

What is Needed for Semantic Service Descriptions?

A Proposal for Suitable Language Constructs

Michael Klein¹

Birgitta König-Ries²

Michael Müssig¹

¹Institute for Program Structures and Data Organization,
Universität Karlsruhe, 76128 Karlsruhe, Germany

²Institute of Computer Science

Friedrich-Schiller-Universität Jena, 07743 Jena, Germany

Abstract: The big promise of service-oriented computing is the ability to form agile networks, i.e. networks of loosely-coupled participants that cooperate by dynamically discovering and invoking each other's services at run-time. The major prerequisite for this promise to be redeemed is an appropriate semantic service description. In this paper, we identify requirements towards such a service description language and show that neither of the two main current approaches, namely OWL-S and WSMO, is able to fully meet these requirements. We then proceed to suggest additional language constructs and a prototypical language, the DIANE Service Description (DSD) that implements these constructs. We explain how service offers and requests can be described and matched using DSD.

Keywords: Semantic Service Descriptions; Matchmaking; OWL-S; WSMO

Biographical notes: *Michael Klein* studied computer science at the University of Karlsruhe. Since 2001, he is working towards his PhD at the Institute for Program Structures and Data Organization at the University of Karlsruhe. His main research area are semantic descriptions for mobile services.

Birgitta König-Ries is the head of the endowed Heinz-Nixdorf chair for Practical Computer Science at the Department of Mathematics and Computer Science of the Friedrich-Schiller-Universität in Jena. Before joining the University of Jena, she has held a temporary professorship at the Technische Universität München (2003-2004) and has worked as an assistant professor with the University of Karlsruhe (since 2001). Before that, from 1999 until 2001 she was a postdoctoral research associate at the Center for Advanced Computer Studies at the University of Louisiana at Lafayette and the IT2 Institute of Florida International University. She received my PhD degree from the University of Karlsruhe in 1999. Her main research interest is on resource usage in dynamic environments, such as mobile (ad hoc) networks and peer to peer systems. The solutions developed are based on the service-oriented computing paradigm.

Michael Müssig studied computer science at the University of Karlsruhe. Since 2005, he is working at Accenture as analyst in the area of communications and high tech.

1 INTRODUCTION

Service-oriented computing (SOC) is widely regarded as *the* computing paradigm of the near future [24]. However, it will only be advantageous over other approaches, if it reaches its full potential, which is the building of agile networks. Here, the participants are loosely coupled as services are discovered and invoked dynamically during run-time.

There exists a wealth of scenarios where this is desirable. Consider, for instance, a manufacturer, that regularly needs to order a large quantity of a certain part, e.g., a bolt. Nowadays, typically, the purchasing department will evaluate possible suppliers once (or at best at periodic intervals). Then, the supplier that currently matches the requirements best is chosen. Web Services may be used to subsequently automate the repetitive order process. However, it would be much more desirable to allow for an automatic selection of the best suited supplier for each new order since prices, the quality of the product and supply times may change.

Consider as a second example a guest speaker using a seminar room. This room will be equipped with cameras, projectors, lamps, screens, blinds and so on. The speaker needs to attach her laptop to the equipment already in the room. Instead of fiddling with a number of switches as is necessary today to connect the laptop, choose a projector, choose a screen, ensure that lighting is appropriate, and so on, it would be desirable for all this to happen automatically. Ideally, the speaker just chooses a "start presentation" button on her laptop resulting in ideal room setting for her presentation. Again, since the equipment available in the room and the environmental conditions (light, noise, number of people present, kind of presentation,...) will change, it is necessary to dynamically find appropriate services. On a cloudy day with only a handful of people in the audience it may be sufficient to just turn on a projector and dim the lights. On a sunny day with a large audience and a need to conserve the presentation, it may be necessary to lower the blinds, switch on a projector and microphones, adjust camera settings for recording and so on. All this should be as transparent as possible for the speaker.

In this paper, we will use an easy to understand "toy scenario". In this scenario, we envision an application running on a mobile computer that allows its users to purchase movie tickets. Again, since the location of the user will change over time, it is not possible to statically assign certain service providers. Instead, a dynamic, automatic selection of appropriate providers is necessary.

All these scenarios will only become a reality if an appropriate semantic description of the services is available. However, finding descriptions that allow for fully automatic and also semantically precise execution of services is very hard. At the moment, a large research community is dealing with this problem and already two quasi-standards (OWL-S and WSMO, see Section 3) are evolving. Although both state that fully automatic service usage is possible, many problems – often stemming from the ontology language – are hampering the success. Therefore, in this paper, we examine which requirements a semantic service description has to meet in order to support dynamic service discovery and invocation.

In Section 2, we will present these requirements. We will explain in some detail why we need a specialized ontology languages, the ability to capture in detail what a service does, the means to express the relationship between message-flow and effect of a service, and a separate treatment of service offers and requests.

Then, in Section 3, we analyze OWL-S and WSMO with respect to their degree of fulfillment of the requirements.

After that, we present a prototypical language, the DIANE Service Description (DSD), which shows ideas how to fulfill the requirements concretely. The language is not based on one of the existing ontology or logic languages such as the often used description logic or f-logic, but uses an ontology language and a reasoning mechanism that is specialized for service discovery: the DIANE Elements (DE). In Section 4, we explain in some detail the specific concepts of DE that enables this. These are in particular: full state-orientation and the provision of sets and variables and a reasoning based on the determination of set inclusion.

In Section 5, we explain how the DIANE Ele-

ments are used within the DIANE Service Description (DSD) to unambiguously yet flexibly describe service offers and requests. In Section 6, we show how we can implement and use the reasoning functions in order to match service descriptions. DSD and its underlying ontology language DE can be seen as a pool of ideas which should be adapted by or integrated into the major approaches. The paper ends with a summary and an outlook to future work in Section 7.

2 REQUIREMENTS

The main motivation for the development of semantic service description languages is the desire to enable fully automatic service processing, i.e. starting with sending a service request, an appropriate service provider has to be selected and invoked in a correct manner. Ideally, the following process needs to be performed: A user or an application (or more general: the *service requestor*) describes a function that he needs as a service request. Also, the *service providers* publicly describe their offered functionalities as service offers in the same description language. Then, the request description is compared to all these descriptions of offered services. A service is matching, if it offers the desired functionality within certain bounds of deviation. It should not be necessary that the services have a syntactically equal description or identical interfaces. From the set of matching services, one provider is chosen and invoked automatically, i.e. without asking the service requestor again.

It is evident that an appropriate service description language is the key when realizing such a service oriented process. Typically, approaches try to achieve this goal by using concepts from the Semantic Web, i.e. making the descriptions machine interpretable by separating a concrete service description from the common knowledge of the world, which can be stored in an ontology. However, in the following, we will show that this alone is not enough to support the process described above. Rather, additional concepts are needed. These are:

- An unambiguous functional description of an of-

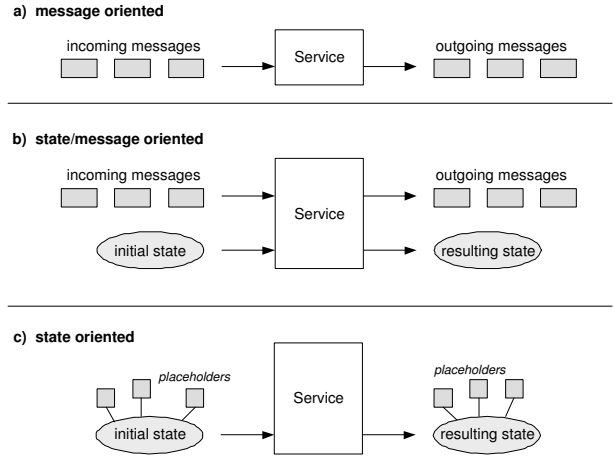


Figure 1: Classification of service description languages.

ferred service, which exactly explains which configuration leads to what result.

- An unambiguous description of a required functionality, which is different from a service offer description and exactly captures the service requestor’s needs.
- A reasoning support that is domain-specific and specialized on service discovery.
- An ontology language that contains special elements for the peculiarities of service descriptions.
- A shared ontology or the concept of mediation.

In the remainder of this section we will explain in detail, why we believe these extensions to be indispensable for successful service processing.

2.1 Unambiguous Functional Description of Offered Services

If we analyze existing approaches to service description, we can see that often only the data flow of the service is described, i.e. the service is described by its incoming and outgoing data (see Figure 1a). The most prominent example for such a description is the

Web Service Description Language WSDL [32]. However, such languages are not adequate for automatic processing. On the one hand, they do not offer any advantage in comparison to a mere interface description, where the functionality has to be guessed from the message flow, on the other hand, the description cannot broker requestor and providers that require or offer different messages (apart from trivial reorderings or renamings of their fields) even though their functionality may match.

Also a separate description of the states before and after service execution is not sufficient (see Figure 1b) as it remains unclear how the exchanged messages influence these states. Thus, again, the semantics of the service are not fully specified. Further on, the expectation for equal message types remains.

Thus, what is needed to unambiguously capture the semantics of a service is *pure state orientation*. A service should be described by the state change that takes place upon successful execution of the service. A precondition describes prerequisites for service execution, the effect describes the state of the world after the service has been executed. Note that this also includes effects on the knowledge of participants. In order to make explicit the influence the message flow has on these state changes, inputs and outputs need to be incorporated into the states. By doing so, messages are no longer regarded separately from the state change (cf. [18]). Furthermore, it becomes possible to match offers and requests that have different expectations with respect to the message flow (see Figure 1c, cf. [15]).

Services often do not provide exactly one effect, but are able to deal with several similar objects and transfer them to different similar states. For instance, a printer is able to print an arbitrary document, if it receives the appropriate input file. For this reason, a service description can be seen as a *family of effects*, which can be captured by introducing the notion of a *set of instances*. Furthermore, variables have to be included at certain points of the description. They enable the service requestor to choose or narrow the desired effect from the set and allow the service provider to return information over the achieved state. By this means, the influence of the parameters on the result is clearly visible. Such sets are called

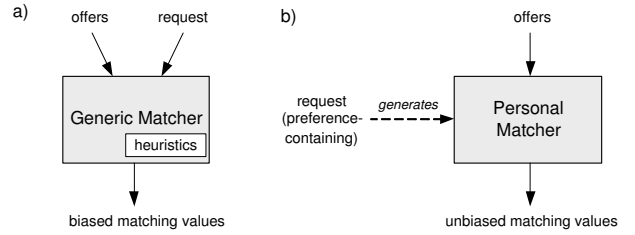


Figure 2: Generic vs. Personal Matcher

configurable sets.

