Web Service Automation Supported by Lightweight Semantic Annotations

dissertation

by

Jacek Kopecký

submitted to the Faculty of Mathematics, Computer Science and Physics of the University of Innsbruck

in partial fulfillment of the requirements for the degree of doctor of science

advisor: Univ.-Prof. Dr. Dieter Fensel,
Semantic Technology Institute Innsbruck

Innsbruck, October 22, 2012

Abstract

Service-oriented computing has brought special attention to service modeling, especially in connection with semantic technologies. The expected proliferation of publicly accessible services (both WS–∗ services based on WSDL and SOAP, and RESTful services — including Web APIs — based on native Web technologies) will necessitate tool support and automation. The research on Semantic Web Service (SWS) addresses this expected need with semantic technologies, creating SWS frameworks that especially address service discovery, composition and execution.

As the first SWS standard, expressing a consensus of the SWS research community, in 2007 the World Wide Web Consortium produced a lightweight bottom-up specification called SAWSDL for adding semantic annotations to WSDL service descriptions. SAWSDL intentionally only defines an annotation mechanism, but does not specify any semantic model for those annotations. As such, it is intended as a point of convergence for SWS research and development.

The development of SAWSDL created an empty space in need of filling by semantic models suitable for use as annotations in WSDL descriptions. To fill this space, this thesis defines **WSMO-Lite**, an ontology for service semantics that fits directly into SAWSDL annotations, covering functional, nonfunctional, behavioral and information semantics of Web services, which together form a basis for semantic automation.

SAWSDL is, by design, specific to WSDL, a standardized Web service description technology. However, for a number of reasons, WSDL is not used to describe RESTful services, and the increasingly common RESTful services were overlooked by SWS research. There was a need for a description language aimed at RESTful services that would be lightweight and easily combine with existing practice. To tackle this need, this thesis proposes two microformats, **hRESTS** and **MicroWSMO**, which mirror WSDL and SAWSDL on top of human-oriented HTML documentation of RESTful services.

Further, this thesis defines a minimal semantic service model that is an abstraction of WSDL and hRESTS, with which RESTful services can seamlessly be included in semantic processing with WSMO-Lite.

To demonstrate the viability of WSMO-Lite, in this thesis we have adapted **several SWS automation algorithms** to this semantic model, namely algorithms for Web service discovery, ranking and composition.

WSMO-Lite is intentionally lightweight, in order to smoothen the learning curve for adopters of SWS technologies. Our work is intended to serve as a basis for consolidation of SWS technologies and as an easy way of extending service-oriented and RESTful systems with semantic automation.

WSMO-Lite was submitted to the W3C for consideration towards standardization, and acknowledged as a *Member Submission* [27]. The W3C Team Comment on the submission stated that it "is a useful addition to SAWSDL for annotations of existing services and the combination of both techniques can certainly be applied to a large number of semantic Web services use cases."

Acknowledgements

WSMO-Lite and its supporting technologies hRESTS and MicroWSMO were developed as pieces of a larger body of work in several collaborative projects. This thesis consolidates and expands the parts of those efforts that were principally developed by the author of this thesis. Several acknowledgments and thanks are in order:

Primarily, I give thanks to Prof. Dieter Fensel, whose vision directly caused the research on WSMO-Lite and MicroWSMO. Also, I thank Dr. Elena Simperl for her guidance in methodology and in framing the thesis.

Special thanks go to Dr. Tomáš Vitvar, who through discussions helped shape the semantic models of WSMO-Lite.

The cornerstone of WSMO-Lite is the distinction of four types of semantics (functional, nonfunctional, behavioral, and information model) by Prof. Amit Sheth. WSMO-Lite continues the research thread started by Prof. Sheth and his team in WSDL-S.

Finally, I also thank my colleagues, especially the main developers of the systems that helped me perform an evaluation of the contributions of this thesis.

for Maya and Nicole

Contents

List of Figures

List of Tables

List of Listings

Part I

Problem Statement and Background

Chapter 1

Introduction

The emergence of service-oriented computing has brought special attention to the area of service modeling. Formal service descriptions are a fundamental element that enables tool support and automation for such tasks as service discovery, composition, execution and mediation. A 2001 vision article about the integration of services and ontologies [38] focuses on how the presence of service descriptions can enable computers to "find possible ways to meet user needs and offer the user choices for their achievement." Various proponents of concrete frameworks share an even stronger vision, where "the program can discover web services and invoke them fully automated" [26], for example to make complete travel arrangements for the user, going as far as filling out the user's expense claim [77]. [26] goes on to claim that "without mechanization of these processes, Internet-based e-commerce will not be able to provide its full potential in economic extensions of trading relationships."

In the decade since those articles, research into semantic Web service (SWS) models and descriptions has received much attention and funding: the major research efforts include the US-based OWL-S initiative,[1] and the European projects DIP, SUPER and SOA4All.[2] In these and other projects, researchers have proposed numerous frameworks for semantic Web services, especially OWL-S [83], WSMO [105] and WSDL-S [1].

SWS research has been highly fragmented, usually top-down, semantics-first, and by design detached from the underlying Web service technologies. Sometimes, semantic automation algorithms are even developed outside formal SWS frameworks (examples will be given later in this thesis). This situation has arguably hurt industrial adoption of SWS technologies, which, to our knowledge, has been minimal — to illustrate, searching for WSMO and OWL-S service descriptions, we have only found research examples and artificial test collections.

Service-oriented computing, centered on the so-called "WS–∗" family of technologies (based on the lightweight XML message protocol SOAP and the service description language WSDL), is heavily driven by standardization. Until 2007, there were no standards for semantic descriptions of Web services, then the World Wide Web Consortium (W3C) produced its Recommendation called SAWSDL: Semantic Annotations for WSDL and XML Schema [107]. SAWSDL represented the consensus of the SWS research community, but it is not a fully-

[1] `http://www.daml.org/services/owl-s/`

[2] `http://dip.semanticweb.org/`, `http://ip-super.org/`, `http://www.soa4all.eu/`

fledged SWS framework; instead it only provides hooks in WSDL where semantic annotations can be attached, leaving the specification and standardization of concrete service semantics for later.

In parallel to the development of WS–∗ technologies, another approach to service-oriented computing has started gaining traction. Built around Web applications, using the core Web protocols and formats, and following the Web architectural style REST [30], the approach is often called *RESTful Web services* [101] or *Web APIs*. Such services have naturally occurred in Web applications, as the application functionalities are increasingly used by programmatic clients, such as other Web applications. RESTful services are especially present in Web applications that use rich user interfaces (cf. Ajax [32]), where the JavaScript code running in the browser is a programmatic client, for which the Web application must provide appropriate services.

RESTful Web services use native Web technologies around HTTP, XML and JSON, resulting in better integration with the Web. However, RESTful services have been largely ignored by Semantic Web Services research, likely because there is no widely-accepted format for machine-oriented (formal) descriptions of RESTful services that could form the basis for semantic descriptions; and in fact there has been considerable discussion on whether RESTful services, with their hypertext structure, even need further formal service descriptions.[3]

An important difference between RESTful and WS–∗ services is in their client-facing structure: where a WS–∗ service usually exposes a single network protocol endpoint that handles all the operations of the service, a non-trivial RESTful service exposes a number of independently meaningful and accessible resources. In our work, we reconcile the two different kinds of services and we show that most semantic automation tasks can disregard the difference. In this thesis, we generally use the unqualified term "Web service" to encompass both WS–∗ services and RESTful services.

In summary, this thesis presents work that has been **motivated by the emergence of SAWSDL**, a standard that aims to directly address the fragmentation of SWS research, and further motivated by the lack of SWS support for RESTful services; the goal of this work is **to provide a SAWSDL-based unifying ontology for lightweight semantic descriptions of WS–∗ and RESTful Web services**, which will support automation of Web service tasks such as discovery, selection, and composition. This lightweight unifying ontology has been called WSMO-Lite.

In this first chapter of this thesis, we define and motivate more concretely the research problems addressed in our work (in Section 1.1), we describe our approach to tackling the problems (in Section 1.2), and we summarize the contributions of our work (in Section 1.3). Finally, we give an overview of the structure of the thesis (in Section 1.4).

1.1 Problem Statement

Semantic Web Services is a research area that applies semantic technologies towards automation of the discovery and use of Web services. The key to automation is a machine-readable representation of information about the func-

[3] `http://lists.w3.org/Archives/Public/public-web-http-desc/2005Jun/0000.html` is an example mailing list posting that started a long thread with such discussions.

tionality and other aspects of known services — a set of *semantic Web service descriptions*. A system that implements service automation will process the formal descriptions to perform or to support tasks such as discovery, ranking, composition, invocation etc.

SWS research primarily focuses on how to automate (wholly or partially) the various tasks. Every proposed approach must specify the information that it needs to know about the existing services, which serves as the input to the approach. *SWS frameworks* specify languages and ontologies for formal description of Web services, with ties to underlying technologies for Web service description and communication. Note that proposed automation approaches are not always related to any particular SWS framework. For example, [42] is a composition approach (discussed in Chapter 7 of this thesis) that employs a purely logical (mathematical) formalization of services; to use this approach with actual Web services, it must be adapted to some concrete SWS framework.

We can identify two high-level requirements on SWS frameworks: a SWS framework must i) capture information about service semantics which supports service automation tasks, ii) attach to relevant Web services technologies. The first requirement is discussed more in Chapter 2, where we provide a detailed survey of relevant literature in SWS research, from which we derive the types of information that need to be expressed in service descriptions to support service automation. The second requirement is discussed in further detail here because it motivates and frames the focus of this thesis.

Several SWS frameworks have been proposed in literature, chief among them (by the body of published work that pertains to them) are OWL-S [83] and WSMO [105]. For representing the service models and domain ontologies, OWL-S uses the Web ontology language OWL, and WSMO uses a family of languages called WSML.[4] In both frameworks, the semantic description of a service is conceptually independent of the underlying technical description (e.g. WSDL), and there is a *grounding* mechanism to connect the semantic description with an underlying technical description to support invocation. In contrast, WSDL-S [1] was another SWS framework which focused on attaching semantics directly to WSDL descriptions, independent of ontology representation languages.

In 2005, the W3C held a workshop to judge the various SWS technologies as inputs to potential standardization. The workshop report [9] indicated an apparent "consensus that better connections are needed with the realm of work based on the existing WS stack"; proposing an activity "to define simple specifications for the low-hanging fruit", which "would likely include semantic annotations in Web services descriptions" with "a simple incremental set of extensions to WSDL." Ultimately, the outcome of the workshop was the formation of a working group on Semantic Annotations for WSDL, which produced SAWSDL [107], a thin layer of annotations for WSDL and XML Schema documents: one generic annotation attribute modelReference that simply points from a WSDL or schema element to a semantic concept of any kind, and two specific attributes, liftingSchemaMapping and loweringSchemaMapping, used for pointing to data mappings.[5]

The work on SAWSDL intentionally avoided defining actual types of semantics that would be pointed to by model references. Annotating the inputs and

[4] WSML provides a formal mapping to RDF and OWL, with some extensions.

[5] SAWSDL layers semantics on top of syntax, hence *lifting* and *lowering*.

outputs of web service operations in SAWSDL is straightforward, but expressing other types of semantics is left unspecified. Therefore, SAWSDL cannot be viewed as a SWS framework itself; instead it is intended as a base on which further work should be done, eventually consolidating SWS approaches for standardization. This opened a **research question:**

1. *What kinds of semantics should be pointed to by SAWSDL model references to support SWS automation, and how should they be represented?*

To scope our work, we must specify the desired degree of SWS automation that we intend to support. The main area where SWS automation can be beneficial is in helping human users deal with large numbers of services — finding and selecting appropriate services and composing them together in valid processes. Therefore, we have chosen discovery, selection and composition as the primary scope of our work. In Section 2.1, we discuss these and other SWS automation tasks that are commonly targeted by SWS frameworks.

In parallel to the above developments, RESTful services started gaining prominence, but they were ignored by most SWS researchers. The primary barrier to applying SWS technologies to RESTful services is that despite the existence of several proposal for formal descriptions of RESTful services (mainly WADL [36]), providers seem reluctant to use them; instead most RESTful services are only described using plain, unstructured HTML documentation useful only to a human developer [72]. WSMO and OWL-S could, in theory, be *grounded* in RESTful services (e.g. in their WADL descriptions), but we are not aware of anybody actually researching such groundings. WSDL, the base for existing groundings and for WSDL-S and SAWSDL, can in theory be used to describe RESTful services, but very few RESTful services have WSDL descriptions [72].

The emergence of RESTful services, along with the lack of formal and semantic descriptions of such services, leads to a **supplemental research question:**

2. *Can SAWSDL and the ontology for service semantics be applied to RESTful services despite the architectural differences between WS–∗ and REST?*

There is no doubt in general that semantic descriptions of some form can support automation of the use of RESTful services; the question above focuses on the particular suitability of the SWS standard SAWSDL, and of the ontology with which we answer the first question, for describing RESTful services as SWS. If we can answer this question positively, we will have a single framework that can support SWS automation for WS–∗ and RESTful services alike.

These two research questions focus our work. They leave several common SWS concerns **out of scope** of this thesis:

- Creation of semantic annotations: we do not tackle the problem of knowledge acquisition; in particular, we do not work on tools and methodologies that would support the process of describing services semantically. For the workings of the algorithms presented in this thesis, we assume service descriptions are already available.
- Invocation: we do not deal in depth with the complexities of mapping between a semantic client and the Web services it wants to invoke — the creation of on-the-wire messages, and handling of errors and faults.

- Goal modeling: in contrast to WSMO, we do not propose a common form for expressing client goals — no common form is required to formalize or implement automation algorithms that work with semantic service descriptions. Indeed, different automation algorithms require different descriptions of what the user wants: for example, algorithms for discovery and composition need to know *what* the user wants to achieve, expressed through a description of the desired capability or the desired effect that the service(s) should have, while algorithms for service ranking and selection need the user to specify *how* it should be done, including such nonfunctional preferences such as that the functionality be done *cheaply*, *fast* and *reliably*, captured for example as weights for various service parameters. Some automation algorithms can further use a description of the *input data* that the client can make available to the service(s), which may affect the suitability or performance of services. The structure of client goals specific to various algorithms is discussed primarily in Chapter 7.
- Mediation: practical deployments of semantic technologies often have a need for mediating between different domain ontologies with overlapping semantics; similarly, service integration often requires mediation of processes. In our work, mediation is assumed to be handled by the infrastructure.

1.2 Approach and Methodology

From the two research questions posed in the preceding section stem four high-level methodological steps for our research work:

First, we analyze the types of semantics necessary to support SWS automation, by studying relevant literature especially on automation algorithms and on SWS frameworks, and by performing conceptual analysis of our own. Based on our analysis, we propose an ontology (a codification of formal definitions) for expressing the semantics on top of SAWSDL and WSDL.

Second, we analyze the structure of RESTful services, with the aim on applying the SWS semantics. From another round of literature examination and our own conceptual analysis, it turns out that we can devise a common model that distills service descriptions for WS–∗ and RESTful services; this model can be annotated with SAWSDL and the semantics from the first step.

These first two steps result in an ontology that has a service model and concepts for service semantics; we call this ontology WSMO-Lite.

Third, to evaluate the viability (fitness for purpose) of the ontology, we adapt several SWS automation algorithms to WSMO-Lite. The algorithms include our offer discovery, developed in parallel with the main contributions of this thesis. In this evaluation step, we check our main **success criteria**: i) that our service semantics ontology is sufficiently expressive to support the desired degree of automation (comprising service discovery, selection and composition), and ii) that the automation works equally well with RESTful services as it does with WS–∗ services.

Fourth and final, we perform further qualitative and quantitative evaluation of our contributions. We describe several prototypical implementations that make use of WSMO-Lite, and to illustrate that our approach is comparable

to the state of the art, we evaluate the performance of the adapted discovery mechanisms against other known service matchmaking tools. We also compare WSMO-Lite to state-of-the-art SWS frameworks, to verify the appropriate positioning of our approach in the evolving SWS field.

While carrying out the above methodological steps, we have further adopted the following interrelated design principles, originating from discussions with service practitioners and SWS researchers, and aimed to increase WSMO-Lite's chances of adoption:

Proximity to underlying standards. To a certain extent, WSMO-Lite is a continuation of the WSDL-S line of work, and a rethinking of WSMO and OWL-S, with SAWSDL in the center of its focus. Where WSMO and OWL-S make the semantic description of a service conceptually independent of the underlying technical description (e.g. WSDL)[6], the approach of WSDL-S, continued in our work, splits the semantic service models into their constituent semantic pieces, and uses those directly in SAWSDL. This way, the semantic description stays close to the underlying WSDL and thus could be easier to publish and maintain.

Light weight. In the spirit of SAWSDL, WSMO-Lite is very lightweight: a) it defines a minimal vocabulary for service semantics, b) the vocabulary is defined in the most basic Semantic Web ontology language, RDFS, with very limited reasoning requirements,[7] but it can easily accommodate more expressive languages, especially including languages for logical expressions and rules; c) WSMO-Lite builds on WSDL, which is already well-known to Web services practitioners; d) in the mechanism for describing RESTful services, the choice of microformats (or alternatively RDFa) limits the need for new syntax, and e) the two microformats proposed in this thesis are also tightly scoped to fit already existing service documentation. In effect, WSMO-Lite adds very few new constructs on top of underlying technologies that are already well-known, and it carries no inherent requirements on reasoning power.

Modularity. WSMO-Lite distinguishes four types of service semantics that together support automation of all the major service consumer's tasks: discovery, composition, negotiation, ranking, invocation etc. WSMO-Lite annotations are modular: in any given service-oriented system, increasing needs for automation can be met incrementally by adding and refining various types of semantic annotations. For example, when the number of services becomes hard to manage, functional annotations can be added to support service discovery and composition; when many services are being composed together, information model annotations can be added to facilitate data mediation; and later nonfunctional properties can be added so that the system can adaptively respond to service failures with replacements that have similar nonfunctional parameters (quality of service, policy, price) as the service that failed.

[6] Both WSMO and OWL-S use a "grounding" mechanism to connect a cohesive semantic model to service technologies, and both have groundings that use SAWSDL ([62, 75]).

[7] We discuss the difference between lightweight and heavyweight ontologies in Section 3.5.3.

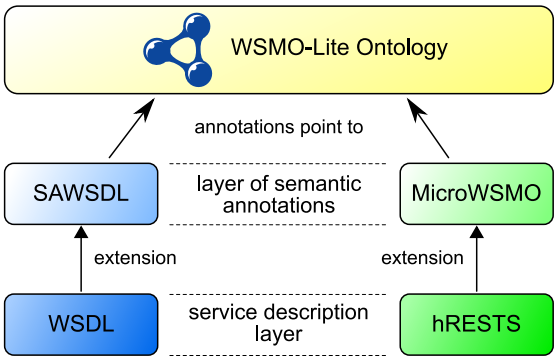


Figure 1.1: Semantic Web service descriptions with WSMO-Lite

1.3 Main Contributions

Figure 1.1 shows the core technologies that are among the contributions of this thesis. Our work stems from the standardization of SAWSDL, which, as illustrated on the left-hand side of the figure, extends the WS–∗ service description language WSDL with semantic annotations. For the content of the annotations, we propose the **WSMO-Lite ontology of service semantics**, capturing the types of semantics that are necessary to support SWS automation.

The right-hand side of the figure shows our proposed technologies for describing and semantically annotating RESTful services: hRESTS and MicroWSMO. WSMO-Lite includes a unified minimal service model that extracts the concepts common in WS–∗ and RESTful Web services. Based on this model, **hRESTS** (HTML for RESTful Services) is a microformat[8] for structuring common HTML documentation of RESTful APIs to make it machine-processable, analogously to how WSDL provides machine-processable descriptions on the WS–∗ side.[9] **MicroWSMO** (Microformat for WSMO-Lite) is an extension of hRESTS that adds SAWSDL annotation properties, where WSMO-Lite semantics can be attached.

We can summarize the contributions of this thesis in the following list:

1. An ontology that enables expressing clearly-delineated types of service semantics over SAWSDL (the first half of WSMO-Lite).
2. A minimal service model that extracts key service description and semantic annotation concepts for processing in semantic tools (the second half of WSMO-Lite).
3. An analysis of the mapping from the structure of RESTful services into the operation-oriented service model above.
4. Two microformats (hRESTS and MicroWSMO) for marking up machine-oriented service descriptions as part of HTML documentation of RESTful services.

[8] Microformats are an approach for annotating mainly human-oriented Web pages so that key information is machine-readable. [58]

[9] There is no accepted equivalent of WSDL for RESTful services; the Web Application Description Language (WADL, [36]) is a RESTful (resource-oriented) alternative to WSDL, but it is not commonly accepted.

5. The adaptations of several SWS automation algorithms for WSMO-Lite, covering discovery, offer discovery, ranking and composition.

WSMO-Lite, together with its supporting technologies hRESTS and MicroWSMO, is a key piece of a larger body of work by researchers in several collaborative projects, chiefly in SOA4All [66]. This thesis consolidates and expands those parts of the aforementioned efforts that were principally developed by the author of this thesis.

1.4 Overview of this thesis

This thesis is organized in three parts:

I. Problem Statement and Background, which analyzes the problem attacked by this work (Chapter 2), and describes all the background technologies necessary for understanding of this thesis (Chapter 3);

II. Semantic Web Service Description Languages, which contains the main contribution of this work, i.e., the semantic models (Chapter 4) and the languages (Chapters 5 and 6) for lightweight semantic description of Web services;

III. Evaluation and Conclusions, which shows the algorithms and prototypes with which lightweight semantic Web service description can be used to achieve concrete automation tasks (Chapters 7 and 8), describes our evaluation of the lightweight semantic description languages presented in this thesis (Chapter 9), and adds concluding remarks, including on future work (Chapter 10).

Figure 1.2 shows the chapters of this thesis as they fit within the three parts, including the dependencies between them.

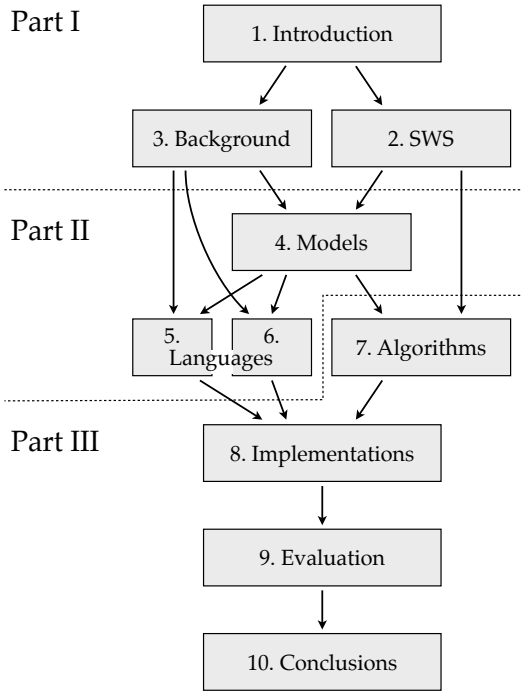


Figure 1.2: Thesis structure, with dependencies between chapters

Chapter 2

Semantic Web Services

In this chapter, we survey existing literature on Semantic Web Services, to provide a concrete description of the research problem addressed by this thesis, which is, in short, automation of the usage of Web services through lightweight semantic service descriptions.

SWS research has two main threads: algorithms that automate various tasks in using Web services, and frameworks that provide models for service descriptions intended as the inputs to the automation algorithms. This chapter analyzes literature in algorithms in Section 2.1 and in frameworks in Section 2.2. Section 2.3 discusses the open problems.

2.1 SWS Automation Tasks and Algorithms

The service consumer's tasks are identified in the Reference Architecture for Semantic Execution Environments (SEE-RA, [56]). Concretely, the SEE-RA defines a semantic broker between the service consumer and the services themselves; as such a broker can equally be deployed privately by service consumers, we talk about tasks of the consumer (or client), automated to some degree with semantic technologies.

In order to achieve any automation, the computer needs to have machine-readable and understandable information about the available services — *Web service descriptions*. These descriptions capture the relevant aspects of the meaning of service operations and messages. WS descriptions are processed by a semantic execution environment (SEE, for instance WSMX [37]). A user can submit a concrete *goal* to the SEE, which then accomplishes it by finding and using the appropriate available Web services. Abstractly, the SEE contains a *service registry*, a knowledge base that integrates the semantic service descriptions. In this thesis, we do not concern ourselves with how a service registry is populated, whether by explicit registrations from service providers or by an automated crawler that attempt to find service descriptions on the open Web, or by any other means.

In Figure 2.1, we illustrate a selection of the tasks that can be automated by a SEE. In the figure, the user wants to arrange the accommodation for a June vacation in Rome. We show four services with published descriptions: the airline Lufthansa, and hotel reservation services for New York, Rome, and one

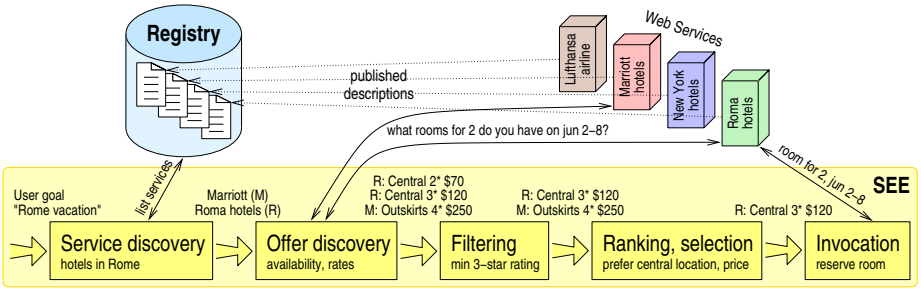


Figure 2.1: Semantic Execution Environment (SEE) automation tasks

for the Marriott hotel chain worldwide.

To achieve the goal, the SEE first *discovers services* that may have hotels in Rome, discarding Lufthansa which does not provide accommodation, and the New York service which does not cover Rome. Then the SEE *discovers offers* by interacting with the discovered services. The available offers in this particular example are only three hotels: a 4* Marriott at the outskirts of Rome, and one 2* and one 3* hotel in the city center. Then the SEE *filters* the offers depending on the user's constraints and requirements (minimum 3-star rating), *ranks* them according to the user's preferences (central location is more important than price) and selects the best offer, in the end *invoking* the corresponding service.

For simplicity of the illustration, the figure does not show *service composition*, the task of combining multiple services in order to achieve more complex functionalities. For example, if the user requests a comprehensive vacation package, the Lufthansa service would be used to book the travel, in combination with the hotel services to book accommodation.

The following subsections discuss relevant literature on approaches to automating these tasks, especially looking for requirements on the semantics that must be captured in service descriptions to support the automation. Due to necessary limits on scope, this thesis does not deal in depth with *semantic mediation*, which resolves any data and process heterogeneities, and *service invocation*, especially the mapping between the semantic layer and on-the-wire Web service messages. Therefore, mediation is omitted from the figure, and the invocation step is included only for completeness of illustration.

2.1.1 Semantic Discovery and Matchmaking

The scope of the term "discovery" can be understood very widely, encompassing a Web crawler that finds existing service descriptions, a matchmaker that selects known services that match a given user goal, and a ranking mechanism that sorts the matched services according to some measure of suitability. In this section, we focus on service matchmaking, characterized by Klusch [61] as "pairwise comparison of an advertised service with a desired service (query) to determine the degree of their semantic correspondence."

In 2001, Trastour et al. analyzed the features that a matchmaking system should have, and from this analysis they derived requirements on languages for service descriptions [119]. They identified the following requirements: i) high degree of flexibility and expressiveness, ii) ability to express semi-structured

data (by which they seem to mean incomplete data), iii) support for types and subsumption, and iv) ability to express constraints. They observed that neither UDDI or ebXML provided enough expressivity for advanced matchmaking with rich and flexible metadata, and they suggested that "there is a need for using ontologies," for which they turned to the technologies of the nascent Semantic Web. These insights are well reflected in later literature.

As an early effort, the actual matchmaker proposed by Trastour et al. relied heavily on ontology-specific matching rules (e.g. a Sale service with the role seller matches a Sale query with the role buyer), where later literature has focused on identifying concrete types of semantics that support generic matching rules.

Klusch [61] presents the most recent survey of semantic service discovery approaches. He categorizes semantic service matching approaches according to

- how matching is performed in terms of non-logic-based or logic-based processing, or a hybrid combination of both, and
- what kinds/parts of service semantics are considered for matching.

In this thesis, we focus on logic-based matchmaking. Non-logic-based and hybrid matching, which includes techniques of graph matching, data mining, linguistics, or content-based information retrieval, uses measures such as text similarity and path-length-based concept similarity to determine the degree of service match. The results of a Semantic Service Selection Contest[1] indicate that logic-based and non-logic-based approaches have different strengths and weaknesses and that hybrid approaches may give the strongest results.

Among logic-based approaches, Klusch investigates what kinds and parts of service semantics are considered for matching in the various approaches, especially pointing out how various approaches use different combinations of the descriptions of service inputs, outputs, preconditions and effects (together known as IOPE). For example, a *PE* matchmaker called PCEM [16] only uses preconditions and effects, an *IO* matchmaker OWLSM [51] only uses the inputs and outputs, and [54] presents an *IOPE* matchmaker for WSMO that uses all of them. Notably, Klusch does not mention semantic matching based on subsumption in hierarchies (classifications) of service types, which was identified among service description requirements by Trastour et al.

In summary, semantic technologies, logics and ontologies are commonly acknowledged as suitable for addressing service matchmaking. Service discovery and matchmaking can use the following types of service semantics: inputs, outputs, preconditions, effects, and service classifications.

2.1.2 Semantic Offer Discovery

Semantic Web service offer discovery, as introduced in this chapter, is a subset of a wider range of *contract agreement* and *negotiation* techniques, as discussed for instance by Preist [93]. In this thesis, we use offer discovery as a representative example of negotiation techniques that is especially useful for e-commerce services on the open Web.

The term *negotiation* has been used for different purposes in a variety of computer science research areas, e.g. electronic commerce, grid computing, distributed artificial intelligence and multi-agent systems. In electronic commerce,

[1] `http://www-ags.dfki.uni-sb.de/~klusch/s3/`

Beam and Segev [10] define negotiation as "the process by which two or more parties multilaterally bargain resources for mutual intended gain." As illustrated by Bartolini and Preist [7], there are several different types of negotiations in e-commerce: auctions (multiple buyers bid for price), double auctions (both buyers and sellers bid for price, e.g. stock exchanges), one-to-one bargaining, and even catalogue provision (price fixed by seller). Offer discovery is similar to catalogue provision, as offer discovery effectively accesses and retrieves the relevant parts of the offer catalogue.

Research on contract agreement and negotiation, for example in multi-agent systems, have generally presumed a controlled environment with a predefined set of interaction protocols for various tasks; for instance, a marketplace would dictate a bargaining and auctioning protocol. Such an approach can be applied to Web services, implying that a bargaining/auctioning protocol or a common offer-query protocol would need to be standardized. An example of work in this direction is the research of Paurobally et al. [86, 87], who propose an extension to the WS-Agreement and WS-Conversation languages [129, 132] with XML-based structures defining standard speech-act-like messages and transitions as in interaction protocols; they employ an iterative Contract Net Protocol [112]. [86] deals with the level of communication languages and interaction protocols, while [87] considers the negotiation subject and strategies.

In Semantic Web Services, we know of two published attempts that address automation of negotiation and dynamic offer discovery: an "estimation phase" of discovery by Küster et al. and the use of a "contracting interface" by Zaremba et al; additionally, we have proposed an alternative offer discovery approach (presented in Chapter 7) that relies on the *safety* property defined by the Web architecture.

Küster et al. [67] describe a service basically as a template for the offers, and parts of the offer data structure are marked as "estimation phase" parameters, with simple ordering of predefined interactions by which the client can retrieve the relevant offer data. In effect, this approach is similar to the one by Paurobally et al. in that Küster et al. prescribe a protocol (the estimation interactions) that would presumably be standardized. Their use of semantic annotations on the offer data structures allows heterogeneity in the data layer, but no heterogeneity is allowed on the communication layer of service operations.

Zaremba et al. [127, 149] talk about a so-called "contracting interface" with an explicitly described operation choreography. In Zaremba's contracting phase of Web service discovery, the SEE client follows the predefined choreography and the semantic annotations of the inputs and outputs to get information about the relevant offers from a discovered Web service. In effect, the SEE can adapt to the specific contracting interface of any Web service, as long as it is explicitly described.

