An ontology for semantic middleware: extending DAML-S beyond web-services

Daniel Oberle[1], Marta Sabou[2], and Debbie Richards[3]

[1] Institute for Applied Informatics and Formal Description Methods (AIFB)
University of Karlsruhe
Germany
`oberle@aifb.uni-karlsruhe.de`

[2] Department of Artificial Intelligence
Vrije Universiteit Amsterdam
The Netherlands
`marta@cs.vu.nl`

[3] Computing Department
Macquarie University
Sydney, Australia
`richards@ics.mq.edu.au`

Abstract. Describing software entities using Semantic Web technology is a growing research area. Our work investigates the semantic description of software entities that provide an application programmer's interface (API) to allow reasoning with the descriptions and their interrelationships. We present an ontology for our Application Server for the Semantic Web where it is used to facilitate implementation tasks and semantic discovery. Building on an emerging standard from the Semantic Web community, our work includes a number of extensions to DAML-S which currently allows semantic description of a particular type of software entities, viz. web-services, in order to facilitate their automatic discovery and integration. We conclude that while some of the parts and many of the principles embodied in DAML-S provide a good starting point, it was necessary to replace two of the core DAML-S ontologies with ontologies designed for semantic middleware application in order to allow inter-operation of Semantic Web software modules.

1 Introduction

Describing software entities using Semantic Web technology is a growing research area. For example, much work has been done on describing a particular type of software entities, viz. web-services, both on a syntactic level [4] as well as on a semantic level [3]. This paper investigates the semantic description of software entities that provide an application programmer's interface (API), so-called software modules, in order to allow for module and API discovery, classification of modules and to facilitate implementation tasks in semantic middleware. We perform our work in the context of a particular semantic middleware, namely our Application Server for the Semantic Web (ASSW) which facilitates plug'n'play engineering of software modules and, thus, the development and maintenance of comprehensive Semantic Web applications [19, 13, 14].

We want to take advantage of established research and align it to our problem of describing software modules. We have extended DAML-S [3] which is a major initiative in this area and allows semantic description of web-services, in order to facilitate their automatic discovery and integration. We adapted principles and actual parts from DAML-S and extended it for our purposes.

Classical Software Reuse Systems also need to index software modules appropriately for efficient and precise retrieval. Techniques like faceted classification [6] represent features of the providers rather than the goals it achieves. Techniques such as analogical software reuse [18] share a representation of modules that is based on goals achieved by the software, roles and conditions. Zaremsky and Wing [22] describe a specification language and matching mechanism for software modules. They allow for multiple degrees of matching and they consider only type information. UPML [7] represents inputs, outputs, preconditions and effects of tasks. However, none of these approaches provides means for semantic module and API discovery, semantic classification of modules or facilitation of implementation tasks.

Another body of related work are adaptations of DAML-S to particular domains. For example, [21] come up with a DAML-S ontology for describing web-services in the bio-informatics domain. [9] describe speech-acts in agent based web-services. However, none of them actually considers software description at the application interface level.

The work reported in this paper seeks to fill both of these identified gaps. The paper is structured as follows: In section 2 we describe scenarios in which semantic description can be useful for software modules and we motivate them within our Application Server for the Semantic Web setting. We extract a set of requirements for the ontology that will facilitate these descriptions. Section 3 briefly introduces DAML-S and analyzes the degree to which it corresponds to our goals. Based on our analyses in sections 2 and 3, we present in section 4 the main contribution of our work: our ontology for semantic middleware together with a discussion of our design decisions. Finally, section 5 provides some examples of actual descriptions based on the ontology before we conclude and present future work in sections 6 and 7.

2 Motivation

In this section we describe a set of tasks in which semantic description can be useful for software modules and motivate them within our Application Server for the Semantic Web setting. Finally, we extract requirements for the ontology that will facilitate these descriptions.

2.1 Application Server for the Semantic Web

Building a complex Semantic Web application typically requires more than a single software module. Ideally the developer of such a system wants to easily combine different — preferably existing — software modules. So far, however, such integration had to be done ad-hoc, generating a one-off endeavour, with little possibilities for reuse and future extensibility of individual modules or the overall system. We already presented an infrastructure in [19, 14, 13] that facilitates reuse of existing modules, e.g. ontology stores, editors, and inference engines and, thus, the development and maintenance of comprehensive Semantic Web applications, an infrastructure which we call the *Application Server for the Semantic Web (ASSW)*. 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 the terminology of [20], an Application Server for the Semantic Web would be called a *Broker*, i.e. transactions from the client application to the software module are always intermediated and possibly modified by interceptors. Its architecture basically uses a 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[4] in order to be managed by the Microkernel. Thus, a software module becomes a *Component*. In order to distinguish between components that are of direct interest to the developer and components providing functionality for the Application Server itself (e.g. connectors or the registry), we call the first *Functional Components* and the latter *System Components*.

Note that, unlike Multi-Agent or Peer-to-Peer systems where a Broker would have to deal with thousands of agents or peers, only a few components are deployed to a particular Application Server for the Semantic Web. Typically those that are necessary to build the desired applications. The developer can use the Component Loader that facilitates the deployment of the required components. A client application uses surrogate objects for components, similar to stubs in CORBA, that take care of the communication details. For example an ontology editor might use surrogate objects for an ontology store and an inference engine — both would be deployed as functional components in the Application Server.

2.2 Scenarios

In this section we consider a number of scenarios that can be more easily realized if semantic descriptions of software modules are available. Just the development of an ontology is beneficial to gain conceptual agreement between the Application Server and the developer. For example, the ontology formalizes relationships between the internal components of the server. Note that the scenarios listed below are generic for semantic middleware. However, we detail them in our Application Server for the Semantic Web setting.