To summarize, in order to unambiguously describe the semantics of the service, a **purely state based service description** is needed, which includes sets and variables to incorporate inputs and outputs into the state changes and to account for a certain flexibility with respect to the achieved effect.

2.2 Unambiguous Description of Required Functionality

Many existing service description languages use the same technique for describing requests and offers. Typically, the service requestor formulates the description of his required functionality as the perfectly matching service. Alternatively, in WSMO, the requestor specifies the desired effect of service execution. Then, in either case, a matcher calculates a similarity distance to all offer descriptions. However, this leads to a biased matching result as the matcher has to use heuristics to estimate deviations between offer and request (see Figure 2a). As these heuristics are not known to the requestor, he will often not be willing to accept an automatically chosen service, but will rather insist upon selecting the service himself from the list of matching results.

The problem can be solved when understanding the fundamental differences between an offer and a request description:

Offer descriptions. The service provider knows all details of his service. Thus he can describe it as a set of single instances which is customizable via variables.

Request descriptions. The service requestor wants to have a certain operation done and does not think inevitably of a certain service. Often, several different services are suitable to fulfill his requirements. Thus, it is not very reasonable to denote one single instance. Additionally, often, the requestor will be willing to accept slightly different effects. Thus, it would be more appropriate for the requestor to describe a set of services that match his requirements together with preferences stating which members of this set he prefers over which others. Obviously, nothing is gained, if the requestor is forced to explicitly write down the individual instances of the set and his preferences for each of them. If, however, the requestor would be able to specify *declaratively* the set and the preferences, a giant step towards automatic service binding would be taken.

Therefore, what is needed in an ontology language is the possibility to declaratively specify sets. Additionally, it needs to be possible to express preferences for individual members of these sets, which could be done by using a fuzzy membership function for the set. Such a **preference-containing request description** can be used to generate a personal matcher, which calculates unbiased matching results (see Figure 2b, cf. [15, 17]).

2.3 Service Discovery Oriented Reasoning Support

The most important step during service discovery is the matching between service offer and request descriptions. To make use of the semantic description, the matching algorithm should be supported by the underlying reasoning possibilities of the ontology language. However, one should not blindly rely on the offered reasoning mechanisms of a certain language, but analyze what reasoning operations are really needed when matching offer and request description. Therefore, rather the reasoning needs of the matching algorithms should influence the choice of the ontology language, than building a matcher around the more or less appropriate reasoning capabilities of the predetermined language.

With the state oriented and preference-containing descriptions from above, the matcher has to deal with configurable, declarative and sometimes fuzzy sets of instances (cf. [16]). It would be largely supported by the following reasoning service, which we call *best subset configuration*:

Given a declaratively defined, configurable set of instances o (the service offer) and a declaratively defined, fuzzy set of instances r (the service request). Configure o in such a way that it is the *best subset* of r , whereas 'best' is defined via the membership values of r .

Thus, the underlying ontology language should provide reasoning support for **sets** that on the one hand are configurable via **variables** and on the other hand are **fuzzy** due to integrated preferences. Existing ontology languages based on description logics or f-logic do not offer this support.

2.4 Specialized Ontology Language

As we can derive from the above, we have several requirements for an ontology language that is appropriate as a basis for semantic service descriptions:

- First, it has to cope with configurable sets. In contrast to the the concepts used to describe the lasting knowledge of the ontologies, sets are used locally, temporary and only within service descriptions. Their configuration must be linked to the service discovery process.
- Second, the ontology language should have a formal semantics allowing for reasoning possibilities such as presented in the previous section.
- Finally, the language should be intuitive to use by a broad community.

Many existing service description languages (DL) base their ontology description on description logics (see [2] for an overview). However, it is not suitable with regard to the above-mentioned requirements.

First, the concept of a set of instances is not included. Thus, sets have to be modelled as new static types, which clutters the common ontologies with personal and highly specific concepts definitions. As stated in [10], these *attributions* should be avoided. Second, the main reasoning service in DL is subsumption, which is similar to subset testing. However, dealing with configurable or fuzzy concepts is not possible. And finally, modelling with DL is rather difficult and unintuitive even for experts in the field.

An modelling concept that is widely used and well understood is object orientation. With its integrated concepts of classes, attributes, and instances as well as an explicit type hierarchy, it could be a good starting point for an ontology language as basis for an service description language. However, it has to be extended by the concept of **configurable sets** and algorithms that provide the desired reasoning services. In Section 4.2, we show means how this is achieved in our prototypical object oriented ontology language *DIANE elements*.

We will see that it becomes important whether an element of a configurable set can be created via its attributes or have to be chose from a pool of valid attribute combinations. This behavior is dependent on the type of the instances in the set. Thus, the ontology language has to **distinguish different types of classes**, such as *value-based* and *entity classes* in our DSD elements.

2.5 Shared Ontology or Mediation

When seeking for a reasonable, automatic processing of service descriptions, a unification of the used vocabulary with ontologies is vital. Two approaches are thinkable:

- Ontologies can be seen as conceptualizations of the real world which a group of users (the *community*) has agreed upon [9]. This agreement leads to the fact that entities speaking of the same situation use the same words, which enables a computer to compare these statements directly and in a semantically correct manner. However, this notion of ontologies does not lead to one huge world ontology, but of small disjunct

ontology modules for the different domains.

- Ontologies can be seen as conceptualizations of the real world that do not demand for a community agreement. In this case, the unification is done within a semi-automatical step of *ontology alignment* before processing descriptions or automatically with the help of mediators during the processing.

Both approaches are applicable. For reasons of simplicity, we present our prototypical language based on ontologies in the first notion, i.e. a community has agreed upon the used concepts.

3 COMPARISON TO OTHER APPROACHES

When looking for semantic service description languages that aim at automating service usage, beside others, two influential approaches can be found: OWL-S and WSMO. In this section, we analyze whether they fulfill the requirements from the previous section. A direct comparison of both languages can be found in [19].

3.1 OWL-S

OWL-S [1] is an ontology for describing web services based on the Web Ontology Language (OWL) [5]. Its goal is to develop tools and technology to enable automation of services on the semantic web. When it was first introduced in 2001 as DAML-S, it was based on three major ideas:

- *Ideas from the Semantic Web*. It separates common knowledge of the world from the concrete descriptions by describing this knowledge in a description logic based ontology.
- *Ideas from aspect oriented software engineering*. It separates three concerns of the description: what does the services do (service profile), how does it work (service model), and how is it accessed (service grounding)?

- *Ideas from the agent community.* It does not only describe the message flow, but also the state transition.

Although, the approach of OWL-S had a great impact on the community, it suffers from many problems with respect to the requirements from the previous section.

First, OWL-S is not purely state oriented, but describes a service by explaining inputs, outputs, preconditions, and effects separately. The interrelationship of these IOPEs is not specified in the service profile. The Semantic Web Rule Language (SWRL) [11] shall fill this gap, but first examples in OWL-S 1.1 are not very promising as they only seem to fit in domains that can be easily formalized.

Second, in OWL-S there are no additional features to specify request descriptions. A service request is expressed as a single instance of the perfectly fitting service. A generic, all-purpose matcher has to use a set of built-in heuristics to calculate the degree of matching between an offer and a request description, leading to unintuitive and unacceptable matching results.

Moreover, requests cannot be expressed for a single purpose (like booking a cinema ticket for *Spiderman 2* this evening), but have to be generalized for a multi-purpose service (like a service that can book a cinema ticket for a movie at a time that are given as inputs).

Third, with the description logic based OWL, OWL-S is based on a language that is rather unintuitive for modelling. Besides primitive types, OWL allows to define complex types with DL-specific type constructors. Reasoners allow to detect implicit subsumes-relationships among these types, which is used within the known matching algorithms. However, when modeling in practice, the complex types are often "misused" in order to emulate object oriented language (see e.g. [6]). Then, matching quickly degrades to a simple type checking as specialized reasoning support is missing.

Much effort has been put into the development of matchmakers for OWL-S:

- The *Semantic Matchmaker* of the Software Agent Group at CMU [26] operates on OWL-S descriptions. It tests if a request can provide

all required inputs and if the offer's output satisfies the requestor's demands. Such an *exact match* of the types is very uncommon, so additional matching degrees are introduced: **plugIn**, **subsumes** and **fail**, which allow a greater deviation from the original type. In any case, the approach relies on the message flow only. Moreover, in cases of a non-exact match, the offer cannot be used directly as the messages of request and offer are not compatible. In the *Mind Swap* project [29] and within the *DAML Agents for Service Discovery* [23], a similar approaches are pursued.

- The *LARKS-Matcher* is an extension of this approach [31, 30]. As in OWL-S, LARKS describes services by four functional parameters **input**, **output**, **inConstraints** and **outConstraints**. The algorithm uses up to five filters to match the description. The first three filters use techniques from information retrieval, the fourth filter is the Semantic Matchmaker described above, the fifth filter separately compares the pre- and postconditions. In summary, the approach suffers from the same problems as the Semantic Matchmaker as it directly compares the message descriptions, too.
- Many approaches rely on logical reasoning services, especially subsumption from description logics [8, 25, 21, 20]. However, they only consider the service's message flow, disregarding its state transition or model.
- In [4] a matcher is presented that also takes the information given in the service model into account. Basing on the Semantic Matchmaker described above, it recursively matches the operations in the request's model against the operations in the offer's model. However, this approach is very problematic as requestor and offerer have to specify nearly exact models to come to a match.

The problems with OWL-S become evident when performing use-case studies with it [28, 3]. Several further problems stemming from the missing ontological foundation of OWL-S are listed in [22].