In our work [63], we have proposed an offer discovery approach that opportunistically uses operations classified as Web-safe to gather information about service offers. We do not use an explicit contracting interface; instead, we use AI planning on the inputs and outputs of safe operations. With this approach, the SEE may be able to find out itself what the contracting interface is, lowering the complexity of service descriptions.

We can see that to support automation of negotiation tasks such as offer discovery in view of heterogeneous services, we require semantic descriptions both of the services' data and operations.

2.1.3 Semantic Filtering and Ranking

To select the most appropriate service or offer for the client's goal, the SEE must rank the results of discovery according to the client's preferences, and filter out those that do not match the client's constraints.

While functional matchmaking can provide coarse-grained matching degrees (commonly for *exact*, *plug-in* and *subsume* matches, see [61]), these degrees are not a good basis for service ranking because any matching service, independent of the match degree, can potentially satisfy the user's request.

To illustrate the inadequacy of functional match degree for ranking, let us revisit the example of looking for a hotel in Rome. The matching degree expresses how close a discovered service is to offering hotel reservations in Rome: a local service for Rome is a better match than a nation-wide service with hotels in Italy, which is itself a better match than a global service specialized in Marriott hotels world-wide. But the matching degree does not evaluate anything but the potential to be able to satisfy the client's goal: here, if the user prefers a high-quality hotel even if it is not centrally located, a Marriott hotel may be the best option, regardless of the Marriott service being the worst functional match.

Therefore, ranking is considered separately from functional matchmaking, and involves different service description parameters. Typical filtering and ranking parameters are nonfunctional Quality of Service (QoS) characteristics such as reliability, availability, processing performance etc., and other information such as location, price, or policies for security and data handling. O'Sullivan et al. [81] presents an extensive analysis of nonfunctional service properties.

Nonfunctional ranking is inherently a multi-criteria decision-making problem (see [50]) due to the presence of multiple parameters that cannot be directly compared to one another. For example, where one user may settle for a low-quality but cheap offer (such as a basic hotel located far from the center of Rome), another user would rather pay more for better quality (e.g., a centrally located four-star hotel). The relative importance of parameters such as price and performance varies for different users and situations.

In practice, nonfunctional filtering is a simpler version of the problem of ranking. For filtering, the client must specify a set of requirements that are evaluated to produce a single binary (pass/fail) result for each discovered service. For ranking, the client must specify preferences (sometimes called "soft requirements") that, when evaluated, must produce a combined ranking value (usually on a numeric scale) that allows comparing services with different nonfunctional properties.

Common NFP filtering and ranking approaches (such as [94] for filtering and [70] for ranking) consider only numerical or keyword values for nonfunctional properties. Toma et al. [118] argue that such approaches are inflexible and that an ontological representation with the help of logical expressions allows *semantic ranking* to provide more accurate results. In Toma's approach, the value of a nonfunctional property of a service may depend on the concrete goal data: the service NFP description includes logical expressions that compute concrete NFP values at run-time. For example, given a package described in the goal data supplied by the user, the NFP expressions can compute the actual price and the expected duration of shipping the package. The computed values can then be used for filtering and ranking.

In summary, to support automation of service filtering and ranking, we need descriptions of the services' nonfunctional properties. To be able to model rich and dynamic scenarios, nonfunctional properties should be described semantically, with the help of logical expressions.

2.1.4 Semantic Service Composition

There are many different approaches to service composition, as shown by the survey of Dustdar and Schreiner [25]. Among other aspects, the survey emphasizes the distinction between *manual composition* and *automatic composition*. To support manual composition, research focuses on languages that describe compositions, and tools that help the user with composing selected services. For automated composition, research investigates languages for describing services, and algorithms that use service descriptions to compose services that together can achieve a specified goal. In the area of Semantic Web Services, the focus is on automated composition.

Different automated composition approaches use varying levels of detail in service descriptions. A recent example by Lecue and Leger [68] is a representative of many approaches that match services into a sequence based on their inputs and outputs. Such approaches assume that the inputs and outputs of a service implicitly reflect the service's functionality. More expressive approaches such as the one proposed by Hoffmann et al. [42] use the preconditions and effects of Web services as *explicit* functional descriptions, decoupling message types from service functionality.

Composition on inputs and outputs, or on preconditions and effects, is commonly called *functional-level composition*, treating Web services as functions with single points of input and output. In contrast, *process-level composition* (e.g. [91]) takes into account the behavioral interfaces of the composed services, treating services as processes rather than atomic functions.

Functional-level composition is tractable, but because it does not take into account the services' behavioral interfaces, the composition solutions are not guaranteed to be actually executable. Rather, they support the human designer with rich information about possible compositions. Ideally, only minor modifications should correct any remaining mismatches.

A further distinction in composition approaches, described by Petrie et al. in [90], focuses on the end result of the composition process: whether it is a reusable workflow supposed to cover many use cases that combine the composed services, or it is a process instance that uses the services for a single use case. Composing process instances is simpler and may result in more optimal compositions, however it requires run-time handling of unexpected states.

Finally, it is important to consider the nonfunctional properties of service compositions, such as the overall Quality of Service (QoS). For example, Cardoso et al. [18] deal with QoS management in workflows made up of Web services. Their work presents a model for computing the combined QoS parameters of a composition from the QoS parameters of the constituent services; in particular, the work deals with the time, cost and reliability of services. As automatic composition tools can potentially produce large numbers of possible compositions, their combined QoS properties can be used for nonfunctional ranking and filtering of the compositions, along the lines of ranking and filtering of services, discussed in the preceding subsection.

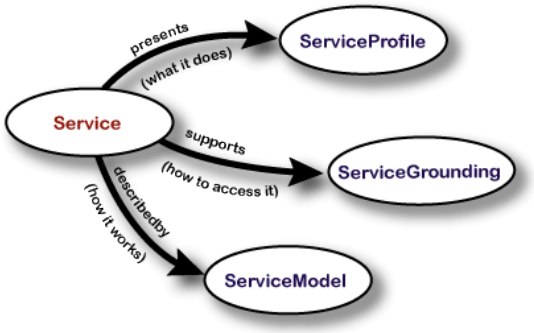


Figure 2.2: The structure of OWL-S (Fig. 1 from [74])

In terms of the requirements on service description, functional-level composition of process instances can use the descriptions of service inputs, outputs, preconditions and effects, in accord with the descriptions used by service discovery. Process-level composition, and composition into reusable workflows, further requires the descriptions of the behavioral interfaces of the available services. The computation of combined nonfunctional properties of compositions relies on the presence of nonfunctional descriptions of the individual services.

2.2 SWS Description Frameworks

Both Web Services and Semantic Web are visions that stem from the growth of the Web. As discussed by Lemahieu [69] in 2001, while the two visions and their families of technologies were often perceived as radically different, the ultimate goal of both is similar — making the Web a medium for machine-to-machine interaction. Lemahieu argues that semantic technologies can and should be applied to Web services, and points to DAML-S as an early example of what we would call a SWS framework.

Among many later research proposals, three SWS frameworks stand out: OWL-S and WSMO as two competing proposals with large bodies of associated literature, and WSDL-S as a lightweight approach that became the basis for SWS standardization. In this section, we analyze these frameworks, identifying the problems they have left open; Section 2.3 gives a summary of these problems and identifies those that we address in this thesis.

2.2.1 OWL-S: Semantic Markup for Web Services

As the W3C worked on the Web ontology language OWL, the early proposal DAML-S [77, 117] was evolved into OWL-S [83, 74]. OWL-S provides three types of knowledge about a service (as shown in Figure 2.2, from [74]):

- *Service profile*, used to advertise the service, describes what the service provides for prospective clients.
- *Service model* "tells a client how to use the service, detailing the semantic content of requests, the conditions under which particular outcomes will

occur, and, where necessary, the step by step processes leading to those outcomes." [74]

- *Service grounding* adds details such as the communication protocol and message formats and data serializations, allowing clients to invoke the service.

The service profile in OWL-S mainly describes the inputs, outputs, preconditions and effects (called *results* in OWL-S), and it also adds properties (deprecated in later versions of OWL-S) for pointing to categories is some ontologies or taxonomies of services and products.

The service model views a service as a process, with one or more steps that describe how a client can interact with the service. In practice, OWL-S descriptions use atomic processes to describe the WSDL operations of a service, and composite processes to describe the dependencies and interactions among the operations. Composite processes support data flows and common control constructs such as conditional branching, loops, and parallel execution with synchronization. Each (sub)process is specified through its inputs, outputs, preconditions and effects; expecting that ideally "the IOPE's published by the Profile are a subset of those published by the Process." [74]

OWL-S provides grounding to WSDL services. [74] presumes that for communication with the service, "the message parts will be serialized in the normal way for class instances of the given types, for the specified version of OWL," which would presumably be some RDF syntax. However, we know no WSDL services (or Web APIs either) that would communicate using OWL data, therefore there is a need for *lowering* OWL data into real-world messages, and *lifting* data from real-world messages into OWL. This is partially addressed by work on grounding OWL-S into SAWSDL [85, 75], where OWL-S gains the use of SAWSDL's lifting and lowering schema mapping annotations.

We are not aware of any publication that would specify the grounding of OWL-S service descriptions onto RESTful services/Web APIs that are not described in WSDL.

2.2.2 Web Service Modeling Ontology WSMO

The Web Service Modeling Ontology WSMO [105, 22] is a refinement and an extension of the fully-fledged Web Service Modeling Framework WSMF [26], providing a formal ontology and language for capturing actual descriptions. WSMF introduced and WSMO refined a top-down conceptual model for SWS with four top-level components: web services, goals, ontologies, and mediators.

Web services are described in WSMO mainly through their functional *capabilities* and behavioral *interfaces*. A capability defines the functionality of a Web service through preconditions, assumptions, postconditions and effects, all captured as logical expressions. WSMO makes a distinction between preconditions and postconditions on the information space of the Web service, and assumptions and effects that describe the state of the world. While WSMO capability description does not explicitly mention inputs and outputs, these aspects are still covered in the preconditions and postconditions, seeing the incoming/outgoing messages as part of the service's information space.

An interface of a Web service describes how the functionality of the service can be achieved. Here, WSMO distinguishes two sides of the service — an outer

choreography interface that describes how the clients should interact with the service, and an inner *orchestration* that defines how the service makes use of other Web services in order to achieve its capability.

Both choreography and orchestration may involve grounding to concrete service(s). In WSMO, service descriptions can be grounded in WSDL or in SAWSDL, with data serialization (lifting and lowering) handled with ontology mediation. We know of no effort to provide grounding for WSMO descriptions in RESTful services or Web APIs that do not use WSDL.

In addition to the functional and behavioral aspects, a Web service description can be annotated with nonfunctional properties, which is especially useful for comparing services that are functionally equivalent with respect to some particular client goal.

WSMO further discusses the description of client **goals**, structurally similar to service descriptions. Mainly, a goal can *request* a particular capability (which is then matched against known services in discovery), and any desired nonfunctional properties (for filtering and ranking of discovery results).

WSMO provides a rich family of languages for describing **ontologies**, which are the basis for WSMO Web service and goal descriptions. Defined before the W3C standardization efforts on OWL 2 and RIF, which improve the logical layering and rule support of the Semantic Web standards, the ontology languages (collectively named WSML for Web Service Modeling Language) cover a wide range of expressivity and computational characteristics. The least expressive language WSML-Core is extended in two separate branches: towards description logics in WSML-DL, and towards logic programming in WSML-Flight and WSML-Rule. The branches are joined in WSML-Full, which corresponds to an extended first-order logic.

Finally, WSMO deeply analyzes various types of **mediators**, which other frameworks relegate to a role of infrastructure components. Here, WSMO acknowledges the heterogeneity inherent in global service-oriented systems. In particular, WSMO declares four types of mediators: *ggMediators* link two goals, making it possible to map between goals, or to define one goal in terms of others. *ooMediators* are used in any WSMO descriptions to import ontologies through alignment, merging or mapping. *wgMediators* link Web services to goals, for example to state that a Web service (totally or partially) fulfills the given goal, providing the necessary data or process mappings. Another use for wgMediators is to allow for run-time late binding of third-party services in an orchestration. Finally, *wwMediators* can be used especially in service orchestrations to resolve data or process mismatches when one Web service uses another.

2.2.3 WSDL-S and SAWSDL

WSDL-S [1] is an extension of WSDL that brings semantic descriptions closer to the underlying Web services technologies. It was developed in the METEOR-S[2] project, and it defines five extension elements and attributes for WSDL and XML Schema:

- `modelReference` extension attribute that specifies the association between a WSDL or XML Schema entity and a concept in some semantic model.

[2] `http://lsdis.cs.uga.edu/projects/meteor-s/`

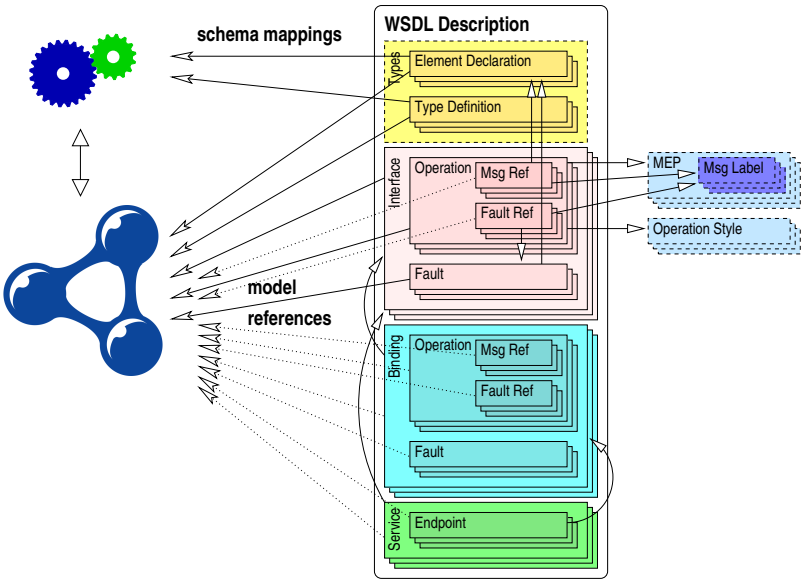


Figure 2.3: The structure of WSDL with SAWSDL annotations

- `schemaMapping` extension attribute for XML Schema elements and complex types, used for handling structural differences between the schema and their corresponding semantic model concepts. For data serialization, [1] discusses the use of XSLT and XQuery.
- `precondition` and `effect` elements, for use as children of the WSDL `operation` element, mainly to support service discovery. WSDL-S does not prescribe or define any actual logical language for capturing the precondition and effect expressions.
- `category` element, for use in the WSDL `service` element, also intended to support dynamic service discovery.

The positioning of WSDL-S as an extension of WSDL removes all need for grounding, but it also means that WSDL-S, as such, cannot support RESTful services that do not have WSDL descriptions.

WSDL-S was taken as the basis for SWS standardization in the W3C, which resulted in the *recommendation* called Semantic Annotations for WSDL and XML Schema (SAWSDL, [107, 64]). The resulting specification adopted the `modelReference` attribute from WSDL-S, and elaborated the `schemaMapping` attribute into a pair of attributes, one for a lifting mapping, and the other for lowering.

Figure 2.3 sketches the structure of WSDL and shows the expected uses of SAWSDL annotations in solid arrows. However, SAWSDL does not preclude the use of its annotation to the elements discussed in the specification, therefore the dotted arrows show that model references can indeed be present on any WSDL component.

Unlike WSDL-S, SAWSDL does not define concrete means for expressing service preconditions, effects, or categories; it is assumed that such information

can be attached with model references. Indeed, as SAWSDL does not specify any service semantics, it cannot be called a SWS framework.

2.3 Open Problems

In Section 2.1, we have identified the following types of semantics used by automation approaches:

- Inputs and outputs are used by many discovery, offer discovery and composition approaches.
- Preconditions and effects are also commonly used for discovery and composition.
- Classifications (of services and operations) help in discovery and offer discovery.
- Nonfunctional properties enable ranking and filtering.
- Behavioral descriptions of services are necessary for process-level composition, and can also be used in offer discovery.

Describing the semantics of services is the function of SWS frameworks. In Section 2.2, we have looked at three existing frameworks and on the lightweight standard SAWSDL; here is a high-level summary of how they cover the above types of semantics:

- WSMO can express inputs, outputs, preconditions and effects on a service, and within a choreography, which is a behavioral description of the service, it can express IOPEs on operations as well; note, however, that inputs and outputs are described within logical expressions, rather than through direct references to ontology classes, as is common in other frameworks. WSMO has no explicit support for service or operation classifications. On the other hand, it has comprehensive support for nonfunctional properties.
- OWL-S directly describes inputs, outputs, preconditions and effects both on services and on operations, and it describes the process (behavioral) model of services. It deprecates the properties for service categorization, and it has no explicit support for nonfunctional properties.
- WSDL-S annotates the inputs, outputs, preconditions and effects of operations, but not the IOPEs of services. It provides explicit support for categorizations of services, and implicit support for categorization of operations. WSDL-S does not put operations together in any kind of behavioral descriptions, nor does it include support for nonfunctional properties.
- SAWSDL has clear support for annotating the inputs and outputs of service operations. The specification is built on the assumption that all other types of semantics can be attached to service descriptions through the model reference construct, but it does not provide concrete mechanisms for expressing any of them.

WSMO has a comprehensive scope and coverage of various aspects of formal descriptions in service-oriented systems, which can make it seem detached from standards (both Web service and Semantic Web technologies), and intimidating in the required learning curve. OWL-S has a strong focus on low-level process

modeling, which may obscure the simplicity of the framework in common settings. To avoid such perceptions, SAWSDL takes a minimalistic, lightweight and modular approach to providing semantic annotations for service descriptions: it is minimalistic in that SAWSDL only defines the bare minimum of constructs; it is lightweight because SAWSDL does not require any advanced processing or complex descriptions; and it is modular because SAWSDL annotations are not inherently interdependent, so they can be used on any subset of WSDL elements as suits a particular application. Work building on SAWSDL should also adopt these traits.

The finalization of SAWSDL created the first open problem: there is no ontology for service semantics that can be used in SAWSDL model reference annotations to support SWS automation. Any work to address this problem should be able to express the above types of semantics, and it should use mechanisms similar to those developed in the previous frameworks, in order to encourage the adaptation of existing automation algorithms.

It should be noted that SAWSDL, despite its principal relationship to WSDL, is not actually specific to WSDL or XML Schema in any way. The attributes can be used in other XML-based service and schema description languages, and the RDF properties defined by SAWSDL can be used with any semantic service or schema model. The public Web contains increasingly many services that are in fact not described in WSDL, especially the RESTful services and Web APIs mentioned in the introduction chapter. RESTful services pose the second problem: there are no models and approaches for semantic descriptions of RESTful services. In order to extend the benefits of SWS automation to RESTful services as well, there is a need for research on how to overcome the reluctance of RESTful service providers towards machine-oriented service descriptions, and how these descriptions would be amenable to semantic annotations, optimally with SAWSDL.

In this thesis, we address both problems: we propose a minimalistic, lightweight and modular ontology of service semantics for use in SAWSDL annotations, and we also propose lightweight mark-up mechanisms for existing (human-oriented) descriptions of RESTful services to make them machine-processable, and suitable for semantic annotations through SAWSDL.

Chapter 3

Background

This chapter contains the description of all the background technologies used in our work, or directly relevant to it. A knowledgeable reader is welcome to skim over it or even skip it entirely; this chapter does not introduce any new results, in merely serves as a reference, to make this thesis a self-contained work.

We start in Section 3.1 with the reference model of service-oriented architectures, defined in the standardization organization OASIS. Then we proceed to common Web technologies in Section 3.2, WS–∗ technologies in Section 3.3, and technologies specific to RESTful services in Section 3.4. Finally, we talk about relevant Semantic Web technologies in Section 3.5.

3.1 OASIS Reference Model for SOA

The term "service" in its modern usage comes from economics. The services sector now dwarfs agriculture and manufacturing, the original main sectors of economy; its boom has recently spawned a Services Science [19], which deals with terms such as "exchange of goods" and "economic entities". In IT, there have been several efforts to define service models and conceptualize services and service-oriented architectures, activities initiated in academic research (e.g. [6, 93, 29]) as well as in the context of standardization bodies (esp. [100], discussed below).

The term "Web service" started in association with a set of concrete technologies for distributed computing, especially WSDL and SOAP, as shown in the W3C's Web Services Architecture document [130]. Service-oriented architecture (SOA) is an abstraction of Web services: it is oriented toward large-scale distributed systems, and it sheds the technology bias of "Web services". SOA is currently best articulated by the OASIS consortium's industry-standard Reference Model for Service Oriented Architecture (SOA RM [100]), which defines a service as follows:

> *A service is a mechanism to enable access to one or more capabilities, where the access is provided using a prescribed interface and is exercised consistent with constraints and policies as specified by the service description.*

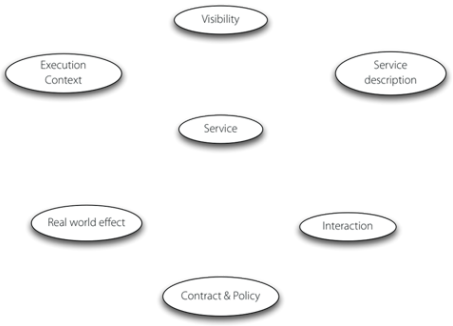


Figure 3.1: Top-level components of the OASIS SOA RM (Fig. 3 from [100])

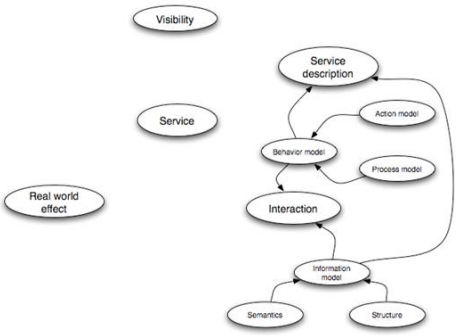


Figure 3.2: Interaction model in the OASIS SOA RM (Fig. 6 from [100])

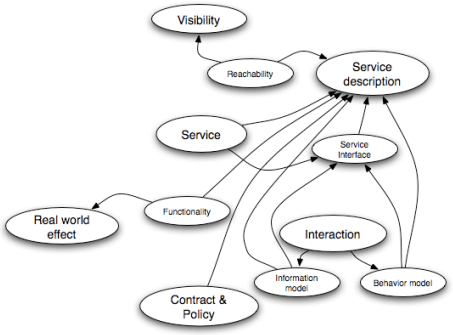


Figure 3.3: Service description model in the OASIS SOA RM (Fig. 9 from [100])

The SOA RM investigates a number of aspects of services; the top-level six (shown in Figure 3.1) are *Service description*[1], *Visibility*, *Execution context*, *Real-world effect*, *Contract and policy*, and *Interaction*. In the following, we give a brief summary of the aspects that are relevant for our work; note that the figures are *concept maps* where arrows denote dependencies.

A service in SOA is generally not a single physical artifact, instead it is a concept useful for manageability. From the point of view of a client, the physical artifacts of a service are its network location (part of the *Visibility* aspect) and the service description.

Figure 3.2 details the components of the interaction model in SOA RM, which deals with how a client interacts with a service. The interaction model is described by the behavior model of the service, and influenced by the information model. The RM splits the behavior model into "actions" — the operations that can be invoked on a service — and "process" that specifies the valid orderings of the operations. The information model is concerned both with the semantics of the data and with data structures and exchange formats.

Figure 3.3 shows the components of a service description: the reachability aspect gives us technical parameters and the network location where the service is available, the functionality aspect describes what the service does, the contract and policy aspect specifies nonfunctional properties, and the service interface aspect describes the data and the behavior of the service.

The SOA RM gives us a general service model structure and it points out useful distinctions that help us define a lightweight service and semantics model in Chapter 4.

3.2 Common Web technologies

In this section, we touch on Web technologies that are common in Web services. Technologies specific either to WS–∗ or to RESTful services are discussed in Sections 3.3 and 3.4, respectively.

We begin with the structured data formats XML and JSON, and then we talk about the foundation technologies of the Web — URI and HTTP.

3.2.1 XML

Extensible Markup Language (XML [138]) is a text format derived from SGML (ISO 8879). In the context of Web services, XML is used mainly for exchanging structured data.

Without delving into complexities that are seldom used, we can say that the data model of XML is a tree, starting with a single root *element* that can have an unordered set of string-valued *attributes* and either a string value within the element, or an ordered set of *child elements*, through which the structure becomes recursive.

Both elements and attributes are named, and to prevent naming conflicts, XML provides a name space mechanism that uses URIs as name space identifiers. Within a single element, the attributes must have unique names, while there is no such requirement on the child elements whose names are unconstrained and can be repeated.

[1] In the text we *emphasize with italics* the terms defined by the SOA RM.

```
1  <app:service xmlns:app="http://www.w3.org/2007/app" xmlns:atom="http://www.w3.
       org/2005/Atom">
2    <app:workspace>
3      <atom:title>Main Site</atom:title>
4      <app:collection href="http://example.org/blog/main" >
5        <atom:title>My Blog Entries</atom:title>
6        <app:categories href="http://example.com/cats/blog.cats" />
7        <app:accept>application/atom+xml;type=entry</app:accept>
8      </app:collection>
9    </app:workspace>
10 </app:service>
```

Listing 3.1: Example XML AtomPub Service Document

Listing 3.1 shows an example XML document, in this case an Atom Publishing Protocol Service Document (discussed in Section 3.4.3). It combines two namespaces (`atom` and `app`) because the service document format reuses vocabulary from the general Atom format. The listing shows elements child elements, elements with attributes (e.g. `app:collection`), and elements with string values (e.g. `atom:title`).

To describe custom data structures, such as the messages accepted and produced by a Web service, there is a schema language called XML Schema [140], itself written in XML. A schema can specify the allowed attributes on an element, the allowed data types of values of attributes and string-valued elements, and the allowed sequences of child elements. An XML Schema can be used both for data format description (to communicate an agreed format) and for structural validation (to check that a particular XML document or message conforms to the expected structure). Later in this chapter, Listing 3.3 contains a self-explanatory example schema.

XML has a large ecosystem of supporting technologies, including the following query and manipulation languages:

- XPath [142] is a language for expressions that identify nodes within XML documents; for example `/app:service//atom:title` selects both `title` elements in our example.
- XSLT [146] is a declarative transformation language, itself in XML, for transformation of XML documents. It uses XPath to select and access data in the source document. XSLT can output XML, HTML or plain text.
- XQuery [143] is a functional query language that works over collections of XML data, also using XPath. XQuery has a specific syntax that is much more concise than XSLT.

3.2.2 JSON

While XML is likely the most common format for structured data exchange, its processing in Web applications that run in the browser was historically slow and complex, therefore an alternative format has emerged, aimed for relatively short messages with structured data: JavaScript Object Notation (JSON [53]), a lightweight data format derived from the JavaScript language.[2]

[2] `https://developer.mozilla.org/en/JavaScript`

```
1  {
2    "service": {
3      "workspace": {
4        "title": "Main Site",
5        "collection": {
6          "href":       "http://example.org/blog/main",
7          "title":      "My Blog Entries",
8          "categories": "http://example.com/cats/blog.cats",
9          "accept":     "application/atom+xml;type=entry"
10       }
11     }
12   }
13 }
```

Listing 3.2: Example JSON, illustrating data from Listing 3.1

A JSON message can represent four primitive types (strings, numbers, booleans and null) and two structured types (objects and arrays). Listing 3.2 shows the data from the XML example earlier, as it might look like in JSON.

In comparison to XML, JSON has no explicit support for namespaces; on the other hand, its syntax distinguishes directly between an object with fields and an array with ordered items — XML does not provide for such distinction. But the main difference in the intended environment (a JavaScript engine) is that JSON can be manipulated natively, whereas XML data needs an access API (called DOM — Document Object Model) which is commonly perceived as unwieldy.

3.2.3 Uniform Resource Identifier

Uniform Resource Identifier (URI [123]) is a syntax for identifiers and addresses on the internet, devised along with the World Wide Web. A typical URI taken from the examples above is

`http://example.com/cats/blog.cats`

which identifies the resource `/cats/blog.cats` at the server `example.com`, accessed through the HTTP protocol. Beside these components, URIs can commonly also have so-called query parameters, which customize or specialize the main resource, and so-called fragment identifiers, which usually identify a part of a document.

URIs can be used as addresses (to locate and access some content or service on the internet), or as names for things, especially on the Semantic Web (see Section 3.5).

In addition to concrete URIs, several syntaxes have been proposed for *URI templates*, e.g. within WSDL 2.0 [134]. A URI template describes a simple schema for creating URIs with parameters (not restricted to be query parameters). For example,

`http://example.com/cats/{type}.cats`

could prescribe the URI for the categories file for a given type (the `{type}` parameter); if the desired type is `blog`, the URI would be the one we've shown above. URI templates are mainly used in service descriptions, to specify how a client should create specific URIs depending on what they need.

3.2.4 HTTP

Hypertext Transfer Protocol HTTP [49] is the application-level network protocol of the World Wide Web. It defines the following main methods for accessing Web resources, identified by URIs:

- GET requests a *representation* of the resource (e.g. a Web page), HEAD requests only its metadata (content type, size, time of last update etc.);
- PUT updates (replaces) the content of a resource with the submitted data;
- DELETE removes the resource;
- POST submits data to the resource for processing, for instance to create a new resource (such as adding an item to a collection) or to update an existing one;
- OPTIONS asks for the HTTP capabilities of a resource; TRACE asks the server to echo the request, for debugging purposes; and CONNECT is used to switch to some other agreed protocol.

Of these, GET, PUT, POST and DELETE are the main functional methods, akin (but not equivalent to) the common database *CRUD* operations Create, Read, Update and Delete. To access PUT, POST and DELETE, a server commonly requires proper authentication, for which HTTP provides several methods.

The four methods have distinct specified semantics: PUT and DELETE are idempotent (repeating the request will have the same end effect), which is useful in face of network failures; and GET is *safe*, as defined in the Web architecture [3]:

> A *safe interaction* is one where the agent does not incur any obligation beyond the interaction. An agent may incur an obligation through other means (such as by signing a contract). If an agent does not have an obligation before a safe interaction, it does not have that obligation afterwards.

A canonical example of a safe interaction in distributed systems is information retrieval: an e-commerce client may, for example, use a catalogue search interface, yet by issuing a search query the client makes no commitment to purchase any items. Safe operations are effectively idempotent (but idempotent operations need not be safe).

Because of the method's safety, GET can be used opportunistically by automated agents — this enables pre-caching, crawling the Web for search engines, and the so-called *following your nose*, which means resolving links in data to retrieve more data.

HTTP is heavily metadata-driven: it can do conditional operations, such as retrieving the resource only if it has changed since the client's last copy (this is part of HTTP's extensive caching machinery); it can also perform content negotiation, where the client expresses preferences over content types (on a data table such as a calendar, a browser would prefer HTML for human viewing, while an automated tool could prefer some machine-oriented format such as iCalendar for data processing).

3.3 WS–∗ technologies

In this section, we give a brief overview of technologies that make up the WS–∗ stack. We start with the protocol SOAP, then detail the service description language WSDL, and finally we touch on other relevant technologies.

3.3.1 SOAP

SOAP [113], originally called Simple Object Access Protocol, is an XML-based messaging protocol for Web services. The specification has three main components: an extensible message structure, a processing model, and a binding framework for transmitting SOAP messages over communication protocols.

The message structure splits the XML message into a set of *headers* with processing metadata, and the *body* which contains the payload of the message.

The processing model defines a notion of an *intermediary* — a message processor that is not the final recipient of the message but that is part of the overall application. Common types of intermediaries would perform tasks such as logging, encryption/decryption and message routing. Individual message headers can be targeted at specific intermediaries. To facilitate distributed extensibility, the processing model specifies that headers can be marked as mustUnderstand — if the receiving processor does not support such a header, it must cease any processing and respond in a predefined fashion, so that the sender of the message learns early what requirements cannot be met.

Finally, the binding framework contains concepts necessary for defining how SOAP messages should be transported over concrete communication protocols. The W3C has defined a normative *HTTP binding*, where SOAP messages can be involved in GET and POST requests; and also an *email binding*[3] that transfers SOAP messages over email infrastructure, especially using the SMTP message delivery protocol. Third parties have defined bindings over UDP [121], JMS (Java Messaging Service, [52]) and other communication protocols.

3.3.2 WSDL

WSDL 2.0 [133] is a language for describing Web services. In particular, it can describe the structure of the messages the service accepts and produces, simple message exchanges (called operations) and all the necessary networking details. On top of this, extensions in WSDL documents can specify that additional features are supported or even required by the service. In effect, WSDL specifies a limited contract that the service adheres to.