Implementation tasks The Application Server itself takes advantage of semantic descriptions within several of its system components. For example,

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

the Component Loader facilitates the deployment process by reading an ontological description of a component. The descriptions usually express required libraries and components as properties. Libraries might rely on others, the same holds for components. By defining corresponding properties transitively, it becomes easy to infer all necessary libraries and components.

Component Discovery Typically, an application developer wants to discover components implementing a specific API[5]. For example, an ontology editor might use a specific ontology store, however, there might be several of them deployed in the Application Server. The registry, a system component and simple ontology store, holds semantic descriptions of all deployed components and can be queried accordingly.

API Discovery In another scenario, the developer wants to find a certain API in order to be able to program against it. Preferably, the developer wants to specify high level details and get a comprehensive list of existing APIs that perform desired tasks. As there can be several related and overlapping APIs, the system should recommend the best fitting one. For example, a developer might want to find any component capable of storing ontologies both with inferencing capabilities and transactions. After discovery, the developer could investigate the component and start programming against it. This scenario is especially interesting when several Application Servers are interconnected and reveal their deployed components to each other.

Classification of software modules The ontology should also facilitate the classification of new software modules. Mostly, their APIs fulfill overlapping tasks, e.g. an ontology store offers both inferencing and storing.

Publishing web-services The developer might want to use the functionality hosted in components as web-services. A web-service connector may publish components' methods correspondingly. Having semantic descriptions stored in the registry, it is a simple task to generate appropriate DAML-S descriptions.

2.3 Requirements

After having discussed motivating scenarios for semantic descriptions of software modules, we are now able to derive requirements for the ontology. The requirements serve as the design principles for our ontology which we will present in the next section.

- The ontology should contain means to describe syntactic information of software modules which is to be used by the middleware itself.
- The ontology should contain means to give high level descriptions of software modules, e.g. the different types of software modules as well as their characteristics.

[5] Note, that there is hard-coded integration of modules by the developer, i.e. she programs against a known API. We do not aim for automatic integration.

- Software module APIs are to be described both at a syntactic level and a semantic level for discovery, classification and for publishing methods as web-services.
- Semantic descriptions of software modules should be reusable. Easy coupling of syntactic and semantic description is another requirement to be met. Hence, the ontology should be divided in several sub-ontologies.
- So far we have presented the need for semantic software module description only in our context of the Application Server for the Semantic Web. However, the ontology should be reusable over a wider range of domains.
- As an ontology is a shared conceptualisation, it should use accepted standards that have been investigated for a long time and have a sound basis.

3 Evaluating DAML-S - Extracting Design Principles

In order to support the required sharing and reuse, as stated in the last section, we have taken DAML-S as a starting point for our ontology. By doing so, we maintain compatibility with the web-service world.

DAML-S is an initiative of the Semantic Web community to facilitate automatic discovery, invocation, composition, interoperation and monitoring of web-services (WSs) through their semantic description [5]. DAML-S is a DAML+OIL ontology conceptually divided into three sub-ontologies (cf. Figure 1) for specifying *what a service does?* (Profile), *how the service works?* (Process) and *how the service is implemented?* (Grounding). The existing grounding allows aligning the semantic specification with implementation details described using WSDL [4], the industry standard for web-service description. There are several interesting design principles underlying DAML-S which inspired us in our work:

1. Semantic vs. Syntactic descriptions DAML-S differentiates between the semantic and syntactic aspects of the described entity. In DAML-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 the description what allows 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 depending on the ontology used to disambiguate the meaning of the syntactic descriptions.

2. Generic vs. Domain knowledge The second principle which underlies the design of DAML-S is the separation between generic and domain knowledge. DAML-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.

3. Modularity Another feature of DAML-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, as depicted in Figure 1, ServiceProfile, ServiceModel and ServiceGrounding.

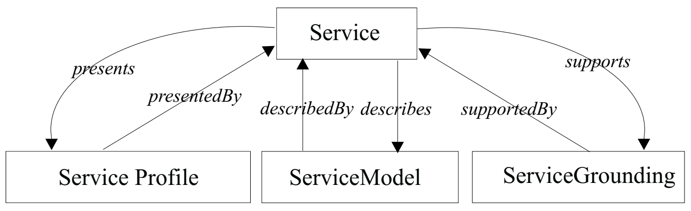


Fig. 1. DAML-S Service Ontology

There are several advantages of this modular modelling:

- Reuse of descriptions. Because 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.
- Flexibility in specification. 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).
- Easy to extend. If the concept that describes a certain aspect is not useful for a certain application domain one can subclass it in a more specialized description.

Despite all these attractive characteristics DAML-S cannot be fully reused for describing software module APIs. This is due to the fact that it was especially designed for web-services descriptions. Therefore,

- the part of the Profile ontology which describes functional characteristics is too simple for our goal of describing APIs and their methods.
- WSDL is not appropriate for syntactic specifications of APIs because it was developed for describing network endpoints.
- the present Grounding was developed for mapping to WSDL, and since we do not use WSDL we cannot use it either.

The above observations played an important role in the design of our ontology. The details of our ontology are given next.

4 The Ontology

4.1 Overview

The conclusion of the previous section is that DAML-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 DAML-S describes software entities that are accessible via a web interface, known as web-services, our goal is to describe software modules, i.e. their APIs as well as other properties. As a result some of the parts of DAML-S were not reusable, however many of the underlying ideas of DAML-S proved to be useful in our modelling effort. Figure 2 presents the main ontologies in DAML-S in comparison with the ontologies we have developed. The following discussion is organised using the design principles in the previous section.

