontologies that specialize parts of the generic ontology presented above. By isolating domain knowledge in separate

sub-ontologies, we conform to the OWL-S design principles and implicitly requirement R4 (Domain Independence).

The first one (*Semantic Web Profiles*) generically describes Semantic Web software modules. We have based our ontology on the outcome of an extensive survey in this domain carried out within the OntoWeb project. The survey [15] distinguishes several categories of software modules (ontology building modules, ontology evaluation modules etc.) and for each category proposes a set of characteristics. These characteristics are used as a framework for comparing the actual modules which are presented.

We transformed this information in a domain ontology as follows. We built a taxonomy of categories according to the document. Each category became a subclass of *Profile*. The characteristics of each category were modelled as sub-properties of the OWL-S *serviceParameter*. For example we have created the *OntologyStore* category and added properties such as *queryLanguage*, *representationLanguage* as suggested by the survey. The additional properties (e.g. *QueryLanguage*) are all specializations of *serviceParameter*. We concluded that it was easy to extend OWL-S Profile (the *serviceParameter* property) for modelling the information in the survey. Also, this will allow easy addition of extra knowledge in the future, since the survey only offers a non-exhaustive, reduced set of characteristics.

The second ontology (*Semantic Web API Description*) describes Semantic Web specific functionalities. It contains a set of API and functionality types (methods) which are generally offered. For example we have declared a *StoreAPI* concept, which denotes APIs for storing engines, and defined it as providing an *AddData* method (for adding data into the store) and a *Retrieve* method (for retrieving the data from the store). Note that by combining simple APIs one can create complex ones. For example a *StoreAndQueryAPI* will be obtained by inheriting methods both from a *StoreAPI* and a *QueryAPI*. Further, within a type of API, specializations can be created by declaring extra methods specializing the existing ones. The schema of this ontology is provided by the *API Description* ontology, where APIs are of type *APIDescription* and their functionalities (such as *AddData*) are of type *Method*. We trust that such an ontology will allow performing a flexible search over the existing APIs.

5. Ontology Deployment

We have incorporated this ontology in our application server[6]. Semantic descriptions of the registered components are stored in a central repository, called registry. For more implementation details as well as actual examples of component descriptions we refer to a technical report [12].

Currently, semantic descriptions support two main scenarios as introduced in section 2.2. First, "implementation details" such as loading components are carried out automatically by using (1) the implementation details in each description (according to the *Implementation* ontology) and (2) the reasoning capabilities of the server. For example, the transitive closure of all required libraries by a certain component can be deduced.

Second, "component discovery" is supported at runtime. Therefore, the client can (1) query the registry for available components, (2) chose a component from the returned list (based on its properties) and (3) use that component. For example, OilEd [1], an ontology editor acting as a client, can query for existing ontology stores or reasoners, and then select and interact with any of the available components. This functionality is supported by Profile descriptions.

6. Related Work

Classical Software Reuse Systems are comparable to our work in that they also need to describe software modules appropriately for efficient and precise retrieval. Techniques like the faceted classification [5] are limited to the representation of the provider's features. Analogical software reuse [10] shares a representation of modules that is based on functionalities achieved by the software, roles and conditions. Zaremsky and Wing [19] describe a specification language and matching mechanism for software modules. They allow for multiple degrees of matching but consider only syntactic information. UPML, the Unified Problem-solving Method Development Language [6], has been developed to describe and implement intelligent broker architectures and components to facilitate semi-automatic reuse and adaptation. It is a framework for developing knowledge-intensive reasoning systems based on libraries of generic problem-solving components that are represented by inputs, outputs, preconditions and effects of tasks. Note that these efforts either describe very different kinds of components or concentrate solely on syntactic or semantic descriptions without blending them together.

Another body of related work are adaptations of OWL-S to particular domains. For example, [18] uses an extension to OWL-S for describing web-services in the bioinformatics domain. OWL-S is enriched with speech-acts when describing agent based web-services in [8]. However, none of them actually considers software description at the API level.

IDL is augmented with concepts specified in Description Logics by [2]. More specifically they consider adding

6 Our implementation is called "KAON SERVER" and can be downloaded from `http://kaon.semanticweb.org`

the following kinds of information to an IDL interface: a) data invariants (particularly useful for database-like integrity constraints, b) procedure pre- and post-conditions, c) object behavior models of dynamics. However, this approach just augments the syntactic part of an API's description. It does not deal with semantic information about a method's functionality like our approach.

7. Conclusions and Future Work

The work presented in this paper is motivated by the idea of applying semantic web-services technology to improve application servers in general, and, an application server which supports development of complex Semantic Web applications in particular. So far, we successfully deployed the presented ontology for the "implementation tasks" and "component discovery" scenarios. Current and future work concentrates in supporting the other scenarios as well. Several conclusions emerged during our work.