3.2 WSMO

The *Web Service Modelling Ontology* (WSMO) [27] is an ontology and a conceptual model for the description of semantic web services. It was developed from the *Web Service Modelling Framework* [7] and is based on four top level notions: *Ontologies* provide a formally specified terminology of the information used by all other components, *goals* are objectives of a client when consulting a web service, *service descriptions* express the capabilities and the interface of a service, and *mediators* which handle heterogeneity mismatches between components that have to inter-operate.

In WSMO, the discovery of an appropriate service is seen as a three-step process [12]. The first step is the *goal discovery* where the human user has to formalize the needed functionality that he has in mind. The second step is the *web service discovery* which can be seen as filtering of the available web services. Here, the formalized service offer and request descriptions are matched and a possibly appropriate service provider is selected. In the third step, the *service discovery*, one concrete service that can be invoked through the interface of the web service is chosen and executed. At the moment, only the second step is supported by WSMO and some possible matchmaking approaches are sketched in [12].

WSMO is using a purely effect oriented approach by describing web services by the states that can be reached after a successful execution. However, WSMO omits the description of the information flow altogether, but describes the set of states as a new static complex type in description logics. The matcher uses subsumption to check whether the goals of the requestor are fully contained in the effects of a certain service offer. As it is not visible how to influence the chosen effect (this is left open for the third step, the service discovery, which has not been specified in WSMO up to now), finding a web service with WSMO does not necessarily mean that it produces a desired effect or can be invoked directly.

WSMO takes a step in the direction of differentiating between service offer and request description by introducing the concept of *goals*. Goals are representing a state of the world that a requestor tries

to achieve. By that, the requestor can specify goal-based requests, i.e. he specifies a requests without exactly knowing which offers are existing at the moment. However, the main reason for introducing goals has been to enable reuse. For a concrete service request, the user chooses one of the existing publicly available goals and specializes it for his needs. However, goals in a request are described in the same way as effects or postconditions in a service offer. Thus, a matcher is not able to involve any preferences of the requestor, but has to rely on generic subsumption mechanisms.

4 A PROTOTYPICAL LANGUAGE

The previous section has shown that the most prominent service description languages are not able to fulfill all the requirements identified in Section 2. Therefore, in this and the next section, we introduce the DIANE Elements (DE) and the DIANE Service Description (DSD) as prototypical languages. It has been stated in Section 2.4 that an ontology language is needed that is specialized for describing services. DIANE Elements is such a language. The DIANE elements are intuitive to use, as they are based on object-orientation. The concepts of attributes and their fillers are taken from F-Logic [13], the primitive data types are taken from XML Schema [33], the clean separation between schema and instances is related to the t-box and a-box concept of description logics [2]. Special constructs to describe service such as declarative and fuzzy set as well as variables are added. Furthermore, the language cleanly distinguished different types of classes.

DSD uses the specialized constructs provided by DE to describe services. It is thus able to meet all the requirements identified earlier.

The aim of these sections is to show which additional language constructs are needed to actually meet the requirements and how these constructs can be implemented. This could be used as a guideline for extending the existing approaches.

In this section, we will go through the general constructs of DE. On a first glance, most of them are not necessarily closely related to service descriptions

and could, indeed, be used in other contexts as well. However, they are ideally suited for service descriptions. This will become obvious in Section 5, where we introduce DSD and show how service offers and requests can be expressed using this language. To wrap up the presentation of the languages, Section 6 introduces the matching algorithm for service offers and requests formulated in DSD.

Throughout this paper, we use a graphical notation for the language constructs. Other representations, e.g. a formal one and a Java-based one do exist. More details can be found in [14].

4.1 Shared Ontology within a Community

As stated in Section 2.5, unification of the used vocabulary is essential for a Semantic Service Description. DE and DSD rely on the notion of a community. A *community* is a group of people that wants to cooperate in a certain area, e.g., by using services of each other in the area of service oriented computing. To do this in an automated manner, the community agrees upon a shared vocabulary. In DSD for example, class definitions are maintained by the community, i.e. new classes, changes at existing classes, or class deletions are discussed and decided within the community. However, in a large community, this approach does not scale. Hence, the work can be split up by domain so that community members only have to work on parts of the ontology that are within their expertise. The result is not a big world ontology, but several domain-specific ontology modules which can be combined when describing services.

4.2 Schemas and Instances

Primitive Types. The simplest types in DE are *primitive types*. These are predefined types that model commonly accepted issues like numbers, dates, boolean values, strings and so on. In DE, eight primitive types are supported: **Integer**, **Double**, **String**, **Boolean**, **Date**, **Time**, **DateTime**, and **Duration**. Their value space and lexicographical notation is derived from XML schema [33]. They are depicted as rectangles as in Figure 3.

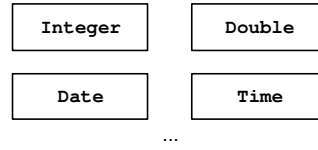


Figure 3: Some primitive types of DE.

For each primitive type, a total order relation and an equality relation are defined.

Classes and Properties. *Classes* are data types that are not predefined by DE. Formally, classes represent a collection of all individuals of a certain kind in the real world. For example, the class **Movie** represents all existing movies in reality. Classes are depicted by a rectangle. If these individuals have common attributes, these can be represented as *properties*. A property of a class has a name and a target type, which defines the instances or values by which this property can be filled. This instance/value is called *filling value* of the property. Properties are depicted by arrows at the rectangle pointing to the target type.

DE distinguishes three types of properties: *definitional* (depicted with a filled circle), *incidental*, and *orthogonal properties* (depicted with an unfilled circle) with the following semantics: If the filling values of all definitional attributes of an instance are defined, the filling values of the incidental properties are also defined.¹ The filling values of the orthogonal attributes are independent from the other filling values and can be assigned individually.

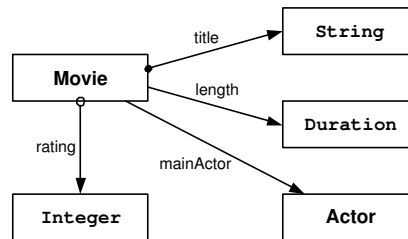


Figure 4: Example for class and attribute definition.

¹In the world of databases, the concept of a *key* is similar.

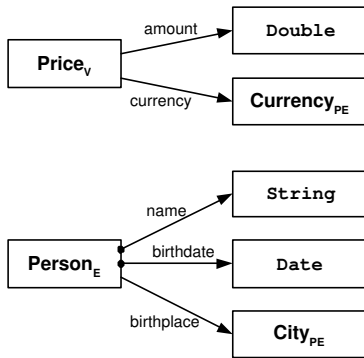


Figure 5: Value based and entity classes.

Figure 4 shows an example. Here, *Movie* has one defining property *name*, two incidental properties *length* and *mainActor*, which means that a given name of a movie defines its length and mainActor (assuming that no two movies have the same name), and one orthogonal property *rating*, which can be assigned individually and is not uniquely defined by the movie’s name.

DE also allows for single inheritance, which is depicted as in UML by a unfilled arrow pointing to the superclass. It is important to notice that the inheritance relationships are explicitly modelled and not derived by reasoning mechanisms.

Value Based and Entity Classes. We distinguish two kinds of classes: *value based* and *entity classes*. This is vital for service descriptions in order to be able to decide whether instances can be created without restrictions or have to be taken from a pool.

Classes where any combination of valid filling values leads to instances that represent existing or producible individuals in the real world, are called *value based classes*. Starting from a given instance of such a class, even a small change at one of its filling values leads to a new instance. An example for such a class is *Price* with *amount* and *currency*. Because of the orthogonality of the properties, it is not reasonable to mark them as definitional or incidental. Moreover, as there are no distinguished filling value combinations in such a class, no names are assigned

to the instances. Therefore, we call them *anonymous instances*. The identity of an anonymous instance is fully defined by its filling values. Often, domain specific comparison functions are defined for value based classes, e.g. a currency aware equality function for *Price*. Value based classes are marked with a subscript *V* (e.g., see *Price* in Figure 5).

In contrast, classes where only certain combinations of filling values lead to instances that represent individuals in the real world, are called *entity classes*. Each of these value combinations describes a discrete entity, which can be labelled with a unique name. Thus, we call instances of entity classes *named instances*. Examples for entity classes are *Person*, *Movie* or *Currency*. As small changes at the filling values of a named instance does not necessarily lead to another entity (think of the hair color of a person), the identity of named instances cannot be defined via their properties, but has to take the instance name into account. Entity classes are marked with a subscript *E* (e.g. see *Person* in Figure 5).

Named instances are published in the public instance pool². Because of the shared ontology, the community is responsible for maintaining this pool, i.e. new instances, instances changes or deletions are discussed and decided within the responsible part of the community. However, especially with service description, there are entity classes where it is not reasonable to publish all instances but keep them for private usage only (like *CreditCard* with *number* and *validity*). Note: These instances still have a (private) name and are not anonymous. However, they are not published and maintained in the public instance pool, but kept in a private one. Thus in service descriptions, only declarative statements over the properties of such an instance can be made. In DE, it has to be declared whether an entity class allows private instances or not. We distinguish:

- *Public entity classes.* In public entity classes each instance that is used is of common interest and has to be published under a unique name in the instance pool. Examples for such classes could be *Movie* or *Currency*. They are marked

²Typically, this pool will be partitioned and physically distributed.

with a subscript PE for public entity class (e.g. see `Currency` in Figure 5).

- *Partially public entity classes.* In partially public entity classes, it is allowed to assume the existence of private, unpublished instances. This has to be borne in mind when matching service descriptions with this kind of classes. The declaration of definitional and incidental properties is of particular importance with these classes as queries often can only be done declaratively over their filling values. An examples for such a class could be `Person` or `CreditCard`. They are not marked with a special subscript other than E for entity (e.g. see `Person` in Figure 5).

4.3 Instance Sets

Sets of instances are a fundamental and novel concept of DE. They are used in situations when it is not possible or reasonable to specify a single instance by its name or its filling values. Especially in combination with variables (see next section), which leads to *configurable sets*, or non-sharp conditions, which leads to fuzzy sets, they are essential in service descriptions:

- *In service offer descriptions.* Often, the service cannot achieve one effect only, but is able to achieve a family of effects, which can be selected through one or more parameters. Thus, the service offer description cannot be a single instance, but has to use configurable sets of instances to represent this fact.
- *In service request descriptions.* As seen above, the service requestor wants to use a certain functionality, which can be achieved by different services. Therefore, it is necessary to describe a request as a set of suitable services. The preferences among the elements can be represented by a fuzzy degree of set membership.