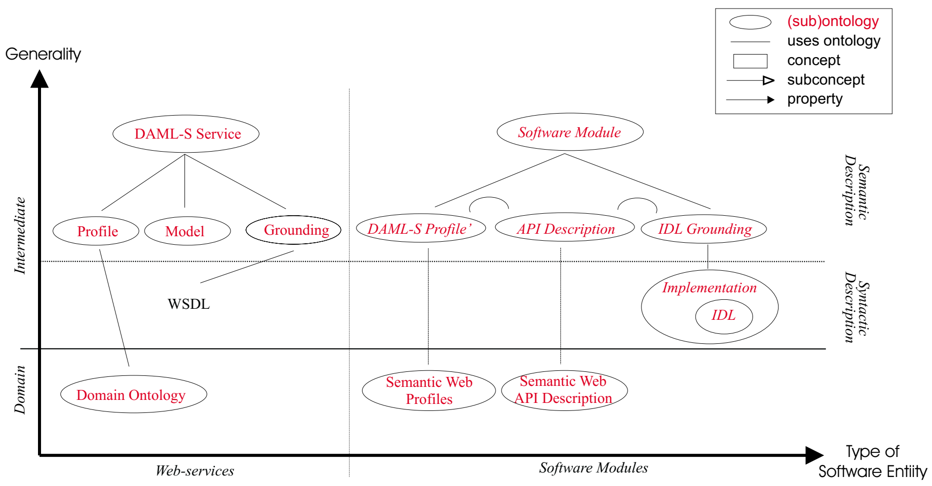


Fig. 2. Ontology overview

1. Semantic vs. Syntactic descriptions We have adopted the separation between semantic and syntactic descriptions in order to achieve a flexible mapping. 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 DAML-S ontologies as follows:

- we have kept the DAML-S Profile ontology for specifying semantic information about the described components. Also we have extended it with a few concepts for describing APIs at the conceptual level. This was necessary because the Profile ontology's constructs for specifying functional descriptions were too shallow. These extensions are grouped in a small ontology called API Description which is described in section 4.2.
- we did not use the Process ontology since we are not interested in the internal working of the modules.
- we defined our own language for describing APIs syntactically since WSDL is designed for specifying network endpoints. We decided to formalize a subset

of IDL (Interface Description Language, cf. [10]) terms in an ontology and to use them to describe the syntactic aspects of APIs.
- as a consequence of the changes above, we could not reuse the existing DAML-S Grounding, rather we wrote our own Grounding Ontology 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. 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 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 [8] 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 is reusable over several domains. From this point of view the DAML-S ontology (and WSDL) is considered to be at the Intermediate knowledge level. The same is true for our extensions of DAML-S, as shown in Figure 2. The intermediate knowledge can be specialized in *Domain* ontologies as done in DAML-S and our approach.

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 making an effort to centralize related content to a certain concept whose instance can be reused at description time. We decided to group together chunks of information that are most likely to be reused. Also, we have grouped this information in small ontologies which are used by other 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

As mentioned above and depicted in Figure 2, our ontology is made up of several sub-ontologies which we now discuss. Note that due to the lack of space we only sketch the ontologies in the figures and present them in isolation to improve the readability. We give an overview in the Appendix.

Software Module ontology This ontology is similar to the DAML-S Service ontology: it contains the main concept and the top concept for each type of description. However, we performed some changes:

– we have renamed the service: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, since we are not interested in the internal working of the modules.
– we have added a SoftwareModuleImplementation concept which groups together implementation details described in the Implementation ontology.

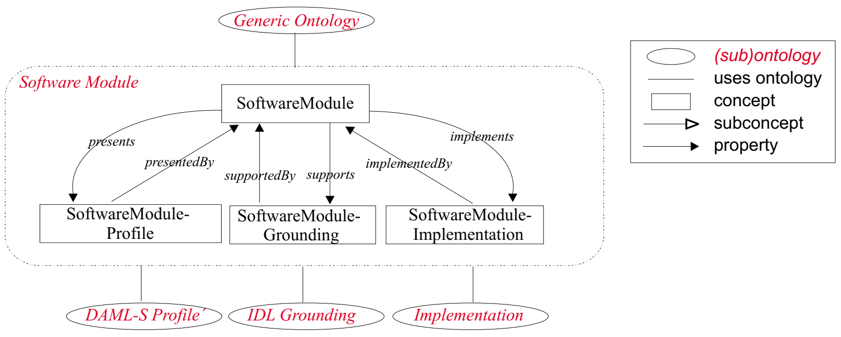


Fig. 3. The Software Module ontology

The three concepts that describe a SoftwareModule (cf. Figure 3) can be specified using the corresponding ontologies as described in what follows.

