Towards Ontologies for Formalizing Modularization and Communication in Large Software Systems

Daniel Oberle,[1] Steffen Lamparter,[2] S. Grimm,[3] D. Vrandečić,[2]
S. Staab,[4] A. Gangemi[5]

[1]SAP Research, CEC Karlsruhe, Germany, d.oberle@sap.com
[2]AIFB, Universität Karlsruhe, Germany, lastname@aifb.uni-karlsruhe.de
[3]FZI Research Center Karlsruhe, Germany, grimm@fzi.de
[4]ISWeb, University of Koblenz-Landau, Germany, staab@uni-koblenz.de
[5]LOA, Laboratory of Applied Ontology, Italy, aldo.gangemi@istc.cnr.it

May 16, 2006

Abstract

Large software systems are modularized in order to improve manageability. The parts of the software system communicate in order to achieve the desired functionality. To better understand, develop, manage, and maintain the resulting complexity, this paper presents a framework of ontologies. The ontologies range from very general, foundational ones to ontologies that elucidate the specificities of particular modularization and communication paradigms. We support two specific paradigms. First, we define an ontology for software components that may be used in traditional middleware architectures, e.g., application servers. Second, we specify an ontology for Web services. Through the reuse of existing foundational ontologies and our new Core Software Ontology, our proposal offers several advantages. In particular, it avoids the typical shortcomings related approaches exhibit and it allows for the concise definition of commonalities and differences of the two paradigms.

1 Introduction

The understanding and management of *complex systems* often require theories that may cover several levels of abstraction. The reason is that even if a complex system is completely defined at a fine-grained micro level, it is often impractical if not impossible for some spectator or user to understand how the very same system behaves at a more abstract, a macro level.

Large software systems are complex systems for which these observations obviously hold. Even though the lines of code define precisely the behavior

of the system, such a view is impractical and often infeasible when someone considers a large software system. There are formal theories for *smaller software systems* that account for the meaning of the code and the behavior of such systems and they have led to important fields of application, e.g., software verification. However, there is a lack of corresponding abstracting formal theories of how software systems behave at large that would also take account of the concepts that software experts use to develop large software systems, as well as to monitor, maintain, and run them.

This paper proposes several ontologies as formal theories to describe some crucial aspects of software systems and software systems behavior at large. In particular, this paper focuses on aspects of *modularization* of and *communication* within large software systems. Modularization and communication have been understood by the software engineering community as core vehicles to organize large software systems. Correspondingly, concepts and technology have been developed to support this organization.

Currently, we see two major strands of such concepts and technologies. The first one is focused around various notions of software components and middleware technologies such as collected, e.g., in *application servers*. This is established practice and the fibre of many large-scale software systems [Mah04]. The second strand accentuates the need for an even stronger decoupling of software and wider distribution of software processes through *service-oriented architectures*, frequently realized through *Web services* concepts and technologies [ACKM03]. The efforts in this second area are still under heavy development.

So far, ontological descriptions of modularization of and communication within/between large software systems have dealt almost exclusively with the moving target of Web services: OWL-S [MBH+04], WSMO [RKL+05], or METEOR-S [POSV04] are prominent representatives of this line of research. In spite of their seminal roles that led to a lot of fruitful research into the formulation and application of Web service ontologies, they exhibit several characteristics that are undesirable for ontologies that aim to become *reference ontologies* and, hence, wide-spread standard. For instance, (*i*), they are conceptually ambiguous, leading to undesirable misunderstandings and, (*ii*), they are hard to integrate into overarching frameworks, because they are not well-founded upon an upper-level ontology. These shortcomings have been analyzed in depth in [MOGS04] and, hence, led to the effort to define a well-founded reference ontology in this domain that would be able to deal with common use cases (such as the ones elaborated in Section 2) and meet quality criteria as briefly, though incompletely indicated by *(i)* and *(ii)*. In particular, our proposal also deals with the lopsided situation that the ontological description of Web services has received great attention, but the ontological description of the (so far) practically more relevant middleware concepts and technologies has been entirely neglected. In doing so, we show that the conceptual difference between the two paradigms is in

fact quite small.

Structure of this Paper In the following, we illustrate two use cases (from a larger set of use cases described in [OESV04, OLES05]), which allow us to derive requirements for the targeted ontologies (Section 2) and we explain the typical shortcomings of existing ontologies that we avoid. To respond to the requirements in a flexible, extensible manner, we provide a framework of a set of ontologies. In fact, some of them (i.e., DOLCE, Descriptions & Situations, Ontology of Plans, and the Ontology of Information Objects as also depicted in Figure 1) we just reuse. These ontologies are surveyed in Section 3 to make this paper self-contained. The original contribution of this paper follows subsequently:

- The Core Software Ontology (CSO): provides a new extensible foundation for describing software, in general (Section 4).
- The Core Ontology of Software Components (COSC): provides specialized concepts needed for software components and application servers (Section 5).
- The Core Ontology of Web Services (COWS) reuses all the other ones in order to establish a well-founded ontology for Web services (Section 6).

All ontologies can be obtained from `http://cos.ontoware.org`. Finally, Section 7 uses the shortcomings introduced in Section 2 in order to validate our proposed approach. The description is illustrated by an elaborate example running through from the use cases to the validation of the ontologies. Before we finally conclude, we survey related work. In order to restrict this paper to a reasonable size, we only elaborate on why and how to *engineer* the ontologies described in this paper. We demonstrate how to *use* the ontologies, i.e., how to exploit them in a middleware environment and how to provide formalized data corresponding to the ontologies, in the complementary article [OSE06].

2 Use Cases and Requirements

Although our approach can be applied to large software systems in general, we provide two short use cases to demonstrate the difficulties in managing distributed applications in this section (2.1 and 2.2, respectively). We argue that one reason for that is the missing conceptual coherence of application server and Web service descriptors. We motivate that a careful and rigorous modelling of the computational domain is necessary to succeed. Finally, 2.3 extracts and lists modelling requirements for the ontology to be formalized in the remainder of this paper.

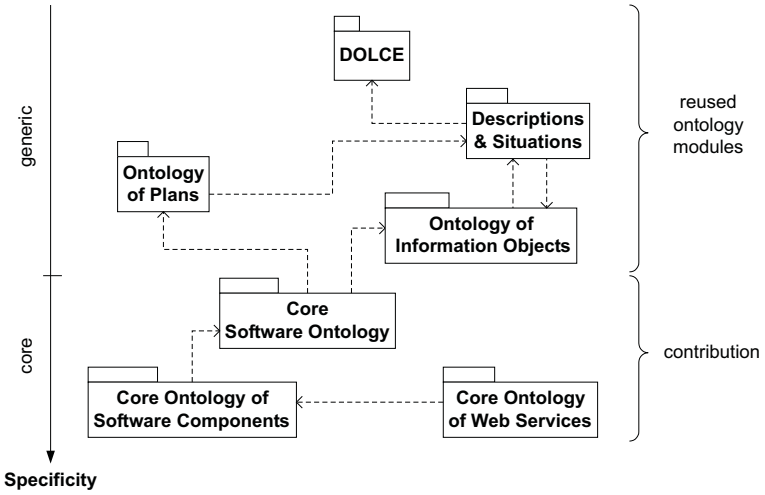

Figure 1: Overview of the ontologies as UML package diagram. Packages represent ontologies; dotted lines represent dependencies between ontologies. An ontology O_1 depends on O_2 if it specializes concepts of O_2, has associations with domains and ranges to O_2 or reuses its axioms.

2.1 Access Rights of Software Components

The increasing use of the Web as a channel to access information systems forced conventional middleware platforms to provide support for Web access. This support is typically associated with *application servers* [ACKM03]. On the upside, application servers offer a wealth of functionality to the developer. On the downside, application servers have become very complex software systems often subsuming conventional middleware in one product. Examples are IBM WebSphere or JBoss. Application servers foster component-based software engineering and introduce the use of deployment descriptors. *Deployment descriptors*, or short descriptors, are XML files that describe and configure components in a declarative way.

Developing and managing software components in an application server is a difficult task even though deployment descriptors reduce coding efforts. The reason is that the conceptual model underlying the different descriptors is only *implicit*. Thus, its bits and pieces are difficult to retrieve, check for validity, and maintain. Rather, it would be desirable to query a system from different perspectives, e.g., are there any user accounts with indirect permission to resources? And if yes, what are those resources? Answering such questions requires to make the conceptual model underlying the different descriptors and security realms *explicit* by an ontology. When confronted with this example, however, one might wonder: What is the difference between the user accounts in the operating system, in the database system and within the application server's realm (where user accounts are called principals)? Are there any ontological differences except their placement in a different

realm? Also, we might be interested in the relationship between a user account in an information system and the corresponding natural person. To infer the total of access rights granted for a natural person who might have several user accounts in and across information systems, might reveal further security holes. Thus, modelling user accounts, roles, and access rights demands a rigorous ontological modelling.

We here present a tricky case that commonly occurs when you must link legacy components. It deals with indirect permissions due to context switches in application servers (cf. Figure 2). Suppose a customer, identified by the user account `dob`, logs into a Web shop via HTTP basic authentication. The script on this page — say, a servlet — might connect to the `CustomerEntityBean`, an Enterprise JavaBean, which in turn accesses the `Customer` table in a legacy database. The legacy database defines its own set of user accounts, which differs from the user accounts in the J2EE (Java 2 Platform Enterprise Edition) realm. We assume that only the `dbuser` (typically the administrator) can access the `Customer` table. So, the EJB must perform an explicit context switch (frequently called the run as paradigm). The call succeeds because `dbuser`'s credentials are propagated. [OSE06]

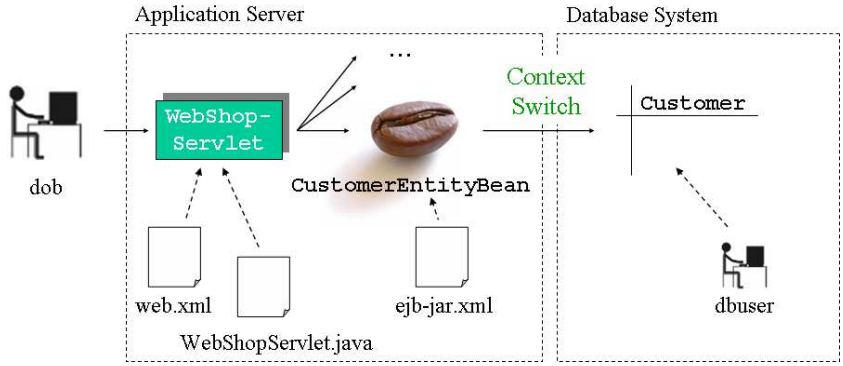

Figure 2: Example of indirect permission. [OSE06]

To overlook the situation outlined above, the developer or administrator would have to analyze two different deployment descriptors, as well as the source code. First, the descriptor of the servlet engine (`web.xml`) states that only authenticated users may access the `WebShopServlet`. Second, the `WebShopServlet` itself accesses the `CustomerEntityBean`. The servlet's `doGet()` method serves the incoming HTTP requests. In our case it queries user account information out of the `Customer` table by means of the bean in order to display it to the user. After retrieving a handle to the bean via the Home interface, the `getCustomerName()` method of the bean is invoked by the servlet. Third, the deployment descriptor of the `CustomerEntityBean`, called `ejb-jar.xml`, states that the bean performs a context switch via the `<run-as-specified-identity>` tag. It thus accesses the database table

with the `dbuser`'s credentials. We list all of the descriptors and source code snippets in the Appendix starting on page 44.

2.2 Policy Handling of Web Services

The paradigm of service-oriented architectures (SOA) factorizes the functionality of an application in loosely coupled software building blocks, viz., services. The Web-based middleware for SOA-based applications is called Web services. Web services subsume a set of protocols and XML-languages for interface description, invocation, discovery, and composition of services. [ACKM03]

Web services are built on top of existing middleware platforms, such as application servers, and introduce additional complexity. Similar to deployment descriptors in application servers, WS* descriptions manage orthogonal aspects in an application-independent way. By WS*, we mean Web service specifications, such as WSDL (Web Service Definition Language), WS-Security, or WS-Policy.[1] WS* descriptions are XML files that declaratively describe how developers should deploy and configure Web services. So, WS* descriptions are exchangeable, and developers might use different implementations for the same Web service description. WS* descriptions' disadvantages, however, are also visible. Although the different standards are complementary, you might produce models composed of different WS* descriptions that are inconsistent but don't easily reveal their inconsistencies. This happens because no coherent formal model of WS* descriptions exists, so it's hard to query the system for conclusions that come from integrating several WS* descriptions.

We here present a use case that shows a conclusion derived from both a WS-BPEL and WS-Policy description,[2] consider the following case. Let's assume we realize a Web shop with internal and external Web services composed and managed by a WS-BPEL engine. After the submission of an order, we have to check the customer's credit card for validity. We assume that credit card providers (VISA, MasterCard, etc.) offer this functionality via Web services. The corresponding WS-BPEL process, `checkAccount`, thus invokes one of the provider's Web services, depending on the customer's credit card.

Suppose now that the Web service of one credit card provider, say MasterCard, only accepts authenticated invocations conforming to Kerberos [NYHR05] or X509 [HFPS99]. It states such policies in a corresponding WS-Policy document. The invocation will fail unless the developer ensures that the policies are met. Without automatic policy matching, the devel-

[1] `http://www.ibm.com/developerworks/views/webservices/standards.jsp`

[2] The Business Process Execution Language (WS-BPEL) is similar to existing workflow languages. It is used to specify how Web services can be composed. WS-Policy allows expressing simple access policies for Web services.

oper has to check the policies manually at development time. We list both descriptors in the Appendix starting on page 45.

Also in this example we are confronted with fundamental ontological questions. What is the difference between a policy of a Web service and an access right on a software component? Are they the same? Can workflows of Web services be modelled such as the invocation chain of software components?

2.3 Modelling requirements

The two use cases above, access rights of software components and policy handling of Web services, give us indications of what concepts a suitable ontology must contain. It is necessary to model information about user accounts, access rights, software components, workflow information, and policies.

In a similar manner, we have introduced further use cases in [OESV04, OLES05] where semantic descriptions of components and services can be exploited to automate — or at least facilitate — some development and management tasks. The use cases in [OESV04] consider reasoning with software components: *libraries and their dependencies*, *conflicting licenses of libraries*, *capability descriptions*, *component classification and discovery*, *semantics of parameters*, *support in error handling*, *reasoning with transactional settings* and *reasoning with security settings*. The use cases in [OLES05] consider Web services: *analyzing message contexts*, *selecting service functionality*, *detecting loops in the interorganisational workflow*, *incompatible inputs and outputs*, *relating communication parameters*, *monitoring of changes*, *aggregating service information* and *quality of service*.

Altogether, the use cases let us derive a set of modelling requirements for deciding which aspects our ontology should formalize. The modelling requirements are: (*i*) *libraries*, *licenses*, *component profiles*, *component taxonomies*, *API descriptions*, *semantic API descriptions*, *access rights* and *workflow information* of software components and (*ii*) *service profiles*, *service taxonomies*, *policies*, *workflow information*, *API descriptions*, as well as *semantic API descriptions* of Web services. We do not claim that the modelling requirements are exhaustive. However, they allow us to constrain our modelling horizon. The resulting ontology will be designed in an extensible way such that further modelling requirements can be met easily.

With the modelling requirements available, it is possible to check whether there exist ontologies that already meet the requirements. In fact, we did analyze two existing ontologies, namely [SOR04] and [MBH+04]. The analysis in [MOGS04, GMSO03] yielded that both existing ontologies could be reused for our purposes but are afflicted with the following shortcomings which are typical for commonly built ontologies:

Conceptual Ambiguity When it is difficult for users to understand the intended meaning of concepts, the associations between these concepts, as well as how they relate to the modelled entities, we speak of conceptual ambiguity of an ontology.

Poor Axiomatization Unlike conceptual ambiguity, poor axiomatization reflects the lesser problem when the definition of concepts is clear, but axiomatization in the ontology itself needs improvement.

Loose Design Loose design of an ontology means that it contains modelling artifacts. Modelling artifacts are concepts and associations which do not bear ontological meaning.