Crisp Declarative Sets. Sets are in between classes and single instances: classes represent *all* individuals of a certain type, instances represent *exactly one* individual, whereas instance sets contain any collection of instances, which can be zero, one, several

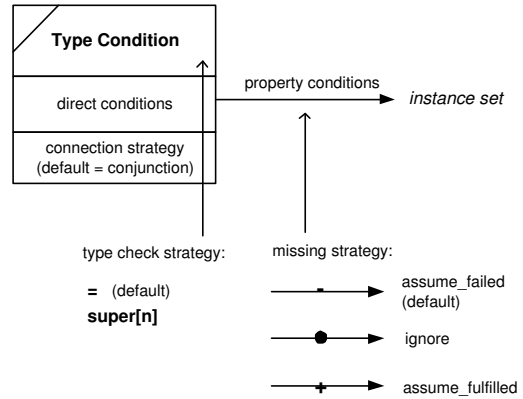


Figure 6: Building blocks of a set.

or all instances of a given type. Sets are depicted by a rectangle with a small cross line in the left upper corner.

The definition of a set is done declaratively, i.e. by specifying a function which decides whether a given instance is in the set or not. In the DE, the space of possible membership functions has deliberately been structured and limited by providing a number of constructors which have to be used to build up sets. This restriction ensures that no undecidable problems are encountered during the matching phase.

We distinguish between two kinds of constructors: conditions and strategies. The following three *conditions* are possible in a definition (see Figure 6):

- A *type condition* for type t . All instances in the set have to be of this type t (or a subtype of it). This restriction can be weakened towards supertypes by a different type check strategy (see below). The type condition is written as name of t in the rectangular. We also say, the set is of type t .
- A list of *direct conditions*. Direct conditions are conditions that are pointed to the potential members of the set directly. Each instance in the set has to obey all these conditions. If the type of the set is a primitive type, all typical comparison operations for this type (like `==` or `<=`) are allowed; if the type is an entity class, only direct

comparisons to named objects of this type are allowed (like ‘== harryPotter3’). Direct conditions are not reasonable for sets with a value-based class as type.

- In case of a type as class with attributes: a list of *property conditions* which are depicted as named arrows at the rectangular. Only properties of the corresponding type (or its super types) can be used as property conditions. Each of these property conditions p points to another instance set y and leads to the following restriction for the members of the set x : An instance can only be member of set x if its property p is filled and the filling value is member of y . By default, all property conditions of the set are connected conjunctively. This can be changed by a different connecting strategy (see below).

Figure 7 shows examples for set definitions. On the right side, a set of actors is defined. Due to its direct condition, it only contains one element: danielRadcliffe. The set of durations contains all duration values below 180 minutes because of the direct condition. Thus, on the left side, the set contains only movies that have Daniel Radcliffe as main actor and are shorter than 3 hours.

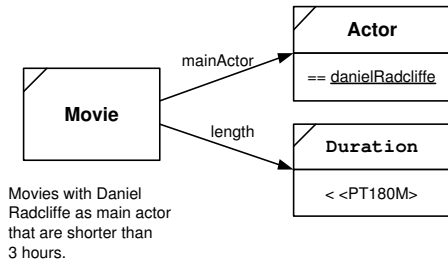


Figure 7: Examples for sets defined by conditions.

The default behavior of a set can be changed by specifying different *strategies* (once more, see Figure 6):

- The *type check strategy* (tcs) weakens the type the instances in the set have to have. By default, only objects of the given type t (and subtypes of

it) are allowed in the set. This is expressed by the tcs ‘=’. If also supertypes of t are allowed, the tcs **super**[n] can be used where the parameter n defines the maximum distance in the ontology. For example, **super**[1] means that objects of type t and t ’s father are allowed (as well as all their subtypes, i.e. t ’s siblings are allowed, too).

- The *connection strategy* (cs) changes the way the single results of the different property conditions are connected. By default, they are connected conjunctively. The cs is specified as a Boolean expression over the property names where the operations **and** and **or** are allowed.
- The *missing strategy* (ms) specifies the behavior in case of a missing filling value of an instance that is tested for set membership. If for example a property condition for a set of type **Movie** specifies that the movie’s length has to be below 3 hours, but the current instance has no filling value for this property, the missing strategy decides how to proceed with this instance: By default, the missing strategy is *assume_failed*, which means that the condition should be regarded as if it had failed. This strategy is depicted by a minus on the property condition arrow (or left blank as it is the default). More strategies are *ignore* depicted by a circle, which skips the condition if it is not defined in the object, and *assume_fulfilled* depicted by a plus, which means that the condition should be regarded as if it had succeeded.

Figure 8 shows an example for a set definition with strategies.

Fuzzy Declarative Sets. Up to now, sets have been defined as *crisp sets*, i.e. each instance is totally within or not within the set. When expressing request description, this is not sufficient as the requestor wants to express her preferences among the elements by different membership degrees. Therefore, DE was extended by the concept of *fuzzy sets*. Fuzzy sets can be defined by a continuous membership function, which assigns a value from the interval $[0, 1]$ to each instance. 0 denotes that the instance is

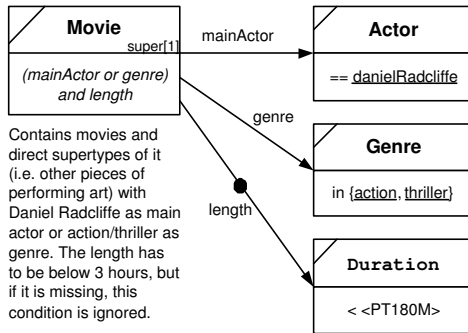


Figure 8: Examples for a set definition with strategies.

no element of the set, whereas values > 0 stand for a gradual membership. Crisp sets are a special case of fuzzy sets as they only use the membership values 0 and 1.

DE offers the following additional elements to define fuzzy declarative sets:

- *Fuzzy direct conditions.* Additional comparison operators are introduced. They do not compare two values in a binary manner, but are somewhat tolerant. With the new operators $\sim==$, $\sim<=$ etc., deviations up to 10% are allowed by default, which leads to linearly decreasing membership values. However, other deviation limits can be given, too.
- *Fuzzy type check strategies.* An additional parameter f can be used at the tcs: `super[n, f]`. While n defines the maximum distance of inheritance relationships in the ontology, $f \in [0, 1]$ defines the penalty factor for each of these steps. Thus, if with `super[2, 0.5]` an instance stems from a supersuperclass, its membership function is multiplied with $0.5^2 = 0.25$.
- *Fuzzy connecting strategies.* As the single results of the property conditions of fuzzy sets need not be boolean values but can be numbers from $[0, 1]$, also the connection procedure has to be adapted. Thus, as a default, **and** is calculated as multiplication \cdot , **or** as modified sum \oplus ³. Addition-

³with $a \oplus b = a + b - ab$, for $a, b \in [0, 1]$

ally, the following connectors are available: `min`, `max`, weighted sum `ws`, and amplification `exp`.

- *Fuzzy missing strategies.* An additional generic missing strategy can be used: `assume_value[n]`. In case of a missing filling value, the given value $n \in [0, 1]$ is assumed as fulfillment value.

4.4 Variables

Variables are a special kind of instance sets. They are needed within offer and request descriptions where it is not possible to specify a concrete value or instance at the time of service description. In contrast to sets, this value has to be bound during the service usage, i.e. before or after the service execution by the service requestor or provider. In any case, the bound value has to be taken from the underlying set (the so called *basic set*). Such a *filled* variable is regarded as singleton set, which contains exactly this filled value. Thus, variables can be used to build configurable sets.

In contrast to schema, instance, and set, the semantic of variables is only defined within a service description, i.e. variables are the first service description oriented constructs. In DE, variables are divided into categories depending by whom they have to be filled during the service usage process. We distinguish two types: IN and OUT variables:

- *IN variables* have to be filled by the service requestor during the service usage. IN variables in the request (so called ReqIN variables) are reasonable if the service requestor wants to execute the request several times in different parametrization. Before sending the request, all ReqIN variables have to be filled. IN variables in the offer (so called OffIN variables) have to be filled in by the service requestor to configure the offered service before it can be invoked.
- *OUT variables* have to be filled by the service provider during the service usage. OUT variables in the request (so called ReqOUT variables) represent information which the requestor is interested in. OUT variables in the offer (so called OffOUT variables) stand for information that the

service provider will announce after the service has successfully executed.

Variables are depicted like sets, but have a light grey background. Moreover, in the left upper corner, the category of the variable is denoted.

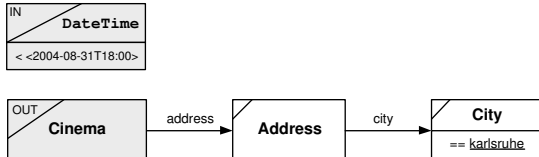


Figure 9: Example for variables in DE.

Within an offer descriptions, a variable v with the basic set s has the following semantics:

- As long as v is *unbound*, v functions like its basic set s , i.e. it only is assured that after service execution a concrete value from s is chosen from the set.
- Once v is *bound* with the value x (i.e. directly before service execution with OffIN variables, and directly after service execution with OffOUT variables), v functions like a singleton set that only contains that element x .

An example for this is given in Figure 10. It deals with a CinemaTicket that is partly undefined yet. However, it is assured that its price will be below 8 Euros. Because of the IN variable, the genre of the movie which the ticket is valid for can be chosen by the service requestor, thus leading to a configurable Movie set. After a successful service execution, the Ticket is fully specified and because of the OUT variable, the amount of its price is returned to the requestor.