The core WSDL specification defines a fairly simple set of components; at the top level, the main ones are the Interface, Binding and Service components, as illustrated in Figure 3.4.

An **Interface** describes the abstract interface of a Web service — the operations, messages and faults. Every operation follows some pre-defined message exchange pattern (MEP). An MEP in WSDL prescribes the number and directionality of messages, and the operation populates them with concrete XML elements, defined in XML Schema. Most MEPs also allow fault messages for

[3] `http://www.w3.org/TR/soap12-email`

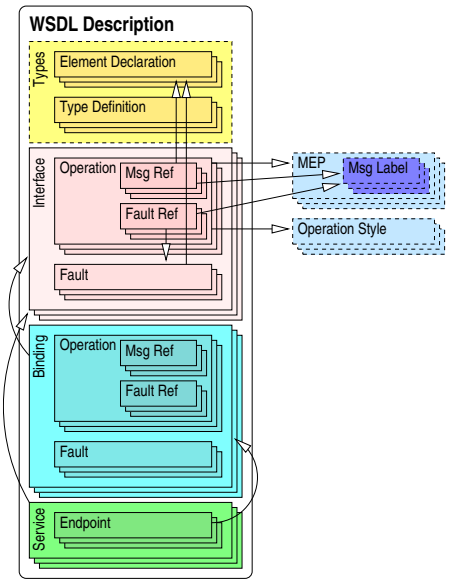


Figure 3.4: The structure of WSDL, with components in solid boxes, and other concepts in dashed boxes.

expected application-level errors. Faults are modeled on the same level as operations, that is, an interface defines a number of faults which are then used by operations.

WSDL 2.0 pre-defines eight **Message Exchange Patterns**; we will show the core three to illustrate the concept:

- *In-Only* is the simplest MEP — it defines that there is only a single input message from the client to the service. This is used for *fire-and-forget*-style interactions; the service can notably not even respond with a fault.
- *In-Out* is by far the most common MEP, as it defines the standard request-response style operations. The MEP starts with an input message, which is followed either by an out message or by an out fault.
- *Robust In-Only* is a messaging-style MEP where the client sends an input message and does not expect a response, except that the service can still send a fault to notify the client if anything is wrong.

While WSDL does not constrain the number of messages in MEPs, we assume in our work that there can only be one input message and one output message in any MEP. This fits all eight of the WSDL-predefined MEPs. [80] proposes a multi-party MEP that can use multiple input messages and multiple output messages, but it still defines one message type for all the input messages and one for all the output messages, therefore the assumption still holds on the level of semantic descriptions.

An interface is specified on the level of XML messages; the networking details about how the messages are represented on the wire are specified in the next top-level WSDL component — the **Binding**. This component follows the structure of the Interface and uses extensions to specify any protocols and networking parameters.

The last top-level WSDL component — **Service** — provides a number of Endpoints where a service is available. An Endpoint specifies an actual address, together with a Binding that indicates how the client should communicate with that address.

Listing 3.3 shows an example WSDL document (adopted from the WSDL Primer [135]) that describes a simple hotel service. Its interface provides a single operation for checking the availability of rooms for a given date. Abstractly (i.e., on the Interface level), the operation accepts the dates of check-in and check-out and the required type of room, and it returns the daily rate of available rooms, or zero if nothing is available. In case the room type or input dates are erroneous, a fault can be generated. The Binding level specifies that the data will be transmitted with SOAP over HTTP, plus several other details. Finally the Service level gives one endpoint address, at which the hotel service is available.

WSDL is, by design, a very extensible language, and in fact some parts of the standard are built as predefined extensions (see [134]). WSDL is mainly extended through predefined *extension points* — those places in WSDL descriptions where a number of options is defined by WSDL, but the list is open. For example, WSDL defines two binding types and eight message exchange patterns, but third parties are free to define new ones, where required.

Further, WSDL is open to XML-based extensibility, i.e. any WSDL element can contain any number of attributes or elements from a foreign (non-WSDL) name space. Such extensions are not constrained in what they may mean. In general, extensions add properties to the existing WSDL components, so that the processor can use the extended information. As we cannot expect all WSDL processors to know all the extensions they might encounter, extension elements in WSDL are by default *optional*, but they may be marked *mandatory*. Optional extensions can be ignored by a processor that does not recognize them, whereas mandatory extensions must be understood by a processor that needs to process the parent WSDL element. This allows mandatory extensions to change the meaning of the parent WSDL element.

SAWSDL is an example of an optional WSDL extension that uses the XML-based extensibility.

3.3.3 Brief overview of other relevant WS–∗ technologies

The WS–∗ family of specifications is quite extensive,[4] here we mention several of them that pertain to service description.

Universal Description Discovery and Integration (UDDI [120]) is a specification for a registry of businesses and their Web services. It was originally backed by a public, openly-accessible deployment, which was discontinued[5] in 2006. As pointed out by [26], UDDI provided "limited support in mechanizing service recognition, service configuration and combination (i.e., realizing complex workflows and business logics with web services), service comparison and automated negotiation," which led to further service description efforts. As UDDI was not extensible, it was unable to support these advanced descriptions. No other service registry standard has been proposed yet.

[4] Its size is illustrated by an overview poster available at `http://www.innoq.com/soa/ws-standards/poster/`

[5] As reported by `http://soa.sys-con.com/node/164624`

```
1  <description targetNamespace="http://hotel.example.com/wsdl" ...>
2    <types>
3      <xs:schema ...>
4        <xs:element name="checkAvailability" type="tCheckAvailability"/>
5        <xs:complexType name="tCheckAvailability">
6          <xs:sequence>
7            <xs:element name="checkInDate" type="xs:date"/>
8            <xs:element name="checkOutDate" type="xs:date"/>
9            <xs:element name="roomType" type="xs:string"/>
10         </xs:sequence>
11       </xs:complexType>
12       <xs:element name="checkAvailabilityResponse" type="xs:double"/>
13       <xs:element name="invalidDataError" type="xs:string"/>
14     </xs:schema>
15   </types>
16
17   <interface name="hotelIface">
18     <fault name="invalidDataFault" element="tns:invalidDataError"/>
19     <operation name="checkAvailability"
20           pattern="http://www.w3.org/2006/01/wsdl/in-out">
21       <input element="tns:checkAvailability"/>
22       <output element="tns:checkAvailabilityResponse"/>
23       <outfault ref="tns:invalidDataFault"/>
24     </operation>
25   </interface>
26
27   <binding name="hotelSOAPBinding"
28     interface="tns:hotelIface"
29     type="http://www.w3.org/2006/01/wsdl/soap"
30     soap:protocol="http://www.w3.org/2003/05/soap/bindings/HTTP">
31     <fault ref="tns:invalidDataFault"
32           soap:code="soap:Sender"/>
33     <operation ref="tns:checkAvailability"
34           soap:mep="http://www.w3.org/2003/05/soap/mep/soap-response"/>
35   </binding>
36
37   <service name="hotelService" interface="tns:hotelIface">
38     <endpoint name="hotelEndpoint"
39           binding="tns:hotelSOAPBinding"
40           address="http://hotel.example.com/service"/>
41   </service>
42 </description>
```

Listing 3.3: Example WSDL hotel service description

Web Services Policy [131] is a general-purpose model to describe the policies of entities in a Web-services-based system. A policy can describe any capabilities, requirements, and general characteristics of both services and their clients, using combinations of *policy assertions*. As an example of policy assertions, we can mention the W3C WS-SecurityPolicy [2] specification. There has been work such as [124] on semantic representation and matching of policies, usually treating policy information as nonfunctional properties.

3.4 RESTful Web services

As we've discussed in the introduction chapter, there is a growing number of Web services that do not use SOAP and WSDL; instead they are built more directly on HTTP. They are commonly called RESTful services (those that follow the REST architectural style of the Web) or Web APIs (programming interfaces for Web applications). We use the terms RESTful services and Web APIs interchangeably in this thesis; however, we must note that many Web APIs disregard most of the REST principles.

In this section, we discuss the main technologies that are used by RESTful services and Web APIs; we start with a description of the REST architectural style, then we move to the various ways of describing the services, both in human-oriented and machine-oriented forms.

3.4.1 REST

Representational State Transfer (REST, [30]) is the name of an architectural style developed by R. Fielding as a formalization of the architectural principles underlying the World-Wide Web. REST is composed of a number of constraints that ensure certain beneficial properties of the resulting architecture. These properties have made the Web scalable and evolvable, so it could have grown to the size and popularity it has today, without showing any signs of inherent barriers to future growth.

The Web was designed to be (and is) an internet-scale distributed hypermedia system. This goal implies certain requirements, which affect REST as well. In particular, the Web needs to be:

- simple, with a low barrier of entry, to attract users and developers;
- extensible, to be able to grow past the initial simplicity;
- distributed hypermedia, to be able to use the power of many internet hosts;
- anarchically scalable, to isolate performance issues of independent parts;
- independently deployable, both in terms of hosts and in terms of protocols and data formats, to allow gradual evolution and coexistence of old and new components;
- human-oriented, both optimized for better user experience, and tolerant of humans' erratic interactions with the system.

The REST architectural style contains the following ingredients (which are themselves simpler architectural styles):

Client-server. This style separates the concerns of the server (serving data, processing user inputs) from those of the client (user interface, presentation and interaction). This simplifies portability of the user interface even to platforms that would not support servers, and it also allows the components to evolve independently.

Layering. While an actual system may consist of hierarchical layers that build one on another, the components are constrained only to see the immediate layers with which they interact. This restriction puts a bound on the overall system complexity and promotes component independence, while adding overhead and latency to the interactions, which are mitigated by the increasing performance of computers and networks.

Stateless communication. "Each request from client to server must contain all of the information necessary to understand the request, and cannot take advantage of any stored context on the server."[30] To understand this constraint, we must separate the state of an application into the state of resources on the server, and the state of the interaction (also known as the *session*) between the client and the server.

In traditional client-server systems, the server keeps a separate session open for each client, which simplifies the communication. For instance, in a search engine, the response to a search request (e.g. "hotels in Rome") is a list of the ten most relevant results. In a stateful system, the client may follow up with a request for *the following 10 results*, and the server knows that the client wants results 11–20. If the same request (*the following 10 results*) is repeated, the response will contain results 21–30, and so on. In a stateless system, the client must always tell the service what exactly it wants: it would request *results 21–30 of the search for "hotels in Rome,"* for instance.

This constraint adds communication overhead (again, mitigated by increased performance of newer networks), and it makes servers relinquish some of the control over how the application proceeds. On the positive side, stateless communication improves the scalability of servers (by freeing their resources between requests) and the reliability of applications (by simplifying the task of recovering from partial failures), as discussed in [101].

Uniform interface. All the components in a RESTful system must support a single uniform interface. In Section 3.2.4, we have discussed the methods of HTTP, especially including GET, POST, PUT and DELETE.

The fact that the interface is uniform means that all components can speak to each other. With a single interface, a Web browser can access any Web resource, and there is no need for specialized browsers for different resources; implementations are decoupled from the applications. REST uniform interface is optimized for large-grain hypermedia data transfer, which is not necessarily efficient for all applications.

Caching. To improve network efficiency and server scalability, components on the Web are allowed to cache responses marked as cacheable. ISPs (Internet Service Providers) and organizations may deploy large caches to lower

the bandwidth used by the users of the Web; but also the client browser incorporates a cache to improve the perceived performance. Caching introduces the potential problem of data inconsistency, but the human users of the Web handle this problem easily.

Code on demand. Finally, REST allows client functionality to be extended by downloading and executing code from the server. Such code is common as scripts inside Web pages (most commonly in JavaScript), or as embedded programs such as Java applets and Flash programs. By allowing code-on-demand, the client software only needs to implement a reduced set of required features. A common example of user interface extensions through code-on-demand is in-page menus, not natively supported by HTML and Web browsers.

These constraints are all applied to the architecture of the Web, as embodied mainly in the Hypertext Transfer Protocol HTTP. Nevertheless, some of these constraints cannot be easily enforced, and it is common for Web sites to break some of them (most notably, the stateless communication constraint is often broken by using *cookies* for session maintenance), which may result in suboptimal user experience.

Further, the hypermedia aspect of the Web leads to a further pair of requirements that affect Web architecture, especially in the area of document formats:

Links and connectedness. Resources on the Web must be able to refer (link) to other resources, the user must be allowed to navigate the resulting graph of links freely.

Addressability. Stemming from the requirement for links, it is necessary that all resources are addressable. For this, REST uses URIs.

REST was designed with the human-oriented Web in focus; however, the constraints can also be applied to machine-oriented Web services. An automated, machine-oriented Web application or service is said to be *RESTful* when it uses the uniform interface (using all the methods as appropriate), when its communication is stateless, and when it enables cacheability.

In contrast to RESTful Web services, traditional SOAP-based Web services commonly only use the POST method, they use transient messages that are not cacheable, and they keep conversational sessions between the server and the client. These violations lead to tighter coupling between the client and the service, and they limit interoperability and scalability of the resulting systems.

3.4.2 HTML, RDFa, Microformats, Microdata

Majority of the content on the Web is captured in the Hypertext Markup Language (HTML [45]). For example relevant to this thesis, Web APIs are commonly described in HTML documentation. HTML is the language of Web pages, defining how content and media should be laid out in a client's browser. The language has structure similar to XML, and it defines numerous tags (that would be called *elements* in XML) to mark up the page's content.

Typical structural and layout tags are `<h1>` for a first-level headline, and `<p>` for a paragraph. HTML allows external styling to be applied to its tags, using

an attribute called `class`; for example, a particular paragraph at the beginning of an article can be marked with the class *"abstract"*, and the style can typeset the paragraph in a smaller italic font, as the abstract of the article.

The `class` attribute in effect makes it possible to annotate the semantics of the page's content. This is used in microformats (discussed below) to mark up page's content as machine-processable data.

HTML, as a hypertext language, can also mark up links between pages, for example like this:

```
<a href="http://example.com">follow this link</a>
```

The `<a>` tag with an `href` attribute signifies a hyperlink. HTML defines two attributes, `rel` and `rev`, that can be used to indicate the relationships between the page that contains the link, and the page that is linked to. For example, a page in a collection (for example a blog entry) can mark up the links to the *next* and *previous* pages in the same collection. With such markup, a browser can provide a common mechanism for navigating in a collection, without the user having to locate and click the actual link in the page. Microformats also often make use of these two attributes.

Further, HTML defines *forms*, a set of tags for various input fields, along with a mechanism for the user to submit the data they entered in the input fields to the Web application. Forms are the primary technique for users to perform interactions such as submitting a search query, purchasing products, or commenting on Web stories. Forms can be seen as rich hyperlinks that not only point to Web resources (where the data is to be submitted) but also define what data should be submitted and what HTTP method should be used for the submission. Effectively, without forms, the Web would be read-only, with browsers relegated to viewing content and following hyperlinks.

Microformats are an "adaptation of semantic XHTML[6] that makes it easier to publish, index, and extract semi-structured information" [58], an approach for annotating mainly human-oriented Web pages so that key information is machine-readable. A microformat is mainly a collection of keywords that are used as `class` names on HTML elements to indicate the type of data contained by the elements, and keywords used on hyperlinks to specify relations between resources. An HTML page with microformat annotations works in a Web browser as any other HTML page, but it also allows programmatic extraction of the contained data, regardless of the presentation structure of the page. There are already microformats for contact information, calendar events, ratings etc., supported by a variety of tools.[7]

Microformats take advantage of existing XHTML facilities such as the `class` and `rel` attributes to mark the fragments of interest in a Web page. A microformat translates the hierarchy of HTML elements into a hierarchy of objects and their properties. For example, a calendar microformat marks events with their start and end times and with the event titles, and a calendaring application that supports this microformat can then directly import the data from what otherwise looks like a normal Web page that can be syntactically *valid* HTML.

To make data from a Web page available for processing, GRDDL [35] is a

[6] HTML in XML; originally, HTML syntax differs from the rules of XML.

[7] See `microformats.org` for examples and further information about microformats.

mechanism for extracting RDF information from Web pages, particularly suitable for processing microformats. GRDDL defines a way for Web pages to point to XSLT transformations that process the page and output RDF triples. To elaborate on the above calendar microformat example, if the page includes a GRDDL transformation pointer, the calendar data will be available to any data browser, not only to applications that implement support for the particular microformat.

Since microformats use simple `class` names to denote the semantics of the content, they are exposed to the risk of naming conflicts when data in multiple formats is included in a single Web page.

Microdata is a draft specification that "allows machine-readable data to be embedded in HTML documents in an easy-to-write manner, with an unambiguous parsing model." [46] The intent of the microdata specification, in development together with HTML5 [48], is to replace the need for ad-hoc microformats, by specifying five HTML attributes, namely `itemid`, `itemprop`, `itemref`, `itemscope`, and `itemtype`, which together support a model that consists of groups of name-value pairs, known as items. While microdata does not reuse any of the existing HTML attributes, it still carries the risk of naming conflicts in data properties, similarly to microformats. Also, the use of new attributes poses a problem for validating the syntactic structure when publishing HTML documents.

Microdata is primarily convertible into JSON structured data. [47] discusses ways of converting microdata into RDF, which may include vocabulary-specific rules. Potentially, also GRDDL can be used to extract RDF from microdata in HTML documents. When microdata is standardized, it will likely be adopted to replace the use of microformats for structured data embedded in HTML pages.

RDFa is a collection of attributes and processing rules for extending XHTML to support RDF [97], and it constitutes an alternative to microformats for including machine-processable data (limited to RDF in this case) in Web pages. RDFa specifies a collection of generic XML attributes for expressing RDF data in any markup language, and especially in HTML. RDFa shares some of its use cases with microformats, but with different design principles: where microformats aim to be especially easy to use for Web content authors, RDFa is better prepared for proliferation of data vocabularies.

In contrast to microformats and microdata, RDFa avoid any risk of naming conflicts through use of XML namespaces to distinguish vocabularies, and it uses the attributes `typeof`, `about`, `rel`, `resource`, and `property` (among others) to express any RDF content. However, like microdata, RDFa also uses extension attributes in HTML and thus can pose validation problems.

3.4.3 Protocol-specific Service Descriptions, AtomPub

Standardized types of Web resources that are amenable to machine processing, such as search engines or online publishing systems (e.g. blogs), often provide machine-readable descriptions in order to facilitate capability discovery and automated interactions.

Let us first illustrate such machine-readable descriptions a simple case: search engines can be integrated in other software, for example in Web browsers in

a conveniently located search field. OpenSearch [21] is a proposed "collection of simple formats for the sharing of search results," and it defines an XML-based format for describing the interface of a search engine. With this format, Web browsers (and other systems that integrate third-party search engines) can learn about a new search engine simply by downloading its "OpenSearch description document" that specifies how search queries should be formulated for the particular engine, and what the responses will look like. In effect, browsers no longer need search-engine-specific functionalities, and conversely, search engine providers no longer need to make browser-specific efforts to be supported.

The situation can be generalized as follows: there is a well-known protocol — in the case of search engines, the protocol consists of a single *search* operation — that can be implemented by many services. Importantly, the protocol does not prescribe the form of resource URIs — it only specifies the types of resources and how they interlink. To support clients, the protocol also specifies a machine-readable description format so that services can provide various types of information about themselves — in the search-engine case, a service especially needs to say how search terms are passed to the engine, for example as URI parameters (and how the parameters are named). The machine-readable description format makes it possible for clients to adapt automatically to any service, and for the services to be accessed automatically by any client.

The Atom Publishing Protocol (AtomPub, [5]) presents a more interesting case. The protocol defines how a client can create, edit and delete resources in collections; collections are represented as Atom feeds citeatom. The primary use is for online publishing systems such as blogs, where the clients can add/update/remove articles and related media files. Like above, AtomPub defines the protocol along with a *Service Document* format that describes the service.

A typical AtomPub client is a desktop blog editor[8] that allows its user to manage their blog and write new entries with the performance and rich user interface of a native desktop application; the user can even compose entries offline and upload them when they connect to the internet again.

With the AtomPub Service Document, a server can advertise to the clients what collections it hosts. The collections of a single service are further grouped into so-called "workspaces", but AtomPub does not provide any operations for managing workspaces so they are not relevant to our discussion here.

For each collection, the Service Document can specify the URI where the collection can be accessed, and the following metadata about the collection: a title, the media types accepted as items in this collection, and a list of categories that can be assigned to items in this collection. For example, a blogging service (workspace) can have two collections: one for the text of the blog entries, with possible categories such as *work*, *link*, *fun* etc., and a second one for pictures that may accompany the blog posts. This way, when a user composes a blog entry with a picture in their desktop blogging application, the application can upload the picture in the media collection, and embed the picture's newly assigned URI (returned by the upload operation) in the entry text as it uploads the entry in the main collection.

Listing 3.1, shown earlier on page 28, contains an example AtomPub Service Document, with only a single collection for textual (Atom entry) data in it.

[8] For instance ECTO, `http://illuminex.com/ecto/`

3.4.4 WADL

As discussed above, specific protocols such as search or blog publishing can have service descriptions tailored to the structure of the protocol's resources. While this allows the descriptions to be concise and to-the-point in what they describe, the protocol-specific approach also causes a proliferation of service description formats.

Having to support a plethora of service description formats to deal with RESTful services is not a desirable situation for service registries, therefore there are efforts for a generic RESTful service description language. Chief among them is the Web Application Description Language (WADL [36]), a format that has slowly gained certain traction, stronger than any other such approach that we have seen. In this section, we give a brief overview of the main characteristics of WADL.

The top-level concept of WADL is an *application*. The language defines a Web application as "a HTTP-based application whose interactions are amenable to machine processing", typically "promoting re-use of the application beyond the browser", and "enabling composition with other Web or desktop applications". WADL recognizes that Web applications require semantic clarity in content (representations) exchanged during their use.

WADL describes an application as a set of *resources*. For each resource, WADL mainly captures its address, and the *methods* that are available on the resource. The address of a resource is specified as a URI template, with various types of parameters that can be filled in the URI and also in request HTTP headers.

A resource in WADL does not have to be fully described *in situ*, instead the description can refer to a reusable *resource type*. This way, WADL supports the reusability of parts of service descriptions.

For every method on some resource, WADL identifies the HTTP method, additional request parameters that extend those defined on the resource itself, and the request and response (input and output) data formats. Interestingly, the response types can be associated with HTTP response codes so that the client knows what to expect depending on the result of the method execution.

Finally, WADL can even point out pieces of data (both in requests and responses) that serve as links to other resources. This way, WADL can capture some information about the expected hypertext structure of the application.

Since WADL aims to be a generic service description language for RESTful services, it would be possible that if it gains adoption, formats such as AtomPub Service Document, or OpenSearch Description, would be expressed in WADL, for example as predefined resource types.

3.5 Semantic Web Technologies

This thesis presents a lightweight semantic approach to automating the use of Web services. The preceding sections of this background chapter have focused on Web and service technologies; here we look at technologies developed to make the Web semantic.

First, in Section 3.5.1 we describe the basis of the Semantic Web, the Resource Description Framework RDF. Then we touch on several ontology languages in Section 3.5.2, and in Section 3.5.3 we discuss the distinction of lightweight and heavyweight ontologies. Finally, in Section 3.5.4, we present XSPARQL, a recent technology proposed for bridging the gap between the Semantic Web with RDF, and data exchanges in XML.

3.5.1 Resource Description Framework RDF

Resource Description Framework (RDF [95] is a data model and a set of syntaxes for representing statements about resources on the Web. RDF makes extensive use of Uniform Resource Identifiers (URIs), not only for the resources that are being described, but also for the vocabularies used to describe them.

The basic construct in RDF is a *triple* $\langle s, p, o \rangle$, where s is the *subject* of the triple, i.e., the resource that is being talked about; o is the *object* of the triple, i.e., a resource or another value to which the subject is related; and p is the *predicate* of the triple, i.e., the relationship between the subject and the object.

An RDF triple object can be either a concrete resource (identified by a URI), or a so-called *blank node*, which is a resource that has no identifier. A predicate is always a concrete resource, and the object can be a concrete resource, a blank node, or a *literal value*, as we will show in an example below.

Multiple RDF triples put together form an oriented *graph*. In RDF, graphs are formally defined as sets of triples, which means that a single triple cannot repeat in the graph, and that the graph need not be connected. A *named graph* is a graph that has a URI identifier. Note that an RDF graph is free to contain statements about itself, but it is not required to do so — the identifier of the graph need not be present in the graph at all.

An RDF *document* is a serialization of an RDF graph in one of the available syntaxes, especially Notation 3[9] and Turtle (a subset Notation 3), and RDF/XML [99]. Listing 3.4 shows an example RDF graph serialized in Turtle.

The graph in the listing talks about two separate resources, one identified as `http://example.com/serviceDescription.html#svc`, and another a blank node, written as `_:x`.

Note that blank node names (like `_:x`) are only meaningful within a single RDF document — if two different RDF documents use the same blank node name, they do not automatically refer to the same blank node.

The example users a shorthand notation for URIs, called *namespace prefixes*. Instead of writing `http://www.wsmo.org/ns/wsmo-lite#Service` as the full URI of a resource, the document declares a prefix called `wl` and then uses the prefix together with the final part of the URI, i.e., `wl:Service`. A single prefix can be used for multiple different resources — our example also refers to `wl:Operation`. Namespace prefixes belong to RDF documents — a document

[9] `http://www.w3.org/DesignIssues/Notation3.html`

```
1  @prefix hr: <http://www.wsmo.org/ns/hrests#> .
2  @prefix wl: <http://www.wsmo.org/ns/wsmo-lite#> .
3  @prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
4
5  <http://example.com/serviceDescription.html#svc> a wl:Service ;
6    rdfs:label "ACME Hotels" ;
7    hr:hasOperation _:x .
8  _:x a wl:Operation .
```

Listing 3.4: Example RDF graph

must define all the prefixes it uses, different documents can define prefixes differently, and the RDF graphs represented by RDF documents are not influenced by the choice of namespace prefixes.

Turtle also allows another special-case shorthand — one can write `a` instead of `http://www.w3.org/1999/02/22-rdf-syntax-ns#type` (or `rdf:type` with the commonly-used prefix), which is a well-known predicate that puts a resource in a class of resources. Our example gives classes to both subjects: the first one belongs to the class `wl:Service` (defined later in this thesis), and the blank node belongs to the class `wl:Operation`. Effectively, the example graph talks about a service and an operation.

Finally, apart from the two `rdf:type` statements, the graph also gives a label to the service resource, and it says that the operation `_:x` is in fact an operation of the service. Note that in Turtle, multiple statements with the same subject can be given without repeating the subject, instead the predicate-object pairs are separated by semicolons. The final statement about a given subject is always marked with a period.

In addition to the Turtle syntax (and Notation 3), which we have used in the example, RDF/XML is an important alternative syntax for RDF graphs. It serializes RDF statements in XML elements, and it gives the author many options on how XML constructs can represent RDF triples. In this thesis, we exclusively use Turtle to show RDF graphs, because it is easier to read, but software tools commonly use RDF/XML because of the ready availability of XML-handling libraries.

Using URIs to identify resources and relationships in RDF removes many issues that other data formats (including XML and JSON) have with vocabulary conflicts. In effect, two RDF graphs can be *merged* simply by making a union of the two sets of triples (any triple present in both sets will only be present once in the merged set). If the two graphs both happen to give statements about some given subject resource, all the statements will be present in the merged graph. While an application may recognize the merged data as inconsistent (for instance it may be seen as a problem if two graphs give different labels to a single service), on the RDF level, the graphs can be merged without difficulty.

Merging any graphs is an aspect in which RDF differs greatly from other data formats such as XML and JSON, where two documents cannot be merged generically without knowledge of the concrete application structure of the data.

3.5.2 Ontology Languages

In this section, we briefly discuss languages for describing semantic vocabularies, focusing on those that are directly usable with RDF. In particular, we discuss RDF Schema (RDFS), Web Ontology Language (OWL), Rule Interchange Format (RIF), and Web Service Modeling Language (WSML).

RDF Schema [98] extends RDF with simple means of defining RDF classes and properties. RDFS can mainly express the following notions:

- Resource C is a *class* — it groups instances with somehow related characteristics (an instance is attached to a class using the `rdf:type` predicate mentioned above).
- Resource D is a *subclass* of a class C — all instances of D are also recognized as instances of C.
- Resource P is a *property* — it should be used as a predicate in RDF triples.
- Resource R is a *subproperty* of property P — every triple that has R as its predicate also implies a triple with the same subject and object and with P as the predicate.
- Property P has class A as its *domain* — every resource that has property P belongs to class A.
- Property P has class B as its *range* — every value of property P (a resource that is the object of a triple with property P as the predicate) belongs to class B.

In addition, RDFS defines several utility properties such as `rdfs:label` for giving resources human-oriented names, and `rdfs:seeAlso` for referring to somehow related resources.

Importantly, RDFS is intended for inference and not for consistency checking, so the forms above will imply new knowledge rather than constraint violations. Most importantly, this difference demonstrates itself in the case of domain and range descriptions: where one might understand a range with the meaning that one can only use instances of class B as values of property P (for example, the property `wl:hasOperation` should always point to an instance of the class `wl:Operation` and it would be an error for it to point to something else), RDFS will simply infer that if something is the value of P, it belongs to B (if `wl:hasOperation` mistakenly points to an instance of `wl:Service`, RDFS will infer that the instance of `wl:Service` is also an instance of `wl:Operation`). Barring datatype clashes (outside the scope of this section, see [96]), RDF and RDFS data cannot be inconsistent.

The Web Ontology Language OWL [82, 84] is conceptually an extension of RDFS for greater expressivity. OWL defines sublanguages/profiles that range in expressivity and computational complexity: In OWL 1, OWL-Lite allows expressing simple constraints and has the lowest formal complexity of the three sublanguages, OWL-DL corresponds to Description Logics and guarantees computational completeness and decidability, and OWL-Full embraces the syntactical freedom of RDF with no computational guarantees. OWL 2 defines profiles that support different use cases: “OWL 2 EL enables polynomial time algorithms for all the standard reasoning tasks; it is particularly suitable for applications where very large ontologies are needed, and where expressive power can be traded for performance guarantees. OWL 2 QL enables conjunctive queries to

be answered in LogSpace using standard relational database technology; it is particularly suitable for applications where relatively lightweight ontologies are used to organize large numbers of individuals and where it is useful or necessary to access the data directly via relational queries (e.g., SQL). OWL 2 RL enables the implementation of polynomial time reasoning algorithms using rule-extended database technologies operating directly on RDF triples; it is particularly suitable for applications where relatively lightweight ontologies are used to organize large numbers of individuals and where it is useful or necessary to operate directly on data in the form of RDF triples." [84]

OWL can express all the RDFS constructs, and it adds the following, among others:

- Class and property equivalence, resource equality and inequality, and class disjointness (where two classes cannot share instances).
- Class intersections, unions and complements.
- Property characteristics such as being transitive, symmetric, functional and inverse-functional, or a property being the inverse of another property.
- Cardinality constraints, expressing that some property (on instances of a given class) must have a given minimum (or maximum) number of values.
- Local range constraints, expressing that some property (on instances of a given class) must have values from some range class (either all the values, or at least some of them).
- Describing ontologies, including some support for versioning and reuse.

Like RDFS, OWL is also intended for inferring new knowledge rather than for consistency checking, but the added constructs of OWL make it possible to detect some types of inconsistencies. For instance, an OWL ontology can express that `wl:Service` and `wl:Operation` are disjoint classes, which means that if an instance is inferred to belong to both classes, the data must be inconsistent.

In 2010, the W3C finalized the Rule Interchange Format RIF [104], a format for exchanging rules among rule systems. It defines the syntax for capturing rules, as well as their rigorous semantics, with several *dialects* for exchanging logical axioms and/or rules with actions. The logics-based dialect is called "Basic Logic Dialect" (RIF-BLD, [103]), which corresponds to definite Horn rules with equality and a standard first-order semantics. RIF-BLD can be used to facilitate the exchange of rules that access RDF and OWL data, and extend RDFS and OWL ontologies.

In addition to the Web standards RDF, RDFS, OWL and RIF, we will also mention here one research proposal for an ontology language: Web Service Modeling Language WSML [24]. The ontology language WSML is part of the larger WSMO framework, already mentioned in Section 2.2.2.

Like OWL, WSML also defines sublanguages (called *variants*) with various levels of expressiveness: WSML-Core is the intersection of Description Logics and Horn Logic, a low-expressivity base on which all the other variants are built. WSML-DL largely coincides with OWL-DL. WSML-Flight and WSML-Rule extend WSML-Core in the direction of Logic Programming, providing a powerful rule language. Finally, WSML-Full is an umbrella that combines the DL branch with the Rule branch, but without a complete specification for its semantics.