DAML-S Profile (extension) We use the DAML-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 OntologyStore module would have a service parameter specifying the used representationLanguage. Therefore, our Profile describes the module as a whole. We found that the current functional description specification of DAML-S is too shallow: often one cannot specify inputs and outputs of a web-service since it offers complex functionality. Here, 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 (cf. Figure 4), 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 did this because we expect that many modules will be able to reuse such functionality descriptions (much more than the contact information of the providers)

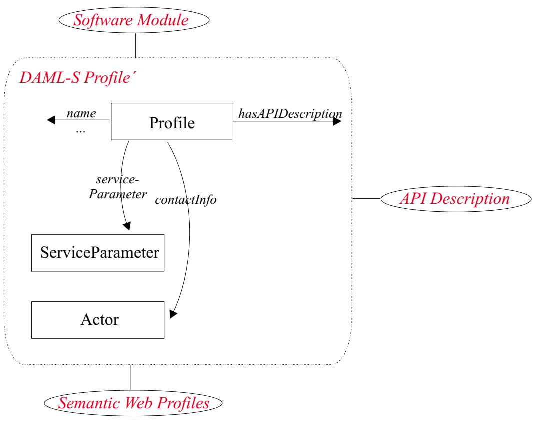


Fig. 4. The DAML-S Profile' ontology

API Description The API Description ontology deals with the API of the module, and as such complements the DAML-S Profile for our purposes.

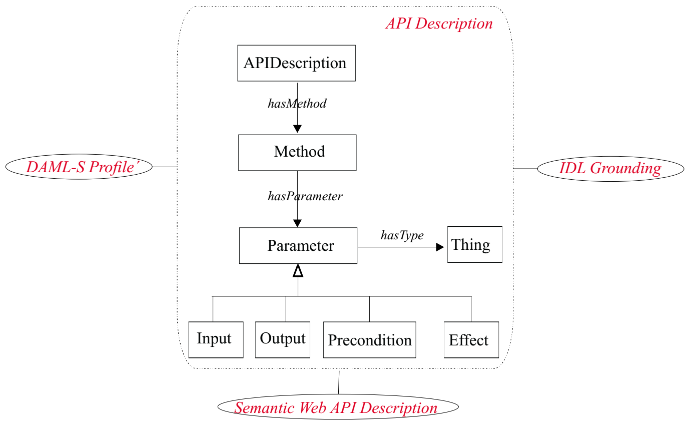


Fig. 5. The API Description ontology

An APIDescription can have multiple hasMethod properties for instances of type Method as depicted in Figure 5. 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 the domain ontology.

Implementation This ontology shown in Figure 6 contains implementation level details of a module. 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.
- the signature of the interface. The name of the methods and their parameters are modelled using the ontology presented next (IDL).

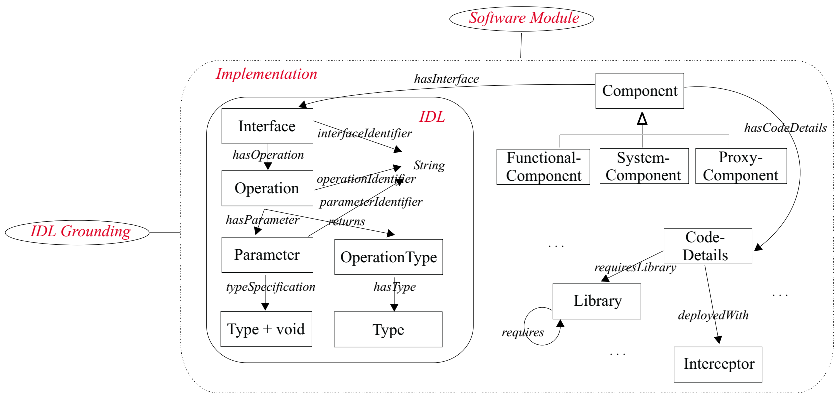


Fig. 6. The Implementation ontology

The main concept, Component (which is a subclass of SoftwareModule-Implementation) links both to an instance of CodeDetails and to an instance of Interface (the class which describes the signature of the API).

IDL We have formalized a small subset of the IDL (Interface Description Language, cf. [10]) 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. Like shown in Figure 6, each Operation can have a set of (input) Parameters of a certain type. Also each Operation returns an OperationType of a certain type (which can be also void). Interfaces, Operations and Parameters have identifiers (which correspond to the names by which they are used in the code).

IDL Grounding The IDL Grounding ontology provides a mapping between the APIDescription and the Interface description. The mapping is straightforward (cf. Figure 7): concepts InterfaceGrounding, MethodGrounding, InputGrounding and OutputGrounding map between respective concepts from the API Description and Implementation sub-ontologies.

We acknowledge the possiblity of redundancy in our approach (given that both the IDL and the API Description ontologies look similar) but easy reuse was 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.

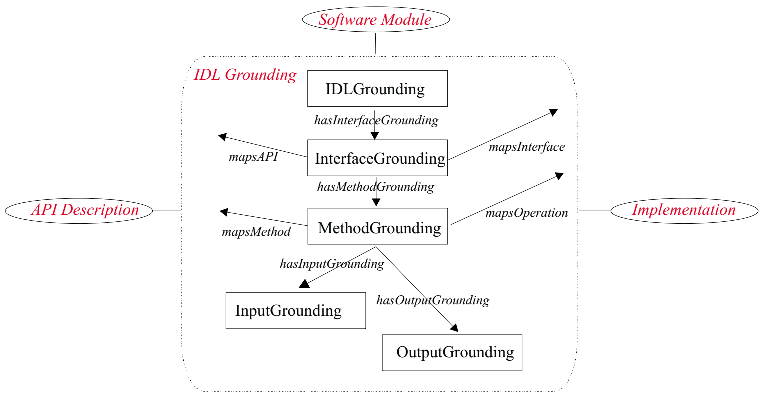


Fig. 7. The Grounding ontology

Domain Ontologies We have built two domain ontologies shown in Figure 8. The first one (Semantic Web Profile) 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 [17] 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 is a kind of damls:Profile, therefore we regard them as subclasses of Profile. The characteristics of each category were modelled as properties of the concept denoting the category and described as sub-properties of the damls:serviceParameter property.