Narrow Scope An ontology is afflicted with narrow scope when it is unclear how a distinction could be made between the objects and events within an information system (regarding data and the manipulation of data) and the real-world objects and events external to such a system. As an example consider the distinction between a user account and its corresponding natural person(s).

We concluded that the problems can be avoided when a carefully engineered foundational ontology is used as a modelling basis. Therefore, section 3 discusses an appropriate foundational ontology.

3 Modelling Basis

In the following Sections 3, 4, 5, and 6, we design an appropriate ontology. To be appropriate the ontology should fulfill the following criteria: (*i*) the ontology should meet the modelling requirements mentioned in the previous section. (*ii*) the ontology should avoid the typical shortcomings introduced in the previous section. (*iii*) the ontology should capture the idiosyncracies of software components and Web services *and* should be easily reusable in different platforms.

When designing such an ontology, it is desirable to start with an extensive and sound modelling basis. Hence, our methodology is geared towards reuse of generic ontologies in order to reduce modelling efforts. Figure 1 already provided an overview of the reused ontologies and the ontologies we contribute. We begin in this section by briefly surveying the reused ontologies DOLCE [OGGM02], Descriptions & Situations, the Ontology of Plans, and the Ontology of Information Objects (the latter are described in [GBCL04]).

3.1 DOLCE

Using a foundational ontology as a modelling basis means relating core concepts and associations to some proposed invariant categories (which are

reflected in the foundational ontology itself). This prompts the ontology engineer to sharpen his notions with respect to the distinctions made in the foundational ontology. What is typically gained is an increased understanding of one's own ontology.

We choose the DOLCE foundational ontology[3] for three reasons. First, DOLCE provides theories for modelling contexts, plans, and information objects. All of them are required for our ontologies and are explained below. Second, DOLCE commits to ontological choices (perdurantism, possibilism, being multiplicative, being descriptive) which are suitable for our domain. Third, DOLCE comes both in a reference and in an application version, axiomatized in quantified modal logic and implemented description logics (OWL DL), respectively. That allows us to formalize our own ontology with a maximum of expressiveness and to use it for run time reasoning later on.

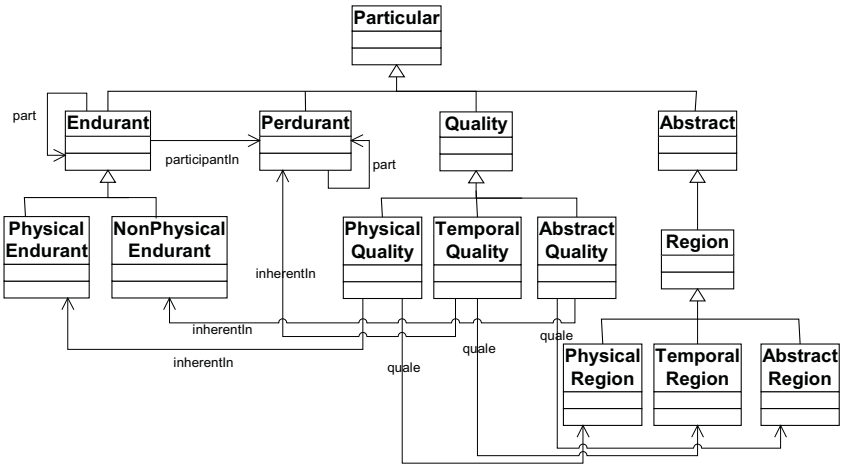

Figure 3: Sketch of DOLCE as UML class diagram. [GBCL04]

DOLCE (Descriptive Ontology for Linguistic and Cognitive Engineering) classifies entities into four categories. These are, as shown in Figure 3, Endurants, Perdurants, Qualities and Abstracts [MBG+02]. The main relation between Endurants (i.e., objects or substances) and Perdurants (i.e., events or processes) is that of participation: an Endurant "lives" in time by participating in a Perdurant. DOLCE introduces Qualities as another category that includes the "individualized" properties of objects or events which we can perceive, measure, or conventionally assert (e.g., color, density, legal validity). Finally, Abstracts do not have spatial or temporal qualities, nor are they qualities themselves. In particular, Regions are used to encode the representation of Qualities as conventionalized in some metric or conceptual space (e.g., a color space, a musical pitch space, a set of legal values). The

[3]DOLCE-related documents can be found at `http://dolce.semanticweb.org`.

corresponding association is called quale in DOLCE. Every category features a whole taxonomy of specializations.

3.2 Descriptions & Situations (DnS)

The domain we want to model, namely that of software components and Web services, requires an ontological formalization of context. The most prominent examples for the need of context modelling are the different views that might exist on data. Data can play the role of both input and output, depending on the context considered.

DOLCE can be augmented by an ontological theory of contexts called Descriptions & Situations (DnS). DnS can be considered an *ontology design pattern* for structuring core and domain ontologies that require contextualization. The following paragraphs provide a brief introduction. For a more detailed description please cf. [GBCL04, GM03a].

When Descriptions & Situations is used with DOLCE, the entities from the DOLCE domain of quantification are called *ground* entities, while the newly introduced entities from the domain of quantification of Descriptions & Situations are called *descriptive* entities. We also visualize this distinction in Figure 4. Parameters, Roles, and Courses are the descriptive entities which are special kinds of ConceptDescriptions (a DOLCE:NonAgentiveSocialObject).[4] The descriptive entities "describe" the ground entities in the following way:[5] Parameters are valuedBy DOLCE:Regions, Roles are playedBy DOLCE:Endurants and Courses sequence DOLCE:Perdurants. The descriptive entities are aggregated by a SituationDescription via the defines association. A SituationDescription ontologically represents a context type, while a Situation can be understood as representing a context occurrence.[6]

Furthermore, the ontology can be used to talk about the association between a set of (assertional) axioms holding for the ground entities, and a set of descriptive axioms. This association is called satisfies. The first set is reified as a Situation, which groups ground entities via the setting association. The second set of axioms is reified as a SituationDescription. A Situation s satisfies a SituationDescription d if the axioms for the ground entities grouped by s are in accordance with the axioms required by d.

The Descriptions & Situations ontology only defines the most generic

[4]Throughout the paper, concepts and associations are written in sans serif and are labelled in a namespace-like manner. Namespace-prefixes indicate the ontology where concepts and associations are defined. If no namespace is given, concepts and associations are assumed to be defined in the ontology currently discussed.

[5]The reader may note, that we occasionally use concept and association names (written in sans serif and preceded by a namespace to clarify their origin) as subjects, objects, and predicates of the sentences in the text.

[6]In the OWL implementation and throughout the related literature, ConceptDescription is also known as Concept, and SituationDescription is also known as Description. We use alternative names here for consistency with past work.

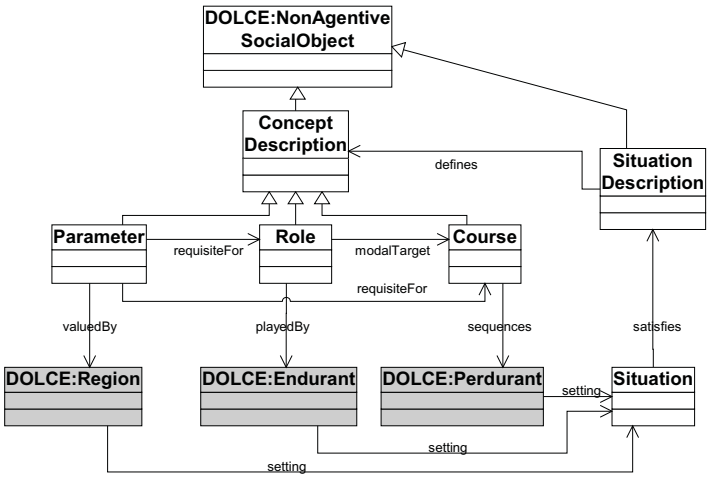

Figure 4: The Descriptions & Situations (DnS) ontology as UML class diagram. Grey classes represent the *ground* entities of DOLCE. *Descriptive Entities* are Parameters, Roles and Courses. [GBCL04]

satisfies association implying that at least some components of a Situation-Description must describe entities in the Situation. This constraint is minimal and for specialized SituationDescriptions additional constraints should be given in order to reason with the satisfaction of candidate Situations. One example is the ontology discussed in the next section: the Ontology of Plans.

Figure 4 also features the modalTarget and requisiteFor associations. The modalTarget association holds between Roles and Courses, and can be used to represent (reified) meta-level constraints on DOLCE's participantIn association (cf. Figure 3) between Endurants and Perdurants. For example, rights, duties, obligations, liabilities, etc., are cases of the more specific attitudeTowards association.

The requisiteFor association holds between Parameters and either Roles or Courses, and can be used to represent (reified) meta-level constraints on the DnS:locatedIn association between DOLCE:Regions and either DOLCE:Endurants or DOLCE:Perdurants.[7]

In the remainder of this article, we also use other notions from the overall DnS ontology in conjunction with DOLCE, notably the Collection concept, the member association, and the unifies association. Collections are DOLCE:SocialObjects that are used to represent (reified) sets of entities sharing some common property, e.g., the set of components in a system, the set of transactions in a time-span, or even a set of tasks to be performed in a same scheduled activity. Collections have a member association to the

[7]The DnS:locatedIn(x, y) association is a composition of DOLCE:inherentIn(z, x), and DOLCE:quale(z, y).

members of a reified set. Notice that the members can change in cardinality and identity, without affecting the identity of their collection, unless explicitly required. A Collection is identified by means of characteristic Roles or Courses, and ultimately there is at least one SituationDescription that unifies it by defining those Roles or Courses.

3.3 Ontology of Plans (OoP)

One of the explicit requirements derived from the *reasoning with transactional settings*, *analyzing message contexts* and *detecting loops in interorganizational workflows* use cases is the possibility to model workflow information between software components or between Web services. The Ontology of Plans (OoP), formalizes a theory of plans in a generic way. It can be reused to model workflow information as well.

The Ontology of Plans applies the ontology design pattern of Descriptions & Situations to characterize planning concepts. The intended use of the ontology is to specify plans at an abstract level independent from existing calculi. It is expected that the concepts of the ontology are implemented as a framework to define detailed or approximate plans for any use (social, personal, computational) by appropriate tools. The resulting plans would then be grounded in some system that implements a set of functionalities and reasons according to the specifications given here. For a detailed description the reader is referred to [GBCL04].

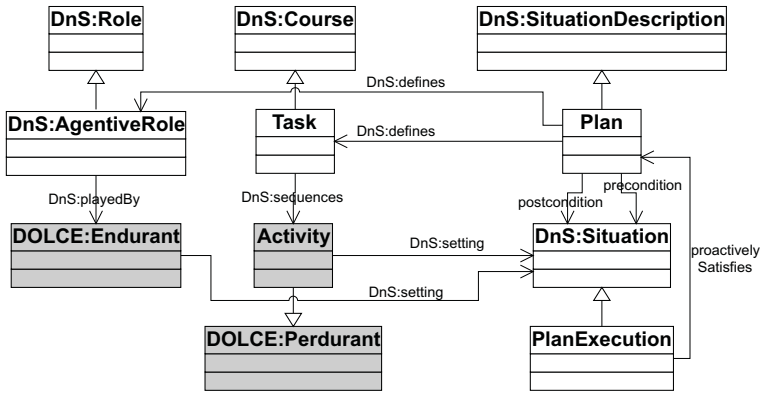

Figure 5: The Ontology of Plans as UML class diagram. Grey classes represent *ground* entities. Concepts from Descriptions & Situations are labelled namespace-like with DnS. [GBCL04]

Plans are special kinds of DnS:SituationDescriptions, which DnS:define Tasks (a special kind of DnS:Course). A typical hierarchy of Tasks (case, branching, synchronization, concurrency, cycling, etc.) is characterized with the help of succession relations. Furthermore, Tasks DnS:sequence Activities — a specialization of DOLCE:Perdurant. Activities are complex actions that

are at least partly conventionally planned.

Specializations of the satisfies association of Descriptions & Situations are applied to express preconditions, postconditions, and several types of satisfaction between a Plan and specific DnS:Situations, e.g., proactivelySatisfies.

As an example we might consider the `CustomerEntityBean` which modifies the `Customer` table (cf. our motivating example in 2.1). In order to formalize this setting, we introduce the *CustomerEntityBeanPlan* which DnS:defines the *ModifyTable* task. An actual execution of this task is represented via the *23:58:00* instance to reflect its timestamp, i.e., DnS:sequences (*ModifyTable*, *23:58:00*). We keep this as a running example, refine and extend it as we move along.

(Ex1) OoP:Plan(*CustomerEntityBeanPlan*)
(Ex2) DnS:defines(*CustomerEntityBeanPlan*, *ModifyTable*)
(Ex3) OoP:Task(*ModifyTable*)
(Ex4) DnS:sequences(*ModifyTable*, *23:58:00*)
(Ex5) OoP:Activity(*23:58:00*)
(Ex6) OoP:PlanExecution(*ModifyTableExecution*)
(Ex7) DnS:setting(*23:58:00*, *ModifyTableExecution*)

3.4 Ontology of Information Objects (OIO)

In our motivating examples we have encountered fundamental ontological questions, e.g., how to model the relationship between a user in an information system and its corresponding natural person. Hence, another requirement for our ontology is a concise distinction between entities in an information system and the real world.

The DOLCE library provides another ontology that allows us to formalize such relationships: the Ontology of Information Objects (OIO). Information objects are the core notion of a *semiotic ontology design pattern* which we briefly discuss here. For a more detailed discussion please cf. [GBCL04].

A content (information) transferred in any modality is assumed to be equivalent to a kind of social object called InformationObject. InformationObjects are spatio-temporal entities of abstract information as described in Shannon's communication theory, hence they are assumed to be in time and realized by some entity.

Figure 6, which depicts the concepts and associations of the ontology, is best explained by a concrete example. The encoding of the `CustomerEntityBean` in Java could be considered an InformationObject. In this case, the InformationObject would be orderedBy the Java language (the InformationEncodingSystem) and realizedBy a specific appearance of the algorithm in main memory (e.g., the contents between memory addresses *0x2112-0x5150*).

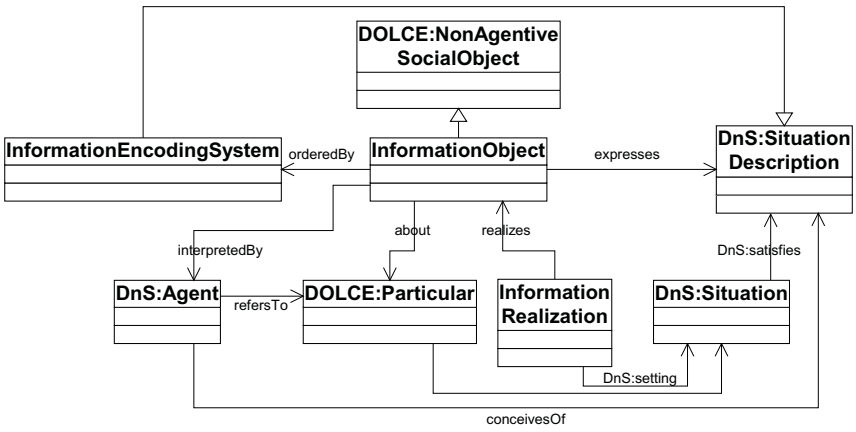

Figure 6: The Ontology of Information Objects as UML class diagram. Concepts defined in DOLCE and Descriptions & Situations (DnS) are labelled with corresponding namespaces. [GBCL04]

The `CustomerEntityBean` expresses a specific OoP:Plan of computational tasks (such as *ModifyTable*) and is interpretedBy a CPU.[8]

(Ex8) OIO:InformationObject(*CustomerEntityBean*)
(Ex9) OIO:orderedBy(*CustomerEntityBean*, *Java*)
(Ex10) OIO:InformationEncodingSystem(*Java*)
(Ex11) OIO:realizedBy(*CustomerEntityBean*, *0x2112-0x5150*)
(Ex12) OIO:InformationRealization(*0x2112-0x5150*)
(Ex13) OIO:expresses(*CustomerEntityBean*, *CustomerEntityBeanPlan*)
(Ex14) OIO:interpretedBy(*CustomerEntityBean,CPU*)
(Ex15) DOLCE:PhysicalObject(*CPU*)

4 Core Software Ontology (CSO)

In order to model the required aspects of components and services, it is necessary to identify fundamental concepts, such as software or data, and to formalize them by reusing our modelling basis. In this section, we design a *Core Software Ontology*, which formalizes such fundamental concepts. The Core Software Ontology is later reused to formalize the required aspects of components and services. Thus, the Core Software Ontology acts as a common basis for the Core Ontologies of Software Components and Web Services which are built in the subsequent sections.