5 DIANE SERVICE DESCRIPTION

In this section, the DIANE Service Description (DSD) is presented. DSD is described by the DIANE elements. First, we analyze the general goals of service descriptions, than we explain about their general

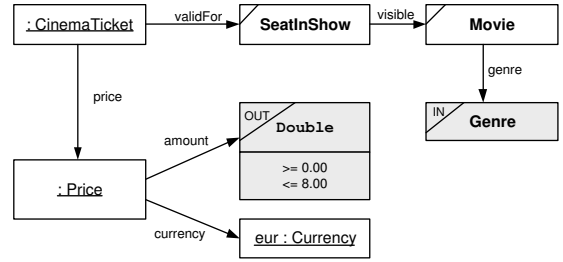


Figure 10: Example for configurable sets.

structure in DSD and the process of their creation. Finally, the concepts are illustrated by two detailed examples.

5.1 Goals of Service Descriptions

As stated in Section 2.2, the goal of a service description is different for offer and request descriptions:

5.1.1 Service offer descriptions

An offer description expresses what effect can be generated by the service. Typically, this description can be split in two parts: the description of the individual that is processed by the service and the description of the new state, which is achieved for this individual during service execution. Examples for that could be: a cinema ticket is owned, a document is printed, a pdf document is made available, the value of a stock is made known, etc.

In general, offered services do not only provide one single effect, but can deal with several individuals and transfer them in several similar states. As a result, a service offer description has to capture a *family of effects*. Thus, sets have to be included into the description. By default, only one of these effects is actually achieved when executing the service. By integrating IN_x variables into the description, the service requestor gets the possibility to choose or at least restrict this actual effect. On the other hand, OUT variables help the service requestor to obtain information about the actual effect, which could be necessary for a proper postprocessing.

5.1.2 Service request description

A request description has to capture what effects the requestor is interested in. Typically, he is interested in getting a certain task done and tolerates a set of effects with different preferences. Thus in requests, fuzzy sets are used to capture the possible effects, where the membership value to the set denotes the preference with this instance.

Also variables have another purpose in request descriptions. ReqIN variables help to reuse the query in various similar situations. ReqOUT variables enable the service requestor to specify what information he needs about the effect.

Such a request description is used to find an appropriate service offer with an effect (consisting of a processed entity and its state change) that can be configured using the OffIN variables to obtain an effect that is highly preferred by the requestor. Moreover, after the service execution, the service requestor has to be aware of all information he specified in the ReqOUT variables.

5.2 Process of Creating Service Descriptions

As a service description is essentially an instance of the class `Service`, the basic technique for creating service descriptions is instantiation. However, for a subsequent matching process it is vital that the descriptions have a relatively structured organization. To achieve this it is reasonable to construct descriptions within a well defined process. This process is attained by continuously layering and instantiating schema definitions (see [18]). We distinguish three layers: the upper service ontology, category ontologies, and domain ontologies (see Figure 11). These are presented in the following subsections.

5.2.1 Upper Service Ontology

The upper ontology is a schema that defines the basic structure of a service description (see Figure 12). The schema is rather simple and adopts the ideas from OWL-S to describe the different aspects of the service separately: in the `ServiceProfile`, an abstract black

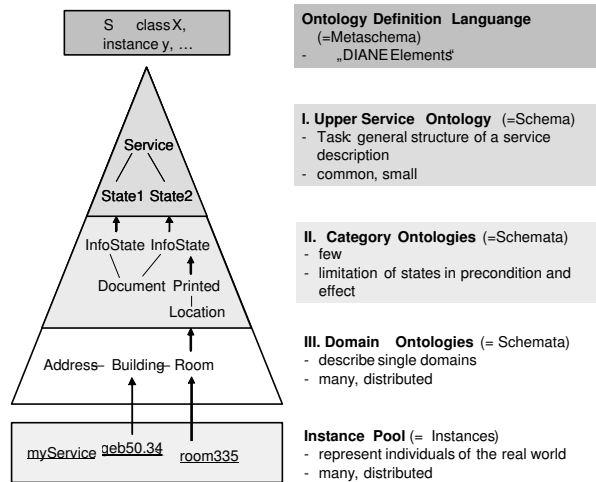


Figure 11: The Ontology Pyramid

box description of the service’s functionality is given; in the `ServiceGrounding`, the connection between this abstract description and the real service is defined. Additionally, the attribute `providedBy` captures the service provider’s address.

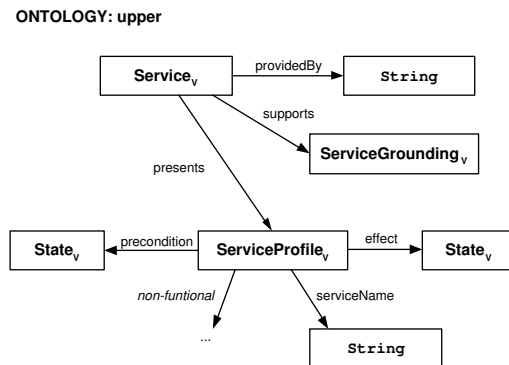


Figure 12: The Upper Service Ontology

The `ServiceProfile` implements the principle of a purely state oriented service description by allowing for two functional attributes, only :

- **precondition.** Describes the state of the world which is needed for a correct service execution.

If the service is called although the world is not in this state, the result will be undefined.

- **effect.** Describes the state of the world after a *successful* service execution, i.e. error cases are explicitly not included into the effect description as they typically do not provide any useful information during service search.

One important class of the profile is **State**, which stands for states in in the real world. The class can be regarded as abstract as it has no own instances. Rather, it has a set of concrete subclasses that are defined within the category ontologies (see next section). Note that such a **State** never fully describes *all* aspects of the real world. In fact, it only stands for the interesting part of the world, while the rest is assumed to be undefined and thus to be unimportant.

Besides the functional attributes, the profile also allows to describe non-functional aspects. However, as they are not in the main focus of DSD, their details are omitted here.

5.2.2 Category Ontologies

The second layer of ontologies for service descriptions are category ontologies. They divide the space of services into clusters with similar state transitions. Thus, the most important task of a category ontology is to concretize the abstract class **State**. In DSD, four service categories are distinguished:

- *Knowledge services* are used to obtain information on an individual. Thus, the state **Known** is introduced, which expresses that additional information about the filling values of an instance is gained. The service achieved this information gain via OUT_x variables that are filled after service execution. Knowledge services are only interesting for entity classes as the filling values of their attributes are dependent. Typical example for such a service could be a movie information service that returns the duration of a movie after entering its title.
- *Information services* are used to change the state of the information system. In general, this state

is defined by the data such as files and database entries that are stored in the system. Thus, the state **LocallyAvailable** is introduced, which expresses that a piece of information is accessible and can be used by the system. The service achieves such an effect by making use of an external exchange protocol for files (such as FTP) or database entries (such as JDBC). Besides **LocallyAvailable**, other states for pieces of information are imaginable.

- *Real object services* are used to change the state of an object in the real world. The most important state (especially when thinking of e-commerce services) is **Owned**, which expresses that a thing is created or acquired in order to get a new possessor. Such an effect is achieved outside the service framework, e.g., by triggering real world actions via messages to persons or usage of external devices such as a printer.
- *Capability services* are service that enable the service requestor for a certain capability. Thus, the state **Enabled** is introduced, which has an attribute **entity** pointing to a **Capability**. Typically, the duration of the state limited. An example for such a service could be an internet access service, which allows the service requestor to send and receive TCP/IP packets within a period of time.

5.2.3 Domain Ontologies

The third layer are domain ontologies. They conceptualize a certain application domain such as books, locations, files etc. They are used to concretize the attributes of the state change, especially to fill the entity which the state refers to. In contrast to the few and rather small ontologies on the upper two levels, there are many and also large domain ontologies. They are not provided by DSD, but should be brought in by experts in the community. Thus, they are typically distributed among the users. Domain ontologies consist of classes, which structure the area semantically, and instances, which represent real individual from the area. Moreover, domain specific comparison functions can be introduced, if they are

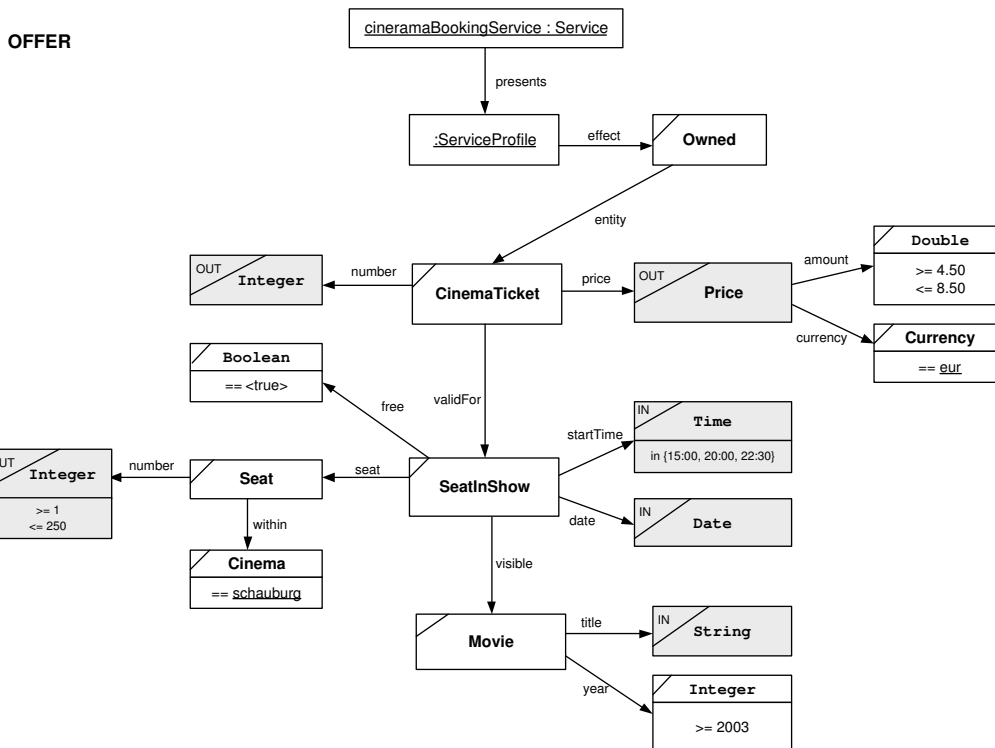


Figure 13: Example for a description of an **offered** service.

of general interest. Very important are the domain ontologies that describe general value-unit measures (such as weight measures, length measures etc.), files, persons, and locations.