For example we have created the OntologyStore category and added properties such as queryLanguage, representationLanguage suggested by the survey. We concluded that the serviceParameter construct of DAML-S was very easy to extend for modelling the information in the survey. Even more the current ontology can be easily extended with extra knowledge. This will be very helpful since the survey only offers a reduced set of characteristics which can be easily extended.

The second ontology (Semantic Web API Description) is meant to describe specific functionalities offered by different modules. This domain ontology introduces a set of API types and functionalities (methods) which are generally

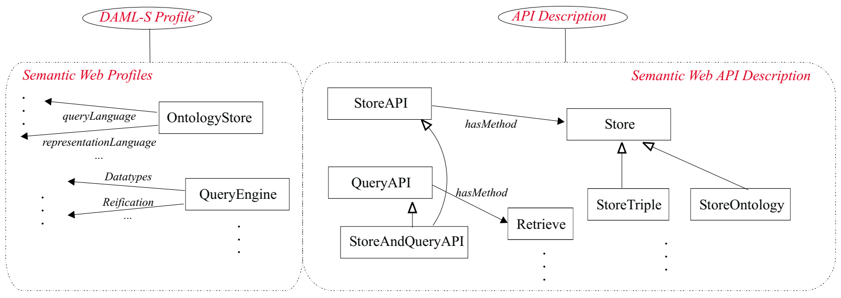


Fig. 8. The Domain ontologies

offered. For example we have declared a StoreAPI concept which denotes APIs for storing engines. A StoreAPI provides a Store method (for saving data into the store) and a Retrieve method for retrieving the data from the store. This is a generic declaration which can be specialized according to the case. For example, StoreOntology can be a special Store method which saves a whole ontology in the store. Similarly, StoreTriple, adds a single RDF [12] statement. The methods differ in the type of their inputs, but nevertheless they perform the same action of adding data to the store. StoreOntology is declared as a subclass of Store with a restriction that the input can be of type Ontology only.

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 of specializing the existing ones. We trust that such an ontology will allow performing a flexible search over the existing APIs.

All domain data is a specialization of the API Description ontology, where APIs are of type APIDescription and their functionalities (such as Store) are of type Method.

5 Examples

In this section we will demonstrate what descriptions of software modules look like. We consider three existing OntologyStores that take RDF [12] as their representationLanguage. We assume that all of them are deployed to an Application Server for the Semantic Web as functional components:

– KAON RDF Main Memory - an OntologyStore that is transient, implementing the RDF API as used in the Karlsruhe Ontology and Semantic Web Toolsuite (KAON, cf. http://kaon.semanticweb.org and [1]).
– KAON RDF Server - a store that implements the same API as above, however, it applies a database system for actual storage.
– Sesame [2] - a well-known store that implements its own API.

For the sake of brevity we only want to illustrate two methods for each API. Note, that we omit fully qualified classnames to improve the readability.

```
KAON RDF API:
  void  add(Statement statement)
  Model find(Resource subject, Resource predicate, RDFNode object)
Sesame API:
  int addDataFromUrl(String dataURL, String baseURL)
  String[][] evalRqlQuery(String query)
```

We provide a semantic description for each software module where one store will be described completely, for the others we will just show how they relate to the first one. We declare KAONRDFMainMemory as a SoftwareModule that links to three instances containing different aspects of the description.

```
<softwareModule:SoftwareModule rdf:ID="KAONRDFMainMemory">
  <damlservice:presents rdf:resource="#MM_Profile"/>
  <damlservice:supports rdf:resource="#KAONRDFAPIGrounding"/>
  <softwareModule:implements rdf:resource="#MM_Impl"/>
</softwareModule:softwareModule>
```

From the profile point of view KAONRDFMainMemory is an OntologyStore, therefore we can describe all the associated properties which we declared in the Semantic Web Profiles ontology. The Profile also includes contact information and a pointer to the APIDescription instance.

```
<swProfiles:OntologyStore rdf:ID="MM_Profile">
  <damlservice:presentedBy rdf:resource="#KAONRDFMainMemory"/>
  <damlprofile:serviceName>
    KAONOntologyStore</damlprofile:serviceName>
  <swProfiles:platform rdf:resource="swProfiles#Any"/>
  <swProfiles:ontologyLanguage rdf:resource="swProfiles#RDF"/>
  <ourdamlprofile:hasAPIDescription rdf:resource="#RDFAPI"/>
</swProfiles:OntologyStore>
```

Note that all the above information is specific to KAONRDFMainMemory. However, the APIDescription can be reused by other modules as well. In terms of our domain ontology the RDFAPI is a StoreAndQueryAPI since it offers both adding data and querying the repository. The description of the API is as follows:

```
<swApis:StoreAndQueryAPI rdf:ID="RDFAPI">
  <apiDescr:hasMethod>
    <swApis:StoreTriple rdf:ID="RDFAPI_StoreTriple">
      <apiDescr:hasParameter>
        <apiDescr:Input rdf:ID="StatementForStore">
          <apiDescr:hasType
            rdf:resource="swProfiles#OntologyStatement"/>
        </apiDescr:Input>
      </apiDescr:hasParameter>

      <apiDescr:hasParameter>
        <apiDescr:Output rdf:ID="NoOutput">
          <apiDescr:hasType
            rdf:resource="swProfiles#NoOutput"/>
        </apiDescr:Output>
      </apiDescr:hasParameter>
    </swApis:StoreTriple>
  </apiDescr:hasMethod>
  <apiDescr:hasMethod>
    <swApis:Query rdf:ID="RDFAPI_Query">
     ...
    </swApis:Query>
 </apiDescr:hasMethod>
</swApis:StoreAndQueryAPI>
```