WSML is in many ways similar to (and interoperable with) the Web standards RDFS and OWL[10], and WSML rules can potentially be exchanged using RIF. In contrast to the Web standards, however, WSML has a human-readable syntax that is tailored for direct authoring and reading of ontologies and rules. As the contributions of this thesis are not tied to any particular rule language or expressivity layer beyond simple subclass relationships, the main consideration for the rule language used in our examples is its readability. Therefore, this thesis, and especially Section 4.5, uses WSML.

3.5.3 Lightweight Ontologies

One of the aims of our work is that the resulting ontologies should be *lightweight*. In this section, we discuss what it means for ontologies to be lightweight. We use a simple one-dimensional spectrum of the level of detail in ontologies, adapted here in Figure 3.5 from a drawing by McGuinness [76].

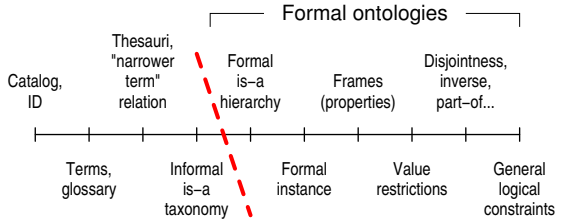


Figure 3.5: An ontology spectrum (adapted from [76])

On the left-hand side of the dashed red line there are the simpler, less formal ways to specify terms in a domain of discourse. Catalogs and glossaries define finite lists of terms, with varying degrees of ambiguity in their interpretation. Thesauri and taxonomies provide unambiguous, yet very weak, relations between the defined terms: for example, a thesaurus may specify some terms to be synonymous, or narrower-than/wider-than one another, and taxonomies define hierarchies of terms without a strict is-a relationship. For illustration, in the eCl@ss classification of products and services[11], the category "Entertainment electronics (maintenance, inspection)" is a subcategory of "Entertainment electronics", even though the task of maintenance is clearly not a piece of electronics. Such informal is-a relationships are useful when categorizing products and services, but they limit the inferences possible in a knowledge management system.

On the right-hand side of the dashed red line, the figure has ontologies with increasing levels of expressivity, starting with formal and strict is-a hierarchies and adding the notions of instances, properties, value restrictions and further logical axioms. McGuinness also defines requirements on what should be considered a formal ontology, and only specifications that fall on the right-hand side of the red line satisfy these requirements.

Ontologies close to the red line are lightweight, with limited expressivity and very efficient reasoning techniques. The far-right end of the spectrum contains

[10] For details on compliance of WSML with W3C languages see [23].

[11] eCl@ss Standardized Material and Service Classification, `http://eclass-online.com/`, version 5.0.1

ontologies that can be very detailed and powerful specifications of their respective domains, but very complex reasoning algorithms are required in order to use all the intended inferences.

The level of expressivity does not only affect the complexity of reasoning algorithms; it also influences the ease of use and reuse. A lightweight ontology is easier to learn and it imposes fewer constraints on the data and other ontologies with which it might be combined. Altogether, lightweight ontologies generally have a lower barrier of adoption than more heavyweight ontologies with a comparable purpose. In the words of Hepp [40], there is an "expressivity-community-size frontier" that severely limits the expected number of users for expressive and large ontologies.

3.5.4 XSPARQL

There are two machine-oriented data representation technologies standardized specifically for the Web: XML and RDF. XML is very popular as a data exchange format, because its hierarchical structure maps very well into programming language structures and database records; on the other hand, RDF is a less constrained graph data model designed for freely combining and querying data from diverse sources. These qualities make RDF better-suitable than XML for the Semantic Web.

XML and RDF have coexisted for nearly a decade now, and until recently, there has been a gap between XML and RDF because there were no tools that could gracefully handle transformations between the two technologies. The standardization of the RDF query language SPARQL [114] and of the XML query language XQuery [143] has spurred the development of XSPARQL [92], a combination of these two query languages that natively supports both XML and RDF, and thus enables relatively easy transformations between the two data formats.

XQuery is a functional programming language for querying and creating XML documents. A single query can reach into multiple documents (using XPath [142] expressions) and produce a single XML document as its output. Due to the important role of literal data (numbers, strings etc.) in XML, XQuery has a powerful operator and function library [145] for manipulating such data.

SPARQL is a declarative query language for RDF data; a single query can combine multiple RDF data sources and produce either a single yes/no answer, a list of variable bindings, or a new RDF graph. SPARQL provides only limited means for manipulating literal data, which is often seen as a drawback.

XSPARQL can process inputs in XML (using XPath expressions) and in RDF (using SPARQL graph patterns). Literal data from RDF is converted into the XQuery/XPath data model [144], which allows it to be subjected to the full power of the XQuery operator and function library. As the output, an XSPARQL query can produce either an XML document or an RDF graph.

In summary, XSPARQL is particularly suitable for transforming between XML and RDF (as discussed in Section 5.1.5) and for combining XML and RDF inputs. In the context of this thesis, XSPARQL is especially useful for the transformations between XML as Web service communication format, and RDF as the data model of semantic clients. This thesis contains example XSPARQL queries for such transformations between XML and RDF in Section 5.1.5.

Part II

Semantic Web Service Description Languages

Chapter 4

Lightweight Service Ontology

"A little semantics goes a long way." — Hendler's Law, 1996

In the preceding chapters, we have shown the need for Web service usage automation, and motivation for lightweight semantic Web service descriptions, supporting both WS–∗ and RESTful Web services. Here, we define a service ontology which satisfies that need, building on a simple yet sufficient model of Web services.

In Section 4.1, we analyze relevant service modeling and description specifications that underlie our WSMO-Lite service model. In Section 4.2, we analyze the requirements that need to be fulfilled by an ontology for semantic descriptions of Web services. We define the concepts of the ontology in Section 4.3, and the RDFS encoding in Section 4.4. In Section 4.5, we discuss options for capturing logical expressions that are beyond the expressivity of RDFS and OWL, and finally in Section 4.6 we show the service model in relation to the underlying service description technologies, which are discussed in the following chapters.

4.1 Web Service Model

Our service model is derived from relevant work on service modeling and description, mainly from the standard for Reference Model for SOA and from WSDL, which are detailed in Sections 3.1 and 3.3.2. These specifications are part of the WS–∗ family of standards, but Chapter 6 clearly demonstrates that our service model is nevertheless appropriate for RESTful services as well.

The OASIS consortium's industry-standard Reference Model for Service Oriented Architecture (SOA RM [100]) investigates a number of aspects of services; the top-level six (shown in Figure 3.1 on page 26) are *Service description*[1], *Visibility*, *Execution context*, *Real-world effect*, *Contract and policy*, and *Interaction*. A service in SOA is generally not a single physical artifact, instead it is a concept useful for manageability. From the point of view of a client, the physical

[1] In the text we *emphasize with italics* the terms defined by the SOA RM.

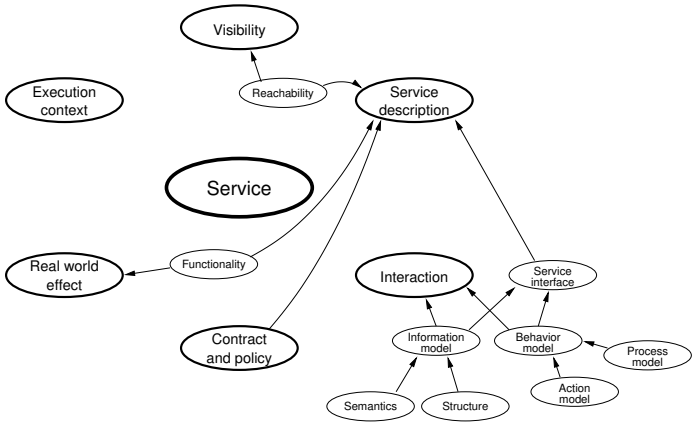


Figure 4.1: Subset of OASIS SOA RM relevant for the service ontology

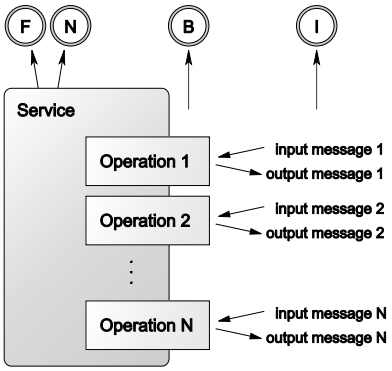
artifacts of a service are its network location (part of the *Visibility* aspect) and the service description.

Service description is key to automation, because a client only has service descriptions to guide it in selecting and using the available Web services. Figure 4.1 (a simplified combination of Fig. 3.2 and Fig. 3.3) details the parts of the SOA RM that must be captured in a machine-readable description: a service has a certain *functionality*, which implements the service capabilities that achieve the *real-world effect*; it may have additional constraints (*contracts and policies*); it is *reachable* (commonly through a computer network, at a given location that gives the service *visibility*); and it has a *service interface*, made up of an *information model* and a *behavior model*, both involved in *interactions* between the service and its clients.

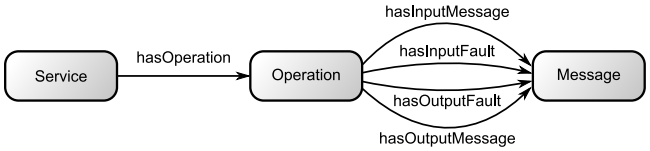
The *information model* of a service interface characterizes the information that may be exchanged with the service. It specifies the *semantics* (i.e., the meaning) of the data, and its *structure* and form. The *behavior model* describes the *actions* (operations) that may be invoked on the service, and the *process* that defines the possible order(s) in which the actions make sense. In the words of the SOA RM, "the process model characterizes the temporal relationships and temporal properties of actions and events associated with interacting with the service." The interaction side of the service is the most detailed part of service description in SOA RM because it represents the majority of the service's interface for use by clients.

For the purpose of tool support and automation of Web service use, a service description must capture the mentioned aspects of the service. Information about message structures, communication protocols and message exchange patterns, and physical service access points (service *reachability*), is already part of technical descriptions such as WSDL; in this work we do not study the semantics of these underlying technical descriptions. In our case, for automation using semantics, we want to represent the semantics of the remaining aspects not covered in the underlying technical descriptions.

Inspired by the distinctions made by Sheth [108], we group the remaining aspects of SOA RM service description into four orthogonal parts:



(a) Structural view with Functional, Nonfunctional, Behavioral and Information semantics



(b) Conceptual view

Figure 4.2: WSMO-Lite Web service description model

- Functional descriptions specifies service *functionality*, that is, what a service can offer to its clients when it is invoked.
- Nonfunctional description defines any *contract and policy* parameters of a service, or, in other words, incidental details specific to the implementation or running environment of the service.
- Behavioral model specifies the *actions* and the *process* (in other words, the operations and their ordering) that a client needs to follow when consuming a service's functionality.
- *Information model* defines the input, output and fault messages of the actions.

In effect, there are four **types of semantics** that must be covered by a service ontology: **functional, nonfunctional, behavioral and information semantics.**

Figure 4.2 shows two different views of the service description model, extracted from the above analysis. Part (a) of the figure shows the structure of a service description, separating the non-semantic service structure from the semantic aspects in the annotations on top. Part (b) of the figure shows the concepts that make up the service model, along with the relationships between them.

The non-semantic service structure can be seen as a straightforward simplification of the structure of WSDL. It starts with the Web service, which offers a

number of operations. Every operation has an input and/or an output message, and possibly also some fault messages[2].

On the semantic level above this structure, functional and nonfunctional semantics are directly properties of a service. Behavioral semantics tie to service actions/operations. Finally, information semantics tie to the data that a service communicates with — to the input, output and fault messages of the operations.

The structure of this model can be seen as a high-level abstraction of the component structure of WSDL, and we detail a concrete mapping from WSDL into our model in Chapter 5. Still, our model applies equally well to RESTful Web services; Chapter 6 details the mapping between sets of hypermedia resources (the structure of RESTful Web services) and sets of operations (our service model).

4.2 Requirements for a Service Ontology

In accordance with most ontology development methodologies to date (cf. [34, 115]), an ontology should have a clear goal, domain and scope, which together shape the requirements that the ontology must fulfill.

The goal of our service ontology is to enable automation of discovery and use of Web services. The domain is the service model described in Section 4.1, which covers both WS–∗ and RESTful Web services. The scope of the service ontology is modeling the four types of semantics (functional, nonfunctional, behavioral and information) to the extent necessary to support automation of the tasks involved in discovery and use of Web services. Additionally, because Web services can cover any kinds of business functionalities, the service ontology must be able to make use of domain-specific ontologies.

In the subsections below, we list requirements that come from the domain (service model), the intended application of the ontology (SWS automation), and from the environment where the ontology will be used. The last subsection summarizes the requirements in a single list.

4.2.1 Domain Requirements

An ontology is a conceptualization of a domain. Therefore, the domain modeled by the ontology is the first source of requirements; it defines what terms should be present in the ontology.

The service model described in Section 4.1 is a common view on Web services, whether they be RESTful or use the WS–∗ technologies. The existence of such a common view allows us to treat both kinds of Web services in the same way for most of the automation tasks performed by a Semantic Execution Environment (SEE).

In our work, we do not attempt to unite the non-semantic technologies of the two different kinds of Web services. Consequently, the technical aspects are out of scope of the service ontology, which serves as a bridge between the non-semantic Web service descriptions (WSDL for WS–∗ services, links and forms for RESTful services), and the functional, nonfunctional, behavioral and information semantics which provide a unified definition of the available Web services.

[2] Operation message exchange patterns are discussed in Section 3.3.2 (Page 32).

Therefore, the service ontology needs to cover the following terms from our service model:

- Web service
- operation
- input message
- output message
- fault messages (in and out)

4.2.2 Application Requirements

An ontology supports an application; in our case the application is a Semantic Execution Environment that automates some tasks involved in the discovery and use of Web services. The application has two major parts—its functionality and its users—both of which provide requirements on the ontology. The functionality is to automate the various tasks, and the users are Web service designers, SEE implementors, and the clients. In this section, we first analyze the functionality requirements, and then we follow with the user requirements.

Functionality Requirements

In Section 2.1, we enumerated the Web service usage tasks that can be (partially or fully) automated with suitable machine-readable descriptions. In Section 4.1, we distinguish four aspects of Web service descriptions: functional and nonfunctional descriptions, behavior and information model. They should be expressed semantically within our service ontology.

The different automation tasks have varying requirements on the extent of semantic descriptions that should support automation. In the following list, we go through the service usage tasks and discuss what semantic annotations are necessary for each task — these requirements stem from our definitions of the various tasks in Section 2.1.

- Service discovery finds services that can functionally satisfy a given goal. Therefore, functional semantics must be expressible in our service ontology. Note that this is an intentionally narrow view of service discovery, which helps us distinguish it from the other tasks.

- Offer discovery communicates with a discovered service and retrieves information about any available offers. Offer discovery deals with the data, therefore it requires information semantics; and it needs to invoke the service's operations in the appropriate sequence, therefore it needs behavioral semantics. Offer discovery incorporates the operation invocation task described below.

- Ranking and filtering for service selection can use any available information, but does not strictly require any. However, most common ranking/filtering parameters fall into the category of nonfunctional semantics, which is therefore marked as required.

- Composition combines functional problem decomposition with service discovery and optionally also ranking/filtering, therefore it requires functional semantics and can benefit from any other information.

- Operation invocation exchanges messages with the service, therefore it needs information semantics to handle the message data.
- Service invocation attempts to execute the selected service to achieve the given goal; therefore it needs behavioral semantics in order to sequence the necessary operation invocations appropriately. Naturally, service invocation also incorporates operation invocation.

Table 4.1 summarizes what descriptions are required (•) or useful but optional (○) for the various tasks.

Service Task	**F**	**N**	**B**	**I**
Service discovery	•			
Offer discovery			•	•
Ranking and filtering	○	•	○	○
Composition	•	○	○	○
Operation invocation				•
Service invocation			•	

Table 4.1: Service usage tasks and the necessary semantics (Functional, Nonfunctional, Behavioral and Information)

We can see that in order to be able to automate all the mentioned tasks, which is the intended application functionality of our service ontology, all four kinds of service semantics are required. The ontology must support the use of domain ontologies for expressing the concrete semantics of particular services, and it must provide a mechanism for connecting these domain ontologies with the service model. The following list summarizes the application functionality requirements, in terms of what must be supported by the service ontology:

- functional semantics
- nonfunctional semantics
- behavioral semantics
- information semantics
- plugging in domain ontologies to express the concrete semantics

User Requirements

There are three types of intended users of the service ontology and the associated SEE application: service designers, SEE implementors, and clients. Here, we analyze these types of user with the aim of identifying the requirements they have on the service ontology.

Service designers create Web services and provide their semantic descriptions. In current practice, the semantic descriptions are often added after a service is created and running, and may even be added by third parties independent from the service provider (for instance, in the evaluation of this thesis, we add semantic annotations to the descriptions of existing public Web services for the purposes of demonstrating the use of our service ontology). Nevertheless, the distinction between the entity who provides a service and the one who provides the semantic description is immaterial to this thesis, therefore we use the simple and understandable term “service designer” for the combined role.

Service designers need to create the technical description (e.g. WSDL) along with the semantic one. This technical description does not express the semantics of the service, but it does describe the structure, corresponding to the service model (Section 4.1). The semantic description can either be embedded in this technical description (annotating it with semantic information, for instance using SAWSDL, which is the basis for our work), or it can be expressed externally. However, since the service ontology encompasses the service model, an external semantic description would duplicate some information from the technical description, introducing space for inconsistencies, especially when the system evolves and the descriptions need to be updated. Therefore we will focus on embedding the semantic descriptions within the technical ones. Naturally, Web service descriptions with added semantic annotations should not break any such uses of the technical descriptions that do not take the semantic annotations into account.

The service designers' requirements, both supported by SAWSDL, can be summarized in the following list:

- The service ontology must be embeddable in the selected technical Web service description technologies.
- The semantic annotations must use backwards-compatible extension mechanisms in the technical description languages.

The implementors create a SEE, the system that uses the semantic descriptions in the process of (semi)automatically fulfilling a client's goal. A SEE is generally decomposed into a set of components that implement the various automation tasks, as described in Section 2.1. The different tasks require different parts of the semantic descriptions, therefore the four parts of the semantic descriptions (functional, nonfunctional, behavioral and information semantics) should be easily separable from each other. In other words, the service ontology should not introduce unnecessary dependencies between its components, resulting in the following requirement:

- The four types of semantics should be expressible independently.

The clients use a SEE to find and use Web services. In order for the SEE to be able to understand and fulfill the client's goal, the goal must be formulated in a machine-processable form. The client may formulate the goal manually or using some end-user tools; and a goal may use any ontologies available to the client. Nevertheless, the client may want to use the ontologies referenced in the descriptions of the available Web services, therefore in order to enable clients to easily find these ontologies, we can formulate the last user requirement as follows:

- The service ontology should not introduce unnecessary indirections between the underlying technical descriptions and the domain ontologies used for the semantic annotations.

4.2.3 Environmental Requirements

Beside the domain and the application intended for our ontology, we also need to consider the context in which the ontology is intended to be used. The context has two major aspects—the pre-existing SWS ontologies and the runtime environment of our concrete SEE implementation.

Semantic Web service automation is already addressed by frameworks such as WSMO and OWL-S. These frameworks are by design detached and independent from the underlying Web service technologies, and they are complex. The preceding section already addresses the first concern with the requirement that our semantic descriptions must be done as annotations of existing Web service description technologies. The second concern, the complexity of the existing approaches, can be viewed from two angles—the complexity for the users to learn the ontology (also addressed in the preceding section), and the complexity of the reasoning algorithms required to work with the ontology.

Part of the reasoning complexity when working with ontologies stems from axioms that enforce consistency constraints, and axioms that infer knowledge implied by the data. Typical axioms of these types are, respectively, class disjointness, and the domains and ranges of properties. The service model part of our ontology is based on underlying technical descriptions such as WSDL, which generally have strong validation support. Relying on this level of validation, we can assume that instances of the service ontology will be created by automatic mapping from valid semantically annotated technical descriptions (as described in Chapters 5 and 6). Therefore, the ontology need not formalize all the constraints that the service model implies, and it can avoid using higher levels of expressivity, with the corresponding reasoning complexity. This becomes the final requirement:

- The service ontology should not formalize constraints that are guaranteed to be satisfied by data generated from valid underlying technical descriptions.

For example, the ontology must have terms for services and operations (coming from the service model), and a property that ties services to the operations they contain, but it need not express the constraints that the class of services is disjoint from the class of operations (no service is an operation); and that a service can only contain operations, and not other services.

4.2.4 Summary of the Requirements

The following list summarizes the requirements we put on our service ontology and on the knowledge representation formalization used to encode the ontology.

1. Domain requirements:
   (a) The service ontology must cover the following concepts: Web service, operation, input message, output message, fault messages.
2. Application requirements:
   (a) The service ontology must support the expression of the functional, nonfunctional, behavioral and information semantics of Web services.
   (b) The service ontology must allow plugging in domain ontologies to express the concrete semantics.
   (c) The service ontology must be embeddable in the selected technical Web service description technologies.

(d) The semantic annotations must use backwards-compatible extension mechanisms in the technical description languages.

(e) The four types of semantics should be expressible independently.

(f) The service ontology should not introduce unnecessary indirections between the underlying technical descriptions and the domain ontologies used for the semantic annotations.

3. Environment requirements:

   (a) The service ontology should not formalize constraints that are guaranteed to be satisfied by data generated from valid underlying technical descriptions.

In the following sections, we identify these requirements as 1a, 2a ... 2f, 3a.

4.3 Service Ontology Conceptualization

In this section, we present the concrete concepts that comprise our ontology, along with the relationships between these concepts. We start by listing the concepts that capture the service model from Section 4.1, then we present the concepts of the various kinds of service semantics, and finally we define the pointers from the components of the service model to the semantics. Section 4.4 then presents the ontology as captured in RDF and RDFS.

4.3.1 Service Model

The service model, discussed in Section 4.1 and illustrated in Figure 4.2, represents the basic structure of Web services. It serves as a common representation of the underlying technical descriptions, parsed in terms of this model for processing in a SEE. Chapters 5 and 6 describe the mapping into our service ontology from WSDL and from the various description technologies for RESTful services.

Our service model is rooted in the concept of *Service*. While some technologies, such as WSDL, abstract the interface of a service from the service itself, so that the interface definition is reusable between multiple services, in our service model the interface would only constitute an indirection between a service and its semantic annotations, with no added value. Therefore, our service model does not separate the interface from the service.

A Web service is a collection of operations. This brings the concept *Operation* and the relation *has operation* linking a service to its operations. Every useful service has at least one operation, since the clients interact with services by invoking their operations, and hence, no client could use a service without operations.

Operations have input and output messages (with "inputs" and "outputs" viewed from the side of the service), and potentially input and output faults as well. Which of these messages should be present depends on the operation's *message exchange pattern*[3]: the most common one is *request-response*, where an operation has a single input (request) message followed at run-time either by a

[3]See Section 3.3.2 (p. 32) for a description of operation message exchange patterns in WSDL.

single output (response) message or by an output fault. Some advanced message exchange patterns can also include input faults (faults going from the client to the service), as discussed for instance in [134, 80].

Messages are pieces of data; the data structure or semantics may be independent of whether the message is used as an input, output, or fault message. For instance, a product information document can be the input of a catalog item update operation, and the output of a product detail listing operation. Therefore, we model the messages with the concept *Message*, and we have four different relations between an operation and its messages, namely *has input message*, *has output message*, *has input fault*, *has output fault*.

4.3.2 Service Semantics

Informally, the four types of service semantics are represented in our service ontology as follows:

- Information semantics are represented using domain *ontologies*, which are also involved in the descriptions of the other types of semantics.
- Functional semantics are represented as *capabilities* and/or functionality *classifications*. A capability defines *preconditions* which must hold in a state before the client can invoke the service, and *effects* which hold in a state after the service invocation. Functionality classifications define the service functionality using some classification ontology (i.e., a hierarchy of *categories*).[4]
- Nonfunctional semantics are represented using an ontology that semantically captures some policy or other nonfunctional properties.
- Behavioral semantics are represented by annotating the service operations with functional descriptions, i.e., capabilities and/or functionality classifications.

We formalize these terms below. Mainly, we define ontology, which is the fundamental building block for all types of semantic descriptions; our definition is only as specific as necessary to capture core ontology elements needed for the purpose of this work, but it is also general enough for us to be able to plug in various knowledge representation languages, such as RDFS, OWL, WSML or RIF (see Section 3.5.2, as appropriate in any particular system.

Further, we formalize classification and capability, which serve for functional description of services and operations. The formalizations allow us to be explicit about the terms we define, and they are also useful for defining algorithms that process WSMO-Lite descriptions, such as the ones shown in Chapter 7 of this thesis.

Definition 4.1 (Ontology) An ontology Ω is a four-tuple

$$\Omega = (C, R, E, I)$$

[4] The distinction of capabilities and categories is the same that is made by Sycara et al. [117] between "explicit capability representation" (using taxonomies) and "implicit capability representation" through preconditions and effects.

where C, R, E, I are sets that, in turn, denote classes (unary predicates), relations (binary and higher-arity predicates), explicit facts — instances of classes and relations (extensional definition), and axioms (intensional definition) that describe how instances are inferred.

A particular axiom common in ontologies is the *subclass* relationship between two given classes c_1 and c_2: if c_1 is subclass of c_2 (written as $c_1 \subseteq c_2$), every instance of c_1 is also an instance of c_2. In general, the subclass relationship forms a partial order on the set of classes (it is a transitive, reflexive and antisymmetric binary relation). To indicate that an ontology contains a subclass axiom between the given classes c_1 and c_2, we write $(c_1 \subseteq c_2) \in I$. Along with the subclass relationship, ontologies may also contain a *subrelation* relationship ($r_1 \subseteq r_2$), defined analogously.

Definition 4.2 (classification) A classification $\Omega_C(c_0) = (C, R, E, I)$ is an ontology whose classes (members of C) represent categories of things. Classification categories form a subclass (subcategory) hierarchy[5] with a single root c_0, i.e., every class in the ontology is either directly a subclass of c_0 (as captured by the subclass axioms within I), or it is a subclass by transitivity through a finite sequence of other classes:

$$\begin{aligned} \forall c \in C: \quad & (c \subseteq c_0) \in I \ \vee \\ & \exists c_1, \ldots, c_n \in C: \quad \forall i \in \{0, \ldots, n-1\} : (c_{i+1} \subseteq c_i) \in I \wedge \\ & \qquad\qquad\qquad\qquad (c \subseteq c_n) \in I \end{aligned}$$

For the purposes of describing the different kinds of service semantics, we distinguish several sub-types of ontologies: an information model ontology (an ontology used as an information model in a service description) is denoted as $\Omega^I \equiv \Omega$; a functionality classification ontology with root $c_0 \in C$ (whose classes form a taxonomy of service functionalities), denoted as $\Omega^F(c_0) \equiv \Omega_C(c_0)$; and an ontology for nonfunctional semantics as $\Omega^N \equiv \Omega$, whose instances (members of E) are concrete nonfunctional descriptions.

Definition 4.3 (capability) A capability is a three-tuple

$$\begin{aligned} K &= (\Sigma, \phi^{pre}, \phi^{\mathit{eff}}) \\ \Sigma &\subseteq V \cup C \cup R \cup E \end{aligned}$$

where K (kappa) represents the capability, Σ is a set of identifiers of elements from C, R, E of some ontology Ω^I complemented with a set of variable names V; ϕ^{pre} is a *precondition* which must hold in a state before the service (or operation) can be invoked, and ϕ^{eff} is the *effect*, an expression which is expected to hold in a state after the successful invocation. Preconditions and effects are defined as logical statements over members of Σ.

[5] Note that it may also be practical in some systems to use less-formal SKOS [111] concept schemes with hierarchies of *broader* and *narrower* concepts: the SKOS *narrowerTransitive* property would replace the subclass axiom $\subseteq$, and a SKOS *top concept* would serve the function of the classification root.

4.3.3 Semantic Annotations of the Service Model

The semantic concepts defined in the preceding section are used to express semantics of concrete services. The resulting semantic descriptions are used to annotate the underlying non-semantic descriptions (such as WSDL) using the standard SAWSDL properties *model reference*, *lifting schema mapping* and *lowering schema mapping*.

A model reference can be used on any component in the service model to point to the component's semantics. In WSMO-Lite, a model reference on a service can point to a description of the service's functional and nonfunctional semantics; a model reference on an operation points to the operation's part of the behavioral semantics description; and a model reference on a message points to the message's counterpart(s) in the service's information semantics ontology. Multiple values of a model reference on a single component all apply to the component; for example, a service can have some nonfunctional properties, pointers to functionality categories, and preconditions and effects which together make up the capability of the service. Each concrete value is always identified with a URI.

The lifting and lowering schema mapping properties are used to associate messages with appropriate transformations between the underlying technical format such as XML and a semantic knowledge representation format such as RDF. Both properties take as values the URIs of documents that define the lifting or lowering transformations. In the lifting and lowering schema mapping properties, multiple values specify alternative transformations. The client is free to choose the alternative it will use, likely depending on what transformation languages it supports.

Table 4.2 formalizes the content of the annotations on our service model. The first column specifies the service model component that is being annotated, the second column specifies the annotation property (model reference, lifting or lowering schema mapping), and the third column specifies what value the annotation can take. The fourth column (Context) shows where the value comes from, using the definitions from Section 4.3.2, and finally, the fifth column (Type) shows which of the four types of semantics this annotation describes: Functional (F), Nonfunctional (N), Behavioral (B) and Information semantics (I).

The values of the lifting and lowering schema mapping properties are formalized as $f(data) \rightarrow X$ and $g(X) \rightarrow data$, where $X \subseteq E$ represents a set of data instances of some information model ontology Ω^I. The functions f and g are transformations between sets of ontological instances (X) and their representations in the underlying technical format, denoted as *data*. We do not constrain the underlying technical formats and the form of the transformation functions, therefore, the terms *f, g* and *data* are not formalized any further.

4.4 The Service Ontology in RDFS

In the preceding section, we discuss the concepts present in our service ontology. Here, we materialize the ontology in the most basic Web ontology language, RDFS [98]. Listing 4.1 presents the ontology, serialized in Turtle (cf. Section 3.5.1) syntax. Figure 4.3 shows the ontology in a graph form, where the

Svc. model	Annotation type / value		Context	Type
Service	*mref*	ϕ^{pre} or ϕ^{eff}	$K = (\Sigma, \phi^{pre}, \phi^{eff})$	F
Service	*mref*	$x \in C$	$\Omega^F(c_0) = (C, R, E, I)$	F
Service	*mref*	$x \in E$	$\Omega^N = (C, R, E, I)$	N
Operation	*mref*	ϕ^{pre} or ϕ^{eff}	$K = (\Sigma, \phi^{pre}, \phi^{eff})$	B
Operation	*mref*	$x \in C$	$\Omega^F(c_0) = (C, R, E, I)$	B
Message	*mref*	$x \in C \cup R$	$\Omega^I = (C, R, E, I)$	I
Message	*lift*	$f(data) \rightarrow X \subseteq E$	$\Omega^I = (C, R, E, I)$	I
Message	*lower*	$g(X \subseteq E) \rightarrow data$	$\Omega^I = (C, R, E, I)$	I

Table 4.2: Service model annotations with SAWSDL properties

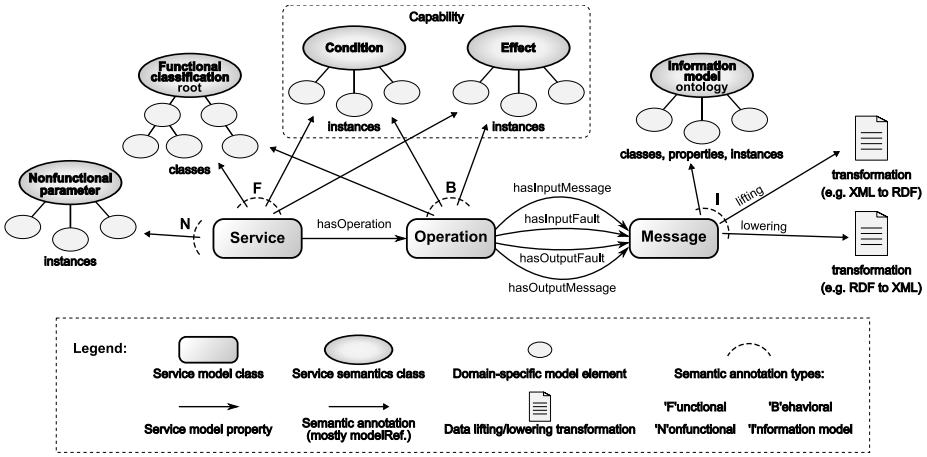


Figure 4.3: The structure and use of our service ontology, annotating the service model from Fig. 4.2(b)

centrally-located components of the service model are annotated with pointers to domain-specific semantic descriptions that fit the service semantics classes defined in WSMO-Lite.