[8] We assume without further mention that for any association there exists an inverse. The naming of associations and their inverses follows an intuitive scheme, e.g., the inverse of realizedBy is called realizes.

Having a common basis is beneficial because it requires modelling the fundamental concepts only once. In essence, the Core Software Ontology meets all modelling requirements which are common to software components and Web services (as derived by the use cases). These are: *API descriptions*, *semantic API descriptions*, *workflow information*, *access rights* and *policies*. The modelling requirements constrain our modelling horizon and give us indications which concepts and associations we have to model. When formalizing concepts and associations, we usually specialize the ontology design patterns provided by the DOLCE library. If such design patterns are not applicable the modelling is left to our discretion. Note that we consider our contributed ontologies as being formalized in DOLCE's representation formalism, viz., modal logic S5. Although we do not explicitly use modal quantifiers, their usage is rooted in DOLCE's concepts and associations, cf. [MBG+03], which we reuse for our modelling.

4.1 Software vs. Data

As mentioned above, the Core Software Ontology formalizes the most fundamental concepts required to model both components and services. We start in this section with a detailed discussion of software and data. In order to clarify both concepts, which are heavily inflicted by polysemy, it is necessary to identify and formalize the entities of the computational domain. The computational domain has a reality of its own, consisting of data manipulated by programs that implement algorithms. The programs that manipulate the data are usually referred to as software. Upon close inspection, it seems that the term software is overloaded and refers to at least three different concepts [GMSO03]:

1. The encoding of an algorithm specification in some kind of representation (i.e., OIO:InformationEncodingSystem). Encoding can be either in mind, on paper, or any other form. The `CustomerEntityBean` can be represented as Java or pseudo code, for instance. This is SoftwareAsCode (which we abbreviate to Software) and is a kind of OIO:InformationObject.

2. The realization of the code in a concrete hardware. These realizations are the DOLCE:PhysicalEndurants that are stored on hard disc or residing in memory. Henceforth, we call them ComputationalObjects (a special kind of OIO:InformationRealization). This could be the appearance of the `CustomerEntityBean` in main memory that can be interpreted and executed by the CPU. Hence, the difference between 1 and 2 is that 2 is physically present in some hardware.

3. The running system, which is the result of an execution of a ComputationalObject. This is the form of software which manifests itself in a

sequence of activities in the computational domain, e.g., the increment of a variable, the comparison of data, the storage of data on the hard disc, etc. This form of software is a DOLCE:Perdurant which we call ComputationalActivity.

ComputationalObjects (item 2) are a specialization of OIO:Information-Realization (any entity that realizes an OIO:InformationObject) as introduced in the Ontology of Information Objects. ComputationalActivities (item 3) are a specialization of OoP:Activity as introduced in the Ontology of Plans. ComputationalObjects and ComputationalActivities are the entities that live in the computational domain.

ComputationalObjects are characterized by the fact that they are necessarily dependent on Hardware which is a DOLCE:PhysicalObject. A suitable dependence association is axiomatized in DOLCE and is called specifically-ConstantlyDependsOn.[9] A ComputationalObject is considered here as a spatio-temporally bounded entity, therefore it exists for the time a memory cell is realizing a certain Software, for instance. Copies of ComputationalObjects in the same or another Hardware are different, although related by some kind of "copy" association. For example, in the case of mobile agents, where people refer to a mobile agent as a piece of software that can move from machine to machine executing the "same" process, it is useful to make agents distinct because the "same" agent can perform differently from machine to machine. The similarity has to be caught via a specialized association, such as copy, which we do not define here, rather than via logical identity.

The execution of a ComputationalObject leads to ComputationalActivities. ComputationalActivities require at least one ComputationalObject as a participant. The definitions below formalize the described properties.[10]

(D1) $\mathsf{ComputationalObject}(x) =_{def} \mathsf{OIO{:}InformationRealization}(x) \wedge$
$\forall y(\mathsf{DOLCE{:}participantIn}(x,y) \rightarrow \mathsf{ComputationalActivity}(y)) \wedge$
$\exists d(\mathsf{DOLCE{:}specificallyConstantlyDependsOn}(x,d) \wedge \mathsf{Hardware}(d))$

(D2) $\mathsf{ComputationalActivity}(x) =_{def} \mathsf{OoP{:}Activity}(x) \wedge$
$\forall y(\mathsf{DOLCE{:}participantIn}(y,x) \rightarrow \mathsf{ComputationalObject}(y)) \wedge$
$\exists c(\mathsf{DOLCE{:}participantIn}(c,x) \wedge \mathsf{ComputationalObject}(c))$

(D3) $\mathsf{DOLCE{:}specificallyConstantlyDependsOn}(x,y) =_{def}$
$\Box(\exists t(\mathsf{DOLCE{:}presentAt}(x,t)) \wedge \forall t(\mathsf{DOLCE{:}presentAt}(x,t) \rightarrow$
$\mathsf{DOLCE{:}presentAt}(y,t)))$

[9] An entity that specificallyConstantlyDependsOn another entity is similar to weak entities in Entity Relationship Models. An entity x specificallyConstantlyDependsOn another entity y iff, at any time t, x cannot be present at t unless y is also present at t. DOLCE formalizes this association by using the $\mathsf{DOLCE{:}presentAt}(x,t)$ association that stands for "x is present (exists) during the time interval or instant t." Note that $ql_T(t',x)$ is the temporal location of x in t'. [MBG+03]

[10] We consider unbound variables in definitions, axioms, and theorems as universally quantified.

(D4) $\mathsf{DOLCE{:}presentAt}(x,t) =_{def} \exists t'(\mathsf{DOLCE{:}ql}_T(t',x) \wedge \mathsf{DOLCE{:}part}(t,t'))$

As an example, consider the ComputationalObject residing in memory between addresses *0x2112* and *0x5150* whose (partial) execution leads to the ComputationalActivity carried out at and identified by the timestamp *23:58:00*. The ComputationalObject could be a concrete appearance of the `CustomerEntityBean` (cf. the motivating example in 2.1) and the ComputationalActivity could be the execution of one of its methods.

(Ex16) ComputationalObject(*0x2112-0x5150*)
(Ex17) ComputationalActivity(*23:58:00*)
(Ex18) DOLCE:participantIn(*0x2112-0x5150*, *23:58:00*)

Regarding item 1, we characterize Software as an OIO:InformationObject. Accordingly, we specialize the design pattern represented by the Ontology of Information Objects (cf. Figure 6 on page 12). First, we constrain the OIO:realizedBy association to ComputationalObjects. Second, we say that Software OIO:expresses an OoP:Plan (cf. Figure 7 for an overview). The OoP:Plan consists of an arbitrary number of ComputationalTasks that DnS:sequence ComputationalActivities (cf. Definition (D6) below). As explained in the Ontology of Plans (Section 3.3), Tasks are the descriptive counterparts of OoP:Activities which are actually carried out. Definition (D5) below captures this intuition of software.

(D5) $\mathsf{Software}(x) =_{def} \mathsf{OIO{:}InformationObject}(x) \wedge \forall y(\mathsf{OIO{:}realizedBy}(x,y) \rightarrow \mathsf{ComputationalObject}(y)) \wedge \exists p,t(\mathsf{OoP{:}Plan}(p) \wedge \mathsf{OIO{:}expresses}(x,p) \wedge \mathsf{ComputationalTask}(t) \wedge \mathsf{DnS{:}defines}(p,t))$
(D6) $\mathsf{ComputationalTask}(x) =_{def} \mathsf{OoP{:}Task}(x) \wedge \forall y(\mathsf{DnS{:}sequences}(x,y) \rightarrow \mathsf{ComputationalActivity}(y))$

The ComputationalObject introduced in (Ex16) can be regarded as a concrete realization of Software (in our case as the `CustomerEntityBean`). We have learned in our motivating example that the bean modifies the `Customer` table. Hence, its corresponding OoP:Plan DnS:defines a ComputationalTask that represents the modification.[11] The ComputationalActivity introduced in (Ex17) could be one specific execution of this task.

(Ex19) Software(*CustomerEntityBean*)
(Ex20) OIO:realizes(*0x2112-0x5150*, *CustomerEntityBean*)
(Ex21) OIO:expresses(*CustomerEntityBean*, *CustomerEntityBeanPlan*)
(Ex22) OoP:Plan(*CustomerEntityBeanPlan*)
(Ex23) DnS:defines(*CustomerEntityBeanPlan*, *ModifyTable*)
(Ex24) ComputationalTask(*ModifyTable*)
(Ex25) DnS:sequences(*ModifyTable*, *23:58:00*)

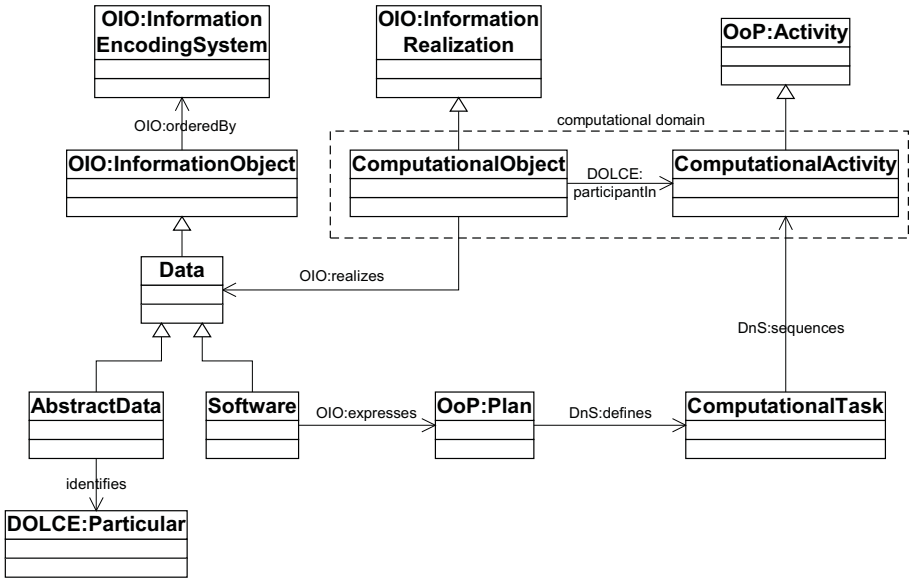

Figure 7: The classification of software and data. Concepts and associations taken from DOLCE, Descriptions & Situations (DnS), the Ontology of Plans (OoP), the Ontology of Information Objects (OIO) are labelled with a namespace.

We consider the data which are manipulated by the programs as ComputationalObjects as well. This reflects the fact that the appearances in the main memory or on the hard disc can be interpreted as instructions for the CPU (i.e., as software) or can be treated as data from the viewpoint of another program. For example, the operating system manipulates application software (loading and unloading it into memory, etc.) much like application software manipulates application data.

Hence, Data can also be considered as a special kind of OIO:InformationObject. The difference to Software is that Data does not OIO:express an OoP:Plan. Furthermore, we introduce AbstractData as a special kind of Data that identifies something different from itself. An example for AbstractData might be a user account in a Unix operating system which has a physical counterpart in the real world. Thus, we say that AbstractData identifies a DOLCE:Particular (a natural person, a company, a physical object) [GBCL04]. The identifies association is a specialization of OIO:about. Definitions (D7), (D8), and (D9) capture these intuitions.

(D7) $\mathsf{Data}(x) =_{def} \mathsf{OIO{:}InformationObject}(x) \wedge \forall y(\mathsf{OIO{:}realizedBy}(x, y) \rightarrow \mathsf{ComputationalObject}(y))$

[11] Note that the detail of modelling ComputationalTasks is a matter of choice. In principle, *ModifyTable* can be considered a complex task and can be broken down to CPU operations.

(D8) $\mathsf{AbstractData}(x) =_{def}$
$\mathsf{Data}(x) \wedge \exists y(\mathsf{DOLCE{:}Particular}(y) \wedge \mathsf{identifies}(x, y))$
(D9) $\mathsf{identifies}(x, y) =_{def}$
$\mathsf{OIO{:}about}(x, y) \wedge \mathsf{AbstractData}(x) \wedge \mathsf{DOLCE{:}Particular}(y) \wedge x \neq y$

As an example, we might introduce another two ComputationalObjects that represent the `dbuser` and the `Customer` table in main memory. The `dbuser` is AbstractData because it identifies a DOLCE:NaturalPerson outside the computational domain, in this case one of the authors.[12]

(Ex26) $\mathsf{ComputationalObject}(\mathit{0x22\text{-}0x23})$
(Ex27) $\mathsf{ComputationalObject}(\mathit{0x316\text{-}0x812})$
(Ex28) $\mathsf{AbstractData}(\mathit{dbuser})$
(Ex29) $\mathsf{AbstractData}(\mathit{CustomerTable})$
(Ex30) $\mathsf{OIO{:}realizedBy}(\mathit{dbuser}, \mathit{0x22\text{-}0x23})$
(Ex31) $\mathsf{OIO{:}realizedBy}(\mathit{CustomerTable}, \mathit{0x316\text{-}0x812})$
(Ex32) $\mathsf{identifies}(\mathit{dbuser}, \mathit{DanielOberle})$
(Ex33) $\mathsf{DOLCE{:}NaturalPerson}(\mathit{DanielOberle})$

The theorem (T1) below is an entailment of our axiomatization. (T1) states that Software must also be considered as Data. As discussed before, this is intuitively clear because an algorithm can be considered as Data from the viewpoint of a compiler, for example. Comparing (D5) and (D7), we find that Software additionally OIO:expresses an OoP:Plan with at least one ComputationalTask. Thus, Software is more specific than Data.

(T1) $\mathsf{Software}(x) \rightarrow \mathsf{Data}(x)$

4.2 API Description

The formalization of fundamental concepts such as Software and Data is a prerequisite for defining API descriptions which is explicitly required by the *support in error handling* and *monitoring of changes* use cases. Assuming the object oriented paradigm (to which we limit ourselves in the remainder of this article), we need to model classes, methods, their inputs, outputs, and datatypes, as well as exceptions. Note that we do not strive to formalize all constructs of object orientation. We limit ourselves to the particular subset that is necessary to formalize simple API descriptions (e.g., we do not formalize specific objects, polymorphism, or inheritance). Below, we give our understanding of those concepts.

(D10) $\mathsf{Class}(x) =_{def} \mathsf{Software}(x) \wedge \forall y(\mathsf{DOLCE{:}properPart}(y, x) \rightarrow$
$(\mathsf{Data}(y) \vee \mathsf{Method}(y)))$

[12] Note that *CustomerTable* is also classified as AbstractData because it identifies a mereological sum of customers.

(D11) $\mathsf{Method}(x) =_{def} \mathsf{Software}(x) \wedge \forall y(\mathsf{DOLCE{:}properPart}(x, y) \rightarrow \mathsf{Class}(y))$
(D12) $\mathsf{Exception}(x) =_{def} \mathsf{Class}(x) \wedge \forall y(\mathsf{methodThrows}(y, x) \rightarrow \mathsf{Method}(y))$
(D13) $\mathsf{DOLCE{:}properPart}(x, y) =_{def} \mathsf{DOLCE{:}part}(x, y) \wedge \neg\mathsf{DOLCE{:}part}(y, x)$

(A1) $\mathsf{methodRequires}(x, y) \rightarrow \mathsf{Method}(x) \wedge \mathsf{Data}(y)$
(A2) $\mathsf{methodYields}(x, y) \rightarrow \mathsf{Method}(x) \wedge \mathsf{Data}(y)$
(A3) $\mathsf{methodThrows}(x, y) \rightarrow \mathsf{methodYields}(x, y) \wedge \mathsf{Exception}(y)$
(A4) $\mathsf{dataType}(x, y) \rightarrow \mathsf{Data}(x) \wedge (\mathsf{Region}(y) \vee \mathsf{Data}(y))$

Definition (D10) considers a Class as a special kind of Software that encapsulates an arbitrary number of Data and an arbitrary number of Methods. Vice versa, a Method is defined as being a part of a Class, having input and output parameters and throwing exceptions.[13] The associations between Methods and their parameters and exceptions are established via methodRequires, methodYields and methodThrows (cf. (D11), (A1), (A2), and (A3)). Exceptions are special kinds of Classes as defined in (D12). dataType relates Data with specific kinds of DOLCE:Regions in the case of simple datatypes, such as strings or integers, or with other Data in the case of complex datatypes, e.g., other classes (cf. Axiom (A4)).