5.3 Examples for Service Descriptions

Figure 13 gives an example description of an offered service. Here, a service for reserving a seat in a cinema is given. The description starts with **Service** and **ServiceProfile** instances from the upper ontology. Then it is expressed that the service operates in the real world (thus being a real object services): after a successful service execution a cinema ticket is owned by the service requestor. The details of this ticket are described by using domain ontologies from the domains movie, cinema, money etc. The cinema ticket is valid for a certain seat in a show. By binding the **IN** variables, the requestor can choose the starting

time, the date, and the title of the movie he wants to see. However, in any case, the show will be located in the **schauburg** cinema and show a new movie (year ≥ 2003). When the seat is reserved, the server provides the details about the number of the booked seat and the reservation number of the ticket. Also the exact price (between 4.50 and 8.50 Euros) will be announced.

In Figure 14, an example of a request description is given. Here, a service requestor is interested in going to the cinema. Thus, a set of suitable effects (here of owning a cinema ticket) is specified. The requestor is interested in a price of 8 Euros or below, however, he also accepts prices that are up to 10% higher (with a decreasing preference value), which is expressed by the fuzzy operator $\sim \leq$. The date of the show should be at 2004-12-28 or on the 27th or 29th with a lower preference value; the time should be between 19:00

REQUEST

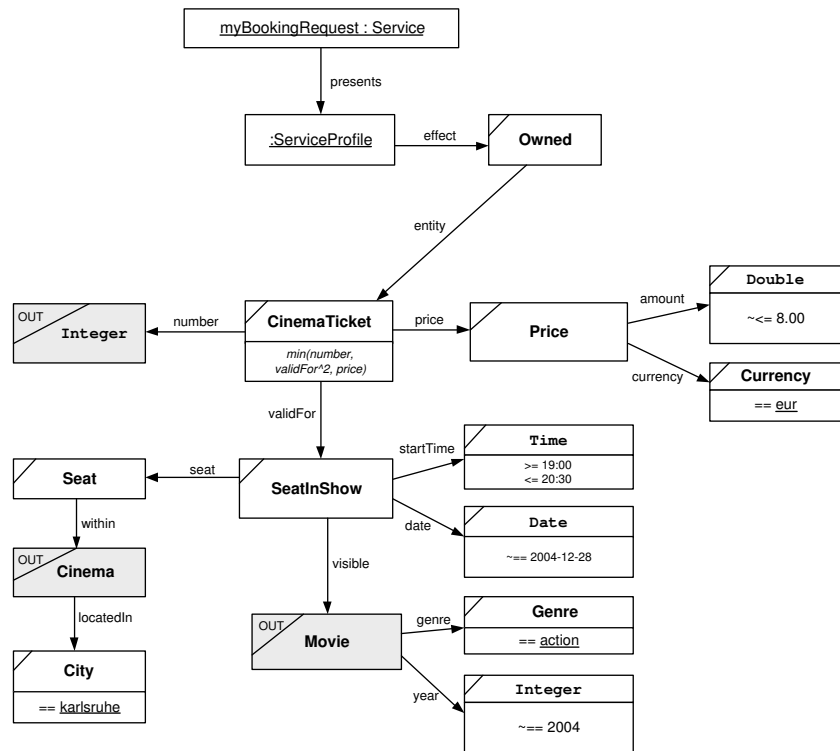


Figure 14: Example for a description of a **requested** service.

and 20:30. The movie should be classified as action movie and preferably from 2004. The exact cinema does not matter as long as it is in Karlsruhe. The conditions in cinema ticket are combined conjunctively which is expressed by the `min` strategy. However, the show is more important than the price, thus `validFor` is emphasized by the exponent 2. After service execution, the service requestor wants to know his reservation number, the cinema where to go, and the exact movie he will see, which is expressed by the OUT variables.

Note that the interfaces and the functionality of the offered and requested service are somewhat different. However, by describing them on the same ontological basis and by integrating the information flow and the preferences into the descriptions, our matcher is able to detect a match between them. It can also derive an optimal variable binding. The matcher is described

in the next section.

6 MATCHMAKING

The matchmaking process has two basic tasks. On the one hand, it has to determine how well a service offer is able to fulfill the needs of a given service request, i.e., it has to compute the matching value between offers and the given request description. On the other hand, it has to enable invocation of the chosen service, i.e., it has to assign values to unbound variables.

Remember that a service offer is basically described by a parameterizable set of achievable effects, while a service request is described by a fuzzy set of desired effects. Intuitively, a service offer matches a request, if the offer set can be parameterized in such a way

that it is a subset of the request set. In such a case, any effect that is achieved by the service provider is useful for the requestor.

In order to express this task more exactly, we formalize the semantics of sets, variables and the corresponding reasoning operation in the following. Let us assume \mathcal{S} is the set of all variable-free, sharp sets that can be defined with the constructors of DE. $\tilde{\mathcal{S}}$ is the set of all variable-free, fuzzy sets definable by DE. Of course, we have $\mathcal{S} \subset \tilde{\mathcal{S}}$. For each set $s \in \tilde{\mathcal{S}}$, its characteristic function

$$\chi_s : I \longrightarrow [0, 1]$$

can be defined which assigns a membership value to any instance i from the set of instances I . I contains all primitive values, named and anonymous instances. These definitions allow us to define the subset relationship between a crisp set $s_1 \in \mathcal{S}$ and a fuzzy set $s_2 \in \tilde{\mathcal{S}}$:

$$\text{subset}(s_1, s_2) = \min_{i \in s_1} \chi_{s_2}(i)$$

i.e. **subset** is defined by the element i that fits worst s_2 and is also in s_1 .

Typically, sets in offer descriptions are not variable-free, but contain IN variables at various places. Thus, \mathcal{S}^n shall contain all sets that are configurable via n IN variables. Such a set $s \in \mathcal{S}^n$ can be configured by an instance tuple $j \in I^n$. The configured set is written as $s[j]$ and is again in \mathcal{S} .

These definition help us to clarify the task of the matcher. Let $o \in \mathcal{S}^n$ be the configurable effect set of the offer description containing n IN variables and $r \in \tilde{\mathcal{S}}$ be the fuzzy effect set of the request description. Then the matching value mv is determined by that variable configuration for o which leads to the highest **subset** value:

$$mv = \max_{j \in I^n} \text{subset}(o[j], r)$$

Or with the definition of **subset** from above:

$$mv = \max_{j \in I^n} \min_{i \in o[j]} \chi_r(i)$$

The optimal IN variable configuration j_{opt} can be determined by using the argmax function.

These formulae also present a theoretical approach how to calculate the match by using a nested loop: The outer loop enumerates all valid variable configurations j , while the inner loop enumerates all elements of the set that has been configured with j . For each element, the membership value for r is calculated. However, in practice, this approach is not feasible as the search space can become extremely large or even infinite. Thus, an explicit enumeration of the instances is not manageable. In most cases, it can be replaced by a symbolic calculation of the **subset** operation using the declarative description elements of the sets. This gets possible as the constructs of DE have especially been developed to possess properties such as orthogonality and monotony that allow for a stepwise, recursive calculation.

We will present the matching algorithm in three steps: First, in Section 6.1, we explain how the reasoning operation **subset** can be efficiently computed on a symbolic level. Second, in Section 6.2, we show how the IN variables in the offer can be bound to obtain a variable-free description that can be used with the efficient **subset** implementation from the first step to calculate the matching value. Finally, in Section 6.3, we explain how the OUT variables of the request can be bound.

6.1 The reasoning operation: subset

Most often, the reasoning operation $\text{subset}(s_1, s_2)$ compares two IN variable-free sets s_1 and s_2 of similar ontological types which have a similar structure. Therefore, an obvious basic technique to efficiently implement this operation is by matching the description graphs symbolically. Beginning with the root element, the two descriptions are traversed synchronously and compared step by step. The graph matching is driven by the request set s_2 , which means that the structure of s_2 's graph determines the order of the matching process because parts of the offer set s_1 's graph that have no corresponding part in s_2 need not be matched.

The matching is done according to the type of s_1 and s_2 :

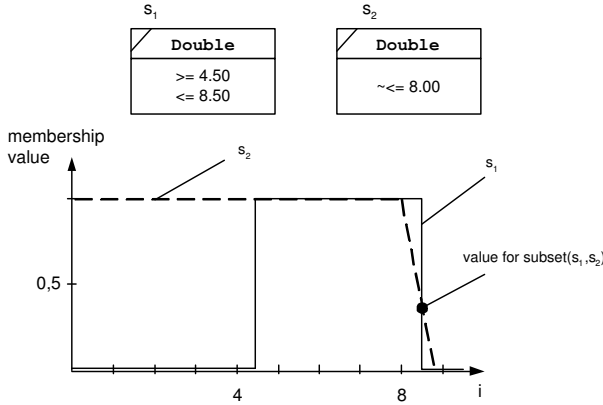


Figure 15: Calculating `subset` for primitive types.

Primitive Type. The calculation of $\text{subset}(s_1, s_2)$ for sets of primitive types is done by creating straight line representations for the membership functions of s_1 and s_2 . Then the smallest value for χ_{s_2} that lies within χ_{s_1} can be easily determined as it lies on one of the intersections or corner points of s_2 's line representation.

In Figure 15, two sets of primitive types have been taken from the offer and request example. In s_2 , the requestor prefers a price that is 8.0 or less, but he also accepts a price which is up to 10% higher with a lowered matching value. In s_1 , the offerer states that the price will be between 4.50 and 8.50. The worst fitting element that is also in s_1 can be found on the marked intersection of the two functions. Thus, in this example $\text{subset}(s_1, s_2) = 0.375$.

Value Based Class. The calculation of $\text{subset}(s_1, s_2)$ for sets of value based classes can be done by recursively calculating `subset` for the attribute conditions of s_1 and s_2 . This becomes possible as the attributes of value based classes are orthogonal by definition, thus the attributes can be matched independently. Moreover, the connecting strategy is a monotonically increasing function in each attribute, thus the minimum can be determined by minimizing each attribute separately.