In the remainder of this section, we define each of the classes and properties present in the ontology. In the interest of simplicity of the RDF form of actual concrete semantic service descriptions, the classes for expressing service semantics are not a straightforward implementation of the formal terms (such as *classification*, *capability*, or *ontology for nonfunctional semantics*). We discuss below some of the considerations that led to the proposed form of the ontology classes.

Lines 2–6 introduce the namespace prefixes used in the listing. The namespace (and location) for our service ontology is `http://www.wsmo.org/ns/wsmo-lite#`, and its namespace prefix is `wl`.[6]

Lines 9–26 contain the RDFS classes and properties for the service model (Section 4.3.1), lines 29–31 are the SAWSDL properties used for annotating

[6]Section 4.6 discusses the relationship between the ontology and the concrete Web service description mechanisms SAWSDL and MicroWSMO shown in Chapters 5 and 6.

```
1  # namespace declarations (this is a comment)
2  @prefix owl: <http://www.w3.org/2002/07/owl#> .
3  @prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
4  @prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
5  @prefix sawsdl: <http://www.w3.org/ns/sawsdl#> .
6  @prefix wl: <http://www.wsmo.org/ns/wsmo-lite#> .
7
8  # service model classes and properties
9  wl:Service a rdfs:Class .
10 wl:hasOperation a rdf:Property ;
11     rdfs:domain wl:Service ;
12     rdfs:range wl:Operation .
13 wl:Operation a rdfs:Class .
14 wl:hasInputMessage a rdf:Property ;
15     rdfs:domain wl:Operation ;
16     rdfs:range wl:Message .
17 wl:hasOutputMessage a rdf:Property ;
18     rdfs:domain wl:Operation ;
19     rdfs:range wl:Message .
20 wl:hasInputFault a rdf:Property ;
21     rdfs:domain wl:Operation ;
22     rdfs:range wl:Message .
23 wl:hasOutputFault a rdf:Property ;
24     rdfs:domain wl:Operation ;
25     rdfs:range wl:Message .
26 wl:Message a rdfs:Class .
27
28 # SAWSDL properties (repeated here for completeness)
29 sawsdl:modelReference a rdf:Property .
30 sawsdl:liftingSchemaMapping a rdf:Property .
31 sawsdl:loweringSchemaMapping a rdf:Property .
32
33 # classes for expressing service semantics
34 wl:Ontology a rdfs:Class ;
35     rdfs:subClassOf owl:Ontology .
36 wl:FunctionalClassificationRoot rdfs:subClassOf rdfs:Class .
37 wl:NonfunctionalParameter a rdfs:Class .
38 wl:Condition a rdfs:Class .
39 wl:Effect a rdfs:Class .
```

Listing 4.1: Service Ontology, captured in Turtle syntax

service descriptions (Section 4.3.3), and lines 34–39 contain classes that capture the various kinds of service semantics (Section 4.3.2). Below, we describe all the classes and properties; examples are shown in the following chapters.

wl:Service (line 9) is a class of Web services, the cornerstone of the service model. Note again that instances of the service model classes are not expected to be authored directly; instead, the underlying technical descriptions are parsed in terms of this ontology for processing in a SEE, as described in Chapters 5 and 6.

wl:hasOperation (lines 10–12) is a property that links services with their operations.

wl:Operation (line 13) is a class of Web service operations. Every operation belongs to a service, and it has at least one message (see below).

wl:hasInputMessage, wl:hasOutputMessage (lines 14–19) are properties that link Web service operations with the messages that are exchanged during operation invocation.

wl:hasInputFault, wl:hasOutputFault (lines 20–25) are properties that link Web service operations with the faults that may occur during operation invocation.

wl:Message (line 26) is a class whose instances represent messages exchanged by Web services and their clients. In WSDL, a message is associated with an XML schema definition of its structure, but in general, our service model here simply needs to represent the *presence* of a message, as its structure is only relevant when processing the lifting or lowering transformations.

SAWSDL properties (lines 29–31) are shown here for completeness; they are defined in the RDF mapping section[7] of the SAWSDL specification.

wl:Ontology (lines 34–35) is a class that serves to mark an information model ontology Ω^I. Similarly to owl:Ontology from the standard Web Ontology Language OWL [82], wl:Ontology allows for meta-data such as comments, version control and inclusion of other ontologies. wl:Ontology is a subclass of owl:Ontology, restricted only to ontologies intended to capture a service information model, as opposed to other kinds of ontologies.[8]

The class wl:Ontology can be used by tools for authoring semantic service descriptions, for instance to primarily suggest explicitly-marked information ontologies when annotating data schemas.

[7] `http://www.w3.org/TR/sawsdl/#rdfmapping`

[8] We have not investigated the possibility that all ontologies would be potentially useful as service information models, in which case, wl:Ontology would be equal to owl:Ontology.

wl:FunctionalClassificationRoot (line 36) is a class that marks the roots of service functionality classifications. In other words, for every $\Omega^F(c_0)$, the root c_0 is an instance of this class. All subclasses of c_0 are included in the particular classification.

Instead of marking the root class, it would be possible to mark the whole containing ontology as a classification, following Definition 4.2. However, in Semantic Web languages, ontology elements are only weakly tied to their containing ontologies. It is much easier to check that a class is a subclass of a functional classification root (the root being a class that is an instance of wl:FunctionalClassificationRoot) rather than checking that a class is defined by a classification ontology. An annotation tool can simply suggest all functional classification root classes and their subclasses when creating functional annotations.

wl:NonfunctionalParameter (line 37) is a class of concrete, domain-specific nonfunctional parameters. For a particular ontology of nonfunctional semantics Ω^N, its instances (members of E) would be instances of this class.[9] As above, we mark concrete nonfunctional parameters rather than the containing ontologies, for the same reasons.

wl:Condition, wl:Effect (lines 38–39) together form a *capability* in a functional annotation. Instances of these classes are expected to contain some logical expressions. The logical expression of a precondition should be satisfied before a service or an operation is invoked, and the logical expression of an effect is expected to hold after the service or the operation succeeds. In common with OWL-S, the WSMO-Lite service ontology does not specify the concrete language for the logical expressions, or their processing. Both WSMO-Lite and OWL-S allow logical expressions to be specified in any suitable language, such as RIF, WSML, SWRL [44] and KIF [60], and embedded in RDF semantic descriptions as literals. To show a concrete example, Section 4.5 below defines the processing of preconditions and effects specified using the language WSML.

The precondition and the effect implicitly make up a capability whose set Σ of terms and variables can be deduced automatically by inspecting the logical expressions. We do not model the capability itself in the RDF ontology, as as it would be an unnecessary indirection between the service (or operation) and its capability's precondition and effect.

In order to create or reuse domain-specific service ontologies on top of the presented Service Ontology, any ontology language with an RDF syntax can be used. This openness preserves the choice of language expressivity according to domain-specific requirements.

[9] In our service ontology, we place no further restrictions on nonfunctional parameters; research in this area, which is out of scope of this thesis, has not yet converged on a common set of properties that nonfunctional parameters should have.

4.5 Using WSML Logical Expressions in Service Capabilities

The service ontology does not prescribe any concrete language for capturing the logical expressions in the preconditions and effects that make up service or operation capability descriptions, nor does it specify any concrete evaluation environment for them. In this section, we specify how the language WSML [24, 136] can be used to capture the preconditions and effects of Web services, and how they are to be evaluated.

In accord with the WSMO-Lite ontology, WSML-based preconditions and effects are represented as RDF resources of type wl:Condition or wl:Effect respectively. Every WSML-based precondition or effect resource has a single value of the property rdf:value, pointing to a literal of the data type wsml:AxiomLiteral.

Definition 4.4 (wsml:AxiomLiteral) The data type wsml:AxiomLiteral[10] is defined as follows: the lexical space contains strings that follow the grammar production

axiomliteral = namespace? expr

where the grammar productions namespace and expr respectively define the syntax for declaring namespaces and for logical expressions, as specified in [136]. The value space of this data type consists of valid WSML logical expressions.

For evaluating a precondition of a Web service or of an operation, the knowledge base contains data from the current user goal (if any), and any other background knowledge available to the evaluator. For evaluating an effect of a Web service, the knowledge base also contains any data that has arrived from the service during its invocation, and any variable bindings introduced by the evaluation of the precondition(s).

Listing 4.2 shows a simple precondition for a hotel reservation service in Rome, Italy; the service can only be invoked for requests for accommodation in Rome, and only for up to ten guests. If the logical expression of an effect of this service would use the variables *?request, ?guests, ?city*, they would be bound to the same values as when the precondition is evaluated.

```
1  ex:RomaHotelsPrecondition a wl:Condition ;
2        rdf:value """
3           namespace _"http://example.org/onto#"
4           ?request [ numberOfGuests hasValue ?guests,
5                      hotel hasValue ?hotel ] memberOf ReservationRequest
6           and ?guests =< 10
7           and ?hotel [ locationCity hasValue ?city ] memberOf Hotel
8           and ?city = "Rome" .
9        """^^wsml:AxiomLiteral .
```

Listing 4.2: Example service precondition in WSML

[10]The prefix wsml: represents the namespace `http://www.wsmo.org/wsml/wsml-syntax#`

4.6 Semantic Web Service Description Layering

In this chapter, we have defined a service ontology intended for semantic descriptions of Web services, yet we have left this ontology free-floating (ungrounded in any concrete Web services technologies), with the service model part representing an abstraction of the concrete existing non-semantic Web service description technologies (such as WSDL). This is intentional, as this ontology is all that is necessary for a Semantic Execution Environment to do many automation tasks; the non-semantic descriptions are necessary only when actually communicating with a Web service.

Nevertheless, our semantic descriptions of Web services are based on the non-semantic, technical description languages. In contrast, the pre-existing SWS technologies such as OWL-S and WSMO both have the semantic descriptions separated from the technical ones, connected by links called *grounding*.

The following two chapters of this thesis discuss how WSMO-Lite fits on top of two semantic Web service description languages: SAWSDL for semantic annotations of Web services described in WSDL, and MicroWSMO for semantic description of RESTful Web services. Both chapters define how concrete service descriptions with semantic annotations are transformed into the service ontology described in this chapter. This semantic form (as RDF data) is required for processing in a Semantic Execution Environment (SEE), which implements the semantic automation algorithms, such as described in Chapter 7 of this thesis.

Consequently, there is no need for service designers to author manually the RDF form of the service description; this is only necessary for the actual pieces of the description of the service semantics, such as service capabilities and non-functional properties.

Figure 4.4 shows the layering of the various levels of service description. The technical descriptions are at the bottom, annotated with semantics. The annotation technology (SAWSDL or hRESTS/MicroWSMO) dictates how the description maps into our service model. The semantic annotations point to actual service semantics, which can then be used for automation in the SEE.

The figure also highlights how the common service model applies equally to the two main Web service technologies — the so-called "WS–*" stack based on SOAP and WSDL, and described semantically with WSMO-Lite; and the RESTful service technologies based on HTTP and HTML, described semantically with MicroWSMO.

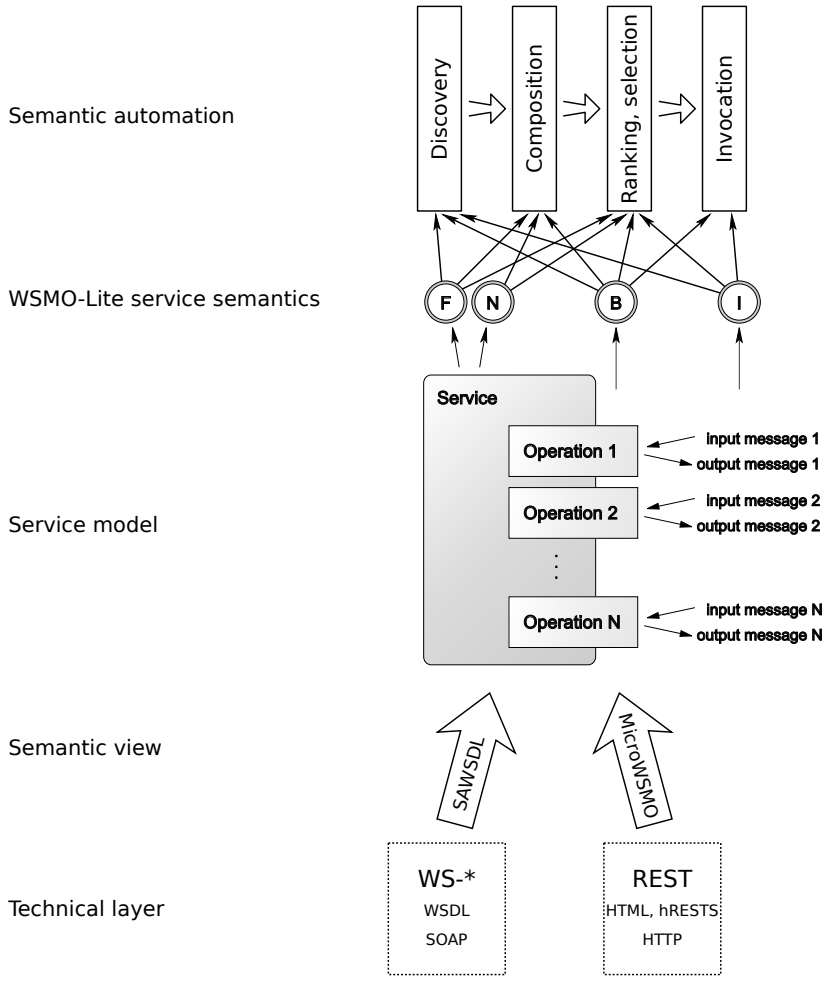


Figure 4.4: Semantic Web service description layering

Chapter 5

Annotating WS–∗ Services with SAWSDL and WSMO-Lite

The previous chapter defines WSMO-Lite, a lightweight ontology for semantic description of Web services. Our ontology, however, is only a part of a semantic Web service description; the lightweight approach to semantic description requires that we build on existing standard technologies that are already in place for non-semantic service description. In this chapter, we define the use of WSMO-Lite on top of SAWSDL and the industry-standard Web Services Description Language (WSDL).

First, in section 5.1 we detail how WSDL descriptions are annotated with the WSMO-Lite semantics, along with some examples. The way we annotate WSDL implies a mapping to the WSMO-Lite service model; in order to avoid any ambiguity, we specify the mapping in Section 5.2. Section 5.3 then discusses the deployment options for the ontologies that define the semantics of concrete services, and in Section 5.4, we analyze some possibilities for validating WSMO-Lite-based annotations of WSDL service descriptions.

5.1 Annotating WSDL with SAWSDL

The WSMO-Lite service model, based on WSDL and SAWSDL, intentionally simplifies the structure of WSDL. In Chapter 4, we defined how the service model can be annotated with semantics, but that does not directly translate to how the annotations should apply to the more complex model of WSDL. Here we therefore detail how WSMO-Lite is used in the full WSDL structure.

We focus on how service descriptions are annotated, i.e., how someone who knows the semantics of a service can specify them in the service's WSDL description. In Table 4.2, we have defined the semantic annotations for the WSMO-Lite service model. Further below in this subsection, we discuss how these annotations fit on components of WSDL descriptions; Table 5.1 presents a concise summary: the first column indicates the type of semantics (functional, non-functional, behavioral and information-model), the second column shows the

WSMO-Lite service model component where the semantics gets attached, and the third column enumerates the corresponding WSDL components where the annotations belong.

Sem. type	WSMO-Lite svc. model	WSDL component
F	Service	Service or Interface
N	Service	Service
B	Operation	Interface Operation
I	Message	Element Declaration or Type Definition

Table 5.1: WSMO-Lite semantics in WSDL

Note that while SAWSDL only describes the use of its modelReference annotations on WSDL *interface* components (along with some of their subcomponents, such as *operations*) and on XML Schema *element declaration* and *type definition* components, it allows the annotation of all the other components in WSDL, including *service*. Our use of modelReference annotations on *service* components in the following subsections is fully within the spirit of SAWSDL.

For the purpose of illustrating the WSDL annotations in the subsections below, we use a simple example service with the following structure: there is a WSDL interface HotelReservation, with operations search for looking up room availabilities, reserve for making reservations, and cancelReservation for canceling them. The interface is provided by one service, RomaHotels, which is constrained to hotels in the city of Rome, Italy.

In the following subsections, we explain in detail the contents of Table 5.1 and we show examples of the various types of semantic annotations in WSDL.

5.1.1 Functional Annotations

In the WSMO-Lite service ontology, we represent functional semantics either with functionality classifications or with formalized capabilities, captured as preconditions and effects. Functionality categories, preconditions and effects are all identified with URIs, and these URIs can be the value of a modelReference annotation in WSDL.

Most directly, the functional semantics of a service can be described with annotations on the WSDL *service* components (represented by `<wsdl:service>` elements), shown in Listing 5.1. The annotations in this listing specify[1] a functionality category AccommodationReservationService, and the specific precondition and effect of this particular service.

In WSDL, a major part of a description is the service *interface*, which "describes a Web service in terms of the messages it sends and receives", doing it "by grouping related messages into operations" [133]. From the point of view of service semantics, WSDL makes no assertions about different services that implement the same interface, only that they will accept and emit messages with the structure defined in the interface operations, and sequenced according

[1] We omit listing the definitions of these annotations for brevity; their meaning can be taken intuitively.

```
1  <wsdl:service name="RomaHotels"  interface="HotelReservation"
2      sawsdl:modelReference="
3          http://example.org/onto#AccommodationReservationService
4          http://example.org/onto#RomaHotelsPrecondition
5          http://example.org/onto#RomaHotelsEffect" >
6    <wsdl:endpoint ... />
7  </wsdl:service>
```

Listing 5.1: Example service functional annotation

to the operation message exchange patterns. In other words, the WSDL specification does not mandate that an interface should be tied to any particular functionality that can be achieved using its operations.

Nevertheless, if a WSDL interface is created to support certain functionality, the description of this functionality can be added as a semantic annotation on the WSDL interface component. In our case, if the interface of our service is specific for the general functionality of the service (hotel reservation), the annotations should be put on the interface, as illustrated in Listing 5.2, where we include the AccommodationReservationService along with general hotel reservation precondition and effect. These annotations then apply to all WSDL services that implement this interface.

```
1  <wsdl:interface name="HotelReservation"
2      sawsdl:modelReference="
3          http://example.org/onto#AccommodationReservationService
4          http://example.org/onto#HotelReservationPrecondition
5          http://example.org/onto#HotelReservationEffect" >
6    ... interface operations come here ...
7  </wsdl:interface>
```

Listing 5.2: Example interface functional annotation

It is also possible that both the service and its interface are annotated with functional descriptions, when the service restricts or extends the functionality of the interface. Furthermore, in WSDL 2.0, interfaces may *extend* other interfaces. The functionality of an interface then includes the functionalities of the interfaces extended by it. In Section 5.2, we define how the functional annotations of interfaces and services are combined when translating a WSDL description into the WSMO-Lite service model that does not deal with service interfaces.

5.1.2 Nonfunctional Annotations

Nonfunctional properties define incidental details and policies specific to the implementation or running environment of a service; therefore, they are naturally expressed as annotations of the WSDL service component. For example, a hotel reservation service may want to specify its price-per-reservation as a nonfunctional property. Such an annotation is shown in Listing 5.3.

```
<wsdl:service name="RomaHotels" interface="HotelReservation"
    sawsdl:modelReference="
        http://example.org/onto#RomaHotelsPricePerReservation" >
</wsdl:service>
```

Listing 5.3: Example nonfunctional annotations

5.1.3 Behavioral Annotations

WSMO-Lite does not define a specific construct for behavioral descriptions; instead, we use functional annotations of service operations and we expect that the client will be able to decide the ordering of operation invocations based on what the operations do.

In WSDL, service operations are described in the service interface; the intended behavior of an interface can be described with annotations on its operation components. For example, hotel reservation services have the following behavioral constraints: the client can use search and reservation operations at any point in time, but it can only cancel a reservation if it has previously made one. Listing 5.4 sketches how preconditions and effects can be used to describe such operation dependencies.

```
<wsdl:interface name="HotelReservation">
  <wsdl:operation name="search"> ... </wsdl:operation>
  <wsdl:operation name="reserve"
      sawsdl:modelReference="
          http://example.org/onto#EffectReservationConfirmed">
    ...
  </wsdl:operation>
  <wsdl:operation name="cancelReservation"
      sawsdl:modelReference="
          http://example.org/onto#PreconditionReservationConfirmed
          http://example.org/onto#EffectReservationCancelled">
    ...
  </wsdl:operation>
</wsdl:interface>
```

Listing 5.4: Example behavioral annotations

In [125], we have shown that the WSMO-Lite behavioral semantics (functional annotations on service operations) can be translated into a WSMO *choreography* (cf. [105]), which is an explicit behavioral description based on Abstract State Machines [15]. A WSMO choreography is a tuple (Σ, R), where Σ is the state machine's signature of symbols (ontology elements), and R is a set of transition rules of the form $r : r^{cond} \rightarrow r^{\mathit{eff}}$, specifying when the rule should fire, and how the state will change. The symbols in Σ can be *input* and/or *output* symbols, respectively corresponding to the input data sent to the service, and the output data produced by the service.

The translation from WSMO-Lite behavioral annotations turns every annotated operation of a given service into one or two rules, and the inputs/outputs of the service are added to the symbol signature, as detailed in the following list (we denote x_I the model reference values of the input message of the operation, x_O the model reference values of the output message, and $\phi^{pre}, \phi^{\mathit{eff}}$ the

precondition and effect of the operation's capability):

1. x_I is added to Σ as input symbol(s);
2. x_O is added to Σ as output symbol(s);
3. an *in-out* operation[2] is translated to a single rule $r : x_I \wedge \phi^{pre} \rightarrow x_O \wedge \phi^{eff}$
4. an *in-only* operation is translated to the rule $r : x_I \wedge \phi^{pre} \rightarrow \phi^{eff}$
5. an *out-only* operation is translated to the rule $r : \phi^{pre} \rightarrow x_O \wedge \phi^{eff}$
6. an *out-in* operation, which is initiated by a message from the service, is translated to two rules, $r_1 : \phi^{pre} \rightarrow x_O$ and $r_2 : x_I \wedge x_O \rightarrow \phi^{eff}$

With a choreography constructed according to these steps, a WSMO-based client is able to automatically invoke a service, i.e., calling its operations in the correct and expected order.

5.1.4 Information Model Annotations

The semantics of the exchanged data is expressed through annotations on the message schemas. A modelReference on an XML Schema element declaration or type definition points to a description of the semantics of the data described by the schema. An annotation pointing to an ontology class or relation means that the data will define an instance (or multiple instances) of that class/relation. For example, the annotations in Listing 5.5 specify that a `<ReserveRoom>` element contains the stay dates and the number of persons for which the client wants to make the reservation, the name under which the reservation will be made (all the data so far wrapped in a ReservationRequest class), and the particular hotel. These are the inputs to the operation.

```
1  <wsdl:types><xs:schema ...>
2    <xs:element name="ReserveRoom"
3      sawsdl:modelReference="http://example.org/onto#ReservationRequest
4                             http://example.org/onto#Hotel"
5      sawsdl:loweringSchemaMapping="http://example.org/ReserveLowering.xsp" >
6      ...
7    </xs:element>
8  </xs:schema></wsdl:types>
9
10 <wsdl:interface name="HotelReservation">
11   <wsdl:operation name="reserve">
12     <wsdl:input element="ReserveRoom" />
13     <wsdl:output element="Reservation" />
14   </wsdl:operation>
15   ...
16 </wsdl:interface>
```

Listing 5.5: Example information model annotations

The following section discusses the data lifting and lowering aspect of information model annotations, necessary for communication between a Web service and a semantic client.

[2]Operation message exchange patterns are discussed in Section 3.3.2 (Page 32).

5.1.5 Data Lifting and Lowering

A SEE works on the semantic level, with its data represented in RDF. In contrast, Web services and their clients usually exchange messages in XML or in some other non-semantic structured data format. In order to enable the SEE to communicate with actual Web services, its semantic data must be *lowered* into the expected input messages, and the data coming from the service in its output messages must be *lifted* back up to the semantic level.

If there were Web services that would accept and produce RDF data in their messages, lifting and lowering would be identity mappings; the SEE would only need to serialize and parse the data in on-the-wire messages. However, RDF-driven Web services are extremely rare. In fact, most Web services use XML-based messages, therefore we must support lowering from RDF to XML, and lifting back. In this section, we describe the use of XSPARQL, introduced in Section 3.5.4, for implementing both transformation directions.

In WSDL, both lifting and lowering transformations are attached to message descriptions, using the SAWSDL attributes `liftingSchemaMapping` and `loweringSchemaMapping` respectively. A message in WSDL is described with an XML Schema element declaration. A lifting transformation should accept documents valid according to the schema of the element, and produce the equivalent RDF data. A lowering transformation takes RDF data as its input, and should produce an XML document that is valid according to the schema of the message element. Alas, verifying that a transformation would accept all valid documents, or that all its possible results are going to be valid, is a known hard (if not generally impossible) problem of proving program correctness. Transformation authors must rely on testing.

To illustrate XSPARQL lifting and lowering transformations, we continue with the example hotel reservation service. We show simple example message schemas for the reservation operation in Listing 5.6 and the respective data ontology in Listing 5.7.

We need a lowering transformation for the reservation request element so that a SEE client can transform the data of a user goal (booking a room) into the appropriate XML/SOAP message; this transformation is shown in Listing 5.8. Further, we need a lifting transformation for the resulting confirmed Reservation; we show such a transformation in Listing 5.9.

The lowering transformation takes parts of the request data and puts it into the resulting XML structure in a straightforward way. The lifting transformation illustrates that it can use the input data together with the incoming response message to construct the reservation graph in RDF. In this case, the reservation response message does not repeat the reservation data, and to have a complete model of the reservation, the lifting transformation copies the relevant data from the request.

5.2 Mapping Annotated WSDL to WSMO-Lite Service Model

In the previous section, we have discussed how to annotate a WSDL document with service semantics; here, we show how an annotated WSDL document is interpreted in our service model introduced in Chapter 4. In effect, this section

```
<xs:schema targetNamespace="http://example.org/reserve.xsd"
        xmlns="http://example.org/reserve.xsd"
        xmlns:sawsdl="http://www.w3.org/ns/sawsdl#"
        xmlns:xs="http://www.w3.org/2001/XMLSchema" >
    <xs:element name="ReserveRoom"
        sawsdl:modelReference="http://example.org/onto#ReservationRequest
                               http://example.org/onto#Hotel"
        sawsdl:loweringSchemaMapping="http://example.org/ReserveLowering.xsp" >
      <xs:complexType>
        <xs:all>
          <xs:element name="hotelID" type="xs:string"/>
          <xs:element name="arrivalDate" type="xs:date"/>
          <xs:element name="numberOfNights" type="xs:short"/>
          <xs:element name="numberOfGuests" type="xs:short"/>
          <xs:element name="name" type="xs:string"/>
        </xs:all>
      </xs:complexType>
    </xs:element>

    <xs:element name="Reservation"
        sawsdl:modelReference="http://example.org/onto#Reservation"
        sawsdl:liftingSchemaMapping="http://example.org/ReserveLifting.xsp" >
      <xs:all>
        <xs:element name="confirmationID" type="xs:string"/>
        <xs:element name="description" type="xs:string"/>
      </xs:all>
    </xs:element>
</xs:schema>
```

Listing 5.6: XML Schema for example service messages

```
@prefix ns:  <http://example.org/onto#> .
@prefix rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix xs:  <http://www.w3.org/2001/XMLSchema#> .

ns:Reservation a rdfs:Class .
ns:ReservationRequest a rdfs:Class .
ns:Hotel a rdfs:Class .

ns:arrivalDate a rdf:Property .
    # domain: either a reservation request or reservation
    # range: xs:dateTime
ns:numberOfNights a rdf:Property .
    # domain: either a reservation request or reservation
    # range: xs:short
ns:numberOfGuests a rdf:Property .
    # domain: either a reservation request or reservation
    # range: xs:short
ns:primaryName a rdf:Property .
    # domain: either a reservation request or reservation
    # range: xs:string
ns:hotel a rdf:Property ;
    # domain: either a reservation request or reservation
    rdfs:range ns:Hotel .

ns:description a rdf:Property ;
    rdfs:subPropertyOf rdfs:comment .
```

Listing 5.7: Ontology for example service message data

```
1  declare namespace ns="http://example.org/onto#";
2  declare namespace nsxml="http://example.org/reserve.xsd";
3
4  for $hotel $peopleCount $arrivalDate $nightCount $name from <input.rdf>
5  where { _:req a ns:ReservationRequest ;
6               ns:arrivalDate $arrivalDate ;
7               ns:numberOfNights $nightCount ;
8               ns:numberOfGuests $peopleCount ;
9               ns:primaryName $name ;
10              ns:hotel $hotel . }
11 return
12   <nsxml:ReserveRoom>
13     <nsxml:hotelID>{$hotel}</nsxml:hotelID>
14     <nsxml:arrivalDate>{$arrivalDate}</nsxml:arrivalDate>
15     <nsxml:numberOfNights>{$nightCount}</nsxml:numberOfNights>
16     <nsxml:numberOfGuests>{$peopleCount}</nsxml:numberOfGuests>
17     <nsxml:name>{$name}</nsxml:name>
18   </nsxml:ReserveRoom>
```

Listing 5.8: XSPARQL example: lowering transformation

```
1  declare namespace ns="http://example.org/onto#";
2  declare namespace nsxml="http://example.org/reserve.xsd";
3
4  let $reservation := //nsxml:Reservation
5  for $hotel $peopleCount $arrivalDate $nightCount $name from <input.rdf>
6  where { _:req a ns:ReservationRequest ;
7                ns:arrivalDate $arrivalDate ;
8                ns:numberOfNights $nightCount ;
9                ns:numberOfGuests $peopleCount ;
10               ns:primaryName $name ;
11               ns:hotel $hotel . }
12 construct {
13    { $reservation/nsxml:confirmationID } a ns:Reservation ;
14          ns:arrivalDate $arrivalDate ;
15          ns:numberOfNights $nightCount ;
16          ns:numberOfGuests $peopleCount ;
17          ns:primaryName $name ;
18          ns:hotel $hotel ;
19          ns:description { $reservation/nsxml:description } .
20 }
```

Listing 5.9: XSPARQL example: lifting transformation

specifies a WSMO-Lite parser for WSDL; a program or a library that transforms a WSMO-Lite description (a WSDL document with SAWSDL annotations pointing to semantics according to our service ontology) into RDF data useful for semantic processing. Various implementation components, including a WSMO-Lite/WSDL parser, are discussed in Chapter 8.

Figure 5.1 illustrates the mapping from WSDL to our service model; it shows the main component correspondences. More formally, Table 5.2 contains the whole mapping: it defines how WSDL components (and their properties) get mapped onto RDF data using the classes and properties of our service model.

The table is phrased in the terms of the WSDL 2.0 component model defined in [133]. In WSDL, components have properties, denoted with curly brackets: for instance, the Service component has the properties {name}, {interface} and {endpoints}; we adopt this notation in the text below. A property value may be a single component, a set of components, or a literal.

In the table, the function prop(?x, property name) serves to access the value of the property {property name} from the component bound to the variable ?x. The functions id(?x) and id(?y,?x) return the URI identifier for the component in the variable ?x (in context of the component ?y); these identifiers may for example have a similar forms as the WSDL component identifiers defined in Appendix C of [133].

The mapping/parsing process starts with a WSDL Service component and descends recursively through the component properties. Note that the order in which the various components are mapped is not significant, since the result is an RDF graph with no inherent ordering.