First, we were able to identify many possible usage scenarios for Semantic Web technology within application servers (in response to our second research question). We also observed that, even if derived in the context of a particular application server, the scenarios lead to requirements which are generally applicable.

Further, we showed that existing work on describing web-services (OWL-S) serves as a good basis for the extension towards an ontology for describing API based software modules (in response to the third research question).

The resulting ontology provides general concepts to describe any API based software. Its similarity with OWL-S is motivated by the observation that web-services often simply reflect all the functionalities offered by an underlying API. Therefore, we believe that the descriptions of a software module offering multiple interfaces should have a single conceptual part and different groundings, one for each interface type. In order to express these considerations formally, to increase the clarity of semantics and offer a richer axiomatization for our ontology we plan to align it to the DOLCE [14] foundational ontology.

References

[1] S. Bechhofer, I. Horrocks, C. Goble, and R. Stevens. OilEd: a reason-able ontology editor for the Semantic Web. In *Proc. of the Joint German Austrian Conference on AI*, volume 2174 of *LNAI*, pages 396–408. Springer, 2001.

[2] 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.

[3] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web Services Description Language (WSDL). Working Draft, March 2003. http://www.w3.org/TR/wsdl.

[4] D. S. Coalition. DAML-S: Semantic Markup for Web Services. DAML-S v. 0.9 White Paper, May 2003.

[5] R. P. Diaz. Implementing Faceted Classification for Software Reuse. *Communications of the ACM*, 34(5):88–97, May 1991.

[6] D. Fensel, R. Benjamins, E. Motta, and B. J. Wielinga. UPML: A framework for knowledge system reuse. In *Proc. of the 16th IJCAI 99*, pages 16–23, 1999.

[7] A. Gangemi, D. M. Pisanelli, and G. Steve. An overview of the ONIONS project: Applying ontologies to the integration of medical terminologies. *Data and Knowledge Engineering*, 31(2):183–220, Sep 1999.

[8] N. Gibbins, S. Harris, and N. Shadbolt. Agent-based Semantic Web services. In *Proc. of the 12th Int. WWW Conference*, pages 710–171. ACM, 2003.

[9] O. M. Group. IDL / Language Mapping Specification - Java to IDL, August 2002. version 1.2.

[10] P. Massonet and A. van Lamsweerde. Analogical reuse of requirements frameworks. In *3rd IEEE International Symposium on Requirements Engineering*, pages 26–39. IEEE Computer Society, 1997.

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

[12] D. Oberle, S. Staab, R. Studer, and R. Volz. KAON SERVER Demonstrator. WonderWeb Project Deliverable, D7, 2003.

[13] 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.

[14] A. Oltramari, A. Gangemi, N. Guarino, and C. Masolo. Sweetening ontologies with DOLCE. In *13th International Conference, EKAW 2002*, volume 2473 of *LNCS*. Springer.

[15] A. G. Perez. A survey on ontology tools. OntoWeb Deliverable 1.3, May 2002. www.ontoweb.org.