As an example, both the `CustomerEntityBean` and the `WebShopServlet` would be Classes. For the bean, we just specialize the instance introduced in (Ex19) on page 16. The set of instances below also formalizes the servlet's `doGet()` method:

(Ex34) $\mathsf{Class}(\mathit{WebShopServlet})$
(Ex35) $\mathsf{Class}(\mathit{CustomerEntityBean})$
(Ex36) $\mathsf{DOLCE{:}properPart}(\mathit{doGet}, \mathit{WebShopServlet})$
(Ex37) $\mathsf{Method}(\mathit{doGet})$
(Ex38) $\mathsf{methodRequires}(\mathit{doGet}, \mathit{req})$
(Ex39) $\mathsf{methodRequires}(\mathit{doGet}, \mathit{resp})$
(Ex40) $\mathsf{Data}(\mathit{req})$
(Ex41) $\mathsf{Data}(\mathit{resp})$
(Ex42) $\mathsf{dataType}(\mathit{req}, \mathit{HttpServletRequest})$
(Ex43) $\mathsf{dataType}(\mathit{resp}, \mathit{HttpServletResponse})$
(Ex44) $\mathsf{Class}(\mathit{HttpServletRequest})$
(Ex45) $\mathsf{Class}(\mathit{HttpServletResponse})$

[13] The OoP:Plan of the Class contains all Plans of its Methods as alternatives.

4.3 Semantic API Description

Another explicit requirement of the *component classification and discovery*, *semantics of parameters*, *selecting service functionality*, and *incompatible inputs and outputs* use cases is to model semantic API descriptions. The use cases propose to model the meaning of methods and parameters in order to allow for a more powerful search over a large and unfamiliar API, for instance.

Our modelling so far already allows to achieve this goal. As depicted in Figure 8, the meaning or behavior of a Method can be modelled via OIO:expresses and a corresponding OoP:Plan. We already gave an example, namely the *CustomerEntityBeanPlan*, in (Ex22) on page 16. The semantics of parameters, as opposed to their datatypes, can be modelled via OIO:about which can point to any concept in the ontology. Thus, it is possible to model that the `getPrice()` method returns a specific Currency (a specialization of DOLCE:AbstractRegion), for example.

(Ex46) Method(*getPrice*)
(Ex47) methodYields(*getPrice*, *result*)
(Ex48) Data(*result*)
(Ex49) dataType(*result*, *xsd:float*)
(Ex50) OIO:about(*result*, *Euro*)
(Ex51) Currency(*Euro*)

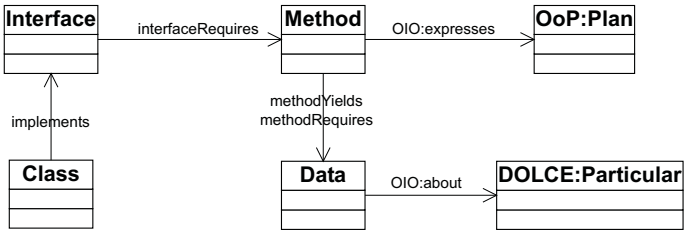

Figure 8: Semantic API description.

We here introduce the notion of an Interface in order to group methods and parameters independently of the Classes they belong to (cf. (D14) and (A5) below). The Interface extends the notion of Java interfaces because it allows to grasp additional information as explained above. In our ontology, the Interface has to be classified as Data as it cannot be executed, i.e., it does not OIO:express an OoP:Plan. Different Classes may implement the same Interface as stated in (A6). In doing so, we are able to model that different Classes provide names for Methods with comparable functionality (e.g., `getPrice()` vs. `getCost()`).

(D14) $\mathsf{Interface}(x) =_{def} \mathsf{Data}(x) \wedge \forall m(\mathsf{inferfaceRequires}(x, m) \rightarrow$
$(\exists p(\mathsf{OIO{:}expresses}(m, p) \wedge \mathsf{OoP{:}Plan}(p)) \wedge \forall d(\mathsf{methodRequires}(m, d) \rightarrow$
$\exists e(\mathsf{DOLCE{:}Particular}(e) \wedge \mathsf{OIO{:}about}(d, e)))))$

(A5) $\mathsf{interfaceRequires}(x, y) \rightarrow$
$\mathsf{DOLCE{:}properPart}(y, x) \wedge \mathsf{Interface}(x) \wedge \mathsf{Method}(y)$

(A6) $\mathsf{implements}(x, y) \rightarrow \mathsf{Class}(x) \wedge \mathsf{Interface}(y) \wedge$
$\forall m_1 \exists m_2(\mathsf{interfaceRequires}(y, m_1) \rightarrow \mathsf{DOLCE{:}properPart}(x, m_2))$

4.4 Workflow Information

The possibility of modelling workflow information, such as information about the `WebShopServlet` invoking the `CustomerEntityBean`, is explicitly required by the use cases *reasoning with transactional settings*, *reasoning with security settings*, *analyzing message contexts* and *detecting loops in interorganizational workflows*.

For modelling workflow information, we use and specialize the ontology design pattern of the Ontology of Plans (cf. Figure 5 on page 11) which in turn builds on Descriptions & Situations. We do so because the design pattern allows abstracting from concrete, i.e., actually executed, workflows. That means, we use ComputationalTasks, which are OoP:Tasks, to represent invocations, the addition of two integers, etc., rather than the actual executions of such tasks (which would be ComputationalActivities). ComputationalTasks are grouped and linked via the OoP:successor and OoP:predecessor associations in an OoP:Plan (a DnS:SituationDescription).[14]

The workflow information we need to model is twofold. First, we have to model invocations between software. Second, we also need to model the inputs and outputs of tasks because the Ontology of Plans does not provide such capabilities.

4.4.1 Invocations Between Software

We start with two associations, viz., executes and accesses, to formalize invocations between Software. Below, (D15) introduces executes as "shortcut" between Software, such as Class or Method, and a ComputationalTask. For example, the `doGet()` method of our `WebShopServlet` executes an invocation task.

(D16) introduces accesses as "shortcut" between the ComputationalTask and the Software or Data that is being called or modified by the task. For example, the invocation task of the `WebShopServlet` accesses the `CustomerEntityBean`. The sequence of executes and accesses can be further abbreviated by invokes which is declared as being transitive (cf. (D17) and (A7)).

[14] The OoP:predecessor and OoP:successor associations hold between OoP:Tasks, and are different from OoP:precondition and OoP:postcondition associations, which hold between OoP:Plans and DnS:SituationDescriptions.

Axioms (A8) and (A9) are introduced for convenience. Regarding (A8), we say that also a Class executes a ComputationalTask when one of its Methods executes this task. Regarding (A9), we state that invokes also holds when we have succeeding tasks.

(D15) $\mathsf{executes}(x, y) =_{def} \mathsf{Software}(x) \wedge \mathsf{ComputationalTask}(y) \wedge$
$\exists co, ca, p(\mathsf{ComputationalObject}(co) \wedge \mathsf{ComputationalActivity}(ca) \wedge$
$\mathsf{OoP{:}Plan}(p) \wedge \mathsf{OIO{:}realizedBy}(x, co) \wedge \mathsf{OIO{:}expresses}(x, p) \wedge$
$\mathsf{DnS{:}defines}(p, y) \wedge \mathsf{DnS{:}sequences}(y, ca) \wedge \mathsf{DOLCE{:}participantIn}(co, ca))$

(D16) $\mathsf{accesses}(x, y) =_{def}$
$\mathsf{ComputationalTask}(x) \wedge \mathsf{Data}(y) \wedge \exists ca, co(\mathsf{DnS{:}sequences}(x, ca) \wedge$
$\mathsf{ComputationalActivity}(ca) \wedge \mathsf{DOLCE{:}participantIn}(co, ca) \wedge$
$\mathsf{ComputationalObject}(co) \wedge \mathsf{OIO{:}realizes}(co, y))$

(D17) $\mathsf{invokes}(x, y) =_{def} \exists z(\mathsf{executes}(x, z) \wedge \mathsf{accesses}(z, y))$

(A7) $\mathsf{invokes}(x, z) \leftarrow \mathsf{invokes}(x, y) \wedge \mathsf{invokes}(y, z)$

(A8) $\mathsf{executes}(x, y) \leftarrow$
$(\mathsf{executes}(z, y) \wedge \mathsf{Method}(z) \wedge \mathsf{DOLCE{:}properPart}(z, x) \wedge \mathsf{Class}(x)$

(A9) $\mathsf{invokes}(x, z) \leftarrow \mathsf{executes}(x, y) \wedge \mathsf{OoP{:}successor}(y, t) \wedge \mathsf{accesses}(t, z)$

In some environments, calls are executed on behalf of a user account whose identity can vary at run time or the authentication can be changed explicitly (called the run-as paradigm). Our running example requires us to express the context switch of the `CustomerEntityBean`, for instance. In order to model this kind of information we introduce the association contextUser as shown below.

(D18) $\mathsf{contextUser}(x, y) =_{def}$
$\mathsf{DnS{:}attitudeTowards}(x, y) \wedge \mathsf{User}(x) \wedge \mathsf{ComputationalTask}(y)$

Revisiting our example, we have a ComputationalTask that models the `WebShopServlet`'s call of the `CustomerEntityBean`. We also have a task that models the modification of the `Customer` table on behalf of the bean. Note that this task is executed with `dbuser`'s credentials. In the examples below, (Ex55) can be inferred from (Ex34), (Ex36), (Ex37), (Ex52), (Ex53), (Ex54), (A7) and (A8).

(Ex52) $\mathsf{ComputationalTask}(\mathit{CallBean})$
(Ex53) $\mathsf{executes}(\mathit{doGet}, \mathit{CallBean})$
(Ex54) $\mathsf{accesses}(\mathit{CallBean}, \mathit{CustomerEntityBean})$
(Ex55) $(Ex34), ..., (A8) \models \mathsf{invokes}(\mathit{WebShopServlet}, \mathit{CustomerEntityBean})$
(Ex56) $\mathsf{ComputationalTask}(\mathit{ModifyTable})$
(Ex57) $\mathsf{executes}(\mathit{CustomerEntityBean}, \mathit{ModifyTable})$
(Ex58) $\mathsf{contextUser}(\mathit{dbuser}, \mathit{ModifyTable})$
(Ex59) $\mathsf{accesses}(\mathit{ModifyTable}, \mathit{CustomerTable})$

4.4.2 Inputs and Outputs

Besides invocations, we also need to model the Inputs and Outputs of tasks. The Ontology of Plans does not provide such capabilities. Inputs and Outputs are required when we want to represent the information of a WS-BPEL workflow, for instance. Inputs and Outputs are DnS:Roles which are both DnS:playedBy Data and DnS:definedBy an OoP:Plan (cf. (D19), (D20) and (A12)). The relationships between Inputs (Outputs) and ComputationalTasks are modelled by inputFor (outputFor) as specified in (A10), and (A11).[15] The difference between Inputs and Outputs is that the former must be present before the latter (cf. (A13)).

(D19) $\mathsf{Input}(x) =_{def} \mathsf{DnS{:}Role}(x) \wedge \forall y(\mathsf{DnS{:}playedBy}(x, y) \rightarrow \mathsf{Data}(y))$
(D20) $\mathsf{Output}(x) =_{def} \mathsf{DnS{:}Role}(x) \wedge \forall y(\mathsf{DnS{:}playedBy}(x, y) \rightarrow \mathsf{Data}(y))$

(A10) $\mathsf{inputFor}(x, y) \rightarrow$
$\mathsf{DnS{:}modalTarget}(x, y) \wedge \mathsf{Input}(x) \wedge \mathsf{ComputationalTask}(y)$
(A11) $\mathsf{outputFor}(x, y) \rightarrow$
$\mathsf{DnS{:}modalTarget}(x, y) \wedge \mathsf{Output}(x) \wedge \mathsf{ComputationalTask}(y)$
(A12) $\mathsf{Input}(x) \vee \mathsf{Output}(x) \rightarrow \exists p(\mathsf{OoP{:}Plan}(p) \wedge \mathsf{DnS{:}defines}(p, x))$
(A13) $\mathsf{ComputationalTask}(ct) \rightarrow \forall d_1, d_2(\forall i, o(\mathsf{inputFor}(i, ct) \wedge$
$\mathsf{DnS{:}playedBy}(i, d_1) \wedge \mathsf{outputFor}(o, ct) \wedge \mathsf{DnS{:}playedBy}(o, d_2)) \rightarrow$
$\exists t_1, t_2(\mathsf{presentAt}(d_1, t_1) \wedge \mathsf{presentAt}(d_2, t_2) \wedge t_1 < t_2))$

As a concrete example, consider the Input for *ModifyTable* which would be the `Customer` table (cf. (Ex60), (Ex61), and (Ex62) below).

(Ex60) $\mathsf{Input}(\mathit{ModifyTableInput})$
(Ex61) $\mathsf{DnS{:}playedBy}(\mathit{ModifyTableInput}, \mathit{CustomerTable})$
(Ex62) $\mathsf{inputFor}(\mathit{ModifyTableInput}, \mathit{ModifyTable})$

4.5 Access Rights and Policies

The requirement to model access rights and policies stems from the *access rights of software components*, *analyzing message contexts*, and *policy handling of Web services* use cases. In general, access rights are required to state that access is granted for a specific user on a specific resource. Policies can be regarded as a generalization of access rights. They define high-level guidelines that constrain the behavior of an information system.

We use and specialize Descriptions & Situations for modelling access rights and policies. The design pattern represented by Descriptions & Situations (cf. Figure 4 on page 10) provides us with the basic primitives of

[15] Both are specializations of DnS:modalTarget, viz., the generic association holding between DnS:Roles and DnS:Courses.

context modelling, such as the notion of roles, which allows us to talk about subjects and objects of a policy on the abstract level, i.e., independent of the entities that play such roles. As we have learned in Section 3.2, Descriptions & Situations therefore distinguishes between *descriptive* and *ground* entities.

In a first step, it is necessary to introduce further *ground* entities which are required later on. (D21) below specifies a User as a special kind of AbstractData which identifies a DnS:Agent. The intuition behind User is a user account in an operating system. Hence, Users identify DnS:Agents which are either DOLCE:AgentivePhysicalObjects or DOLCE:AgentiveSocialObjects. Most frequently, but not always, a natural person is associated with such an account. We aggregate Users to a UserGroup by exploiting DnS:Collection in (D22).

(D21) $\mathsf{User}(x) =_{def} \mathsf{AbstractData}(x) \wedge \forall y(\mathsf{identifies}(x, y) \rightarrow \mathsf{DnS{:}Agent}(y))$

(D22) $\mathsf{UserGroup}(x) =_{def} \mathsf{DnS{:}Collection}(x) \wedge \forall y(\mathsf{DnS{:}member}(x, y) \rightarrow \mathsf{User}(y))$

In a second step, we specialize the *descriptive* entities of Descriptions & Situations, viz., DnS:Roles, DnS:Courses, DnS:Parameters, and DnS:SituationDescriptions as follows. First, we introduce two DnS:Roles to represent the subject and the object of a policy in (D23) and (D24). PolicySubjects are DnS:AgentiveRoles and can be DnS:playedBy Users or UserGroups. PolicyObjects are DnS:NonAgentiveRoles and can be DnS:playedBy Data. Second, we need to represent the predicate of a policy by a special kind of DnS:Course. (D6) on page 15 already introduced ComputationalTask which meets this requirement. We further aggregate such tasks to TaskCollections in (D25). The intuition behind TaskCollections are the security "roles" in operating or database systems. That means a TaskCollection groups ComputationalTasks, such as read, write, or execute. Third, we introduce Constraints as special kinds of DnS:Parameter. The ComputationalTask or TaskCollections can be constrained in some way, e.g., a Web service policy might state that an invocation is only possible with Kerberos or X509 authentication (cf. (D26)). Finally, we construct a PolicyDescription, viz., a special kind of DnS:SituationDescription, from the aforementioned concepts.[16] Axiom (A14) requires each PolicyDescription to have a PolicySubject, ComputationalTask, and a PolicyObject. Figure 9 provides an overview.