Furthermore, we declare the technical details of the module. It is deployed within an Application Server for the Semantic Web as FunctionalComponent and has some code details as well as an API declaration.

```
<impl:FunctionalComponent rdf:ID="MM_Impl">
  <impl:hasCodeDetails>
    <impl:CodeDetails rdf:ID="MM_CodeDetails">
      <impl:requiresLibrary rdf:resource="#someLibraryDecl"/>
      ...
    </impl:CodeDetails>
  </impl:hasCodeDetails>
  <impl:hasInterface rdf:resource="#KAONRDFInterface"/>
</impl:FunctionalComponent>
```

After providing the syntactic description for the API as shown below, the final step would be writing a grounding of RDFAPI to KAONRDFInterface, called KAONRDFAPIGrounding.

```
<idl:Interface rdf:ID="KAONRDFInterface">
  <idl:interfaceIdentfier>
    edu.unika.aifb.rdf.api.model
  </idl:interfaceIdentfier>

  <idl:hasOperation>
    <idl:Operation rdf:ID="addOp">
      <idl:operationIdentifier>add</idl:operationIdentifier>
      <idl:hasParameter>
        <idl:Parameter rdf:ID="statement">
          <idl:parameterIdentifier>statement</idl:parameterIdentifier>
          <idl:typeSpecification>
            edu.unika.aifb.rdf.api.model.Statement
          </idl:typeSpecification>
        </idl:Parameter>
      </idl:hasParameter>
      <idl:returns>
        <idl:OperationType rdf:ID="response">
          <idl:hasType>void</idl:hasType>
        </idl:OperationType">
      </idl:returns>
    </idl:Operation>
  </idl:hasOperation>

  <idl:hasOperation>
    <idl:Operation rdf:ID="queryOp">
     ...
    </idl:Operation>
  </idl:hasOperation>
</idl:Interface>
```

The definition of the KAON RDF Server is very similar. We provide the same type of descriptions and we can reuse the RDFAPI (the semantic description of the API), KAONRDFInterface (the syntactic description) and KAONRDFAPI-Grounding (the grounding between these two aspects).

When describing Sesame we cannot reuse any API related descriptions from the previous descriptions. We will only present the semantic description and show the main difference that lies in the type of the loading method.

```
<swApis:StoreAndQueryAPI rdf:ID="SesameAPI">
  <apiDescr:hasMethod>
    <swApis:LoadOntology rdf:ID="SesameAPI_LoadOntology">
      <apiDescr:hasParameter>
        <apiDescr:Input rdf:ID="OntologyForLoad">
          <apiDescr:hasType
            rdf:resource="swProfiles#Ontology"/>
        </apiDescr:Input>
      </apiDescr:hasParameter>

      <apiDescr:hasParameter>
        <apiDescr:Output rdf:ID="LoadedStatements">
          <apiDescr:hasType
            rdf:resource="swProfiles#LoadedStatements"/>
        </apiDescr:Output>
      </apiDescr:hasParameter>
    </swApis:LoadOntology>
  </apiDescr:hasMethod>
```

```
      <apiDescr:hasMethod>
      <swApis:Query rdf:ID="SesameAPI_Query">
       ...
      </swApis:Query>
   </apiDescr:hasMethod>
</swApis:StoreAndQueryAPI>
```

Note that the number of inputs in the signature of the method can be different from the number of inputs in the semantic description. In this particular case, semantically the only parameter is the ontology to be loaded. However, when it comes to implementation, there are actually two parameters needed to acquire the ontology.

6 Conclusions

This paper presented a basic set of ontologies for the semantic description of software modules. The ontologies are written for software modules in general. However, they can be used in combination with arbitrary domain ontologies. Here, we have complemented them with domain ontologies specific to Semantic Web software modules in the context of our semantic middleware called Application Server for the Semantic Web. By applying semantics, we are able to allow reasoning with API descriptions and their interrelationships — which, to the best of our knowledge, hasn't been offered before.

We showed that existing work on describing web services (DAML-S, cf. [3]) serves as a good basis for the extension towards an ontology for describing software modules in general and Semantic Web software modules in particular. We've adopted the distinction between profile and grounding from DAML-S and developed our own grounding ontology based on the IDL terminology [10].

We designed the ontologies in a way such that a description of a web service can easily be adapted to a software module description. In addition, reuse of API descriptions, which may be grounded differently, is supported.

7 Future work

The single ontologies presented in this paper have only been sketched. In a next step we have to define them concisely, in particular our domain ontologies, i.e. the Semantic Web API Description and Profile. IDL remains to be formalized as well since only its core was conceptualized and presented.

In the future we plan to detail how the semantic matching will take place. In contrast to simply querying the registry, semantic matching supports the application developer with some intelligence in finding the desired component (cf. [16]). For example, semantic matching can aid selection of a particular component from among a number of similar components even when the capability description given by the developer is not precise.

Another task that was not detailed in this paper and remains to be done is the mapping to a top-level ontology. We plan to use DOLCE [15] and verify our mapping with the OntoClean [11] methodology. In the more distant future, we plan to semi-automatically extract ontological descriptions out of the code.

Acknowledgements This work is financed by WonderWeb, an EU Information Society Technologies (IST) funded project (http://wonderweb.semanticweb.org). Participants are the University of Manchester, the Vrije Universiteit Amsterdam, LADSEB-CNR, Padova, Italy and the University of Karlsruhe, Germany.