The Service component is directly mapped to an instance of wl:Service (row 1 in the table). This instance should further be annotated with the service name as the value of the property rdfs:label, and with a link to the original WSDL document, using the property rdfs:isDefinedBy. The latter is used whenever the

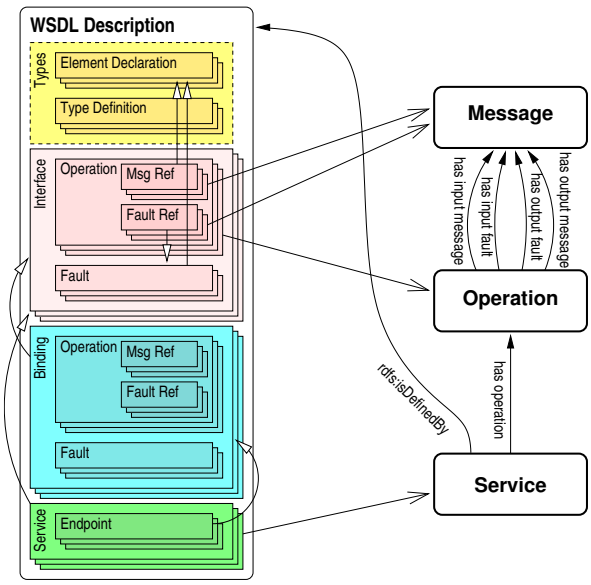


Figure 5.1: Mapping from the WSDL structure to our service model

#	WSDL Components	WSMO-Lite RDF triples
1	Service ?s	id(?s) a wl:Service. id(?s) rdfs:label prop(?s, name). id(?s) rdfs:isDefinedBy ⟨wsdlDocumentURI⟩.
2	?i = prop(?s, interface) ?o ∈ prop(?i, interface operations)	id(?s) wl:hasOperation id(?s,?o). id(?s,?o) a wl:Operation. id(?s,?o) rdfs:label prop(?o, name).
3	$?i_1$ = prop(?s, interface) ∃ $?i_2 \ldots ?i_n$: ∀ k = 2...n: $?i_k \in$ prop($?i_{k-1}$, extended interfaces) ?o ∈ prop($?i_n$, interface operations)	id(?s) wl:hasOperation id(?s,?o). id(?s,?o) a wl:Operation. id(?s,?o) rdfs:label prop(?o, name).
4	?m ∈ prop(?o, interface message references) prop(?m, direction) = *in*	id(?s,?o) wl:hasInputMessage id(?s,?m). id(?s,?m) a wl:Message.
5	?m ∈ prop(?o, interface message references) prop(?m, direction) = *out*	id(?s,?o) wl:hasOutputMessage id(?s,?m). id(?s,?m) a wl:Message.
6	?f ∈ prop(?o, interface fault references) prop(?f, direction) = *in*	id(?s,?o) wl:hasInputFault id(?s,?f). id(?s,?f) a wl:Message.
7	?f ∈ prop(?o, interface fault references) prop(?f, direction) = *out*	id(?s,?o) wl:hasOutputFault id(?s,?f). id(?s,?f) a wl:Message.
8	?r ∈ prop(?s, model reference)	id(?s) sawsdl:modelReference ?r.
9	?r ∈ prop(?i, model reference)	id(?s) sawsdl:modelReference ?r.
10	?r ∈ prop(?o, model reference)	id(?s,?o) sawsdl:modelReference ?r.
11	?e = prop(?m, element declaration) ?r ∈ prop(?e, model reference)	id(?s,?m) sawsdl:modelReference ?r.
12	?li ∈ prop(?e, lifting schema mapping)	id(?s,?m) sawsdl:liftingSchemaMapping ?li.
13	?lo ∈ prop(?e, lowering schema mapping)	id(?s,?m) sawsdl:loweringSchemaMapping ?lo.
14	?f' = prop(?f, interface fault) ?e = prop(?f', element declaration) ?r ∈ prop(?e, model reference)	id(?s,?f) sawsdl:modelReference ?r.
15	?li ∈ prop(?e, lifting schema mapping)	id(?s,?f) sawsdl:liftingSchemaMapping ?li.
16	?lo ∈ prop(?e, lowering schema mapping)	id(?s,?f) sawsdl:loweringSchemaMapping ?lo.
17	prop(?o, safe) = true	id(?s,?o) sawsdl:modelReference wsdlx:SafeInteraction.

Table 5.2: Mapping from WSDL components to WSMO-Lite service model; the functions prop and id are explained in the text.

original WSDL is necessary; for instance in automation tasks that invoke the service's operations, the WSDL contains the binding and endpoint information necessary to perform the invocation.

Rows 2–3 map all the service's operations, using ?s from row 1. A WSDL service *implements* an interface, which is then the value of the service component's {interface} property. All operations of that interface (i.e., Interface Operation components grouped in the {interface operations} property of the interface), and all the operations of interfaces extended by this interface, either directly or through further interface extension steps, are mapped to instances of wl:Operation, and attached with wl:hasOperation to the wl:Service instance. Like services, operations also have names suitable for the rdfs:label property.

Rows 4–7 map the operations' input, output and fault messages, using ?o from rows 2–3. A WSDL interface operation component has the properties {interface message reference} and {interface fault reference}, each a set of correspondingly named components that describe the various messages and faults exchanged by the operation. Each message reference and fault reference component is mapped to an instance of wl:Message; message references whose {direction} property is *in* (or *out*) are attached to the wl:Operation instance using the property wl:hasInputMessage (or wl:hasOutputMessage). Similarly, fault references are attached using the properties wl:hasInputFault and wl:hasOutputFault, depending on the {direction} property.

Rows 8–16 handle the SAWSDL annotations present in the WSDL. Rows 8 and 9 combine model references from the service ?s (from row 1) and its interface ?i (from row 2) on the generated wl:Service instance, and row 10 copies operation model references to the corresponding wl:Operation instance (with ?o from rows 2–3).

Message annotations are gathered from the schema definitions for the messages. An Interface Message Reference component has an {element declaration} property whose value describes the message schema; in rows 11–13, the model references and the lifting and lowering schema mappings of this element declaration are mapped on the corresponding wl:Message instance (with ?m from rows 4–5). In rows 14–16, fault message annotations are collected in the same way from the element declaration specified by the fault components referenced from the operation fault reference components[3] (with ?f from rows 6–7). Note that SAWSDL propagates annotations from type definitions to element declarations; such propagated values are reflected in the component model that is the input to our mapping.

While WSDL focuses on the messaging and networking aspects of a service description, the use of HTTP has necessitated that WSDL introduce a semantic flag for marking operations as *safe* as defined in the Web architecture (we discuss Web interaction safety in Section 3.2.4). An assertion that an operation is safe is a statement about the functionality of the operation; in particular, about the lack of application-significant side-effects and new obligations. While safety does not imply anything about what the operation will actually do, it does mean that the operation can be invoked opportunistically, for example for *offer discovery*, as shown in Chapter 7. Therefore, row 17 in the mapping table maps the safety

[3]WSDL 2.0 has an indirection between operations and fault data definitions: where an operation message reference directly identifies an XML Schema element declaration, an operation fault reference identifies a fault description, which then specifies the element declaration for the data.

Example WSDL	Corresponding RDF View
1 <wsdl:description>	@prefix ex: <.../example/genid#> .
2 <wsdl:types>	@prefix rdfs: <.../rdf−schema#> .
3 <xs:schema ...>	@prefix sawsdl: <.../sawsdl#> .
4 <xs:element name="ReserveRoom"	@prefix wl: <.../wsmo−lite#> .
5 sawsdl:modelReference="	
6 ...#ReservationRequest	35 ex:svc a wl:Service ;
7 ...#Hotel"	35 rdfs:label "RomaHotels" ;
8 sawsdl:loweringSchemaMapping="	35 rdfs:isDefinedBy <*original WSDL URI*> ;
9 .../ReserveLowering.xsp" >	sawsdl:modelReference
10 ...	16 <...#AccommodationReservationService> ,
11 </xs:element>	17 <...#HotelReservationPrecondition> ,
12 </xs:schema>	18 <...#HotelReservationEffect> ,
13 </wsdl:types>	38 <...#RomaHotelsPrecondition> ,
14 <wsdl:interface name="HotelReservation"	39 <...#RomaHotelsEffect> ,
15 sawsdl:modelReference="	40 <...#RomaHotelsPricePerReservation> ;
16 ...#AccommodationReservationService	19 wl:hasOperation ex:op1 ;
17 ...#HotelReservationPrecondition	22 wl:hasOperation ex:op2 ;
18 ...#HotelReservationEffect" >	28 wl:hasOperation ex:op3 .
19 <wsdl:operation name="search" >	
20 ...	19 ex:op1 a wl:Operation ;
21 </wsdl:operation>	19 rdfs:label "search" .
22 <wsdl:operation name="reserve"	
23 sawsdl:modelReference="	22 ex:op2 a wl:Operation ;
24 ...#EffectReservationConfirmed" >	22 rdfs:label "reserve" ;
25 <wsdl:input element="ReserveRoom" />	25 wl:hasInputMessage ex:msg1 ;
26 <wsdl:output element="Reservation" />	26 wl:hasOutputMessage ex:msg2 ;
27 </wsdl:operation>	sawsdl:modelReference
28 <wsdl:operation name="cancelReservation"	24 <...#EffectReservationConfirmed> .
29 sawsdl:modelReference="	
30 ...#PreconditionReservationConfirmed	25 ex:msg1 a wl:Message ;
31 ...#EffectReservationCancelled" >	sawsdl:modelReference
32 ...	6 <...#ReservationRequest> ,
33 </wsdl:operation>	7 <...#Hotel> ;
34 </wsdl:interface>	sawsdl:loweringSchemaMapping
35 <wsdl:service name="RomaHotels"	9 <.../ReserveLowering.xsp> .
36 interface="HotelReservation"	
37 sawsdl:modelReference="	26 ex:msg2 a wl:Message .
38 ...#RomaHotelsPrecondition	
39 ...#RomaHotelsEffect	28 ex:op3 a wl:Operation ;
40 ...#RomaHotelsPricePerReservation" >	28 rdfs:label "cancelReservation" ;
41 <wsdl:endpoint ... />	sawsdl:modelReference
42 </wsdl:service>	30 <...#PreconditionReservationConfirmed> ,
43 </wsdl:description>	31 <...#EffectReservationCancelled> .

Figure 5.2: Consolidated WSDL example and its mapping to RDF, showing for each generated triple the corresponding line of the example WSDL; the URIs are shortened for space and readability.

flag defined in [134] to a semantic annotation — a model reference with the value wsdlx:SafeInteraction (with ?o from rows 2–3), where the prefix wsdlx refers to the namespace `http://www.w3.org/ns/wsdl-extensions#` (cf. Section 6.5). As a functional annotation on operations, the safety property becomes part of the service's behavioral semantics.

To illustrate the mapping, Figure 5.2 consolidates the WSDL example snippets from Listings 5.1–5.5 and presents the corresponding RDF form, in Notation 3 syntax. In the RDF, each number in front of a line indicates the line number in the left-hand-side WSDL document that leads to the particular right-hand-side RDF triple.

5.3 Deployment of Ontologies Used in SAWSDL Descriptions

The preceding sections discuss WSMO-Lite in terms of the annotations that it places in WSDL documents. However, the annotations merely point from WSDL to semantic concepts identified by URIs. The semantic concepts must be defined somewhere, in the form of information ontologies, functionality taxonomies, the definitions of service/operation preconditions and effects in terms of logical expressions, or ontologies of nonfunctional semantics. Here, we discuss the deployment options for these ontologies.

Note that we assume the client already has access to the annotated WSDL descriptions. WSDL documents can be available at the service endpoints or in service registries, such as the semantic registry iServe (see Section 8.3) or the search engine `seekda.com`, which crawls the Web for public WSDL service descriptions.

There are three broad options for where the definitions of the semantic concepts used in the SAWSDL files can be located:

1. the concept definitions are **embedded in the WSDL documents** along with the annotations,
2. the concept definitions are **available publicly on the Web**, discoverable through their URIs,
3. or the concept definitions are **stored in a repository** that the clients are expected to be able to access.

These three options are differently suitable for various envisioned situations, and they can even be combined: for instance, a standard functionality taxonomy can be available publicly on the Web, an internal product ontology can be in a local repository, and concrete precondition and effect definitions can be embedded in the WSDL document. In the following subsections, we sketch the main characteristics of the deployment options and some typical cases where the options should be used. Figure 5.3 illustrates the options graphically.

5.3.1 Embedding

Embedding the relevant concept descriptions in the WSDL document helps make the document self-contained, i.e., the content of the document is all a semantic client needs in order to include the described Web service in its processing. An embedded semantic description is illustrated in Listing 5.10 on lines 6–21: RDF statements are contained in an extensibility element inside the WSDL document. In the interest of readability, the RDF is written in Notation 3 inside a proposed element `<rdf:n3>`; the standard RDF/XML syntax would use `<rdf:RDF>` as the extensibility element in WSDL.

SOA deployments often contain registries of service descriptions (such as UDDI [120]). With semantic concept descriptions embedded in the WSDL documents, no further registry infrastructure is needed. Therefore, this approach enables gradual adoption of semantic automation in existing SOA systems.

A semantic description would typically be embedded in a WSDL document if it is specific to the given Web service. If multiple WSDL files should share the

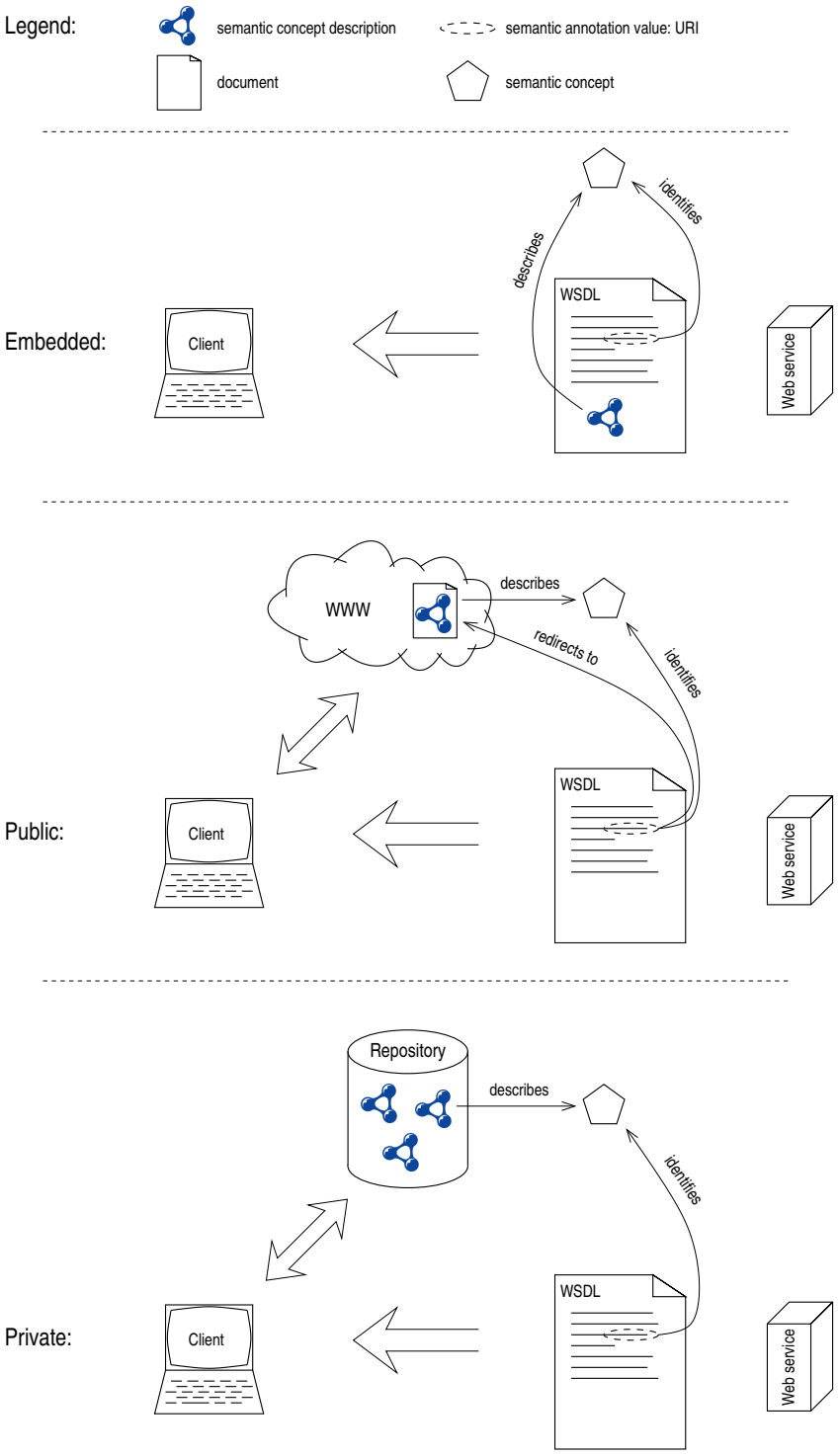


Figure 5.3: Deployment options for semantic descriptions used in WSMO-Lite

```
<wsdl:description>
  <wsdl:service name="RomaHotels"
      sawsdl:modelReference="http://example.com/#RomaHotelsPrecondition">
  </wsdl:service>

  <rdf:n3>
    @prefix ex: <http://example.org/onto> .
    @prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
    @prefix wsml: <http://www.wsmo.org/wsml/wsml-syntax#> .
    @prefix wl: <http://www.wsmo.org/ns/wsmo-lite#> .

    ex:RomaHotelsPrecondition a wl:Condition ;
        rdf:value """
          namespace _"http://example.org/onto#"
          ?request [ numberOfGuests hasValue ?guests,
                     hotel hasValue ?hotel ] memberOf ReservationRequest
          and ?guests =< 10
          and ?hotel [ locationCity hasValue ?city ] memberOf Hotel
          and ?city = "Rome" .
        """^^wsml:AxiomLiteral .
  </rdf:n3>
</wsdl:description>
```

Listing 5.10: Embedding semantic descriptions in WSDL

same description, embedding it in all the files would mean duplication of content, and an increase of the cost of maintenance, with a risk of the shared parts becoming inconsistent. To address such sharing in a WSDL-based infrastructure that supports import of WSDL documents with extensions, the shared semantic descriptions could be embedded in a WSDL document imported from all the other WSDL files that use it.

To process a WSMO-Lite description that embeds the semantic definitions within the WSDL file, the client must perform the following steps:

1. Identify the extension elements in the WSDL description which contain semantic definitions (e.g., the `<rdf:n3>` element, or the standard `<rdf:RDF>` element for RDF/XML). A particular client may support only a limited set of such elements; `<rdf:RDF>` SHOULD be among them.

2. Process the semantic data inside these elements and either merge the definitions with the client's knowledge base, or register them for subsequent retrieval, when processing the SAWSDL annotations themselves.

Abstractly, this process precedes the processing of the actual annotations. In effect, the steps above build a local private repository (as discussed in Section 5.3.3) within the client, which is then queried whenever the client needs to know the meaning of an annotation.

5.3.2 Public Availability

Semantic descriptions and ontologies that are not specific to any particular Web service, especially such as domain and upper ontologies, should be external to the WSDL documents, and many of them will be public.

With the semantic concept descriptions outside the WSDL document, it is necessary for the client to be able to locate the description of each concept indicated by a modelReference. The SAWSDL specification recommends that "the URI used for pointing to a semantic concept resolve to a document containing its definition." [107] To find the definition of a particular concept, the client can make an HTTP request to the concept URI; the HTTP response should either directly contain the definition, or it should redirect to a resource that does.

5.3.3 Private Repository

In private SOA deployments, such as inside enterprises, it may be desirable to keep the semantic concept descriptions external to the WSDL documents and yet hide them from the public Web. For instance, sensitive product descriptions in the company's product ontology could reveal trade secrets if made public. Therefore, there is an option to store the semantic descriptions in a local repository that is shared between the service provider(s) and the semantic clients that may be distributed in different departments of the enterprise.

The private repository can be an intranet server (using public Web technologies in a separated private network), or a dedicated database for the semantic data. A client that encounters WSMO-Lite annotations can resolve them (and find the appropriate semantic concept descriptions) by querying the known internal repository, or if the intranet solution is in use, by following the same steps as described in the preceding section.

5.4 Validation of WSMO-Lite Descriptions in SAWSDL

Validation is a process of verifying that a formal document follows certain application-specific rules. Here, we discuss the validation of WSDL documents annotated with WSMO-Lite semantics, to detect potential problems in service descriptions. However, even a valid description does not guarantee success in using the service, as we cannot eliminate run-time failures such as a product being out of stock, or errors such as a network outage. On the other hand, some automation may succeed even with an invalid description. For example, during a particular run, the client may not encounter the problematic parts of the invalid description.

Therefore, a validator is a useful, yet non-essential, tool for improving the chances of successful automation with WSMO-Lite descriptions, by identifying potential problems and thus saving valuable human debugging effort.

For the combination of SAWSDL and WSMO-Lite, we distinguish four facets of validity, as described in Section 5.4.1. Section 5.4.2 details the validation rules for one of the facets, *consistency*, and Section 5.4.3 then describes the process of WSMO-Lite validation.

5.4.1 Facets of Validity

We distinguish the following facets of validity of SAWSDL/WSMO-Lite descriptions:

1. *Syntax* constrains the basic building blocks of a formal document, which must be a well-formed sentence of some underlying language. WSDL and SAWSDL have well-defined syntax, built on the syntax of XML 1.0. The syntax of XML is described in an Extended Backus-Naur Form (EBNF) notation in the specification. The structure of WSDL and SAWSDL is described in their respective XML Schemas, with further constraints in prose in the specifications, especially in WSDL 2.0. A WSDL validator first checks that the input document is well-formed XML, and then it validates the WSDL schema and any further constraints. A SAWSDL validator checks that the annotation values are URIs, and possibly that the document contains no unknown attributes in the SAWSDL namespace, which would likely indicate typographical errors. A WSMO-Lite-oriented validator should include validation steps for WSDL and SAWSDL.

2. *Consistency:* a consistent description does not contradict itself. We define a number of consistency rules for WSMO-Lite over WSDL in Section 5.4.2. A validator can only verify the consistency of a description to the extent to which it understands the formal semantics of the annotations, therefore some of the rules defined in Section 5.4.2 form rather a guide for humans who check and test the description, as opposed to a computer algorithm for automatic validation.

3. *Completeness:* depending on the tasks that the description should support, it may only need to capture a subset of the service semantics. The relationship between SWS automation tasks and the required semantics is shown in Table 4.1 (on page 56). A validator can take a WSMO-Lite service description and report the automation tasks for which the description has sufficient information, or if the user specifies a set of tasks that need to be supported, the validator can report what annotations are missing, if any.

4. *Correctness:* a description should be a truthful model of its underlying service. A syntactically invalid or semantically inconsistent description cannot be considered truthful. However, above validity and consistency checking, correctness is generally verified by testing or, ultimately, by "adding eyeballs".

As is apparent, automatic validation is only possible to a certain degree. While we strive for correctness of semantic descriptions, the facets of syntax, consistency and completeness are the only aspects for which we have at least partial validation algorithms. Syntactical validity and completeness can be checked entirely, whereas consistency can only be checked to the extent of available reasoning power.

Note that the above formulations of *completeness* and *correctness* differ from those introduced by Preist in [93]. Preist defines both completeness and correctness in the context of service discovery, in terms of the (possible or actual) functionalities of the service: a service description is complete if it describes all the functionalities, and it is correct if it does not describe any functionalities that the service does not actually offer. Preist notes that it is often "possible to achieve completeness but not correctness in service descriptions" — "for example, a bookseller can state 'I sell books' in an advert, but cannot guarantee

that every title is in stock." [93] With the focus on covering the functionalities of a service, neither property (completeness or correctness) can be checked by an automated tool. On the other hand, our definition of completeness deals with the coverage of support for SWS automation tasks by a given service description, which can be checked algorithmically. Our definition of correctness corresponds to a notion of an ideal, perfect description; it would include both Preist's completeness and correctness. It is not automatically verifiable, but useful to conclude the discussion of the facets of validity.

5.4.2 WSMO-Lite Consistency Rules

As we have said in the previous section, a consistent description does not contradict itself. Inconsistencies (or contradictions) can occur between related parts of a formal description. In this section, we identify the relations between different WSMO-Lite annotations, and we define rules that must hold for the annotations to be consistent.

Table 5.3 identifies the pairs of annotations that are related in a way that gives space for inconsistencies. Both rows and columns are types of annotations (Functional, Behavioral, Nonfunctional and Information model semantics); an empty table cell indicates no relationship, the symbol • means potential conflicts for which we have at least some automatic validation algorithms, and the symbol ∘ indicates potential conflicts that our validation rules cannot uncover.

	F	**B**	**N**	**I**
F	•	•		•
B				•
N			∘	•
I				∘

Table 5.3: Consistency relationships among WSMO-Lite annotations

Due to our definition of the types of semantics, nonfunctional annotations cannot conflict with functional or behavioral annotations because the nonfunctional semantics cover aspects orthogonal to the other types. Further, there cannot be internal conflicts in behavioral annotations because operations of a single service may be independent, preventing meaningful comparison of their behavioral annotations. Other than that, there is potential for conflicts among all the other pairs of annotations. In the rest of this section, we discuss these potential conflicts and approaches for their validation.

Conflicts within Functional Annotations

WSMO-Lite annotates WSDL with functional semantics on two levels: both the WSDL *service* and its *interface* can be annotated. As *interface* annotations apply to all services that implement the interface, they are taken in conjunction with the *service* annotations. In case of logical expressions, as used for service preconditions and effects, the conjunction of two expressions may result in an unsatisfiable expression (such as $A \wedge \bar{A}$). Hence,

Rule 1 If a WSDL service and the interface implemented by the service are both annotated with capabilities, the conjunction of the preconditions must be satisfiable, and the conjunction of the effects must also be satisfiable.

To validate Rule 1, we employ a suitable logical reasoner and check the satisfiability of the conjunctions of the preconditions or effects.

With this rule, we can detect inconsistencies in functional annotations with capabilities; however, no such rule can be defined for functional categories annotating WSDL services and their interfaces. Even if two functionality categories were declared to be distinct (for example, a concrete hotel room in Italy cannot also be a hotel room in Germany), a single service can provide access to either category, depending on the request.

Note that within the framework of WSMO-Lite, we cannot compare functional annotations using precondition and effect expressions with functional annotations that use functionality categories. This limitation applies not only to the verification of Rule 1, but also to the rules that follow below.

Conflicts Between Functional and Behavioral Annotations

Functional semantics describe what a service can do. Behavioral semantics describe how a client should communicate with the service — in what order it should invoke the available operations. In WSMO-Lite, we specify the behavioral semantics of a Web service through functional annotations of the service's operations. In case both the service functionality and the operation functionalities are described using capabilities (preconditions and effects), we could be able to check the following two rules:

Rule 2 The behavioral semantics of a service allows at least one successful execution coming from a state that fulfills the service capability precondition and ending in a state that fulfills the capability effect.

Rule 3 For every state that fulfills the precondition of a service capability, the behavioral semantics of the service allows at least one successful execution ending in a state that fulfills the capability effect.

In effect, these two rules both check the consistency of the service functional annotations with the operation functional annotations (checking that the operations can deliver what the service promises), with Rule 2 being a weaker variant of Rule 3. If a Web service description satisfies Rule 3, and the client can satisfy the service's precondition, there exists an order of operations that, barring any runtime failures, will end up satisfying the effect of the service. If a service description only satisfies Rule 2, there may be cases when the client cannot actually find a suitable operation ordering, even if the precondition of the service is satisfied.

For instance, if the service functional description says the service can sell and deliver products, but there is no actual operation that allows for the client to specify delivery options, Rule 3 would be violated because physical delivery cannot be accomplished, but the service would comply with Rule 2 because if delivery is not necessary (for instance when purchasing music in a digital format, such as MP3), the service could be used successfully.

Rules 2 and 3 can be implemented using planning or constraint satisfaction algorithms.

Conflicts within Nonfunctional Annotations

The nonfunctional semantics of a Web service can consist of multiple nonfunctional properties. Since WSMO-Lite does not by itself provide details of how nonfunctional properties are expressed (WSMO-Lite relies on domain ontologies), we cannot formulate a concrete rule for detecting inconsistencies. However, a particular validator may support some set of nonfunctional property specifications and it may be able to check their consistency; for example, a single hotel reservation service cannot specify two different per-reservation prices (unless the prices can be scoped to different kinds of reservations, if the price ontology allows that).

Conflicts within Information Model Annotations

The information model of a Web service mainly describes the input and output messages of the service. There are two kinds of information model annotations in SAWSDL: i) a *model reference* points from an XML Schema component to a class or relation, which means that XML data described by this component will carry instances of the class or relation; ii) *lifting schema mapping* or *lowering schema mapping* annotations (denoted simply as lifting and lowering annotations) point to data transformations that map between XML data and the ontological instances it represents. Naturally, these annotations need to be consistent:

Rule 4 If an XML schema component has both a model reference annotation and a lifting annotation, the lifting transformation must accept XML data valid according to the schema component, and produce instances valid according to the ontology element specified by the model reference annotation. Similarly, if an XML schema component has both a model reference annotation and a lowering annotation, the lowering transformation must accept instance data valid according to the ontology element specified by the model reference annotation, and produce an XML element valid according to the schema component.

For illustration, a message may be described as being a ReservationRequest. If it has a lifting annotation, the lifting transformation must accept an XML document that is valid according to the message schema, and the result of the transformation must contain an instance of ReservationRequest; and if the message has a lowering annotation, the lowering transformation must take an instance of ReservationRequest as its input and return an XML document that is valid according to the message schema.

There cannot be a general algorithm which checks this rule for powerful Turing-complete transformation languages such as XSLT and XSPARQL, however the rules might be implemented for some common cases, or for less powerful transformation languages.

Consistency of Information Model with Other Annotations

While the information model of a Web service mainly deals with the input and output messages, it also describes the terms used in the service or operation preconditions and effects (for functional and behavioral semantics), and in non-functional properties.

A validator can traverse the functional, behavioral and nonfunctional service annotations, along with the model reference annotations of the input and output messages, and check the availability of the formal description of the information model, as described in Section 5.3, and report a problem for every term used for which no formal description can be found.

5.4.3 Validation Process

To summarize, a WSMO-Lite validator takes a semantic Web service description, potentially along with a list of SWS tasks that the description should support, and returns a listing of errors (problems that must be corrected) and warnings (potential problems which may or may not need fixing), if any, according to these steps:

1. First, syntactical validity of the input description should be checked, reporting any problems as errors.
2. The semantic annotations in the document are checked against all applicable consistency rules (to the extent that the rules can be checked automatically), any inconsistencies are reported as errors.
3. If the list of intended SWS automation tasks is available to the validator, the input description is checked for completeness of annotations required by these tasks; any violations are reported as warnings.
4. If the validator is run without a list of intended SWS automation tasks, it runs completeness checks for all tasks and reports those for which the description contains sufficient annotations.

Chapter 6

MicroWSMO: Annotating RESTful Web Services

"There's usually an HTML page." — Anonymous
(regarding Web API descriptions)

Chapter 4 defines a lightweight ontology for semantic description of Web services, and Chapter 5 applies it to annotating WSDL, the industry-standard Web Services Description Language. The Web contains many services that are not described in WSDL. Especially services that are more native to the Web — ones that are "RESTful" — are seldom described in WSDL. In this chapter, we discuss how RESTful services are actually described, and how those descriptions can be made accessible to machine processing and SWS automation.

First, in Section 6.1, we discuss how the WSMO-Lite service model applies to RESTful services. In Section 6.2, we define two microformats that can be used to turn HTML documentation of RESTful services into machine-readable semantic descriptions, then Section 6.3 looks at other already-machine-oriented service and resource descriptions that can also be annotated semantically. In Section 6.4, we address data lifting and lowering for the specific case of RESTful services. In Section 6.5, we briefly analyze some types of semantics that the HTTP protocol imposes on RESTful services. Finally, Section 6.6 discusses the deployment of semantic descriptions for RESTful services, and Section 6.7 sketches the validation of hRESTS/MicroWSMO descriptions.

6.1 Model for Semantic Description of RESTful Services

RESTful Web services (often called "Web APIs") are hypermedia applications consisting of interlinked resources (like Web pages) that are oriented towards machine consumption. In their structure and behavior, RESTful Web services are very much like common Web sites [101].