(D23) $\mathsf{PolicySubject}(x) =_{def} \mathsf{DnS{:}AgentiveRole}(x) \wedge \forall y(\mathsf{DnS{:}playedBy}(x, y) \rightarrow (\mathsf{User}(y) \vee \mathsf{UserGroup}(y))) \wedge \forall z(\mathsf{DnS{:}attitudeTowards}(x, z) \rightarrow (\mathsf{ComputationalTask}(z) \vee \mathsf{TaskCollection}(z)))$

[16] Note that DnS:unifies is the generic association between DnS:SituationDescriptions and DnS:Collections.

(D24) $\mathsf{PolicyObject}(x) =_{def} \mathsf{DnS{:}NonAgentiveRole}(x) \wedge$
$\forall y(\mathsf{DnS{:}playedBy}(x,y) \rightarrow \mathsf{Data}(y)) \wedge \forall z(\mathsf{DnS{:}attitudeTowards}(x,z) \rightarrow$
$(\mathsf{ComputationalTask}(z) \vee \mathsf{TaskCollection}(z)))$

(D25) $\mathsf{TaskCollection}(x) =_{def} \mathsf{DnS{:}Collection}(x) \wedge \forall y(\mathsf{DnS{:}member}(x,y) \rightarrow$
$\mathsf{ComputationalTask}(y))$

(D26) $\mathsf{Constraint}(x) =_{def} \mathsf{DnS{:}Parameter}(x) \wedge \forall y(\mathsf{DnS{:}requisiteFor}(x,y) \rightarrow$
$(\mathsf{ComputationalTask}(y) \vee \mathsf{TaskCollection}(y))) \wedge \forall z(\mathsf{DnS{:}defines}(z,x) \rightarrow$
$\mathsf{PolicyDescription}(z))$

(D27) $\mathsf{PolicyDescription}(x) =_{def}$
$\mathsf{DnS{:}SituationDescription}(x) \wedge \forall y(\mathsf{DnS{:}unifies}(x,y) \rightarrow$
$\mathsf{TaskCollection}(y)) \wedge \forall z(\mathsf{DnS{:}defines}(x,z) \rightarrow \mathsf{Constraint}(z) \vee$
$\mathsf{ComputationalTask}(z) \vee \mathsf{PolicySubject}(z) \vee \mathsf{PolicyObject}(z))$

(A14) $\mathsf{PolicyDescription}(x) \rightarrow$
$\exists s,t,o(\mathsf{DnS{:}defines}(x,s) \wedge \mathsf{PolicySubject}(s) \wedge \mathsf{DnS{:}defines}(x,t) \wedge$
$\mathsf{ComputationalTask}(t) \wedge \mathsf{DnS{:}defines}(x,o) \wedge \mathsf{PolicyObject}(o))$

It is worthwhile to spend some words on the DnS:attitudeTowards association between DnS:Roles and DnS:Courses. The DnS:attitudeTowards association is a special kind of DnS:modalTarget and can be considered the *descriptive* counterpart of the DOLCE:participantIn association. It is used to state attitudes, attention, or even subjection that an object can have with respect to an action or process. In our case, DnS:attitudeTowards is used to state the relationship between PolicySubjects, as well as PolicyObjects, and the ComputationalTask or TaskCollection. Descriptions & Situations provides us with three initial specializations of DnS:attitudeTowards, viz., DnS:rightTowards, DnS:empoweredTo, and DnS:obligedTo. We further refine DnS:rightTowards in (A15) below.

(A15) $\mathsf{computationalRightTowards}(x,y) \rightarrow \mathsf{DnS{:}rightTowards}(x,y) \wedge$
$\mathsf{PolicySubject}(x) \wedge (\mathsf{ComputationalTask}(y) \vee \mathsf{TaskCollection}(y))$

(A16) $\mathsf{computationalRightTowards}(x,z) \leftarrow \mathsf{computationalRightTowards}(x,y) \wedge$
$\mathsf{TaskCollection}(y) \wedge \mathsf{DnS{:}member}(y,z) \wedge \mathsf{ComputationalTask}(z)$

(A17) $(\mathsf{DnS{:}playedBy}(x,z) \wedge \mathsf{PolicySubject}(x) \wedge \mathsf{UserGroup}(z)) \rightarrow$
$\exists y(\mathsf{DnS{:}member}(z,y) \wedge \mathsf{User}(y) \wedge \mathsf{DnS{:}playedBy}(x,y))$

(A16) and (A17) infer the closure of all resulting rights considering UserGroups and TaskCollections. A PolicySubject is granted rights on all tasks which are members of the TaskCollection. Similarly, a User is granted all access rights which are granted for his UserGroup.

An analysis of the descriptor of our `WebShopServlet` (`web.xml`, cf. Appendix on page 44) lets us derive the following PolicyDescription. The HTTP basic authentication allows *anybody* to perform an HTTP GET on the servlet. We consider *anybody* as a UserGroup that has every User of the system as DnS:member.

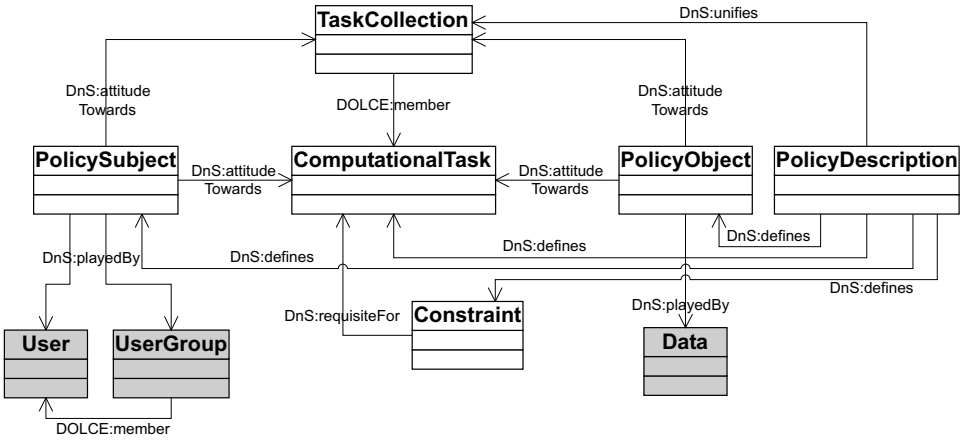

Figure 9: The Policy Description as UML class diagram. Grey classes represent *ground* entities, white classes the *descriptive* entities of Descriptions & Situations or specializations thereof.

(Ex63) PolicyDescription(*WebShopServletPolicy*)
(Ex64) DnS:defines(*WebShopServletPolicy*, *ServletCaller*)
(Ex65) PolicySubject(*ServletCaller*)
(Ex66) UserGroup(*anybody*)
(Ex67) DnS:playedBy(*ServletCaller*, *anybody*)
(Ex68) DnS:defines(*WebShopServletPolicy*, *GET*)
(Ex69) ComputationalTask(*GET*)
(Ex70) computationalRightTowards(*ServletCaller*, *GET*)
(Ex71) DnS:defines(*WebShopServletPolicy*, *ServletCallee*)
(Ex72) PolicyObject(*ServletCallee*)
(Ex73) Class(*WebShopServlet*)
(Ex74) DnS:playedBy(*ServletCallee*, *WebShopServlet*)
(Ex75) DnS:obligedTo(*ServletCallee*, *GET*)

5 Core Ontology of Software Components (COSC)

In the last section we have presented a Core Software Ontology consisting of fundamental concepts and associations such as software, data, users, policies and so on. We separated the fundamental concepts in a core ontology to facilitate reuse.

Although some of the modelling requirements are already met by the Core Software Ontology, there remain further use cases that explicitly require the formalization of software component and Web service idiosyncracies. In this section, we present a possible *Core Ontology of Software*

Components based on the Core Software Ontology that meets the remaining modelling requirements relevant for software components, viz., *libraries and licenses*, *component profiles*, and *component taxonomies*.

We start by formalizing our understanding of the term "software component." It requires special attention as there is a variety of interpretations that leads to ambiguity. We also put libraries and licenses in this core ontology because the *libraries and their dependencies* and *conflicting licenses of libraries* use cases propose to detect inconsistent configurations of components and their required libraries. Finally, we define a component profile that aggregates all relevant aspects of a component. We expect that this aggregation makes browsing and querying for developers more convenient. The component profile is envisioned to act as the central information source for software components rather than having bits and pieces all over the place. We finish by revisiting the motivating example from Section 2.1, and show how it can be formalized.

5.1 Formalization of the Term "Software Component"

Software componentry is a loosely defined term for a software technology proposing that software should be developed by glueing prefabricated components together as in the field of electronics or mechanics. Software componentry also proposes encapsulating software functionality for multiple use in a context-independent way, composable with other components, and as a unit of independent deployment and versioning.[17]

Software components often assume the form of object-oriented classes conforming to a framework specification. However, software components differ from classes. The basic idea in object-oriented programming is that software should be written according to a mental model of the actual or imagined objects it represents. Software componentry, by contrast, makes no such assumptions.

The framework specifications prescribe (*i*) interfaces that must be implemented by components and (*ii*) protocols that define how components interact with each other. Examples of framework specifications are Enterprise JavaBeans (EJB) and the Component Object Model (COM) from Microsoft.

The definitions below formalize this intuition of software component as closely as possible. Assuming the object-oriented paradigm, (D30) below states that a SoftwareComponent is a special kind of CSO:Class that conforms to a FrameworkSpecification. According to the definition above, a FrameworkSpecification is (*i*) a DOLCE:Collection of CSO:Interfaces and (*ii*) a special kind of OoP:Plan which specifies the interaction of components (cf. (D28)). Conformance means that at least one CSO:Interface prescribed by

[17] Wikipedia, `http://en.wikipedia.org/wiki/Software_component`, August 2005.

the FrameworkSpecification has to be implemented by the SoftwareComponent (cf. (D29)).

(D28) $\mathsf{FrameworkSpecification}(x) =_{def}$
$\mathsf{OoP{:}Plan}(x) \wedge \exists y(\mathsf{DOLCE{:}Collection}(y) \wedge \mathsf{DnS{:}unifies}(x, y) \wedge$
$\forall z(\mathsf{DOLCE{:}member}(y, z) \rightarrow \mathsf{CSO{:}Interface}(z)))$

(D29) $\mathsf{conforms}(x, y) =_{def} \mathsf{CSO{:}Class}(x) \wedge \mathsf{FrameworkSpecification}(y) \wedge$
$\exists i, c(\mathsf{CSO{:}Interface}(i) \wedge \mathsf{DOLCE{:}member}(c, i) \wedge \mathsf{DOLCE{:}Collection}(c) \wedge$
$\mathsf{DnS{:}unifies}(y, c) \rightarrow \mathsf{CSO{:}implements}(x, i))$

(D30) $\mathsf{SoftwareComponent}(x) =_{def}$
$\mathsf{CSO{:}Class}(x) \wedge \exists y(\mathsf{conforms}(x, y) \wedge \mathsf{FrameworkSpecification}(y))$

Coming back to our running example, we would define the `Customer-EntityBean` as a SoftwareComponent that conforms to the *EnterpriseJava-Beans* FrameworkSpecification. In essence, the *EnterpriseJavaBeans* specification can be conceived as a set of Jave interfaces (`javax.ejb.*`).

(Ex76) $\mathsf{SoftwareComponent}(\mathit{CustomerEntityBean})$

(Ex77) $\mathsf{FrameworkSpecification}(\mathit{EnterpriseJavaBeans})$

(Ex78) $\mathsf{conforms}(\mathit{CustomerEntityBean}, \mathit{EnterpriseJavaBeans})$

5.2 Libraries and Licenses

The *libraries and their dependencies* and *conflicting licenses of libraries* use cases require the modelling of libraries and licenses. Both use cases discuss the problem of conflicting libraries and incompatible licenses in the current configuration of an integrated software development environment (IDE). In the case of libraries, a `lib1.jar` might conflict with a `lib2.jar` in a specific version. For example, such information can be obtained from expert knowledge or from public sources, such as the RPM package manager.[18] However, the check for conflicts still remains a manual task. In the case of licenses, we find similar problems. Typically, software libraries are released under specific licenses such as GPL, LGPL, Apache, BSD, Public Domain, XFree86, or commercially closed source licenses.[19] The proliferation of different software licenses means increased work for software developers. They have to check whether used libraries have conflicting licenses.

Therefore, the use cases propose an automatic check for conflicting libraries and incompatible licenses in an integrated software development environment (IDE) at development time. In order to realize either use case, we introduce the concepts of SoftwareLibrary and License in (D31) and (D32) below. A SoftwareLibrary consists of a number of CSO:Classes and is classified as CSO:Data because it cannot be executed as a whole. The concept

[18] `http://www.rpm.org`

[19] `http://www.gnu.org/philosophy/license-list.html`

License is a special kind of LegalContract as introduced in the Core Legal Ontology [GST04].

(D31) $\mathsf{SoftwareLibrary}(x) =_{def} \mathsf{CSO{:}Data}(x) \wedge \forall c(\mathsf{DOLCE{:}properPart}(x, c) \rightarrow \mathsf{CSO{:}Class}(c))$

(D32) $\mathsf{License}(x) =_{def} \mathsf{LegalContract}(x) \wedge \exists y(\mathsf{CSO{:}Software}(y) \wedge \mathsf{DnS{:}involves}(x, y))$

Very often there are functional dependencies between libraries that are revealed only during run time (e.g., by `ClassNotFoundExceptions` in Java). For example, a library `lib1.jar` might depend on `lib2.jar` which in turn depends on `lib3.jar` and so forth. It is a very tedious task to keep track of such dependencies and, additionally, to check whether there are conflicts between libraries in this dependency graph. In order to reason with such information, we introduce the following associations and axioms: First, the transitive libraryDependsOn in (A18) and (A19) below. Second, the symmetric libraryConflictsWith in (A20) and (A21). Finally, (A22) formalizes indirect conflicts.

The existence of incompatible licenses further complicates the situation. Even though libraries in the dependency graph do not conflict, they might have incompatible licenses. In order to reason with such information, we further introduce the association releasedUnder between SoftwareLibraries and Licenses in (A23), as well as the symmetric licenseIncompatibleWith in (A24) and (A25).

(A18) $\mathsf{libraryDependsOn}(x, y) \rightarrow \mathsf{DOLCE{:}specificallyConstantlyDependsOn}(x, y) \wedge \mathsf{SoftwareLibrary}(x) \wedge \mathsf{SoftwareLibrary}(y)$

(A19) $\mathsf{libraryDependsOn}(x, z) \leftarrow \mathsf{libraryDependsOn}(x, y) \wedge \mathsf{libraryDependsOn}(y, z)$

(A20) $\mathsf{libraryConflictsWith}(x, y) \rightarrow \mathsf{SoftwareLibrary}(x) \wedge \mathsf{SoftwareLibrary}(y)$

(A21) $\mathsf{libraryConflictsWith}(x, y) \leftrightarrow \mathsf{libraryConflictsWith}(y, x)$

(A22) $\mathsf{libraryConflictsWith}(x, z) \leftarrow \mathsf{libraryDependsOn}(x, y) \wedge \mathsf{libraryConflictsWith}(y, z)$

(A23) $\mathsf{releasedUnder}(x, y) \rightarrow \mathsf{OIO{:}expresses}(x, y) \wedge \mathsf{SoftwareLibrary}(x) \wedge \mathsf{License}(y)$

(A24) $\mathsf{licenseIncompatibleWith}(x, y) \rightarrow \mathsf{License}(x) \wedge \mathsf{License}(y)$

(A25) $\mathsf{licenseIncompatibleWith}(x, y) \leftrightarrow \mathsf{licenseIncompatibleWith}(y, x)$

As an example, let us assume the `CustomerEntityBean` requires `lib1.jar`. Adding `lib1.jar` to the classpath in turn requires `lib2.jar` and `lib4.jar`. Adding `lib2.jar` to the classpath additionally requires `lib3.jar`. Furthermore, let us assume that `lib4.jar` conflicts with `lib3.jar`. Despite

the small number of libraries, the situation becomes quite complex. Compiling and running the application will yield a run time exception. Given the modelling below we can infer libraryConflictsWith(*lib1.jar*, *lib4.jar*) because of (A19), (A21), and (A22).