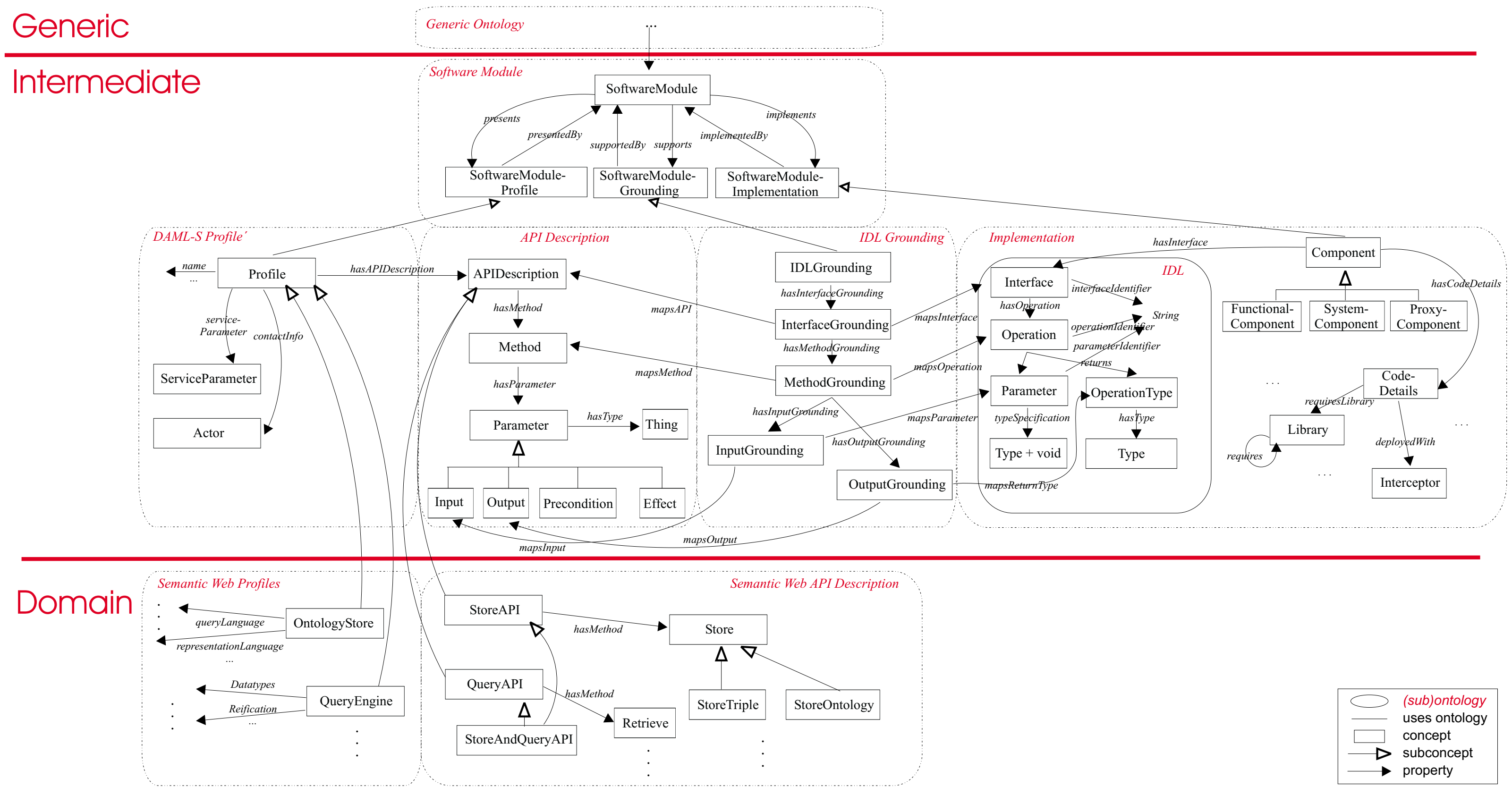
We would like to thank our colleagues, in particular Sudhir Agarwal, Peter Haase, Steffen Staab, Rainer Tellmann and Raphael Volz as well as all WonderWeb researchers for fruitful discussions that helped to shape this contribution.

References

1. E. Bozsak, M. Ehrig, S. Handschuh, A. Hotho, A. Maedche, B. Motik, D. Oberle, C. Schmitz, S. Staab, L. Stojanovic, N. Stojanovic, R. Studer, G. Stumme, Y. Sure, J. Tane, R. Volz, and V. Zacharias. KAON - towards a large scale Semantic Web. In K. Bauknecht, A. M. Tjoa, and G. Quirchmayr, editors, *E-Commerce and Web Technologies, Third International Conference, EC-Web 2002, Aix-en-Provence, France, September 2-6, 2002, Proceedings*, volume 2455 of *Lecture Notes in Computer Science*. Springer, 2002.
2. J. Broekstra, A. Kampman, and F. van Harmelen. Sesame: A generic architecture for storing and querying RDF and RDF Schema. In I. Horrocks and J. A. Hendler, editors, *The Semantic Web - ISWC 2002, First International Semantic Web Conference, Sardinia, Italy, June 9-12, 2002, Proceedings*, volume 2342 of *Lecture Notes in Computer Science*. Springer, 2002.
3. M. H. Burstein, J. R. Hobbs, O. Lassila, D. Martin, D. V. McDermott, S. A. McIlraith, S. Narayanan, M. Paolucci, T. R. Payne, and K. P. Sycara. DAML-S: Web service description for the Semantic Web. In I. Horrocks and J. A. Hendler, editors, *The Semantic Web - ISWC 2002, First International Semantic Web Conference, Sardinia, Italy, June 9-12, 2002, Proceedings*, volume 2342 of *Lecture Notes in Computer Science*, pages 348–363. Springer, 2002.
4. R. Chinnici, M. Gudgin, J.-J. Moreau, and S. Weerawarana. Web services description language (wsdl). Working Draft, Mar 2003. Working Draft.
5. D. S. Coalition. DAML-S: Semantic Markup for Web Services. DAML-S v. 0.9 White Paper, May 2003.
6. R. P. Diaz. Implementing faceted classification for software reuse. *Communications of the ACM*, 34(5):88–97, May 1991.
7. D. Fensel, R. Benjamins, E. Motta, and B. J. Wielinga. UPML: A framework for knowledge system reuse. In T. Dean, editor, *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence, IJCAI 99, Stockholm, Sweden, July 31 - August 6, 1999. 2 Volumes, 1450 pages*, pages 16–23. Morgan Kaufmann, 1999.
8. A. Gangemi, G. Steve, and F. Giacomelli. ONIONS: An ontological methodology for taxonomic knowledge integration. In *Proceedings of ECAI-96 Workshop on Ontological Engineering, Budapest, August 13th.*, 1996.