In the example in Figure 16, `Price` is a value based

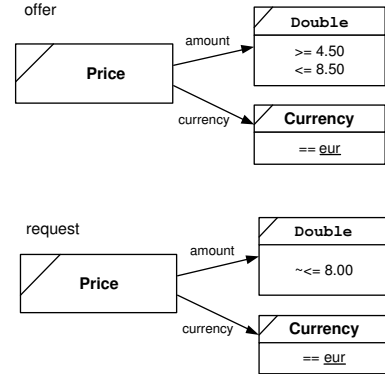


Figure 16: Example for sets of value based classes.

class with the two attributes `amount` and `currency`. Both attributes are matched separately which leads to a subset value of 0.375 for the attribute `amount` and to a subset value of 1.0 for the attribute `currency` since the two `Currency` sets are identical. As no individual connecting strategy is defined, the matcher uses the default connecting strategy and multiplies the two values to get a resulting subset value of 0.375 for the two `Price` sets.

Public Entity Class. The calculation of $\text{subset}(s_1, s_2)$ for sets of entity classes leads to the problem that the attributes are not orthogonal anymore. However, since in the case of public entity classes all instances have been published, it is possible to use the enumeration approach from the definition: $\text{subset}(s_1, s_2) = \min_{i \in s_1} \chi_{s_2}(i)$. Therefore, all elements within s_1 are listed. For each, the membership value according to s_2 is calculated. The value of the `subset` operation is the smallest of these membership values. The operation can be accelerated by using an index over the instances in the pool, which helps to efficiently enumerate the elements within s_1 .

Partially Public Entity Class. If s_1 and s_2 are sets of partially public entity classes, the calculation of `subset` is more difficult. On the one hand, as with every entity class, the attributes are not orthogonal, on the other hand, the instances are not or not com-

pletely available in a public instance pool. Thus, the value of the subset operation can only be estimated by giving a lower bound for it. This lower bound is still useful within a matching of service request and offer descriptions as it determines the matching value in the worst case. The estimation is calculated by assuming the orthogonality of the attributes, i.e. `subset` is calculated by recursively applying `subset` for the attributes and combining them according to the connecting strategy. The estimation could be improved by adding information about the interdependencies of the attributes.

6.2 Calculating a match by using subset

Typically, offer descriptions contain IN variables, so `subset` cannot be used directly to calculate the matching value between an offer description o and a request description r . Therefore, in a preliminary step, o has to be made IN variable-free by finding and binding these variables to their optimal values. To do this correctly, it is necessary to pay attention to the dependencies between them. Thus, the notion of a *variable context* is introduced. Each context contains variables that have to be processed together, whereas the contexts themselves can be optimized independently.

As a rule, two IN variables are not independent, if they are used within property conditions of the same set of entity class type e . If e is a *public* entity class, the interdependencies are completely known via the published instances and therefore can be taken into account by the matcher. In such a case, the IN variables are grouped together in a common context. If e is a *partially public* entity class, the interdependencies are not fully known and cannot be taken into account by the matcher. Thus, the matcher assumes independency here, but instructs to compute *all* possible variable bindings for this subtree in order to be able to retry the service execution if the chosen variable binding is not accepted by the service provider.

Thus, within a concrete offer description, the context of an IN variable can be determined in the following manner:

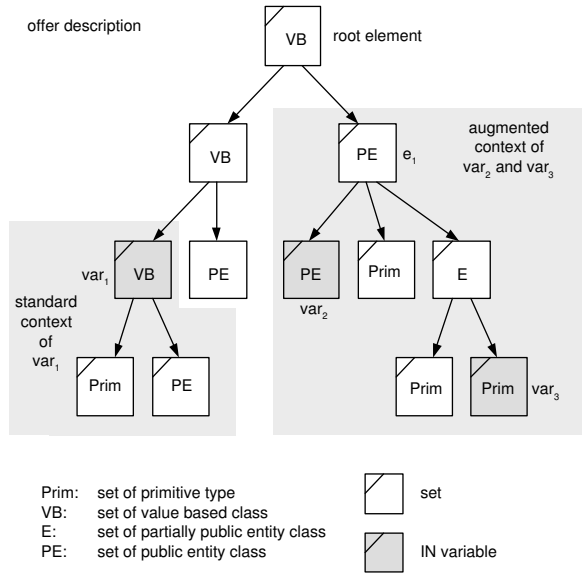


Figure 17: Example for the context of variables

- The *standard context* of an IN variable comprises the variable itself as well as all sets that are directly or indirectly reachable via attribute conditions. A standard context always contains exactly one IN variable.
- The context has to be extended to an *augmented context*, if a set of a public entity class can be found on the path between the root element of the offer description and the IN variable. Then, the context comprises the public entity class e that occurred first on this path as well as all sets that are directly or indirectly reachable from e via attribute conditions. Note that such an augmented context can contain more than one IN variable.

An example for the contexts of IN variables is given in Figure 17. Here, a generic graph of an offer description is shown, consisting of sets (white boxes) and IN variables (gray boxes). These have different types such as primitive types (Prim), value based classes (VB) as well as partially public (E) and public entity classes (PE). In the example, the IN variable var_1 has no public entity class on its path from the root

element, so the standard context is chosen comprising var_1 itself and its successors. In contrast, the IN variables var_2 and var_3 are in the same augmented context as both have e_1 as first public entity class on their paths from the root element.

By definition, the optimal variable binding and also the matching value can be determined separately for each context. The calculation depends on whether we have a standard or an augmented context.

Standard Context. In a standard context, the optimization of the sole IN variable can be done by using `max-subset`, which is defined analogously to the standard `subset` as $\text{max-subset}(s_1, s_2) = \max_{i \in s_1} \chi_{s_2}(i)$. The reason for this lies in the fact that the requestor can select the binding value for the IN variable by himself and is not reliant on the service provider’s choice. As the remaining context is variable-free, the recursive calculation procedure for the different types in the context is analogous to minimizing `subset` from above.

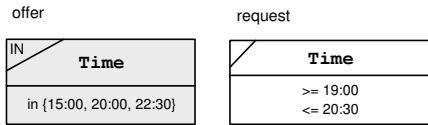


Figure 18:

As an example, we take the `OffIN` variable of type `Time` from our cinema example (see Figure 18), which is optimized within its standard context. It is maximized against the corresponding `Time` set in the offer, which provides 20:00 as optimal variable binding and 1.0 as matching value.

Augmented Context. In an augmented context, all contained IN variables have to be optimized together as they are dependent on each other. The interdependency arises as the variables together are used to indirectly choose one element from the root set of the context c_o . For example, a movie could be uniquely selected via its title or a seat via its row and number. However, in some cases, the IN variables only allow to narrow down the possible elements in

the root set, leaving the final choice to the service provider, e.g. a movie could only be vaguely specified via its genre and year. Thus, the optimal IN variable binding for the context is the binding $j \in I^n$ that leads to the configured context $c_o[j]$ which has the highest value for $\text{subset}(c_o[j], c_r)$, where c_r is the corresponding context in the request.

An efficient implementation for this is the following: while checking each published instance i for set membership in c_o , the corresponding value combination j that can be found at the variables is stored. Only these stored value combination j have to be considered in the outer loop when looking for the optimal variable binding j_{opt} .

In the example in Figure 19, an augmented context starting with a `Movie` set is shown. When the matcher iterates through the movie instances, only the movies from 2003 and later are within in the offer set. Their titles are captured and stored at the `String` variable. For each of these titles, `subset` is calculated between the configured offer set and the request set.⁴ The highest value is the matching value and determines the optimal binding for the IN variable.

As stated above, the presence of partially public entity classes changes the behavior of the matcher as it has to compute *all* possible IN variable bindings and matching results to be sure to get the service executed with one of the configurations. Thus, if there is a partially public entity class on the path between

⁴Here, the configured offer sets only have one element case because `title` is a definitional attribute of `Movie`.

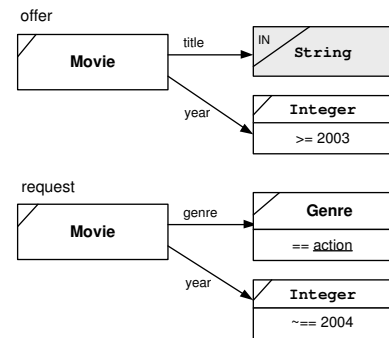


Figure 19: Example for an augmented context.

the root element of the offer description and the root element of a context, the matcher computes all IN variable bindings for this context that have a matching value that is substantially larger than 0.

In our cinema example, `SeatInShow` is a partially public entity class. It changes the matching behavior of the succeeding contexts: for the `Time`, `Date` and `title` variable of type `String`. In case of the `Date` variable, three variable binding are returned: 2004-12-28 with a matching value of 1.0 as well as 2004-12-27 and 2004-12-29 with matching values of 0.5. Also the augmented context for the `title` variable has to provide all variable bindings, which would be several movie titles with different matching values depending on their year, such as "Spiderman 2" with a matching value 1.0 and "Terminator 3" with a matching value 0.8. Altogether, all combinations of these variable bindings are returned.

Completing the match. After all contexts have been analyzed, each of them has been assigned one (or several) variable binding(s) and the corresponding matching value(s). The overall matching value can be determined by replacing all contexts by special nodes representing the optimal matching value(s) of this context. The offer is matched against the request description using the standard `subset` implementation from Section 6.1. When reaching such a special node, its assigned matching value is used instead of recalculating it again. If a context has several matching results for different variable bindings, the matcher also returns several matching results for the complete offer description, depending on the variable binding. If the service is not executable for the best binding, the second best can be tried and so on.

Thus, after these steps, the optimal binding of the IN variables and the corresponding matching value have been calculated by the matcher.

6.3 Filling variables in the request

To finish the matching process, the OUT variables of the request have to be bound. These ReqOUT variables represent additional requirements of the service requestor. He is only interested in a certain service offer when the desired information specified by the

ReqOUT variables is known after a successful service execution. Thus, a service with a high matching value cannot be used, if it does not fulfill the request's information demands.