(Ex79) SoftwareLibrary(*lib1.jar*)
(Ex80) SoftwareLibrary(*lib2.jar*)
(Ex81) SoftwareLibrary(*lib3.jar*)
(Ex82) SoftwareLibrary(*lib4.jar*)
(Ex83) libraryDependsOn(*lib1.jar*, *lib2.jar*)
(Ex84) libraryDependsOn(*lib1.jar*, *lib4.jar*)
(Ex85) libraryDependsOn(*lib2.jar*, *lib3.jar*)
(Ex86) libraryConflictsWith(*lib4.jar*, *lib3.jar*)
(Ex87) (Ex79), ..., (A22) $\models$ libraryConflictsWith(*lib1.jar*, *lib4.jar*)

5.3 Component Profiles and Taxonomies

So far, we have formalized several different aspects relevant for a software component such as interface and policy descriptions or plans. In this section we further aggregate the knowledge in component profiles. We expect that such an aggregation makes browsing and querying for developers more convenient. The component profile is envisioned to act as the central information source for a specific software component rather than having bits and pieces all over the place. Furthermore, the component profiles can be specialized and aligned in a taxonomy as required by the use cases *capability descriptions*, *component classification and discovery*, *reasoning with transactional settings*, and *reasoning with security settings*.

(D33) and (A26) define a Profile as follows: First, it aggregates CSO:PolicyDescriptions, an OoP:Plan, the required SoftwareLibraries, the implemented Interfaces and additional Characteristics of a specific Software entity. Second, the link to the described Software is specified via the describes association. (D34) specializes this definition to ComponentProfile.

Often, we need to express certain capabilities or features of components, such as the version, transactional or security settings. For this purpose, we introduce Characteristics on a Profile in (D35). It is expected that ComponentProfiles are specialized and put into a taxonomy. For example, we might define a DatabaseConnectorProfile as a ComponentProfile that provides for specific Characteristics describing whether the underlying database supports transactions or SQL-99. A taxonomic structure further accommodates the developer in browsing and querying for ComponentProfiles in his system.

Finally, (A27) specifies the profiles association as a "catch-all" for DnS:defines, DnS:unifies, OIO:about, as well as OIO:expressedBy. This is done for convenience in order to relieve the developer, who will certainly have to deal with such information, from such modelling details.

(D33) $\mathsf{Profile}(x) =_{def} \mathsf{OIO{:}InformationObject}(x) \wedge \forall y(\mathsf{profiles}(x,y) \rightarrow (\mathsf{CSO{:}PolicyDescription}(y) \vee \mathsf{SoftwareLibrary}(y) \vee \mathsf{CSO{:}Interface}(y) \vee \mathsf{OoP{:}Plan}(y) \vee \mathsf{Characteristic}(y))) \wedge \forall z(\mathsf{describes}(x,z) \rightarrow \mathsf{Software}(z))$

(D34) $\mathsf{ComponentProfile}(x) =_{def} \mathsf{Profile}(x) \wedge \forall y(\mathsf{describes}(x,y) \rightarrow \mathsf{SoftwareComponent}(y))$

(D35) $\mathsf{Characteristic}(x) =_{def} \mathsf{DnS{:}Parameter}(x) \wedge \forall y(\mathsf{DnS{:}defines}(y,x) \rightarrow \mathsf{Profile}(y)) \wedge \forall z(\mathsf{DnS{:}valuedBy}(x,z) \wedge \mathsf{DOLCE{:}AbstractRegion}(z))$

(A26) $\mathsf{describes}(x,y) \rightarrow \mathsf{OIO{:}about}(x,y) \wedge \mathsf{Profile}(x) \wedge \mathsf{CSO{:}Software}(y)$

(A27) $\mathsf{profiles}(x,y) \rightarrow \mathsf{DnS{:}defines}(x,y) \vee \mathsf{DnS{:}unifies}(x,y) \vee \mathsf{OIO{:}about}(x,y) \vee \mathsf{OIO{:}expressedBy}(x,y)$

The information grouped by a ComponentProfile might have different origins. For example, a specific PolicyDescription might be automatically obtained from `ejb-jar.xml`, while manual modelling or source code analysis would result in an OoP:Plan. Hence, it is important to model also informationTimestamp and informationSource for parts of the ComponentProfile. We omit their definition because both are simple attributes with `xsd:string`.

As an example, we construct a profile for our `CustomerEntityBean` below. We assume the bean requires `lib1.jar`, implements the *javax.ejb.EntityBean* interface and has a policy description.

(Ex88) $\mathsf{ComponentProfile}(\mathit{CustomerBeanProfile})$

(Ex89) $\mathsf{describes}(\mathit{CustomerBeanProfile}, \mathit{CustomerEntityBean})$

(Ex90) $\mathsf{profiles}(\mathit{CustomerBeanProfile}, \mathit{lib1.jar})$

(Ex91) $\mathsf{informationTimestamp}(\mathit{lib1.jar}, \mathit{050805\text{-}9{:}45{:}21})$

(Ex92) $\mathsf{profiles}(\mathit{CustomerBeanProfile}, \mathit{javax.ejb.EntityBean})$

(Ex93) $\mathsf{CSO{:}Interface}(\mathit{javax.ejb.EntityBean})$

(Ex94) $\mathsf{profiles}(\mathit{CustomerBeanProfile}, \mathit{CustomerEntityBeanPolicy})$

(Ex95) $\mathsf{CSO{:}PolicyDescription}(\mathit{CustomerEntityBeanPolicy})$

(Ex96) $\mathsf{informationSource}(\mathit{CustomerEntityBeanPolicy}, \mathit{file://ejb\text{-}jar.xml})$

5.4 Example

In this section, we revisit our running example from Section 2.1 and show how it can be formalized with our ontology. We already introduced some of the instances in a piecemeal manner throughout the paper. We collect the relevant instances to construct PolicyDescriptions and Plans so that a simple query can be used to detect if there are indirect permissions. An overview is given in Figure 12 on page 46 in the Appendix.

The descriptor files of the `WebShopServlet` (`web.xml`) and the `CustomerEntityBean` (`ejb-jar.xml`) result in two CSO:PolicyDescriptions. The third CSO:PolicyDescription below can be extracted from database metadata.

We can now define additional axioms to deduce all indirectly accessible resources for a user. First, Axiom (A28) infers the directly accessible resources r of a user u. The reader may note that axioms (A16) and (A17) on page 24 also infer the accessible resources which are a result of group memberships. Second, Axiom (A29) infers indirectly accessible resources, i.e., ones that are a result of a call with a context switch. With (A28) and (A29) we can infer $\mathsf{indirectlyAccessibleResource}(\mathit{CustomerTable}, \mathit{anybody})$ — a result which otherwise would require tedious manual efforts.

(A28) $\mathsf{directlyAccessibleResource}(r, u) \leftarrow \mathsf{CSO{:}Data}(r) \wedge \mathsf{CSO{:}User}(u) \wedge \mathsf{DnS{:}playedBy}(s, u, t) \wedge \mathsf{CSO{:}PolicySubject}(s) \wedge \mathsf{CSO{:}computationalRightTowards}(s, ct) \wedge \mathsf{CSO{:}ComputationalTask}(ct) \wedge \mathsf{DnS{:}obligedTo}(o, ct) \wedge \mathsf{CSO{:}PolicyObject}(o) \wedge \mathsf{DnS{:}playedBy}(u, r, t)$

(A29) $\mathsf{indirectlyAccessibleResource}(r_3, u) \leftarrow \mathsf{directlyAccessibleResource}(r_1, u) \wedge \mathsf{CSO{:}invokes}(r_1, r_2) \wedge \mathsf{CSO{:}Software}(r_2) \wedge \mathsf{CSO{:}executes}(r_2, t) \wedge \mathsf{CSO{:}ComputationalTask}(t) \wedge \mathsf{CSO{:}accesses}(t, r_3) \wedge \mathsf{CSO{:}contextUser}(u, t)$

6 Core Ontology of Web Services (COWS)

In this section we present a possible *Core Ontology of Web Services* to meet the remaining modelling requirements of *service profiles* and *service taxonomies*. The Core Ontology of Web Services is based on the Core Ontology of Software Components presented in Section 5. We start by formalizing our understanding of the term "Web service," introduce the notion of service profiles, revisit the motivating example (cf. Section 2.2), and show how it can be formalized.

6.1 Formalization of the term "Web service"

On the one hand, Web services often reveal functionality residing in a class or component. Application servers typically provide support to automatically access the functionality via the standardized SOAP protocol and the automatic generation of standardized WSDL interface descriptions. However, the same can be done with the Java Remote Method Invocation (RMI) or CORBA although with different protocols and interface descriptions. On the other hand, a Web service can be defined as a composition of other Web services, e.g., by the Business Process Execution Language (WS-BPEL).[20] Again, this can be done with software components in common workflow engines as well.

So what is the difference between a software component and a Web service? We argue that standardization in terms of Web protocols and descriptions seems to be the major distinction. In any case, Web services are

[20] `http://www-128.ibm.com/developerworks/library/specification/ws-bpel/`

mandatorily accessible via the SOAP protocol and expose an interface description according to WSDL. This is in line with one of the many existing definitions:

"*A Web service is a software system identified by a URI, whose public interfaces and bindings are defined and described using XML. Its definition can be discovered by other software systems. These systems may then interact with the Web service in a manner prescribed by its definition, using XML based messages conveyed by internet protocols*" [BCF+03]

However, there are dozens of other, partly contrary, definitions of the term Web service. In [GMSO03] we list several definitions and conclude that a concise axiomatization of such an overloaded term is necessary to avoid confusion among developers and ontology users.

(D36) follows the definition above and specifies WebService as a special kind of CSO:Software which is OIO:orderedBy a *WSDLEncoding*. The *WSDLEncoding* is an OIO:InformationEncodingSystem as defined in the Ontology of Information Objects. For our middleware domain, (A30) further constrains the intended meaning of WebService by axiomatizing that it either reveals functionality residing in a COSC:SoftwareComponent or a combined service specified by an OoP:Plan.[21]

(D36) $\mathsf{WebService}(x) =_{def} \mathsf{CSO{:}Software}(x) \wedge \forall y(\mathsf{OIO{:}orderedBy}(x, y) \wedge y = \mathit{WSDLEncoding})$

(A30) $\mathsf{WebService}(x) \rightarrow$
$(\exists sc(\mathsf{CSO{:}invokes}(x, sc) \wedge \mathsf{COSC{:}SoftwareComponent}(sc)) \oplus$
$\exists p, t(\mathsf{CSO{:}executes}(x, t) \wedge \mathsf{CSO{:}ComputationalTask}(t) \wedge$
$\mathsf{DnS{:}defines}(p, t) \wedge \mathsf{OoP{:}Plan}(p) \wedge \mathsf{OIO{:}expresses}(x, p)))$

6.2 Service Profiles and Taxonomies

The *analyzing message contexts*, *selecting service functionality*, *relating communication parameters*, *aggregating service information* and *quality of service* use cases require the modelling of service profiles and taxonomies. Similar to COSC:ComponentProfiles, we group the different descriptions relevant for a Web service in a ServiceProfile in (D37) below. We expect that such a grouping makes browsing and querying for developers more convenient. The information grouped by a ServiceProfile might have different origins. Hence, we also add informationTimestamp and informationSource as simple attributes to parts of the profile. We omit their definition because both are simple attributes with `xsd:string`. Furthermore, ServiceProfiles can be specialized and put into a taxonomy.

[21] Note that the symbol $\oplus$ represents the logical xor (exclusive or) connective.

ServiceProfiles differ from COSC:ComponentProfiles in two ways: First, they can have QualityOfService parameters.[22] QualityOfService parameters are specializations of COSC:Characteristics and defined on ServiceProfiles as shown in (D38). Second, the ServiceProfile necessarily OIO:describes a WebService as opposed to COSC:ComponentProfiles which COSC:describe COSC:SoftwareComponents (cf. (A30)).

(D37) $\mathsf{ServiceProfile}(x) =_{def} \mathsf{COSC{:}Profile}(x) \wedge \forall y(\mathsf{COSC{:}describes}(x, y) \rightarrow \mathsf{WebService}(y))$

(D38) $\mathsf{QualityOfService}(x) =_{def} \mathsf{COSC{:}Characteristic}(x) \wedge \forall y(\mathsf{COSC{:}profiles}(y, x) \rightarrow \mathsf{ServiceProfile}(y))$

6.3 Example

In this section we revisit the example of Section 2.2, and show how it can be formalized with our ontology. The WS-BPEL process description can be parsed and relevant information can be extracted leading to an OoP:Plan consisting of several ComputationalTasks. Figure 13 on page 47 in the Appendix provides an overview.

We can now introduce axiom (A31) below to infer all WebServices which CSO:invoke other WebServices with attached CSO:PolicyDescription. With (A9) on page 21 and executes(*WebShopWS*, *checkAccount*), OoP:successor (*checkAccount*, *CallMastercardWS*) and CSO:accesses (*CallMastercardWS*, *MastercardWS*) we can entail invokesWebServiceWithPolicy(*WebShopWS*, *MasterCardWS*). Without our approach, obtaining this result would require tedious manual analyses of the WS-BPEL and WS-Policy descriptors.

(A31) $\mathsf{invokesWebServiceWithPolicy}(x, y) \leftarrow \mathsf{CSO{:}invokes}(x, y) \wedge \mathsf{WebService}(x) \wedge \mathsf{WebService}(y) \wedge \mathsf{COSC{:}describes}(sp, y) \wedge \mathsf{ServiceProfile}(sp) \wedge \mathsf{COSC{:}profiles}(sp, pd) \wedge \mathsf{CSO{:}PolicyDescription}(pd)$

7 Proof of Concept

We have proposed the design of an appropriate ontology. We have defined appropriateness at the beginning of Section 3 as follows: (*i*) the ontology should meet all the modelling requirements derived from our use cases, (*ii*) it should avoid the typical shortcomings of commonly built ontologies and (*iii*) it should enable reuse in specific platforms and reduce modelling efforts to a minimum. In this section, we detail where and how our ontology responds to (*i*), (*ii*) and (*iii*), in 7.1, 7.2, and 7.3, respectively.

[22] One may also specify quality parameters for software components such as done by [vH03], for instance. However, our use cases only require to express quality of service parameters for Web services.

7.1 Meeting the Modelling Requirements

Tables 1 and 2 summarize which parts of the ontology meet the requirements. The requirements comprise modelling (*i*) *libraries*, *licenses*, *component profiles*, *component taxonomies*, *API descriptions*, *semantic API descriptions*, *access rights* and *workflow information* of software components and (*ii*) *service profiles*, *service taxonomies*, *policies*, *workflow information*, *API descriptions*, as well as *semantic API descriptions* of Web services.

Table 1: Modelling requirements for software components and the parts of the ontology that meet the requirements.

Part of ontology \ Requirement	Libraries	Licenses	Component profiles	Component taxonomies	API descriptions	Semantic API descriptions	Access rights	Workflow information
OoP:Plan						×		×
CSO:Class,CSO:Method					×			
CSO:Interface						×		
CSO:PolicyDescription							×	
COSC:SoftwareLibrary	×							
COSC:License		×						
COSC:ComponentProfile			×	×				

Table 2: Modelling requirements for Web services and the parts of the ontology that meet the requirements.

Part of ontology \ Requirement	Service profiles	Service taxonomies	Policies	Workflow information	API descriptions	Semantic API descriptions
OoP:Plan				×		×
CSO:Class,CSO:Method					×	
CSO:Interface						×
CSO:PolicyDescription			×			
COWS:ServiceProfile	×	×				

7.2 Avoiding the Typical Shortcomings

Besides meeting the requirements, another goal was to avoid the typical shortcomings of common ontologies as outlined in Section 2.3, viz., *conceptual ambiguity*, *poor axiomatization*, *loose design*, and *narrow scope*. In the following we give some examples how the shortcomings are eliminated.