From the Architecture of the Web [3] and from the REST architectural style [30], we can extract the following concepts inherent in RESTful services: a *resource*, identified by a URI that also serves as the endpoint address where

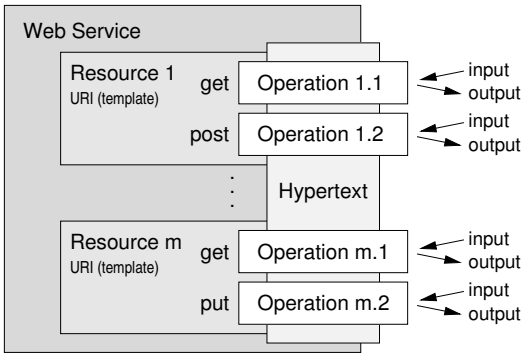


Figure 6.1: Functional model of a RESTful Web service

clients can send requests; every resource has a number of *methods* (in HTTP, the most-used methods are GET, HEAD, POST, PUT and DELETE) that are invoked by means of request/response message exchanges. The messages can carry *hyperlinks*, which point to other resources and which the client can navigate when using the service. A hyperlink can simply be a URI, or it can be a *form* which specifies not only the URI of the target resource, but also the method to be invoked and the structure of the input data.

Note that the Web architecture contains no formal concept of a *service* as a whole; a service is a grouping of resources that is useful for developing, advertising and managing related resources.

The hypertext nature of RESTful Web services differs from the service model discussed in the preceding chapters: hypertext emphasizes the decomposition of a service into Web resources which are preferably units of data, whereas the WS–∗-based model in WSMO-Lite decomposes services into operations, i.e. units of function.

Figure 6.1 depicts a natural model for RESTful services. A Web service has a number of resources whose HTTP methods represent operations, each with potential inputs and outputs. The service also has a hypertext graph structure where the outputs of some operations may link to other operations. In contrast to the model described in Chapter 4, a RESTful service does not have a single location, instead each resource has its own address; a parametrized URI template (cf. Section 3.2.3) can denote a set of resources with the same operations.

In this section, we use an example RESTful Web service to show that, for the purposes of semantic automation, we can map the resources-and-hypermedia structure of a RESTful service into the service model as a set of operations. This allows us to integrate RESTful services with WS–∗ services in a single semantic automation approach.

Below, Section 6.1.1 shows the example RESTful service, and Section 6.1.2 uses this example to show the mapping of a hypermedia structure to our model.

6.1.1 Example RESTful Web Service

Figure 6.2(a) depicts an example RESTful hotel booking service, with its resources and the links among them; we use this synthetic example to demonstrate how a hypermedia service can naturally be viewed as a set of operations.

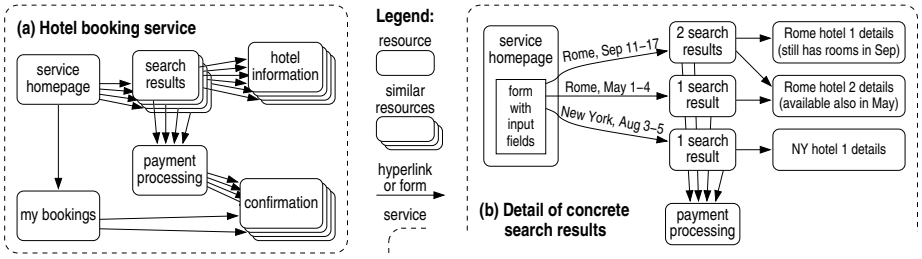


Figure 6.2: Structure of an example RESTful hotel reservation service: (a) showing types of resources, (b) showing details of search results resources with example searches

The "service homepage" is a resource with a stable address and information about the other resources that make up the service. It serves as the initial entry point for client interaction. In a human-oriented Web application, this would be the homepage, such as `http://hotels.example.com/`.

The existence of such a stable entry point lowers the coupling between the service and its clients, and it enables the evolution of the service, such as adding or removing functionality. A client need only rely on the existence of the fixed entry point, and it can discover all other functionality as it navigates the hypermedia. In contrast, in service-description-driven distributed computing technologies, such as WS–∗ Web services, the client is often programmed against a given service description before it uses the service, making it harder to react dynamically to changes of the service.

The homepage resource of our example service contains a form for searching for available hotels, given a number of guests, a start date and the duration of the stay, and a location. The search form serves as a parametrized hyperlink to search results resources, as shown in Figure 6.2(b), one resource per every unique combination of the input data — the form prescribes how to create a URI that contains the input data; the URI then identifies a resource with the search results. As there is a large number of possible search queries, there is also a large number of results resources, and the client does not need to know that all these resources are likely handled by a single software component on the server.

The search results are modeled as separate resources (as opposed to, for instance, a single data-handling resource that takes the query parameters in an input message) because it simplifies the reuse of the hotel search functionality in other services or in mashups (lightweight compositions of Web applications), and because it also supports caching of the results. With individual search results resources, creating the appropriate URI and retrieving the results (with HTTP GET) is easier in most programming frameworks than POSTing the input data in a structured data format to one Web resource, which would then reply with the search results.

In this example, search results are presented as a list of concrete rates available at the hotels in the given location, for the given dates and the number of guests, as also shown in Figure 6.2(b). Each item of the list contains a link to further information about the hotel (e.g. the precise location, star rating and other descriptions), and a form for booking the rate, which takes as input

the payment details (such as credit card information) and an identification of the guest(s) who will stay in the room. The form data is submitted (with the POST method) as a booking request to a payment resource, which processes the booking and redirects the client to a newly created confirmation resource. The content of the confirmation can serve as a receipt.

The service homepage resource also links to "my bookings", a resource listing the bookings of the current user (who is identified through a suitable authentication functionality). This resource links to the confirmations of the bookings done by the authenticated user. With such a resource available to them, client applications no longer need to store the information about performed bookings locally.

The confirmation resources may further provide a way of canceling the reservation (not shown in the picture, could be implemented with the HTTP DELETE method).

Together, all the resources we've described here form the hotel booking service. However, since the involved Web technologies actually work on the level of resources, *service* is a virtual term here and Figure 6.2 shows the service in a dashed box.

6.1.2 Viewing Hypermedia as Operations

While the resources of the service (the *nouns*) form a hypermedia graph (shown in Fig. 6.2), the interaction of a client with a RESTful service is a series of operations (the *verbs* or *actions*) where the client sends a request to a resource and receives a response that may link to further useful resources. The hypermedia graph (the links between resources) guides the sequence of operation invocations, but the meaning of a resource is independent of where it is linked from; the same link or form, wherever it is placed, will always lead to the same action. Therefore, the operations of a RESTful Web service can be considered independently from the graph structure of the hypertext. In fact, RESTful services are commonly documented as sets of operations available to the clients — as APIs.

In Table 6.1, we summarize the mapping from the Web-architecture-based model of RESTful services into the WSMO-Lite service model. Effectively, an automated client (such as a semantic automation system) can view RESTful services through the model from Figure 4.2, with the operations being the methods available on the resources that constitute the RESTful services. This common view allows us to support WS–∗ and RESTful services in WSMO-Lite without regard to their technological differences.

RESTful services	**WSMO-Lite service model**
Service (*a group of resources*)	Service
Resource	– (*not modeled explicitly*)
Resource method	Operation
Method request/response	Operation input/output message
Hyperlink	– (*treated as part of message data*)

Table 6.1: Mapping of RESTful services to WSMO-Lite service model

Our description of the example hotel reservation service in the preceding

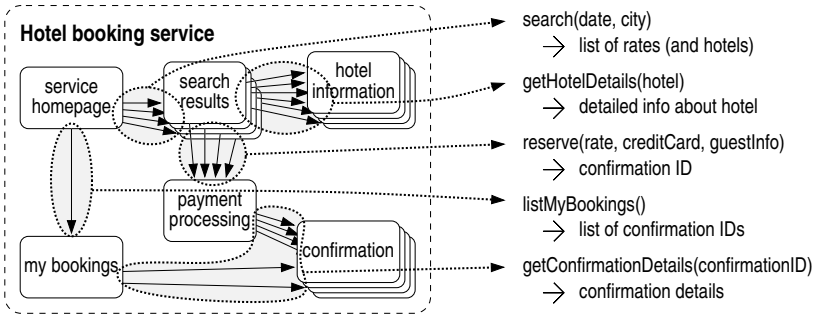


Figure 6.3: Operations of the example service from Figure 6.2

subsection has been based on the hypermedia aspect: we described the resources and how they link to each other. Figure 6.3 demonstrates the mapping: we extract here the operations present in the example service. The search form in the homepage represents a search operation, the hotel information pages linked from the search results can be viewed as an operation for retrieving hotel details, the reservation form for any particular available rate becomes a reservation operation, and so on.

In summary, we have shown here that the service model from Figure 4.2 is an appropriate model for capturing the structure of RESTful Web services for the purpose of semantic automation.

6.2 Describing RESTful Web Services in HTML

In order to provide semantic automation of the usage of RESTful Web services, the semantic client processes machine-readable descriptions of the available services. In the preceding section, we have shown that the WSMO-Lite service model is a suitable abstraction for these descriptions. We proceed now to discuss their concrete syntax.

Public RESTful Web services are commonly described in human-oriented documentation using the general-purpose Web hypertext language HTML. In contrast to WS–∗ Web services, public RESTful services are rarely described in a machine-readable format that could be annotated semantically to support SWS automation, despite the existence of languages such as WADL (see Section 3.4.4). Therefore, in this section we propose a *microformat* (see Section 3.4.2) that can mark up the machine-oriented pieces of information described within the textual HTML documentation. With the microformat markup in place, the information becomes machine-readable and can be annotated with semantics.[1]

Still, while the majority of RESTful Web services only provides HTML documentation, there are nevertheless some machine-readable (and machine-oriented) descriptions of RESTful services and resources on the Web; we turn our attention to such descriptions in Section 6.3.

[1]We have chosen to express the service description markup in microformats and not microdata (cf. Section 3.4.2) because the latter is a working draft in development, and because microformats capture machine-oriented data in *valid* HTML without extensions.

This section is structured as follows: Section 6.2.1 demonstrates how HTML is used to describe RESTful Web services, Section 6.2.2 presents the hRESTS microformat for identifying key pieces of the descriptions in a machine-readable way, and Section 6.2.3 extends this microformat with SAWSDL-like semantic annotations. Finally, in Section 6.2.4 we discuss how alternatively to using microformat syntax, we could use RDFa to encode the WSMO-Lite service structure and semantic annotations in HTML service documentation.

6.2.1 Example HTML Description of a RESTful Web Service

As already stated, public Web services generally come with HTML documentation.[2] Textual documentation in HTML is the prevalent form of Web API descriptions, and in many cases it is the only one. Typically, such documentation will list the available operations (under various names such as *API calls*, *methods*, *commands* etc.), their URIs and parameters, the expected output data, any error conditions and so on; it is, after all, intended as the documentation of a programmatic interface.

The following might be an excerpt of a typical operation description:

> ACME Hotels service API
> **Operation getHotelDetails**
> Invoked using the method GET at `http://example.com/h/{id}`
> **Parameter:** `id` - the identifier of the particular hotel
> **Output value:** hotel details in an `ex:hotelInformation` document

Such documentation has all the details necessary for a human to be able to create a client program that can use the service. This documentation can be captured in HTML as shown in Listing 6.1. In order to tease out the technical details (operations, addresses, HTTP methods, input and output data formats), the HTML documentation needs to be amended in some way; in the following subsection, we describe hRESTS, a microformat developed for this purpose.

```
1  <h1>ACME Hotels service API</h1>
2  <h2>Operation getHotelDetails</h2>
3
4  <p> Invoked using the method GET at http://example.com/h/{id} <br/>
5    <strong>Parameter:</strong>
6      <code>id</code> – the identifier of the particular hotel
7    <br/>
8    <strong>Output value:</strong> hotel details in an
9      <code>ex:hotelInformation</code> document
10 </p>
```

Listing 6.1: Example HTML service description

6.2.2 hRESTS: Service Model Microformat

The preceding section shows a typical HTML description of one operation of a RESTful Web service. Here, we proceed to define hRESTS, a microformat

[2] Such as `flickr.com/services/api` and `docs.amazonwebservices.com/AmazonSimpleDB/2007-11-07/DeveloperGuide`

that adds machine-readable service model structure to such descriptions. Additionally, hRESTS identifies two key pieces of information about an operation: the *address* (URI template) of the resource(s) on which the operation acts, and the *HTTP method* represented by the operation. The information about an operation's address and method enables tool support for invocation. In effect, hRESTS is analogous to WSDL, albeit much less detailed.

The hRESTS microformat is made up of a number of HTML classes that correspond to the various parts of the WSMO-Lite service model:

- `service` specifies the description of the whole service,
- `operation` marks a single operation within that service,
- `address` defines the URI template for an operation,
- `method` defines the HTTP method for an operation,
- `input` marks a block describing the inputs of an operation,
- `output` does the same for the outputs, and
- `label` specifies a human-readable name of a service or of an operation.

Table 6.2 details the use of these classes (the last column specifies an RDF mapping discussed below).

HTML class	Target markup	Children	RDF counterpart
`service`	block	`label`, `operation`, `address`, `method`	wl:Service
`operation`	block	`label`, `input`, `output`, `address`, `method`	wl:Operation, wl:hasOperation
`input`	block	—	wl:Message, wl:hasInputMessage
`output`	block	—	wl:Message, wl:hasOutputMessage
`address`	textual or hyperlink	—	hr:hasAddress
`method`	textual	—	hr:hasMethod
`label`	textual	—	rdfs:label

Table 6.2: The HTML classes of the hRESTS microformat

Listing 6.2 shows the RDFS definition of the two invocation-information properties (hr:hasMethod and hr:hasAddress, in the namespace `http://www.wsmo.org/ns/hrests#`), which extend the RDFS schema for the WSMO-Lite service model defined in Listing 4.1 (page 64):

hr:hasAddress (lines 8–10) is a property that links an operation with the resource URI template. There is currently no standard datatype for URI templates, therefore we introduce one on line 16.

hr:hasMethod (lines 11–13) is a property that links an operation with the HTTP method that should be invoked on the target resource.

```
1  @prefix hr: <http://www.wsmo.org/ns/hrests#> .
2  @prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
3  @prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
4  @prefix wl: <http://www.wsmo.org/ns/wsmo-lite#> .
5  @prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
6
7  # added properties for wl:Operation
8  hr:hasAddress a rdf:Property ;
9     rdfs:domain wl:Operation ;
10    rdfs:range hr:URITemplate .
11 hr:hasMethod a rdf:Property ;
12    rdfs:domain wl:Operation ;
13    rdfs:range xsd:string .
14
15 # a datatype for URI templates
16 hr:URITemplate a rdfs:Datatype .
```

Listing 6.2: MicroWSMO extensions to the service model from Section 4.4

Listing 6.3 shows hRESTS annotations (in bold) in the HTML code of the sample service description shown earlier.[3] This listing helps us illustrate the following detailed definitions of the hRESTS microformat classes themselves and the structural constraints on hRESTS descriptions defined near the end of this section.

```
1  <div class="service" id="svc">
2    <h1><span class="label">ACME Hotels</span> service API</h1>
3    <div class="operation" id="op1">
4      <h2>Operation <span class="label">getHotelDetails</span></h2>
5      <p> Invoked using the <span class="method">GET</span>
6        at <code class="address">http://example.com/h/{id}</code><br/>
7        <span class="input">
8          <strong>Parameter:</strong>
9            <code>id</code> – the identifier of the particular hotel
10       </span><br/>
11       <span class="output">
12         <strong>Output value:</strong> hotel details in an
13           <code>ex:hotelInformation</code> document
14       </span>
15     </p>
16 </div></div>
```

Listing 6.3: Example hRESTS service description

In the following detailed definitions, we refer to RDF classes and properties from the service model from Section 4.4 (with the prefix wl), and the additional properties defined in Listing 6.2 for invocation of RESTful services (with the prefix hr).

The `service` class on block markup (e.g. `<body>`, `<div>`), as shown in Listing 6.3 on line 1, indicates that the contents of the element describe a service API using the hRESTS microformat. An element with the class `service` corresponds to an instance of wl:Service. A service contains one or more operations and may have a label (see below).

[3] Note the added `<div>` and `<span>` blocks that add nested element structure to the description; they do not otherwise affect the presentation of the HTML documentation.

The `operation` class, also used on block markup (e.g. `<div>`), indicates that the element contains a description of a single Web service operation, as shown in the listing on line 3. An element with this class corresponds to an instance of wl:Operation, attached to its parent service with wl:hasOperation. An operation description specifies the address and the method used by the operation, and it may also contain description of the input and output of the operation, and finally a label.

The `address` class is used on textual markup (e.g. `<code>`, shown on line 6, or `<abbr>`) or on a hyperlink (`<a href>`) and specifies the URI of the operation, or the URI template in case any inputs are URI parameters. Its value is attached to the operation using hr:hasAddress. On a textual element, the address value is in the content; on an abbreviation, the expanded form (the `title` of the abbreviation) specifies the address; and on a hyperlink, the target of the link specifies the address of the operation.

The `method` class on textual markup (e.g. `<span>`, shown on line 5) specifies the HTTP method used by the operation. Its value is attached to the operation using hr:hasMethod.

Both the address and the method may also be specified on the level of the service, in which case these values serve as defaults for operations that do not specify them. In absence of any explicit value for method, the default is GET. The RDF form of the service model reflects the default values already applied, that is, an instance wl:Service will never have either hr:hasMethod or hr:hasAddress, while an instance of wl:Operation should always have both.

The `input` and `output` classes are used on block markup (e.g. `<div>` but also `<span>`), as shown on lines 7 and 11, to indicate the description of the input or output of an operation. Elements with these classes correspond to instances of wl:Message, attached to the parent operation with wl:hasInputMessage and wl:hasOutputMessage respectively. While hRESTS does not provide for further machine-readable information about the inputs and outputs, extensions such as MicroWSMO (cf. Section 6.2.3) and SA-REST [109] may add more properties here.

In principle, the output data format can be self-describing through the metadata the client receives together with the operation response (self-description is a major part of Web architecture, and the response message itself may contain a pointer to a semantic lifting transformation), but it is, in general, useful for API descriptions to specify what the client can expect; hence our `output` class.

The `label` class is used on textual markup to specify human-readable labels for services and operations, as shown on lines 2 and 4 in the example listing. The value is attached to the appropriate service or operation using rdfs:label.

Additionally, elements with the classes `service` or `operation` can carry an `id` attribute, which is combined with the URI of the HTML document to form the URI identifier of the particular service or operation. This will allow other statements to directly refer to these instances.

The definitions above imply a hierarchical use of the classes within the element structure of the HTML documentation. The following is a complete list of structural constraints on the hierarchy of elements marked up with hRESTS classes. It reflects the structure of our service model, amended with the defaulting of the `address` and `method` properties:

1. No XHTML element has two or more hRESTS classes at the same time.

2. No element with the class `service` is a descendant[4] of an element with any hRESTS class.
3. Either there is no element with the class `service` in the document, or every element with the class `operation` is a descendant of an element with the class `service`.
4. No element with the class `operation` is a descendant of an element with an hRESTS class other than `service`.
5. Every element with the class `address`, `method` or `label` is a descendant of an element with either the class `service` or the class `operation`.
6. Every element with the class `input` or `output` is a descendant of an element with the class `operation`.
7. An element with the class `operation` can only contain one descendant element with the class `input` and one descendant element with the class `output`.
8. No element with any of the classes `address`, `method`, `input`, `output` or `label` is a descendant of an element with an hRESTS class other than `service` and `operation`.

A single HTML document can define multiple services; such a document will contain multiple elements with the class `service`.

Conversely, and this is a common occurrence, multiple documents can together make up the description of a single service. Indeed, textual service documentation is often split into a number of interlinked pages that describe the service as a whole, the individual operations, data types, error conditions, specific authentication mechanisms etc. In such cases, the Web page describing an operation (or a group of operations) will not contain any element with the class `service` because it is described elsewhere. The documents that together make up the service description should contain metadata links pointing to the first document in the set (`rel="start"` as defined by HTML [45]) and to the documents that make up the set (`rel="section"`) — such links can help a crawler to find related pieces of the service description.

Such a situation is illustrated in Figure 6.4, which shows an overview page on the left that talks about the service as a whole, and three pages on the right that describe one operation each. The `start` and `section` relation links tie the pages together, which can be interpreted in our RDF model as a description of a single service with three operations.

As a consequence, a Web page pointed to by a link with `rel="start"` should contain only a single element with the class `service` so that the assignment of the operations to the service is unambiguous.

The next subsection contains an example of HTML semantically annotated with hRESTS and it shows the RDF data generated from such HTML by a hRESTS parser (cf. Section 8.2.2).

[4]The term *descendant* is defined for XML/HTML elements in XPath [141].

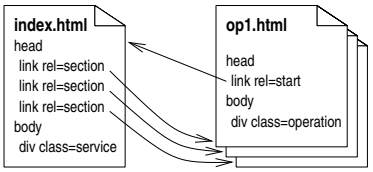


Figure 6.4: Service description in multiple documents

6.2.3 MicroWSMO Sem. Annotations: an Extension of hRESTS

The hRESTS microformat structures the HTML documentation of RESTful Web services so they are amenable to machine processing. The microformat identifies key pieces of information that are already present in the documentation, effectively creating an analogue of the machine-readable WSDL descriptions for WS–∗ services. hRESTS forms the basis for further extensions, where service descriptions are annotated with added information to facilitate further processing. In this section, we present an extension of hRESTS called MicroWSMO ("Microformat for WSMO-Lite"[5]), which adds semantic annotations; another extension is SA-REST [109, 33], which adds information for faceted browsing and discovery of services by client developers.

Similarly to how SAWSDL is a layer for semantic annotations of WSDL (the machine-readable service description language with support in development tools), also MicroWSMO is a layer for semantic annotations of hRESTS (the service description microformat that aims to provide for development tool support). Because the hRESTS view of services (Section 6.1.2) is so similar to that of WSDL, MicroWSMO can adopt SAWSDL-style properties to add semantic annotations conforming to the WSMO-Lite service ontology. See Section 4.3.3 for information about our use of the SAWSDL properties in our model of service semantics.

SAWSDL annotations are URIs that identify semantic concepts and data transformations. The annotation URIs can be added to the HTML documentation of RESTful services in the form of hypertext links. HTML [45] defines a mechanism for indicating the relation represented by a hyperlink; the relation is specified in the `rel` attribute. Along with `class`, the `rel` attribute is also commonly used by microformats.

In accordance with SAWSDL, MicroWSMO consists of the following three types of link relations:

- `model` indicates that the link is a model reference,
- `lifting` and `lowering` then denote links to the respective data transformations.

Listing 6.4 illustrates the use of these link relations on semantic annotations added to the hRESTS description from Listing 6.3.

In the following detailed definitions, we use the prefix sawsdl to refer to the SAWSDL RDF properties included in the service model in Section 4.4.

[5]MicroWSMO is so named for historical reasons; a more direct name "SA-hRESTS" would be confusingly close to SA-REST [109]; and another alternative, "MicroSAWSDL", would imply close ties with WSDL, which would be undesirable with RESTful services.

```
1  <div class="service" id="svc">
2    <h1><span class="label">ACME Hotels</span> service API</h1>
3      <p>This service is a
4        <a rel="model" href="http://example.com/ecommerce/hotelReservation">
5          hotel reservation</a> service.
6      </p>
7    <div class="operation" id="op1">
8      <h2>Operation <span class="label">getHotelDetails</span></h2>
9      <p> Invoked using the <span class="method">GET</span>
10       at <code class="address">http://example.com/h/{id}</code><br/>
11       <span class="input">
12         <strong>Parameter:</strong>
13         <a rel="model" href="http://example.com/data/onto.owl#Hotel">
14           <code>id</code></a> – the identifier of the particular hotel
15           (<a rel="lowering" href="http://example.com/data/hotelID.xsparql">
16            lowering</a>)
17       </span><br/>
18       <span class="output">
19         <strong>Output value:</strong> hotel details in an
20           <code>ex:hotelInformation</code> document
21       </span>
22     </p>
23 </div></div>
```

Listing 6.4: Example MicroWSMO semantic description

The `model` link relation, on a hyperlink present within an hRESTS `service`, `operation`, `input` or `output` block, specifies that the link is a model reference (sawsdl:modelReference in the RDF mapping) from the respective component to its semantic description. We can directly apply WSMO-Lite annotations here, as discussed in Section 4.3.3 and summarized in Table 4.2.

Listing 6.4 shows the use of the `model` link relation on lines 4 and 13. Line 4 specifies that the service does hotel reservations (the URI would identify a category in some classification of services), and line 13 defines the input of the operation to be an instance of the class Hotel, which would be a part of the service's data ontology.

The `lifting` and `lowering` link relations, on hyperlinks present within hRESTS `input` or `output` blocks (mapping to the RDF properties sawsdl:liftingSchemaMapping and sawsdl:loweringSchemaMapping), specify that the links point to the respective data transformations between the knowledge representation format of the service's data ontology and the wire syntax of the messages of the service. Section 6.4 discusses issues of data lifting and lowering transformations in the context of RESTful services.

Listing 6.4 shows a link to a lowering transformation on line 15. The transformation would presumably map a given instance of the class Hotel into the ID that the service expects as a URI parameter.

Finally, Listing 6.5 shows the RDF data that can be extracted from the example MicroWSMO description from Listing 6.4, using the GRDDL XSLT transformation defined in Section 8.2.2.

As shown in this section, MicroWSMO allows HTML service documentation to be annotated with service semantics in the same way that WSDL is annotated with SAWSDL. The RDF data parsed from hRESTS and MicroWSMO has the same structure as the RDF data obtained from WS–∗ descriptions as described

```
@prefix ex: <http://example.com/serviceDescription.html#> .
@prefix hr: <http://www.wsmo.org/ns/hrests#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix sawsdl: <http://www.w3.org/ns/sawsdl#> .
@prefix wl: <http://www.wsmo.org/ns/wsmo-lite#> .

ex:svc a wl:Service ;
  rdfs:isDefinedBy <http://example.com/serviceDescription.html> ;
  rdfs:label "ACME Hotels" ;
  sawsdl:modelReference <http://example.com/ecommerce/hotelReservation> ;
  hr:hasOperation ex:op1 .
ex:op1 a wl:Operation ;
  rdfs:label "getHotelDetails" ;
  hr:hasMethod "GET" ;
  hr:hasAddress "http://example.com/h/{id}"^^hr:URITemplate ;
  wl:hasInputMessage [
    a wl:Message ;
    sawsdl:modelReference <http://example.com/data/onto.owl#Hotel> ;
    sawsdl:loweringSchemaMapping <http://example.com/data/hotelID.xsparql>
  ] ;
  wl:hasOutputMessage [
    a wl:Message ;
  ] .
```

Listing 6.5: RDF data extracted from Listing 6.4

in Table 5.2, and illustrated in Figure 5.2. Thus we can treat RESTful services just like WS–∗ services and provide the same level of semantic automation, as discussed in the later chapters of this thesis.

6.2.4 RDFa: an alternative to hRESTS and MicroWSMO

Alternatively to introducing microformats to capture the service model structure and semantic annotations in the HTML documentation of RESTful Web services, we could also employ RDFa (see Section 3.4.2) and use the RDF-based WSMO-Lite service model directly. RDFa specifies a collection of generic XML attributes for expressing arbitrary RDF data inside HTML.

Since our service description data is ultimately processed as RDF, RDFa would be directly applicable. In our case, the difference between the use of a microformat or RDFa boils down to several considerations:

- the microformat syntax is simpler and more compact than RDFa;
- HTML marked up with our microformat remains valid HTML, whereas RDFa currently only validates against the newest schemas;
- RDFa represents the full concept URIs and thus facilitates the coexistence of multiple data vocabularies in a single document, where microformats may run into naming conflicts;
- processing microformats requires vocabulary-specific parsers (such as our XSLT transformation mentioned in Section 6.2.2), while parsing the RDF data from RDFa is independent from any actual data vocabularies;
- RDFa cannot support domain-specific syntactic shortcuts such as a service automatically getting an `rdfs:isDefinedBy` property linking back to the

```
<div typeof="wl:Service" about="#svc"
     xmlns:wl="http://www.wsmo.org/ns/wsmo-lite#"
     xmlns:sawsdl="http://www.w3.org/ns/sawsdl#"
     xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">
  <span rel="rdfs:isDefinedBy" resource="" />
  <h1><span property="rdfs:label">ACME Hotels</span> service API</h1>
    <p>This service is a
      <a rel="sawsdl:modelReference"
          href="http://example.com/ecommerce/hotelReservation">
        hotel reservation</a> service.
    </p>
  <div rel="wl:hasOperation"><div typeof="wl:Operation" about="#op1">
    <h2>Operation <code property="rdfs:label">getHotelDetails</code></h2>
...
```

Listing 6.6: Example service description with RDFa annotations

service description, or the defaulting of address and method properties from a service to its operations;

- RDFa can be used to embed not only hRESTS and SAWSDL properties, but also any other RDF data (cf. Section 6.6).

We illustrate the RDFa form of the WSMO-Lite service model with SAWSDL annotations (instead of the hRESTS/MicroWSMO microformats) with a brief snippet in Listing 6.6, highlighting the differences from Listing 6.4. The RDFa form uses XML namespaces to distinguish vocabularies, and it uses the attributes `typeof`, `about`, `rel`, `resource`, and `property` (among others) to express any RDF content.

6.3 Other Technologies for Describing RESTful Services

Even though most of the prominent public RESTful Web services are indeed only described in HTML documentation, there are also instances of machine-processable service description formats. Some are application-specific, such as the Service Document format introduced by the Atom Publishing Protocol or the OpenSearch Description Document (both formats are described in Section 3.4.3); others are general description formats[6] such as WADL (cf. Section 3.4.4) that aim to capture the details of the syntactic contract of a set of Web resources. Even Web hyperlinks and forms can be interpreted as limited syntactic contract descriptions of Web resources.

While a comprehensive survey of such Web service description approaches is out of scope of this thesis, in this section we select representative examples and we show that the service descriptions can be transformed to our service model, and that either SAWSDL or hRESTS/MicroWSMO can be adopted to add semantic annotations and thus to integrate these service description approaches (and the underlying services and resources) into our lightweight semantic automation system.

[6] `http://pacificspirit.com/Authoring/REST/` is one list of such description languages.

WSDL 2.0 can also be used to describe RESTful services — its potential limitations in this application are the topic of ongoing debates in the community around RESTful Web services.[7] Inasmuch as WSDL 2.0 can describe RESTful Web services, it can be annotated with SAWSDL and WSMO-Lite as shown in Chapter 5 of this thesis.

Below in Section 6.3.1, we discuss the application of SAWSDL and WSMO-Lite to WADL. In Section 6.3.2, we apply SAWSDL and WSMO-Lite to Atom-Pub Service Documents — an example current and standardized application-specific service description format. Finally, in Section 6.3.3 we analyze how Web hyperlinks and forms can be interpreted as service descriptions and how the hRESTS and MicroWSMO microformats can be utilized to extend them with further structural and semantic annotations.

6.3.1 Web Application Description Language (WADL)

In Section 3.4.4, we described the Web Application Description Language, a generic RESTful service description format that has slowly gained certain traction, stronger than any other such approach that we have seen. Here we show how it fits within our service description approach and how it can be annotated with service semantics.

The top-level concept of WADL is an *application*. WADL describes an application as a set of *resources*. For each resource, WADL captures its address as a URI template, and the *methods* that are available on the resource. For every method, WADL identifies the request (input) parameters, and the request and response (input and output) data formats. WADL can even point out pieces of data that serve as links to other resources and thus describe the hypertext structure of the application.

The structure of WADL descriptions very clearly fits our functional model of RESTful services, illustrated in Figure 6.1. A WADL *application* corresponds to our term *Web service*, and a *method* on a *resource* corresponds to our term *operation*.

In our semantic service descriptions, we put semantic annotations on the Web service, on its operations and on their inputs and outputs. A WADL service description can be annotated in a straightforward way with SAWSDL attributes and WSMO-Lite semantics (applied to WSDL in Chapter 5). To specify functional and nonfunctional semantics of a service, we can put a modelReference on the top-level `wadl:application` element. Behavioral semantics are specified by putting modelReferences with functional annotations on `wadl:method` elements. And finally, the information model semantics are specified with modelReference, liftingSchemaMapping and loweringSchemaMapping inside the `wadl:request` or `wadl:response` element structure. This is shown in Table 6.3, modeled after Table 5.1.

In effect, WADL accepts the same annotations as WSDL, with the same function. In addition, since WADL describes RESTful services, the semantics of HTTP operations and hyperlinking can also be taken into account; we describe the semantics inherent in RESTful services later in Section 6.5.

[7] `http://jonathanmarsh.net/2008/09/15/mapping-rest-services-to-operations/` and `http://sanjiva.weerawarana.org/2008/09/blog-on-describing-rest-with-wsdl-20.html` are blog posts that illustrate the ongoing debates; also see the links in those posts.