9. N. Gibbins, S. Harris, and N. Shadbolt. Agent-based semantic web services. In *Proceedings of the Twelfth International World Wide Web Conference WWW12, 20-24 May 2003, Budapest, Hungary*, pages 710–171. ACM, 2003.
10. O. M. Group. Idl / language mapping specification - java to idl, Aug 2002. 1.2.
11. N. Guarino and C. A. Welty. Evaluating ontological decisions with ontoclean. *Communications of the ACM*, 45(2):61–65, Feb 2002.
12. O. Lassila and R. Swick. Resource description framework (RDF) model and syntax specification. W3C Recommendation, Feb 1999. http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/.
13. D. Oberle, S. Staab, R. Studer, and R. Volz. Supporting application development in the Semantic Web. Technical report, University of Karlsruhe, Institute AIFB, 2003. http://www.aifb.uni-karlsruhe.de/WBS/dob/pubs/ACM.pdf.
14. D. Oberle, R. Volz, B. Motik, and S. Staab. *An extensible open software environment.* International Handbooks on Information Systems. Springer, 2003.
15. A. Oltramari, A. Gangemi, N. Guarino, and C. Masolo. Sweetening ontologies with dolce. In A. Gómez-Pérez and V. R. Benjamins, editors, *Knowledge Engineering and Knowledge Management. Ontologies and the Semantic Web, 13th International Conference, EKAW 2002, Siguenza, Spain, October 1-4, 2002, Proceedings*, volume 2473 of *Lecture Notes in Computer Science.* Springer, 2002.
16. M. Paolucci, T. Kawamura, T. R. Payne, and K. P. Sycara. Semantic matching of web services capabilities. In I. Horrocks and J. A. Hendler, editors, *The Semantic Web - ISWC 2002, First International Semantic Web Conference, Sardinia, Italy, June 9-12, 2002, Proceedings*, volume 2342 of *Lecture Notes in Computer Science*, pages 333–347. Springer, 2002.
17. A. G. Perez. A survey on ontology tools. OntoWeb Deliverable 1.3, May 2002. www.ontoweb.org.
18. P.Massonet and A. Lamsweerde. Analogical reuse of requirements frameworks. In *Proceedidngs of the 3rd IEEE International Symposium on Requirements Engineering (RE 97)*, pages 26–39, 1997.
19. R. Volz, D. Oberle, S. Staab, and B. Motik. KAON SERVER - a Semantic Web Management System. In *Proceedings of the Twelfth International World Wide Web Conference WWW12, Alternate Tracks, Practice and Experience, 20-24 May 2003, Budapest, Hungary*, 2003.
20. H. C. Wong and K. Sycara. A taxonomy of middle-agents for the internet. In *Proceedings of the Fourth International Conference on MultiAgent Systems*, pages 465 – 466, July 2000.
21. 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.
22. A. M. Zaremski and J. M. Wing. Specification matching software components. *ACM Transactions on Software Engineering and Methodology*, 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'
name
...
Profile
hasAPIDescription
service-Parameter
contactInfo
ServiceParameter
Actor
API Description
APIDescription
hasMethod
Method
hasParameter
Parameter
hasType
Thing
Input
Output
Precondition
Effect
mapsAPI
mapsMethod
mapsInput
mapsOutput
IDL Grounding
IDLGrounding
hasInterfaceGrounding
InterfaceGrounding
hasMethodGrounding
MethodGrounding
hasInputGrounding
hasOutputGrounding
InputGrounding
OutputGrounding
mapsInterface
mapsOperation
mapsParameter
mapsReturnType
Implementation
IDL
Interface
interfaceIdentifier
hasOperation
Operation
operationIdentifier
String
parameterIdentifier
returns
Parameter
OperationType
typeSpecification
hasType
Type + void
Type
hasInterface
Component
hasCodeDetails
Functional-Component
System-Component
Proxy-Component
Code-Details
requiresLibrary
Library
requires
deployedWith
Interceptor
...
Semantic Web Profiles
queryLanguage
OntologyStore
representationLanguage
Datatypes
QueryEngine
Reification
Semantic Web API Description
StoreAPI
hasMethod
Store
QueryAPI
hasMethod
Retrieve
StoreTriple
StoreOntology
StoreAndQueryAPI
(sub)ontology
uses ontology
concept
subconcept
property