7.2.1 Conceptual Disambiguation

Common ontologies, such as OWL-S [MBH+04], suffer from conceptual ambiguity [MOGS04]. An example is the notion of OWL-S:Service which is defined twice and differently in the specification. In turn, both definitions stand in conflict with the axiomatization of the concept in the ontology. In [SOR04], we have found a similar dilemma regarding the plethora of meanings and definitions of terms, such as component, software component, or software module. Both ontologies fail to convey their intended meanings of such terms and leave the interpretation to the ontology user.

In contrast to such commonly built ontologies we have captured the intended meanings of concepts and associations as precisely as possible. Our definition of terms such as Web service (Definition (D36) on page 31) or software component (Definition (D30) on page 26) are in line with the natural language definitions prevailing in the middleware community. Comparing both definitions makes evident that very few concepts actually differ when "upgrading" from software components to Web services. Only minor extensions to the Core Software Ontology are required to capture the differences between software components and Web services.

While our definitions of the terms Web service and software component may not be the only ones, the fact that they are highly axiomatized allows comparing them to alternative definitions and allows fostering discussions on alternative conceptualizations. We argue that this will enable mutual understanding which is crucial for information integration of any kind.

7.2.2 Increased Axiomatization

Common ontologies are often reduced to a simple taxonomy with domain and range restrictions on associations. An example taken from [MBH+04] are the OWL-S:ControlConstructs which define how composite processes are combined.

In our ontology we have made use of the Ontology of Plans which provides extensive axiomatization of OoP:Tasks and subconcepts thereof. OoP:Tasks are directly comparable to the OWL-S:ControlConstructs, but provide a heavyweight axiomatization. An example is *SynchroTask* (an instance of OoP:ControlTask) which matches the concept of OWL-S:Join in the OWL-S:SplitJoin control construct. A *SynchroTask* joins a set of tasks after a branching and waits for the execution of all (except the optional ones) tasks that are direct successors to a *ConcurrencyTask* or *AnyOrderTask*. Below we give the axiomatization of the *SynchroTask* as introduced in [GBCL04].

$$\mathsf{ControlTask}(\mathit{SynchroTask}) \rightarrow \exists t_1, t_2, t_3 (t_1 = \mathit{ConcurrencyTask} \vee t_1 = \mathit{AnyOrderTask}) \wedge \mathsf{successor}(t_1, x) \wedge (\mathsf{ComplexTask}(t_2) \vee \mathsf{ActionTask}(t_2)) \wedge (\mathsf{ComplexTask}(t_3) \vee \mathsf{ActionTask}(t_3)) \wedge \mathsf{directSuccessor}(t_2, \mathit{SynchroTask}) \wedge \mathsf{directSuccessor}(t_3, \mathit{SynchroTask})$$

Another example from [MBH+04] is the OWL-S:components association, which is used to relate OWL-S:ControlConstructs to their components. In OWL-S this association is described merely as a special kind of owl:Property with a domain of OWL-S:ControlConstruct. Therefore, the intended meaning of OWL-S:components remains unclear. Is it a parthood association? And if yes, is it temporary, transitive, etc.? Our ontologies are superior because they make use of the Ontology of Plans. The latter exploits the DOLCE:temporaryComponent association which has a firm foundation as a special kind of the more basic DOLCE:component mereological association and DOLCE:partlyCompresent temporally indexing association. Both are characterized by formal restrictions on their application to other basic concepts.

7.2.3 Improved Design

In our ontology we propose to use contextualization as a design pattern. Contextualization allows us to move from monolithic component or service descriptions to the representation of different, possibly conflicting views with various granularity. The Descriptions & Situations ontology provides us with the basic primitives of context modelling such as the notion of roles, which allows us to talk about inputs and outputs on the abstract level, i.e., independent of the objects that play such roles.

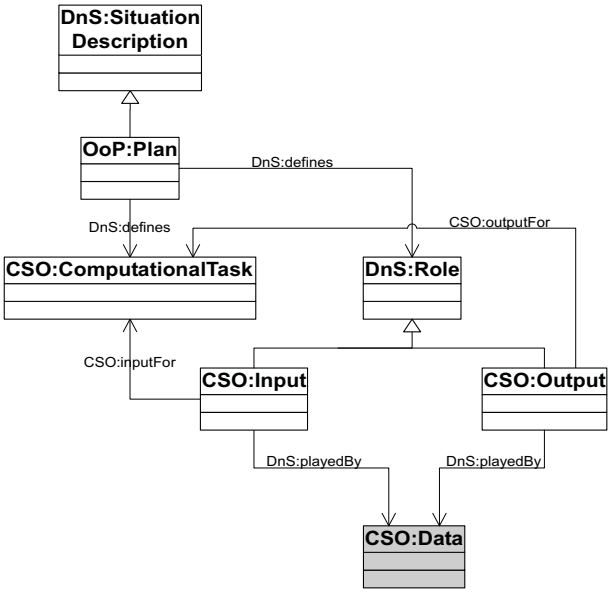

Figure 10: Solution to the attribute binding problem. Data can play both the role of an Input and an Output at the same time. Inputs and Outputs can be linked to ComputationalTasks in a Plan. White classes represent descriptive entities, grey classes represent ground entities.

Using this pattern results in a much more intuitive representation of attribute binding than in OWL-S. This patterns applies clearly defined semantics and scoping provided by Descriptions & Situations. Attribute binding in OWL-S is necessary to express, e.g., that the output of a process is the input to another process. In our ontology, inputs and outputs can be modelled as DnS:Roles which serve as variables. Thus, CSO:Data can play multiple roles within the same or different descriptions. It is natural to express that the given CSO:Data is output with respect to one process, but input to another (cf. Figure 10).

7.2.4 Wider Scope

Components and services exist on the boundary of the world inside an information system and the external world. Web services, in particular, may carry out operations to support a real-world service. Functionality, which is an essential property of a service, then arises from the entire process that comprises computational, as well as real-world activities.

The distinction between information objects, events and physical objects is not explicitly made in most ontologies. In our ontology this separation naturally follows from the use of DOLCE and the Ontology of Information Objects, where the distinction is an important part of the characterization of concepts. In particular, it becomes possible to be more precise about the kinds of relationships that can occur among objects or between objects and events.

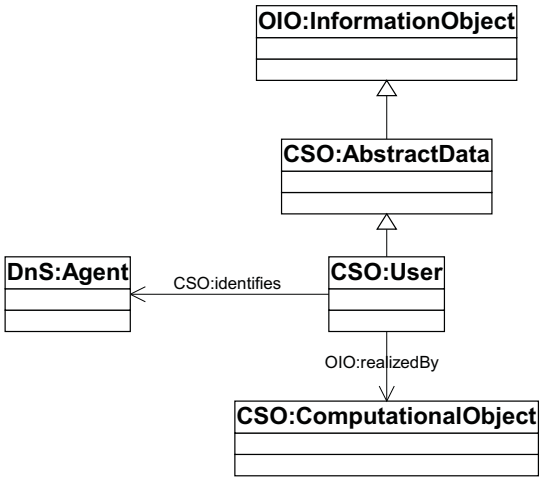

Figure 11: Using the Ontology of Information Objects allows us to model the relationship between a user in an information system and its corresponding agent (e.g., a natural person).

For example, we can distinguish among a physical object (such as a nat-

ural person), an information object (such as user account in an information system), and represent the link between them. The capabilities provided by our ontology are shown in Figure 11. It is worthwhile to make such differences explicit, e.g., when we want to infer the total of access rights granted for a natural person who might have several user accounts in and across information systems.

7.3 Enabling Reuse

Finally, we have designed the ontology in a way to be easily reusable in different platforms *and* as specific as possible at the same time. The following three steps have to be taken in order to allow for reuse in a specific platform: (*i*) specialization of the core concepts and associations to reflect the idiosyncracies of the platform. For example, we have to introduce EnterpriseBean as a special kind of COSC:SoftwareComponent in a J2EE-based platform. Step (*ii*) removes concepts and associations that have been introduced merely for reference purposes. As an example, it is unlikely and not required to model particular ComputationalObjects or ComputationalActivities for the reasoning at run time. Both were introduced to better explain concepts such as Software or Data. Finally, step (*iii*) requires a decision for an ontology language that can be reasoned with at run time. Accordingly, the axiomatization has to be adapted to this language. This might be a description logic, such as OWL DL, which is less expressive than the modal logic S5.

In fact, we took the three steps and reused the ontology in our ontology-based application server. The details are given in [Obe06].

8 Related Work

Related work can be split in two categories: Established schema specifications for application management and existing ontologies. The most prominent example in the first category is the Common Information Model (CIM),[23] an object-oriented specification developed by the Desktop Management Task Force (DMTF). CIM provides a common definition of management information for systems, networks, applications and services, and allows for vendor extensions. For example, there is also a specification for J2EE application servers defining Enterprise JavaBeans and the like. CIM's purpose it to define the details for integration with other management models. Other management models include the Management Information Base (MIB), stemming from network management and modelling information of network devices, or the schema of the Web Service Distributed Management (WSDM) by OASIS.[24] Finally, there are the XML-DTDs of applica-

[23] http://www.dmtf.org/standards/cim/

[24] http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsdm

tion servers and Web service descriptors that formally define characteristics and relationships relevant for our purposes. However, all of those efforts lack formal semantics and are not suitable for reference purposes.

The second category of related work are existing ontologies. There have been several efforts which deal almost exclusively with the moving target of Web services: OWL-S, WSMO, SWSF, and METEOR-S. In spite of their seminal roles that led to a lot of fruitful research into the formulation and application of Web service ontologies, they exhibit the typical shortcomings which we discussed in Section 2.3. OWL-S [MBH+04] is one of the first core ontologies explicitly aiming at automatic discovery, automatic invocation, automatic composition and interoperation as well as automatic execution of Web services. OWL-S enriches an existing WSDL description by an ontological capability description and workflow information. It is split into three modules, viz., "Profile" for describing what the service does, "Process" for describing how the service works and "Grounding" for describing how the service is implemented. The Web Service Modelling Ontology (WSMO) [FB02] has goals similar to OWL-S. However, it additionally defines an Execution Environment (WSMX) for the dynamic discovery, selection, mediation, invocation, and inter-operation of Semantic Web Services. Aspects considering the ontology representation language are separately defined in the Web Services Modelling Language (WSML). The recent Semantic Web Services Framework (SWSF) [BBB+05] also provides an ontology for Semantic Web Services, namely the Semantic Web Services Ontology (SWSO). By its axiomatization of Web service related notions, it addresses deficiencies of OWL-S in a way similar to our work. This framework also defines its own language (called SWSL) for ontological representation, providing a first-order logic style as well as a rule-based variant. Finally, the METEOR-S project at the LSDIS Lab, University of Georgia, aims to extend WS* descriptors with Semantic Web technologies to achieve greater dynamics and scalability.[25] More specifically, METEOR-S focuses on adding semantics to WSDL and UDDI, on adding semantics to WS-BPEL and on a semi-automatic approach for annotating Web services described in WSDL. The endeavor is to define and support the complete lifecycle of Semantic Web Services processes [POSV04].

Furthermore, there has been some work that overlaps with the ideas presented here. For example, the COHSE Java ontology[26] offers a formal schema for expressing a Java software project by an ontology. The open source project Introspector[27] is a back-end to the popular GNU compiler collection gcc,[28] which generates an RDF defined ontology out of gcc compiled source code, and thus works with all languages supported by gcc, for

[25] http://lsdis.cs.uga.edu/Projects/METEOR-S/

[26] http://cohse.semanticweb.org/software.html

[27] http://introspector.sourceforge.net

[28] http://gcc.gnu.org

example C, C++, Java, Fortran and others. [Wel95] offers a more profound and sound ontology-based foundation to these level of detail, analyzing the constructs available when programming. All these works provide support for using ontologies in the area of software development, but on a much finer grained level than the work presented here. Thus, such ontologies could be used complementary to ours.

In addition, there are theories of distributed interaction. Examples for such theories include CCS [Mil80], CSP [Hoa85], and Z [Dil94]. These languages focus mainly on the exact meaning of distributed interaction and little on the relevant concepts, such as software components, user roles etc., involved in such a theory. In contrast, our approach proposes several ontologies as formal theories to describe some crucial aspects of software systems and software systems behavior at large. In particular, this paper focuses on aspects of *modularization* of and *communication* within large software systems. In the future it might be useful to expand our ontology toward such process algebras. The languages mentioned, however, often have an implicit underlying conceptualization that may be hard to align with our ontology. There is also work evolving these well-known languages avoiding such strong commitment and thus may be more interesting with regards to a fruitful integration [Pir94, Qua98].

An example of a higher level software component ontology in use is provided by [AHS03]. Instead of the technological management of software components as provided by the middleware layer and described herein, her work focusses on the social and project-level management of Open Source software projects.

Finally, there are some ontologies which focus on specific aspects, whereas our ontology tries to relate the different aspects in a larger focus. Examples are the Core Plan Representation (CPR) [Pea98] and the Process Specification Language (PSL) [GM03b] which are comparable to the Ontology of Plans. UPML, the Unified Problem-solving Method Development Language [FBMW99], has been developed to describe and implement intelligent broker architectures and components to facilitate semi-automatic reuse and adaptation.

9 Conclusion

We have presented a set of ontologies, CSO, COSC, and COWS, that formalize modularization and communication in large software systems. In doing so, the ontologies cover the most important paradigms in this arena, i.e., software components and Web services. The ontologies are grounded in foundational ontologies, as well as in application use cases. Thus, they fulfil a number of competing purposes.

First, they allow for an accurate discussion of what is meant by terms

such as "Web service," "Profile" and the like. In related, seminal approaches, such as OWL-S, the meaning of these terms were only weakly formalized leaving their disambiguation to the intuition of the reader, a situation that we here improve upon considerably.

Second, the overall set of ontologies surveyed or defined here untangle the different aspects one must consider when formalizing software components and Web services. There are foundational notions, such as "Objects" or "Plans," there are generic notions, such as "Software" (with its different shades of meaning) and there are notions that are specific to modularization and communication, such as "software component," or "invokes." Our organization lays out corresponding concepts into several ontologies of increasing specificity. Thus, we provide a modular basis that lends itself more easily for extension than a monolithic approach.

Finally, the ontologies provide support for developing and managing software systems at large scale, such as has been demonstrated by systems for managing application servers and Web services.

The reader may note that what is presented here are the *reference ontologies* in this domain. They use a powerful logic, viz., modal logic S5. For actual work, these reference ontologies need to be reduced to knowledge representation schemes that are more amenable to operation. Though we have such initial *application ontologies*, what we have is quite poor and we still need to explore the trade-off between representational power and run-time efficiency in order to come up with more powerful application ontologies.

For the future, we foresee that the set of use cases might have to be extended leading to further specifications in CSO, COSC, and COWS. Based on some core assumptions about the top levels, we have done our best in order to allow for monotonic extensions. In case a change of paradigm is necessary, e.g., in the Ontology of Plans in order to accommodate some particular way of reasoning (e.g., a different knowledge representation for a planner with specific properties), we expect that changes of representation will be rather small in the other ontologies.

Acknowledgements We would like to thank our colleagues Peter Mika, Marta Sabou, as well as Rudi Studer, Pascal Hitzler, and Peter Haase for their fruitful reviews and discussions that helped to shape this contribution.

Research for this paper was partially funded by SmartWeb, a German BMBF funded project, by ASG - Adaptive Services Grid, EU IST project FP6 - 004617, and by DIP, an EU IST project 507483 .

References

[ACKM03] Gustavo Alonso, Fabio Casati, Harumi Kuno, and Vijay Machiraju. *Web Services*. Springer, Sep 2003.

[AHS03] Anupriya Ankolekar, James Herbsleb, and Katia Sycara. Addressing Challenges to Open Source Collaboration With the Semantic Web. In Joseph Feller, Brian Fitzgerald, Scott Hissam, and Karim Lakhani, editors, *Proceedings of Taking Stock of the Bazaar: The 3rd Workshop on Open Source Software Engineering, the 25th International Conference on Software Engineering (ICSE)*, Washington, D.C., 2003. IEEE Computer Society.