Each ReqOUT variable r is bound by comparing it against the corresponding set o in the configured offer (i.e. all OffIN variables are bound). We can differentiate three cases:

- o lies within a standard context of an OffIN variable. Then the binding value for r is unique and can be derived from the binding value of the context's OffIN variable.
- o lies within a standard context of an OffOUT variable. Then the binding value for r will be unique after service execution, when the context's OffOUT variable is filled.
- o does not lie within a standard context. Then, the matcher has to determine whether o has exactly one element. This can be done by analyzing the definitional attributes. If yes, r is bound to this element, if not, r cannot be bound and the offer description is discarded.

In our cinema example, three ReqOUT variables have to be bound. The ReqOUT variable for the ticket number lies within the standard context of the corresponding OffOUT variable and can be filled after service execution. The `Movie` and `Cinema` variables do not lie within a standard context. However, the corresponding sets in the offer consist of exactly one element: schauburg for the `Cinema` and the instance of the chosen `Movie` (title is a definitional attribute). Thus, the information demands of the requestor as well as one of his desired effects can be fulfilled by this offer.

To sum up, we showed a matcher that performs its task in three steps: configuring the offer by binding its OffIN variables, calculating the matching value by using the specialized reasoning mechanism `subset`, and providing the requested information by binding the ReqOUT variables. By using algorithms that rely on a symbolic processing of the descriptions, an efficient implementation was gained.

7 SUMMARY AND OUTLOOK

An appropriate service description language is the major prerequisite for automatic dynamic service binding. In this paper, we have first motivated the need for this. We have introduced a number of scenarios that make it obvious why automatic dynamic service binding is desirable. We have then identified requirements towards a service description language that supports this. These are in particular: the ability to unambiguously describe service offers, special constructs to adequately express service requests, a reasoning support that is specialized on service discovery, and an ontology language that contains special elements for the peculiarities of service descriptions. We have shown that existing approaches to semantic service descriptions currently do not fulfill these requirements. We have then introduced a number of language constructs realized in the prototypical language DIANE Elements, the DIANE Service Description (DSD) which uses DIANE Elements to describe service offers and requests, and the associated matching algorithm that relies on efficient reasoning implementations. We have thus shown that implementation of adequate service description languages together with appropriate matching algorithms is indeed possible.

In our opinion, if semantic web services are to become widely used, service description languages need to be adapted to really allow for automatic, dynamic service discovery and invocation. To achieve this, the concepts presented in this paper (or something functionally equivalent) should be included into the existing description languages.

8 References

- [1] Anupriya Ankolenkar, Mark Burstein, Jerry R. Hobbs, Ora Lassila, David L. Martin, Drew McDermott, Sheila A. McIlraith, Srinu Narayanan, Massimo Paolucci, Terry R. Payne, and Katia Sycara, ‘DAML-S: Web service description for the semantic web’, in *Proc. Of the First Intl. Semantic Web Conf. (ISWC)*, Sardinia, Italy, (June 2002).
- [2] F. Baader, D. Calvanese, D.L. McGuinness, D. Nardi, and P.F. Patel-Schneider, *Description Logic Handbook – Theory, Implementation and Applications*, Cambridge University Press, 2002.
- [3] Steffen Balzer, Thorsten Liebig, and Matthias Wagner, ‘Pitfalls of OWL-s: A practical semantic web use case’, in *Proc. of the 2nd Intl. Conf. on Service Oriented Computing*, pp. 289–298, New York, NY, USA, (December 2004).
- [4] Sharad Bansal and José M. Vidal, ‘Matchmaking of web services based on the DAML-S service model’, in *Proc. of the Second Intl. Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pp. 926–927, Melbourne, Australia, (2003). ACM.
- [5] Sean Bechhofer, Frank Van Harmelen, Jim Hendler, Ian Horrocks, Deborah L. McGuinness, Peter F. Patel-Schneider, and Lynn Andrea Stein. OWL Web Ontology Language Reference. W3C Recommendation, February 2004. <http://www.w3.org/TR/owl-ref/>.
- [6] Alex Borgida and Ronald J. Brachman, ‘Conceptual modeling with description logics’, in *Description Logic Handbook – Theory, Implementation and Applications*, eds., F. Baader, D. Calvanese, D.L. McGuinness, D. Nardi, and P.F. Patel-Schneider, chapter 10, 359–381, Cambridge University Press, (2002).
- [7] Dieter Fensel and Christoph Bußler, ‘The web service modeling framework (WSMF)’, in *Database and Information Research for Semantic Web and Enterprise*, (2002).
- [8] Javier Gonzalez-Castillo, David Trastour, and Claudio Bartolini, ‘Description logics for matchmaking services’, in *Proc. of the Workshop on Applications of Description Logics at KI-2001*, Vienna, Austria, (September 2001).
- [9] Tom Gruber, ‘A translation approach to portable ontology specifications’, *Knowledge Acquisition*, **5**, 199–220, (1993).

- [10] Nicola Guarino and Christopher A. Welty, ‘An overview of OntoClean’, in *Handbook on Ontologies*, eds., S. Staab and R. Studer, chapter 8, 151–159, Springer Verlag, (2004).
- [11] Ian Horrocks, Peter F. Patel-Schneider, Harold Boley, Said Tabet, and Benjamin Groszand Mike Dean. SWRL: A semantic web rule language – combining OWL and RuleML. <http://www.daml.org/rules/proposal/rules-all.html>, April 2004.
- [12] Uwe Keller, Rubén Lara, Axel Polleres, Ioan Toma, Michel Kifer, and Dieter Fensel. WSMO web service discovery. WSMO Working Draft, November 2004. <http://www.wsmo.org/2004/d5/d5.1/v0.1/>.
- [13] Michael Kifer, Georg Lausen, and James Wu, ‘Logical foundations of object-oriented and frame-based languages’, *Journal of the ACM*, **42**(4), 741–843, (July 1995).
- [14] Michael Klein, ‘Handbuch zur DIANE service description’, Technical Report 2004-17, Universität Karlsruhe, Faculty of Informatics, (December 2004).
- [15] Michael Klein and Birgitta König-Ries, ‘Combining query and preference - an approach to fully automatize dynamic service binding’, in *Short paper at IEEE International Conference on Web Services (ICWS 2004)*, San Diego, CA, USA, (July 2004).
- [16] Michael Klein and Birgitta König-Ries, ‘Coupled signature and specification matching for automatic service binding’, in *Proc. of the European Conference on Web Services (ECOWS 2004)*, Erfurt, Germany, (September 2004).
- [17] Michael Klein and Birgitta König-Ries, ‘Integrating preferences into service requests to automate service usage’, in *First AKT Workshop on Semantic Web Services*, Milton Keynes, UK, (Dezember 2004).
- [18] Michael Klein, Birgitta König-Ries, and Philipp Obreiter, ‘Stepwise refinable service descriptions: Adapting DAML-S to staged service trading’, in *Proc. of the First Intl. Conference on Service Oriented Computing*, pp. 178–193, Trento, Italy, (December 2003).
- [19] Rubén Lara, Dumitru Roman, Axel Polleres, and Dieter Fensel, ‘A conceptual comparison of WSMO and OWL-S’, in *European Conference on Web Services (ECOWS 2004)*, Erfurt, Germany, (September 2004).
- [20] Lei Li and Ian Horrocks, ‘Matchmaking using an instance store: Some preliminary results’, in *Proc. of the 2003 Intl. Workshop on Description Logics (DL’2003)*, poster paper, Rome, Italy, (2003).
- [21] Lei Li and Ian Horrocks, ‘A software framework for matchmaking based on semantic web technology’, in *Proc. of the Twelfth Intl. World Wide Web Conference (WWW 2003)*, Budapest, Hungary, (May 2003).
- [22] Peter Mika, Marta Sabou, Aldo Gangemi, and Daniel Oberle, ‘Foundations for DAML-s: Aligning DAML-s to DOLCE’, in *In Proc. of the AAAI Spring Symposium on Semantic Web Services*, Stanford, CA, USA, (March 2004).
- [23] M. Montebello and C. Abela, ‘DAML enabled web services and agents in the semantic web’, in *Web, Web-Services, and Database Systems: NODe 2002*, pp. 46–58, Erfurt, Germany, (October 2002).
- [24] Yefim V. Natis. Service-oriented architecture scenario. Gartner Research. AV 19-6751, April 2003.
- [25] Tommaso Di Noia, Eugenio Di Sciascio, Francesco M. Donini, and Marina Mongiello, ‘A system for principled matchmaking in an electronic marketplace’, in *Proc. of the Twelfth Intl. World Wide Web Conference*, Budapest, Hungary, (May 2003).

- [26] Massimo Paolucci, Takahiro Kawamura, Terry Payne, and Katia Sycara, ‘Semantic matching of web services capabilities’, in *Proc. of the First International Semantic Web Conference*, Sardinia, Italy, (2002).
- [27] Dumitru Roman, Holger Lausen, Uwe Keller, Eyal Oren, Christoph Bussler, Michael Kifer, and Dieter Fensel. Web service modeling ontology (WSMO). WSMO Working Draft, September 2004. <http://www.wsmo.org/2004/d2/v1.0/>.
- [28] Marta Sabou, Debbie Richards, and Sander Van Splunter, ‘An experience report on using DAML-S’, in *Proc. Of the Twelfth Intl. World Wide Web Conf. Workshop on E-Services and the Semantic Web (ESSW)*, Budapest, Hungary, (2003).
- [29] Evren Sirin, James Hendler, and Bijan Parsia, ‘Semi-automatic composition of web services using semantic descriptions’, in *Proc. of Web Services: Modeling, Architecture and Infrastructure. Workshop in Conjunction with ICEIS2003*, Angers, France, (2003).
- [30] Katia Sycara, Seth Widoff, Matthias Klusch, and Jianguo Lu, ‘Larks: Dynamic matchmaking among heterogeneous software agents in cyberspace’, *Autonomous Agents and Multi-Agent Systems*, **5**, 173–203, (2002).
- [31] Katia P. Sycara, Matthias Klusch, Seth Widoff, and Jianguo Lu, ‘Dynamic service matchmaking among agents in open information environments’, *SIGMOD Record*, **28**(1), 47–53, (1999).
- [32] World Wide Web Consortium, ‘Web service description language (WSDL)’. <http://www.w3.org/TR/wsdl>.
- [33] World Wide Web Consortium, ‘XML schema part 2: Datatypes’. <http://www.w3.org/TR/xmlschema-2/>.