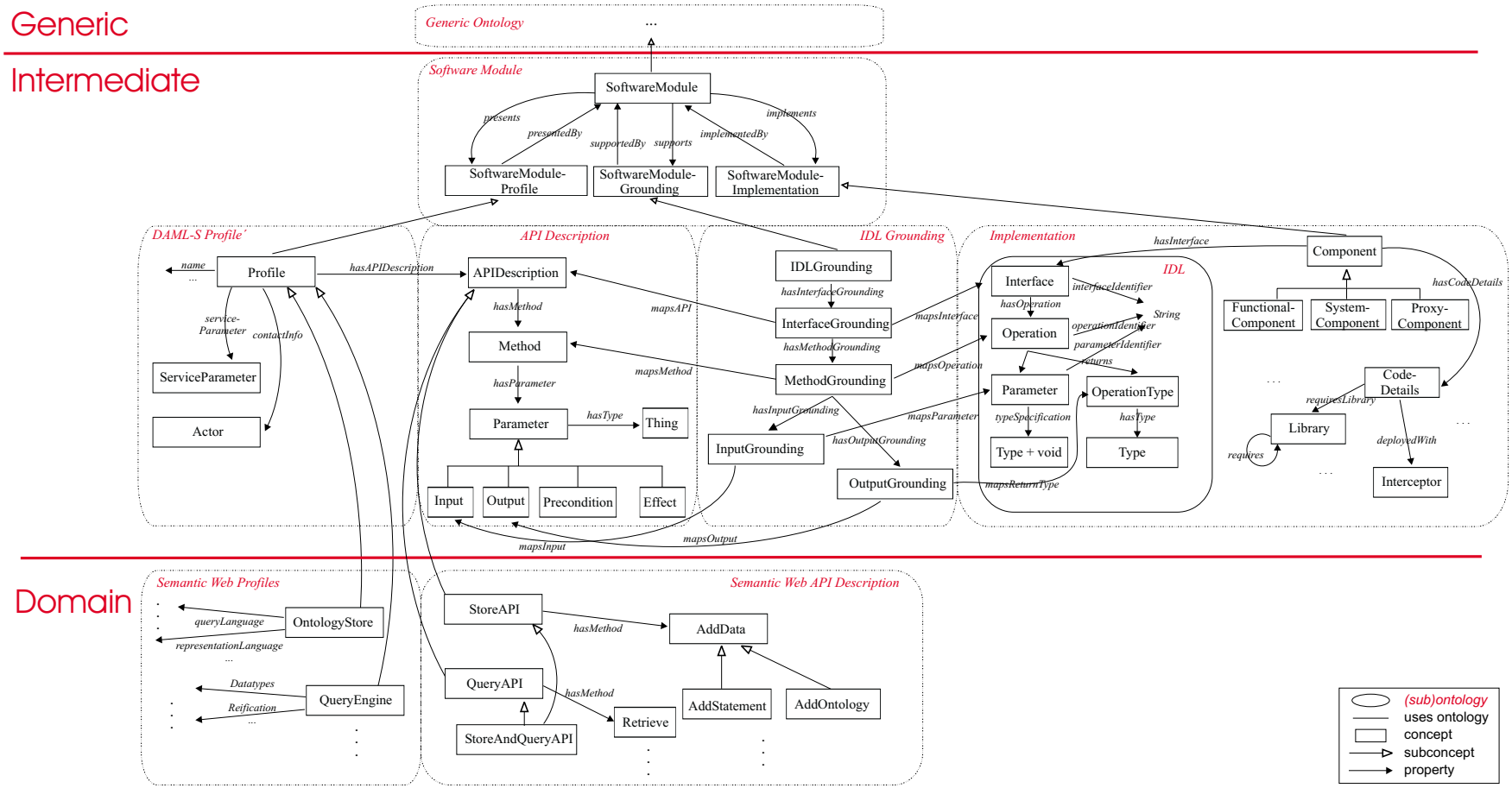
[16] D. Richards and M. Sabou. Semantic Markup for Semantic Web Tools: A DAML-S Description of an RDF-Store. In *2nd Int. Semantic Web Conference (ISWC)*, volume 2870 of *LNCS*, pages 274–289. Springer, Sep 2003.

[17] M. Sabou, D. Richards, and S. van Splunter. An experience report on using DAML-S. In *Proc. of the 12th Int. World Wide Web Conference*, 2003.

[18] C. Wroe, R. Stevens, C. Goble, A. Roberts, and M. Greenwood. A Suite of DAML+OIL ontologies to describe Bioinformatics Web Services and Data. *International Journal of Cooperative Information Systems*, 12(2):197–224, 2003.

[19] A. M. Zaremski and J. M. Wing. Specification matching of software components. *ACM Transactions on Software Engineering and Methodology*, 6(4):333–369, Oct 1997.

APPENDIX


Generic
Intermediate
Domain
Generic Ontology
...
Software Module
SoftwareModule
presents
presentedBy
supportedBy
supports
implementedBy
implements
SoftwareModule-Profile
SoftwareModule-Grounding
SoftwareModule-Implementation
DAML-S Profile
API Description
IDL Grounding
Implementation
IDL
name
Profile
hasAPIDescription
APIDescription
service-Parameter
contactInfo
ServiceParameter
Actor
hasMethod
Method
hasParameter
Parameter
hasType
Thing
Input
Output
Precondition
Effect
mapsAPI
mapsMethod
IDLGrounding
hasInterfaceGrounding
InterfaceGrounding
hasMethodGrounding
MethodGrounding
hasInputGrounding
InputGrounding
hasOutputGrounding
OutputGrounding
mapsInterface
mapsOperation
mapsParameter
mapsReturnType
mapsInput
mapsOutput
hasInterface
Interface
interfaceIdentifier
hasOperation
Operation
operationIdentifier
String
parameterIdentifier
returns
Parameter
typeSpecification
Type + void
OperationType
hasType
Type
Component
hasCodeDetails
Functional-Component
System-Component
Proxy-Component
Code-Details
requiresLibrary
Library
requires
deployedWith
Interceptor
Semantic Web Profiles
queryLanguage
OntologyStore
representationLanguage
Datatypes
Reification
QueryEngine
Semantic Web API Description
StoreAPI
hasMethod
AddData
QueryAPI
Retrieve
AddStatement
AddOntology
StoreAndQueryAPI
(sub)ontology
uses ontology
concept
subconcept
property