[BBB+05] S. Battle, A. Bernstein, H. Boley, B. Grosof, M. Gruninger, R. Hull, M. Kifer, D. Martin, S. McIlraith, D. McGuinness, J. Su, and S. Tabet. Semantic Web Services Framework (SWSF). W3C Member Submission. Available at `http://www.w3.org/Submission/2005/07/`, May 2005.

[BCF+03] David Booth, Michael Champion, Chris Ferris, Francis McCabe, Eric Newcomer, and David Orchard. Web Services Architecture, May 2003.

[Dil94] Antoni Diller. *Z: An Introduction to Formal Methods*. John Wiley & Sons, 2nd edition, 1994.

[FB02] Dieter Fensel and Christoph Bussler. The Web Service Modeling Framework WSMF. *Electronic Commerce: Research and Applications*, 1:113–137, 2002.

[FBMW99] Dieter Fensel, Richard Benjamins, Enrico Motta, and Bob J. Wielinga. UPML: A framework for knowledge system reuse. In Thomas 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.

[GBCL04] Aldo Gangemi, Stefano Borgo, Carola Catenacci, and Jos Lehmann. Task taxonomies for knowledge content. Metokis Deliverable D07, Sep 2004.

[GM03a] Aldo Gangemi and Peter Mika. Understanding the Semantic Web through Descriptions and Situations. In *DOA/CoopIS/ODBASE 2003 Confederated International Conferences DOA, CoopIS and ODBASE, Proceedings*, LNCS. Springer, 2003.

[GM03b] Michael Grüninger and Christopher Menzel. The Process Specification Language (PSL) Theory and Applications. *AI Magazine*, 24(3):63–74, 2003.

[GMSO03] Aldo Gangemi, Peter Mika, Marta Sabou, and Daniel Oberle. An Ontology of Services and Service Descriptions. Technical report, Laboratory for Applied Ontology (ISTC-CNR), Viale Marx, 15, 00137 Roma, 2003.

[GST04] Aldo Gangemi, , Maria-Teresa Sagri, and Daniela Tiscornia. A Constructive Framework for Legal Ontologies. Internal project report, EU 6FP METOKIS Project, Deliverable, 2004. `http://metokis.salzburgresearch.at`.

[HFPS99] R. Housley, W. Ford, W. Polk, and D. Solo. Internet X.509 Public Key Infrastructure Certificate and CRL Profile. Network Working Group. Request for Comments: 2459., Jan 1999.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[Mah04] Q.H. Mahmoud, editor. *Middleware for Communications*. Wiley, 2004.

[MBG+02] Claudio Masolo, Stefano Borgo, Aldo Gangemi, Nicola Guarino, Alessandro Oltramari, and Luc Schneider. The WonderWeb Library of Foundational Ontologies. WonderWeb Deliverable D17, Aug 2002. `http://wonderweb.semanticweb.org`.

[MBG+03] Claudio Masolo, Stefano Borgo, Aldo Gangemi, Nicola Guarino, and Alessandro Oltramari. Ontology Library (final). WonderWeb Deliverable D18, Dec 2003. `http://wonderweb.semanticweb.org`.

[MBH+04] David Martin, Mark Burstein, Jerry Hobbs, Ora Lassila, Drew McDermott, Sheila McIlraith, Srini Narayanan, Massimo Paolucci, Bijan Parsia, Terry Payne, Evren Sirin, Naveen Srinivasan, and Katia Sycara. OWL-S: Semantic Markup for Web Services. `http://www.daml.org/services/owl-s/1.1/`, Nov 2004.

[Mil80] Robin Milner. *A Calculus of Communicating Systems*. Springer, 1980.

[MOGS04] Peter Mika, Daniel Oberle, Aldo Gangemi, and Marta Sabou. Foundations for Service Ontologies: Aligning OWL-S to DOLCE. In *The 13th International World Wide Web Conference Proceedings*, pages 563–572. ACM, May 2004.

[NYHR05] C. Neuman, T. Yu, S. Hartman, and K. Raeburn. The Kerberos Network Authentication Service (V5). Network Working Group. Request for Comments: 4120., Jul 2005.

[Obe06] Daniel Oberle. *Semantic Management of Middleware*, volume I of *The Semantic Web and Beyond*. Springer, New York, Jan 2006.

[OESV04] Daniel Oberle, Andreas Eberhart, Steffen Staab, and Raphael Volz. Developing and Managing Software Components in an ontology-based Application Server. In *5th International Middleware Conference*, LNCS. Springer, 2004.

[OGGM02] A. Oltramari, A. Gangemi, N. Guarino, and C. Masolo. Sweetening ontologies with DOLCE. In Asunción Gómez-Pérez and V. Richard 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.

[OLES05] Daniel Oberle, Steffen Lamparter, Andreas Eberhart, and Steffen Staab. Semantic Management of Web Services. In Boualem Benatallah, Fabio Casati, and Paolo Traverso, editors, *Service-Oriented Computing - ICSOC 2005: Third International Conference, Amsterdam, The Netherlands, December 12-15, 2005. Proceedings*, volume 3826 of *Lecture Notes in Computer Science*, pages 514–519. Springer-Verlag GmbH, DEC 2005.

[OSE06] Daniel Oberle, Steffen Staab, and Andreas Eberhart. Semantic Management of Distributed Web Applications. *IEEE Distributed Systems Online*, 7(4), Feb 2006.

[Pea98] Adam Pease. Core Plan Representation. Object Model Focus Group, Nov 1998.

[Pir94] L. Ferreira Pires. *Architectural Notes: a Framework for Distrbudet Systems Development*. PhD thesis, University of Twente, The Netherlands, 1994.

[POSV04] A. Patil, S. Oundhakar, A. Sheth, and K. Verma. METEOR-S Web Service Annotation Framework. In *The 13th International World Wide Web Conference Proceedings*, pages 553–563. ACM, May 2004.

[Qua98] D.A.C. Quartel. *Action relations. Basic design concepts for behaviour modelling and refinement*. PhD thesis, University of Twente, The Netherlands, 1998.

[RKL+05] Dumitru Roman, Uwe Keller, Holger Lausen, Rubén Lara Jos de Bruijn, Michael Stollberg, Axel Polleres, Cristina Feier,

Christoph Bussler, and Dieter Fensel. Web Service Modeling Ontology. *Applied Ontology*, 1(1):77—106, 2005.

[SOR04] Marta Sabou, Daniel Oberle, and Debbie Richards. Enhancing Application Servers with Semantics. In Shonali Krishnaswamy, Seng W. Loke, and Jian Yang, editors, *1st Australian Workshop on Engineering Service-Oriented Systems (AWESOS 2004) Wednesday, 14 April 2004, Melbourne, Australia. In conjunction with the Australian Software Engineering Conference (ASWEC)*, pages 7–15. Monash University, Australia, 2004.

[vH03] A.T. van Halteren. *Towards an adaptable QoS aware middleware for distributed objects*. PhD thesis, University of Twente, The Netherlands, 2003.

[Wel95] Christopher Welty. *An Integrated Representation for Software Development and Discovery*. PhD thesis, Rensselaer Polytechnic Institute Computer Science Department, 1995.

Appendix

Descriptors of Example in Section 2.1

web.xml

```
...
<security-constraint>
 <web-resource-collection>
  <web-resource-name>WebShopServlet</web-resource-name>
  <url-pattern>/servlet/WebShopServlet</url-pattern>
  <http-method>GET</http-method>
 </web-resource-collection>
</security-constraint>

<login-config>
 <auth-method>BASIC</auth-method>
</login-config>
...
```

doGet() Method of the WebShopServlet

```
public class WebShopServlet extends HttpServlet {
 public void doGet(HttpServletRequest request,
                   HttpServletResponse response)
 {
  ...
  //get customer info via CustomerEntityBean
  CustomerObject cObject = cHome.create()
  out.println(cObject.getCustomerName())
  ...
 }
```

ejb-jar.xml of the CustomerEntityBean

```
...
<ejb-jar>
<enterprise-beans>
 <entity>
  <ejb-name>CustomerEntityBean</ejb-name>
  <ejb-class>edu.unika.aifb.CustomerEntityBean</ejb-class>
  ...
  <security-identity>
   <run-as-specified-identity>
    <role-name>dbuser</role-name>
```

```
   </run-as-specified-identity>
  </security-identity>
 </entity>
</enterprise-beans>
</ejb-jar>
...
```

Descriptors of Example in Section 2.2

WS-BPEL document

```
...
<process name="checkAccount">
 <switch ...>
  <case condition="getVariableData('creditcard')='VISA'">
   <invoke partnerLink="toVISA"
    portType="visa:CCPortType"
    operation="checkCard"...>
   </invoke>
  </case>
  <case condition="getVariableData('creditcard')='MasterCard'">
   <invoke partnerLink="toMastercard"
    portType="mastercard:CCPortType"
    operation="validateCardData"...>
   </invoke>
  </case>
 ...
 </switch>
</process>
...
```

WS-Policy document

```
...
<wsp:Policy>
 <wsp:ExactlyOne>
  <wsse:SecurityToken>
   <wsse:TokenType>wsse:Kerberosv5TGT</wsse:TokenType>
  </wsse:SecurityToken>
  <wsse:SecurityToken>
   <wsse:TokenType>wsse:X509v3</wsse:TokenType>
  </wsse:SecurityToken>
 </wsp:ExactlyOne>
</wsp:Policy>
...
```

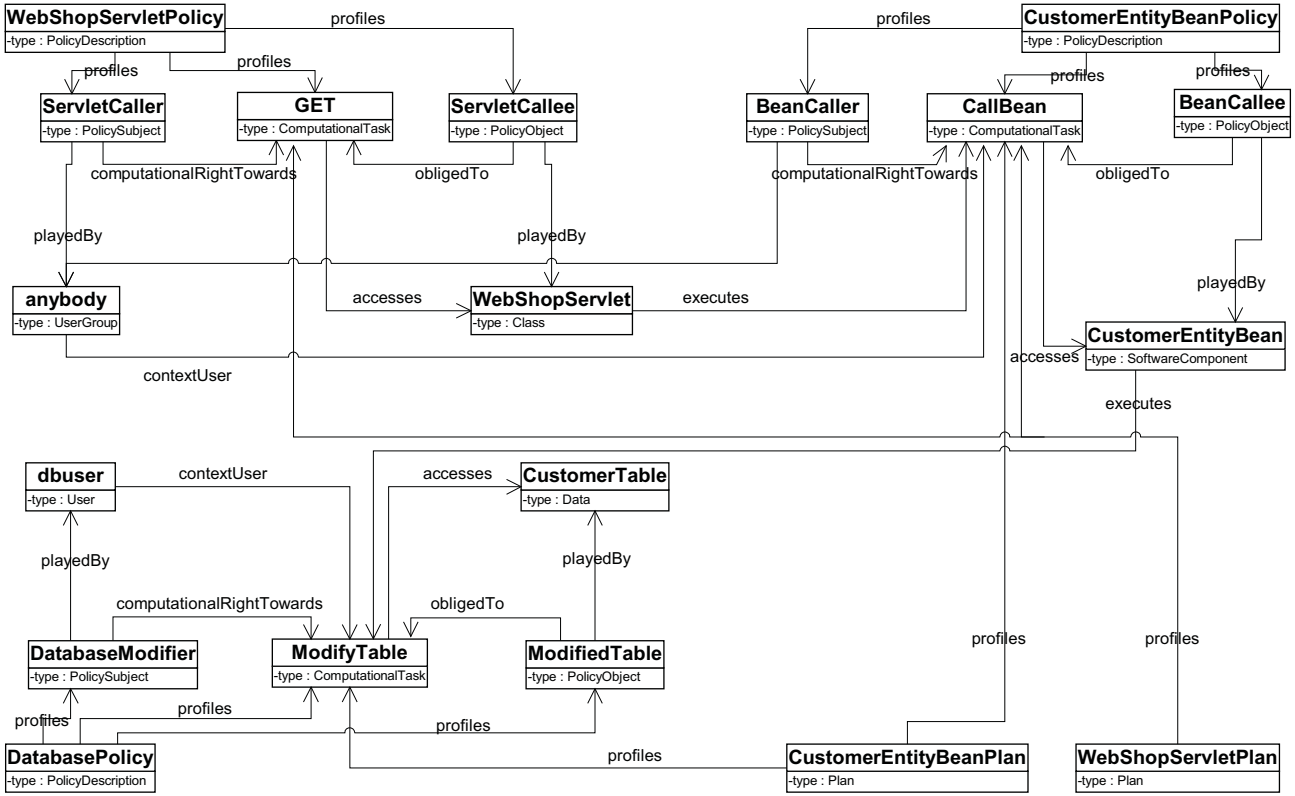

Figure 12: UML diagram of the software component example on page 30.

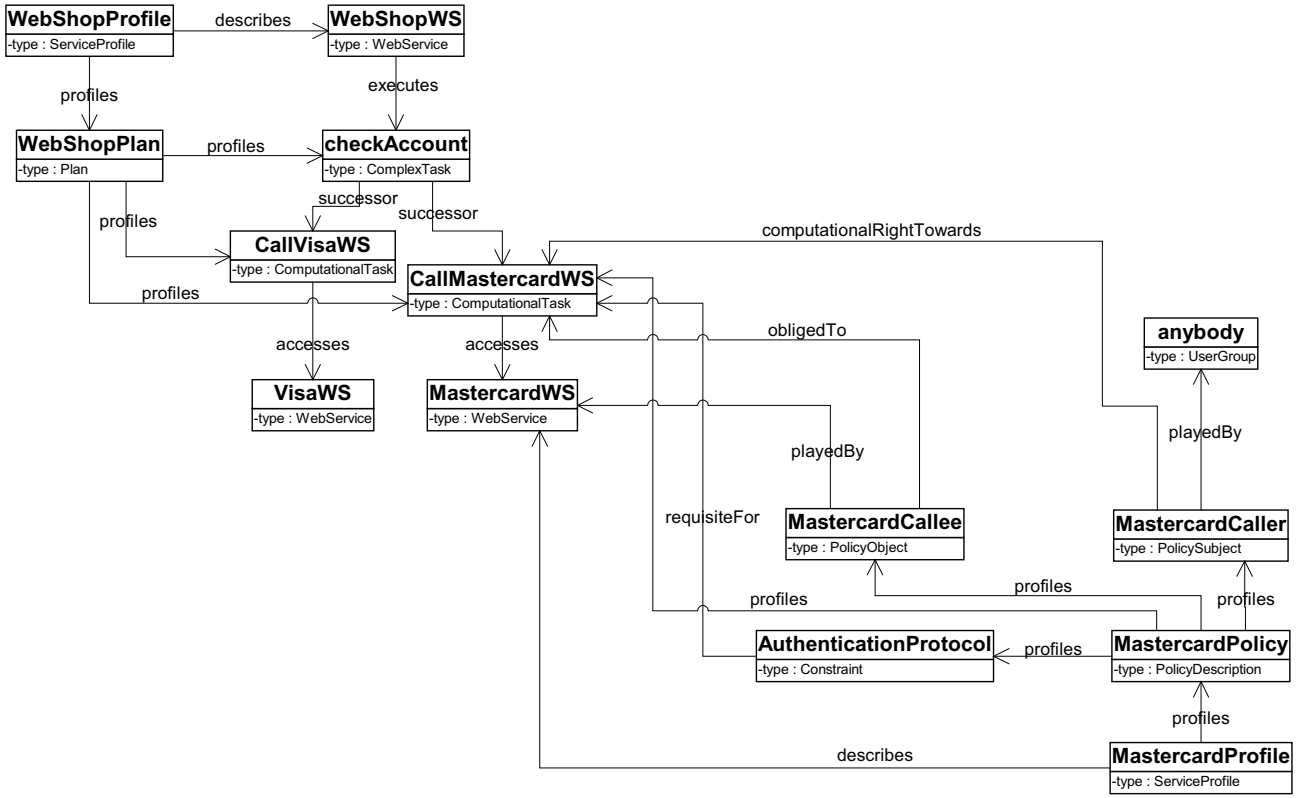

Figure 13: UML diagram of the Web services example on page 32.