Sem. type	WSMO-Lite svc. model	WADL component
F	Service	Application
N	Service	Application
B	Operation	Method
I	Message	Request, response, param, representation, XS element decl. or type definition

Table 6.3: WSMO-Lite annotations in WADL

Optimally, providers of RESTful Web services would adopt a language such as WADL and use it to publish machine-readable descriptions of their services, as this would stimulate the development of tools that simplify client-side code.[8] We have demonstrated here that if a language such as WADL does gain adoption, our semantic service automation approach can be applied to this language in a straightforward fashion.

6.3.2 AtomPub Service Document

In Section 3.4.3, we described the Atom Publishing Protocol, along with the structure of the AtomPub Service Document format. To recapitulate it quickly, an AtomPub Service Document describes a single *service*, which consists of one or more *workspaces*, each made up of one or more *collections*.

The Atom Publishing Protocol defines operations available on *collections*, but it does not define any operations on *services* and *workspaces*, nor does it specify any relationship between collections inside a single workspace, or between the workspaces of a single service. Therefore, an AtomPub *collection* can be seen as an independent Web service with well-known publication functionality. We can translate any AtomPub collection description into an instance of wl:Service, with operations as described below. To avoid confusion with the top-level AtomPub service concept, the rest of this section uses the word "service" exclusively to refer to the Web service that is a single AtomPub collection.

The Atom Publishing Protocol defines the functionality of collection services; we introduce a WSMO-Lite functionality category cat:AtomPubCollection that captures the functional semantics of these services — in other words, the instance of wl:Service that is generated for an AtomPub collection will have a modelReference that points to the category AtomPubCollection. Publications that extend or restrict the Atom Publishing Protocol may advertise these modifications by putting additional category URI(s) in a sawsdl:modelReference attribute on the appropriate `app:collection` element inside the Service Document.

Listing 6.7 shows an example AtomPub Service Document (adopted from Listing 3.1) that indicates (on line 7 using the hypothetical extension category RewritableCollection) that the collection can be rewritten as a whole.

An AtomPub collection service may also advertise its nonfunctional properties (cf. Chapter 4), likewise using the sawsdl:modelReference attribute on the `app:collection` element, as shown in the listing on line 8.

[8] For instance, `wadl2java` (`https://wadl.dev.java.net/wadl2java.html`) generates client-side stubs to replace the need to deal with raw HTTP. Such tools in a limited form could also be developed for hRESTS; this is, however, out of scope for this thesis.

```
1  <service xmlns="http://www.w3.org/2007/app" xmlns:atom="http://www.w3.org/2005/
       Atom"
2            xmlns:sawsdl="http://www.w3.org/ns/sawsdl#">
3    <workspace>
4      <atom:title>Main Site</atom:title>
5      <collection href="http://example.org/blog/main"
6          sawsdl:modelReference="
7                  http://example.com/atompub#RewritableCollection
8                  http://example.com/blog/meta#pricing" >
9        <atom:title>My Blog Entries</atom:title>
10       <categories href="http://example.com/cats/forMain.cats" />
11       <accept>application/atom+xml;type=entry</accept>
12     </collection>
13   </workspace>
14 </service>
```

Listing 6.7: Example semantically-annotated AtomPub Service Document

The Atom Publishing Protocol standard prescribes the operations that are available on the two types of resources (collections and entries) it defines; on a collection the operations are

- *list collection entries,*
- *submit a new collection entry;*

and on an entry they are

- *retrieve an entry,*
- *update an entry,*
- *delete an entry.*

These five operations are available on all AtomPub services. As shown in Listing 6.8, we can generate the respective five wl:Operation instances together with the HTTP method information for each of them, and with the concrete address for the collection operations. The URIs of the entries are determined by the service and linked from the collection; therefore the address URI template for the entry operations is the whole URI as a template parameter (lines 41, 56, 70). The data-retrieval operations (list collection entries, retrieve an entry) are also marked to be *safe* (lines 19 and 43).

The information model of the Atom Publishing Protocol service is the Atom Syndication Format, whose schema can easily be translated into an ontology. Creating such an ontology is out of scope of this thesis; however, Listing 6.8 shows message annotations (e.g. lines 31, 36) using terms that would likely occur in an ontology for Atom and AtomPub, together with example lifting and lowering mapping pointers (e.g. lines 32, 37).

To summarize, we can transform an AtomPub Service Document into a WSMO-Lite description, with specific annotations describing the functionality and the nonfunctional properties of a particular publication.

Just as AtomPub service documents can be annotated with SAWSDL and viewed as a well-known type of RESTful Web services, we can expect to be able to apply annotations on other standardized application-specific service descriptions, and make whole classes of services amenable to semantic automation.

```
1  @prefix hr: <http://www.wsmo.org/ns/hrests#> .
2  @prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
3  @prefix sawsdl: <http://www.w3.org/ns/sawsdl#> .
4  @prefix wsdlx: <http://www.w3.org/ns/wsdl-extensions#> .
5  @prefix wl: <http://www.wsmo.org/ns/wsmo-lite#> .
6  @prefix wlx: <http://www.wsmo.org/ns/wsmo-lite-extensions#> .
7  @prefix wlxt: <http://www.wsmo.org/ns/wsmo-lite-extensions/transform/> .
8
9  <http://example.org/blog/main> a wl:Service ;
10     rdfs:label "My Blog Entries" ;
11     sawsdl:modelReference <http://example.com/atompub#RewritableCollection>,
12          <http://example.com/blog/meta#pricing> ;
13     wl:hasOperation _:listEntries, _:submitEntry, _:getEntry, _:updateEntry, _:
        deleteEntry .
14
15 _:listEntries a wl:Operation ;
16     rdfs:label "List collection entries" ;
17     hr:hasAddress "http://example.org/blog/main"^^hr:URITemplate ;
18     hr:hasMethod "GET" ;
19     sawsdl:modelReference wsdlx:SafeInteraction ;
20     wl:hasOutputMessage [
21       a wl:Message ;
22       sawsdl:modelReference wlx:AtomCollection ;
23       sawsdl:liftingSchemaMapping wlxt:atomlifting.xslt ;
24     ] .
25 _:submitEntry a wl:Operation ;
26     rdfs:label "Submit a new entry" ;
27     hr:hasAddress "http://example.org/blog/main"^^hr:URITemplate ;
28     hr:hasMethod "POST" ;
29     wl:hasInputMessage [
30       a wl:Message ;
31       sawsdl:modelReference wlx:AtomEntry ;
32       sawsdl:loweringSchemaMapping wlxt:atomlowering.xslt ;
33     ] ;
34     wl:hasOutputMessage [
35       a wl:Message ;
36       sawsdl:modelReference wlx:AtomEntry, wlx:AtomEntryCreationConfirmation ;
37       sawsdl:liftingSchemaMapping wlxt:atomlifting-submit.xslt ;
38     ] .
```

```
39 _:getEntry a wl:Operation ;
40     rdfs:label "Retrieve an entry" ;
41     hr:hasAddress "{entryURI}"^^hr:URITemplate ;
42     hr:hasMethod "GET" ;
43     sawsdl:modelReference wsdlx:SafeInteraction ;
44     wl:hasInputMessage [
45       a wl:Message ;
46       sawsdl:modelReference wlx:AtomEntry ;
47       sawsdl:loweringSchemaMapping wlxt:atomlowering-uri.xslt ;
48     ] ;
49     wl:hasOutputMessage [
50       a wl:Message ;
51       sawsdl:modelReference wlx:AtomEntry ;
52       sawsdl:liftingSchemaMapping wlxt:atomlifting.xslt ;
53     ] .
54 _:updateEntry a wl:Operation ;
55     rdfs:label "Update an entry" ;
56     hr:hasAddress "{entryURI}"^^hr:URITemplate ;
57     hr:hasMethod "PUT" ;
58     wl:hasInputMessage [
59       a wl:Message ;
60       sawsdl:modelReference wlx:AtomEntry ;
61       sawsdl:loweringSchemaMapping wlxt:atomlowering-uri-body.xslt ;
62     ] ;
63     wl:hasOutputMessage [
64       a wl:Message ;
65       sawsdl:modelReference wlx:AtomEntryUpdateConfirmation ;
66       sawsdl:liftingSchemaMapping wlxt:atomlifting-update.xslt ;
67     ] .
68 _:deleteEntry a wl:Operation ;
69     rdfs:label "Delete an entry" ;
70     hr:hasAddress "{entryURI}"^^hr:URITemplate ;
71     hr:hasMethod "DELETE" ;
72     wl:hasInputMessage [
73       a wl:Message ;
74       sawsdl:modelReference wlx:AtomEntry ;
75       sawsdl:loweringSchemaMapping wlxt:atomlowering-uri.xslt ;
76     ] ;
77     wl:hasOutputMessage [
78       a wl:Message ;
79       sawsdl:modelReference wlx:AtomEntryDeletionConfirmation ;
80       sawsdl:liftingSchemaMapping wlxt:atomlifting-delete.xslt ;
81     ] .
```

Listing 6.8: Representation of the example from Lst. 6.7 in our service model

6.3.3 Links and Forms

The Web is made up of a tremendous number of resources, most of them oriented for direct human use and interaction through a Web browser. There are, nevertheless, many machine-oriented resources, some of which are explicitly designated as parts of RESTful Web services (or Web APIs). The preceding content of this chapter has discussed how these resources can be described in a machine-processable format with semantic annotations. The Web also contains many machine-oriented resources that are not designated as Web services, probably mainly because their providers have not considered them as such.

Examples of such resources include simple machine-oriented data sources, such as financial data or weather information in an XML format, dictionary translation etc., or even services with effects other than serving information, ranging from simple mailing list subscription/unsubscription interfaces even to applications such as online banking.

Some of these *potential*[9] Web services are made available with HTML forms, and even with HTML hyperlinks (cf. Section 3.4.2) in the case of the simplest data sources with no query or filtering functionality. In this section, we describe how the hRESTS and MicroWSMO microformats can also be used to describe such resources as RESTful Web services.

In the description of our example service in Section 6.1.1, we have already mentioned forms and hyperlinks. This is a natural terminology when discussing Web applications, including RESTful Web services. In the example service, the operation listMyBookings() is a simple link to the "my bookings" resource, which retrieves the list of the bookings of the current user who is identified by an authentication mechanism. The rate-search operation search(date,city) can naturally be represented with a form with two input fields, one for the date and one for the desired location. (Usually the user interface will be more complex, we simplify the example for brevity, without loss of generality.) Note that if the description of listMyBookings() contains said hyperlink, or if the description of search(date,city) contains said form, the developer reading the description can easily test the operations by simply clicking the link or filling-in and submitting the form.

An HTML form or a hyperlink can be taken as a service description (with a single operation). A link specifies that there is a resource at the given address, and that it can be expected to return useful data when the HTTP method GET is invoked on it. A form specifies the address, the HTTP method (HTML forms only support GET and POST, while XForms [137] support all the HTTP methods), and the input parameters in the fields of the form. Table 6.4 shows that in comparison to hRESTS, forms and hyperlinks only lack the notions of service, labels, and any description of the expected outputs.

In order to accommodate semantic annotations of all four kinds of semantics, we need to embed HTML forms and links in the complete hRESTS structure. Hyperlinks can already be used to provide the address of an operation, therefore hRESTS can be used as-is to describe hyperlinks that point to machine-oriented data resources as RESTful Web services. To be able to incorporate forms, the microformat needs one extension, defined in the paragraph below:

[9] They are machine-oriented services, but their providers do not designate them as Web services.

	service	operation	address	method	input	output	label
hRESTS	•	•	•	•	•	•	•
hyperlinks		•	•	•	N/A		
forms		•	•	•	•		

Table 6.4: WSMO-Lite annotations in WADL

The `input` class can also be used on forms (`<form>`). The form's `action` attribute specifies the address of the enclosing operation, and the `method` attribute specifies the operation's method. The various input fields of the form describe the input of the operation. The enclosing operation may further contain a block with the class `output` which describes the response expected when the form is submitted.

While we have defined here how hyperlinks and forms that point to machine-oriented resources can be annotated with hRESTS and MicroWSMO, this support is currently largely theoretical. Beyond the scope of this thesis, as part of future work, we may attempt to crawl the Web and identify those machine-oriented resources, and then we can evaluate whether efforts should be spent on annotating those links and forms, or whether it would be better to document the resources as Web APIs and apply hRESTS to the resulting documentation.

6.4 Data Lifting and Lowering

In Section 5.1.5, we have discussed data lifting and lowering in context of WSDL-described Web services. Since the prevalent protocol used by these services, SOAP, wraps an XML payload, the lifting and lowering transforms straightforwardly between semantic data (generally in RDF) and XML. RESTful services, on the other hand, present three complicating factors for lifting and lowering:

1. XML is not as prevalent in RESTful services as it is in WS–∗ services, therefore lifting and lowering needs to take into account diverse data formats, including form data serialization and JSON (cf. Section 3.2.2).

2. The input data of RESTful service operations is not necessarily transmitted in a single document; instead, there may be URI template parameters, query parameters, and for some methods also the request body (called *request entity* in HTTP).

3. The meaning of the response data (the *response entity*) of RESTful service operations depends on the response status code, since faults are not transferred as entity data.

To deal with the diversity of the data formats (the first problem), a lifting/lowering engine must support sufficiently expressive transformation languages. XSPARQL (cf. Section 3.5.4) can be used if the target format is textual or XML. Alternatively, JavaScript[10] can be used as well, if the engine provides a suitable API for reading and creating semantic data. JavaScript would be especially suitable for textual structured target formats such as JSON (cf. Section 3.2.2).

[10] `https://developer.mozilla.org/en/JavaScript`

```
1  <mw:requestData>
2    <mw:templateParam name="{parameter name}" value="{parameter value}" /> *
3    <mw:queryParam name="{parameter name}" value="{parameter value}" /> *
4    <mw:body mediaType="{media type}"> ?
5       {content as described in the text, dependent on the media type}
6    </mw:body>
7    <mw:formInput name="{parameter name}" value="{parameter value}" /> *
8  </mw:requestData>
```

Listing 6.9: MicroWSMO Request Data Format for Output of Lowering

In case different transformation languages are suitable but not generally supported, MicroWSMO (and SAWSDL) allows multiple lowering (resp. lifting) transformations to be specified on a single input (resp. output) message, to provide equivalent alternatives. A lifting/lowering engine can choose any of the provided transformations based on factors such as the language in which the transformation is written, or the run-time availability of the actual lifting/lowering file.

To deal with the second problem, that of fragmenting the input data into URI parameters and the body entity in the lowering transformation, we define a special XML format, shown in Listing 6.9, to represent the result of lowering. The root of the format is the element `<mw:requestData>`.[11]

In case the input of the operation is described as an HTML form, the element `<mw:requestData>` contains an ordered list of `<mw:formInput>` elements, each of which describes the value of a single input field in the form. No two `<mw:formInput>` elements can have the same value of the `name` attribute. The final serialization of this data then follows the rules of the form, as defined in HTML [45]. A form with the method GET will put all the input form field data in URI query parameters, whereas a form with the method POST will serialize the input data in the request entity, usually with the media type application/x-www-form-urlencoded.

For operations whose input is not described as a form, the `<mw:requestData>` element may contain any number of values for the parameters of the URI template address of the operation (using the element `<mw:templateParam>`); any number of values to be added to the URI as query parameters (using the element `<mw:queryParam>`), and optionally a single element `<mw:body>` that contains the request entity. All these child elements can be present in the `<mw:requestData>` root element in any order. No two `<mw:templateParam>` elements can have the same value of the `name` attribute. The `<mw:queryParam>` elements are processed in the order in which they appear, appending the name/value pairs to the query component of the operation's address URI.

To accommodate diverse data formats, the content of the `<mw:body>` element is dictated by the `mediaType` attribute: for textual media types (`text/*`), the `<mw:body>` must only contain text (which must be escaped to avoid conflicts with XML syntax, e.g. using a CDATA section). For XML media types (`application/xml` and `application/*-xml`, see [139]), the `<mw:body>` must only contain a single XML element child, which will be the root element of the resulting request entity. For any other media types, the content of `<mw:body>` must be the binary form of the request entity, encoded into text using Base 64 [8].

[11] The prefix `mw:` stands for the namespace `http://www.wsmo.org/ns/microwsmo#`.

```
1  <mw:responseData>
2    <mw:status code="{3−digit integer}"/>
3    <mw:body mediaType="{media type}"> ?
4       {content as described in the text, dependent on the media type}
5    </mw:body>
6  </mw:responseData>
```

Listing 6.10: MicroWSMO Response Data Format for Lifting

Finally, to deal with the third problem (dependence of the meaning of the results on the response status code), we define an XML format to represent the response data and to serve as the input to lifting. The format is shown in Listing 6.10.

The root of the format is `<mw:responseData>`. It has only two child elements: `<mw:status>` captures the status code of the HTTP response (a 3-digit number, e.g. 200 for success, 500 for a server-side error etc.), and `<mw:body>` contains the response entity. The format of the body element is the same as above in the request data.

In summary, the lifting and lowering transformations in MicroWSMO do not work on the raw message data; instead they use a simple representation of the HTTP messages so that the significant parts of the HTTP messages are exposed, while the lifting and lowering transformations are still shielded from the full complexity of HTTP messages.

6.5 Semantics Inherent in RESTful Web Services

HTTP, as an application protocol, defines a uniform interface with a prescribed meaning that all Web resources, including RESTful services, are expected to follow. The definitions of HTTP methods can be reflected in a functionality taxonomy, which we can use to automatically add functional descriptions of the operations of RESTful services. This section defines this HTTP method functionality taxonomy and its uses.

As described in Section 3.2.4, the HTTP specification [49] defines eight methods: GET, HEAD, POST, PUT, DELETE, OPTIONS, TRACE and CONNECT. Only the first six are generally useful in RESTful services; TRACE is mostly used for debugging purposes, and CONNECT is used for tunneling different protocols through HTTP.

Most HTTP methods have well-defined limited functionality: GET serves for data retrieval, HEAD retrieves only the HTTP metadata of a resource (used especially for cache validation), PUT replaces the contents of a resource, DELETE removes a resource, and OPTIONS checks the communication options of a resource (for instance, the allowed HTTP methods). All these five methods are defined as *idempotent* — the effect of invoking one of these methods multiple times in a sequence is the same as invoking the method only once. Additionally, GET, HEAD and OPTIONS are *safe* methods (as defined in Section 3.2.4), i.e., they are not supposed to have any application-significant side effects.

In contrast to the above methods, POST is a general-purpose method whose functionality is unconstrained. This makes it possible to implement any kind

```
1  @prefix hr: <http://www.wsmo.org/ns/hrests#> .
2  @prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
3  @prefix wsdlx: <http://www.w3.org/ns/wsdl-extensions#> .
4  @prefix wl: <http://www.wsmo.org/ns/wsmo-lite#> .
5
6  hr:HttpMethod a wl:FunctionalClassificationRoot .
7  hr:IdempotentMethod rdfs:subClassOf hr:HttpMethod .
8  wsdlx:SafeInteraction rdfs:subClassOf hr:IdempotentMethod .
9  hr:ReplaceResource rdfs:subClassOf hr:IdempotentMethod .
10 hr:DeleteResource rdfs:subClassOf hr:IdempotentMethod .
```

Listing 6.11: Functional Classification of HTTP Methods

of functionality in Web sites, as exemplified by the use of HTTP POST to transport SOAP messages with any purpose.

Listing 6.11 shows a functionality classification for the constrained HTTP methods. The class HttpMethod is the root of this classification with one direct subclass, IdempotentMethod which groups all the idempotent operations (currently all the five constrained HTTP methods). The class wsdlx:SafeInteraction, imported from WSDL 2.0, groups the three methods that are *safe*, which are information retrieval operations further differentiated by their inputs and outputs; and the classes ReplaceResource and DeleteResource then identify the remaining two methods PUT and DELETE.

This functional classification has two uses: first, we can automatically attach the classes as functional descriptions of service operations, contributing to the behavioral semantics of the service; and second, the classes can be used explicitly on service operations to indicate their functionality, for instance when the method POST on some particular resource is safe, idempotent, or when it replaces PUT or DELETE.[12]

Table 6.5 summarizes the model references that can be automatically attached to hRESTS operations depending on the HTTP method. These annotations make the HTTP method semantics available to WSMO-Lite semantic clients.

A MicroWSMO parser should automatically attach these model references. However, a small number of Web APIs use HTTP GET to perform side effects, such as deleting an item from a container or confirming mailing list unsubscription. While such misuse is recognized as erroneous, Web search engine crawlers and Web accelerator programs have been using server-provided restrictions (`robots.txt`) and heuristics (such as the presence of URI parameters after '`?`') to guess whether GET will be safe on a given URI, limiting their reach to avoid unintended consequences for broken Web applications. Such heuristics can also be built into a MicroWSMO parser so that the wsdlx:SafeInteraction model reference would only be attached to operations whose address is judged as safe. However, the investigation of such heuristics is out of scope of this thesis.

[12] HTML forms only support GET and POST, therefore some services use POST where PUT or DELETE could be more appropriate.

Method	Automatically attached model reference
GET	wsdlx:SafeInteraction
HEAD	wsdlx:SafeInteraction
OPTIONS	wsdlx:SafeInteraction
PUT	hr:ReplaceResource
DELETE	hr:DeleteResource

Table 6.5: Semantics Inherent in HTTP Methods

6.6 Deployment of Semantic Descriptions

The two microformats defined in this chapter, hRESTS and MicroWSMO, enable semantic annotations in HTML documentation of RESTful Web services. These annotations, however, merely identify semantic concepts by URIs, identically to SAWSDL annotations in WSDL, as discussed in Section 5.3. The semantic client must be able to access the semantic definitions of these concepts (usually in the form of ontologies), and there are two main options for where the ontologies can be deployed:

1. The ontologies are available publicly on the Web, alongside the HTML documentation. See Section 5.3.2 for a brief discussion of this deployment strategy. Since the service documentation is already presumably on the Web, it is natural that the semantic concept definitions should be available likewise.

2. Some concept definitions can be embedded in the HTML documentation of the service, for example using RDFa (see Sections 3.4.2 and 6.2.4). This is particularly suitable for small pieces of semantic definitions limited in applicability to a particular service — for instance the service preconditions and effects. Using GRDDL to extract the MicroWSMO and hRESTS data, the added concept definitions will automatically become part of the resulting service description RDF graph.[13]

Naturally, the options can be combined: for instance, a reusable service functionality taxonomy can be available publicly on its Web site, whereas the service-specific definitions, such as the preconditions, effects, and nonfunctional properties, could be embedded as RDFa in the HTML documentation.

6.7 Validation of MicroWSMO/hRESTS Files

As discussed in Section 5.4, it is useful to define validation rules for formal languages of any kind. Section 5.4.1 defines four facets of validity:

- syntax,
- consistency,
- completeness,
- correctness.

[13] A GRDDL transformation for RDFa is available via `http://ns.inria.org/grddl/rdfa/`

In contrast to WSMO-Lite, where syntactical validity falls on the level of XML, XML Schema and validation rules defined by the WSDL standard, hRESTS and MicroWSMO syntactical validity has two levels:

- HTML and XHTML are defined using DTDs and XML Schemas. However, the Web has been known to work well with formally invalid HTML documents, and HTML validation itself deals with many alternatives and special cases. MicroWSMO and hRESTS are intended to be applied in valid XHTML documents, but it may be possible that hRESTS and MicroWSMO tools could also process HTML-invalid input, including for example documents that purport to be XHTML but are not well-formed XML.
- Sections 6.2.2 and 6.2.3, which define the hRESTS and MicroWSMO microformats, contain structural constraints on the hierarchy of elements marked up with the microformats' classes. An hRESTS and MicroWSMO validator should first validate the underlying HTML, and then proceed to validate the microformat structural constraints.

The validation of consistency, completeness and correctness of hRESTS and MicroWSMO descriptions, which is on the level of the service semantics, is then analogous to the validation of WSMO-Lite descriptions, as defined in Sections 5.4.2 and 5.4.3.

Part III

Evaluation and Conclusions

Chapter 7

Algorithms for Service Discovery and Composition

The preceding chapters show how Web service descriptions can be annotated with semantics, and the ontology in which the semantics are expressed. Semantic descriptions are intended to support tasks such as service discovery and composition, therefore in this chapter we define several algorithms for these tasks, in order to evaluate that the proposed languages can actually support Web service automation (i.e., that they are *fit for purpose*). Mainly adapted from existing literature, these algorithms are not necessarily meant to be the most powerful or the most efficient ones, instead they are meant to demonstrate the versatility of our lightweight semantic descriptions.

Through this evaluation step, we check our main success criteria: i) that our service semantics ontology is sufficiently expressive to support the desired degree of automation (comprising service discovery, selection and composition), and ii) that the automation works equally well with RESTful services as it does with WS–∗ services.

In Section 7.1, we define all the steps that we include in Web service discovery and composition; for each of these steps, the following sections then provide automation algorithms: Section 7.2 deals with functional service matchmaking, Section 7.3 discusses service negotiation and offer discovery, Section 7.4 looks into ranking Web services on their nonfunctional properties, and Section 7.5 discusses when and how to involve users in the final service selection. When no single service can fulfill the user's request, discovery may involve service composition, which is the topic of Section 7.6.

In the Evaluation Chapter 9, Section 9.2 provides further discussion of how the algorithms presented below serve to evaluate the contributions of this thesis.

7.1 Discovery Process in General

The Web contains many Web services: in September 2012, the Web service search engine `seekda.com` knew about over 28000 WS–∗ services, and the API and Mashup list `programmableweb.com` contained more than 7000 RESTful services. Although the numbers are nowhere near the millions or even billions promised by early Web service evangelists, it is nevertheless clear that effective

finding and reuse of services must be supported by powerful search capabilities. In addition to classical full-text search algorithms that can easily be applied to service descriptions and documentation, semantic annotations enable greater detail and more expressivity in expressing user goals (queries), and improvements in precision of the matching of goals to known services.

Semantic service search and matchmaking is commonly called *discovery*. As described in Section 2.1.1, the aim of discovery in general is to find those services that fit the current needs of the user, and potentially to find appropriate concrete service offers. To help select one service (and one offer), a *ranking* mechanism can be employed to sort the services and the offers based on the user's preferences.

Before we proceed to concrete algorithms in Section 7.2 and beyond, we formally define in the remainder of this section the several steps involved in the discovery process. First, in Section 7.1.1 we formalize certain auxiliary terms, and then in Section 7.1.2, we proceed to the actual definitions for discovery and composition.

7.1.1 Auxiliary Definitions

In order to specify the inputs and outputs of the automation algorithms, we define here the terms *goal*, *service registry*, *concrete offer*, and *composite service*. The terms *service* and *service description* are defined earlier in this thesis.

Definition 7.1 (goal) Goal is a data structure that captures the current relevant needs of the user, i.e., what the user wants the SEE to achieve.

In the form of informal commands, the following would be example goals: i) book a skiing trip within three months somewhere in the Tyrolean Alps; ii) find the current price of the shares of IBM; iii) find possible suppliers of laptops for our institute.

Different automation tasks have different requirements on what should be described in a goal. In this thesis, we do not define a common structure for goals; instead, along with each sample automation algorithm, we define the data components that need to be present in a goal as inputs of the algorithm.

Definition 7.2 (service registry) The SEE must be aware of some existing Web services. The knowledge base that contains the (semantic) descriptions of the known Web services is called the service registry.

An example service registry is iServe (see Section 8.3). A service registry does not magically know about every existing Web service, instead it must be populated with service descriptions: a service provider may explicitly submit the semantic descriptions of its services into the registry, or the provider may simply publish the service descriptions on the Web, where they can be found by a Web crawler. The Web service search engine `seekda.com` demonstrates that finding Web services by crawling the Web is viable.

Definition 7.3 (concrete offer) As a service description need not contain all the specific information necessary for a client to be able to evaluate whether the service can indeed satisfy the detailed goal, *concrete offers* may be established by querying the service; an offer describes exactly how the service can satisfy the goal.

As an illustration, let us use the hotel reservation service: the semantic description may define the service as a *hotel reservation service* with its scope limited to the city of Rome, which is a useful granularity for Web service discovery. The service description itself does not guarantee that there will be any hotels available at any concrete requested dates, let alone hotels that fit the user's other constraints such as price and location. Therefore, a concrete offer will then specify a particular hotel and a particular room rate available at that hotel for the dates required by the user. Generally, there could be multiple offers from a single Web service, such as, in our case, different hotels and even different room categories at one hotel, all bookable through the same service.

A semantic description such as above would be called *complete* but not *correct* by Preist [93]. He postulates that a contract agreement phase is necessary for any service whose description is not correct, among other criteria. In this thesis, we do not investigate the full range of contract agreement and negotiation techniques, instead we focus on one particular sub-task of contract agreement, which is offer discovery. Therefore, we deal with *concrete offers* instead of *contracts* or *agreed services* discussed by Preist.

In case of what Preist calls *basic services*, i.e. those that do not require a contract agreement phase, the service description itself is a concrete offer. In the definition of steps such as ranking (below), we can therefore use the term *concrete offer* even when no offer discovery takes place.

Definition 7.4 (composite service) When no single known service can satisfy the goal, it may be possible to compose together multiple services that, taken together, do satisfy it. In composition, several services effectively act as one, which we call a *composite service*.

Typical examples of services that can be composed together would be airplane reservation and hotel reservation for arranging a trip. Preist [93] mentions *service bundles* which are roughly equivalent to what we call *composite services*; in our work, we assume that a composite service can be treated as a service by almost all service automation tasks, as discussed below.

7.1.2 Steps of the Discovery Process

Now we can proceed to the definitions of the various tasks related to discovery in general, illustrated in Figure 7.1.

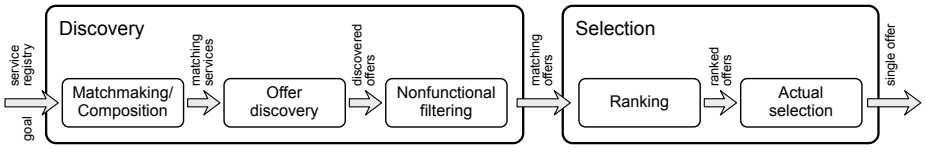


Figure 7.1: Decomposition of Discovery and Selection

On a high level, we can distinguish discovery from selection: discovery finds relevant services that fit the goal, and selection choose which service(s) to use in a concrete execution.

Discovery itself can be split into i) matchmaking and/or composition over a goal and a service registry, ii) offer discovery through interactions with the

services found by matchmaking or constructed by composition, and iii) nonfunctional filtering, which eliminates services and offers that fall outside the client's constraints. The following are definitions of these tasks:

Definition 7.5 (discovery) Overall, discovery is the process of finding pertinent known services and their concrete offers for achieving the user's goal.
Inputs: goal, service registry
Outputs: a set of concrete offers that match the goal

Definition 7.6 (matchmaking) Based on functional semantics[1] and on the information model, matchmaking selects those services from the service registry, that may potentially be able to fulfill the goal. In other words, matchmaking discards services that cannot fulfill the goal (according to their description).
Inputs: goal (functional description of expected functionality), service registry
Outputs: set of services (service descriptions)

Definition 7.7 (composition) Based on functional semantics and on the information model, composition combines services from the service registry into composite services that may potentially be able to fulfill the goal.
Inputs: goal (functional description of expected functionality), service registry
Outputs: set of composite services

Definition 7.8 (offer discovery) As an optional step, offer discovery is a specific kind of contracting: the SEE interacts with the matching services to gather information about concrete offers pertinent to the goal. In case when offer discovery is not (or cannot be) performed, a straightforward mapping turns each matching service into a single offer.
Inputs: goal (concrete instance data), set of matching services
Outputs: set of concrete offers

We should note that what we call *offer discovery* is elsewhere in literature (e.g. [28, 54]) also called *service discovery*, making the distinction between a *Web service* and the *service* it actually provides[2]. We prefer the term *offer* to avoid causing confusion due to overloading of the common word "service". In addition, the term *offer* is easily understood to encompass both offered services and offered products, the basis of e-commerce.

Definition 7.9 (nonfunctional filtering) The filtering step compares the nonfunctional properties of the services and their offers against the user's constraints, expressed in the goal.
Inputs: goal (nonfunctional constraints), set of discovered offers
Outputs: set of matching offers

Examples of nonfunctional constraints include the maximum allowed price, the minimum required trust evaluation, etc.

[1] In this thesis, we do not consider algorithms for process matchmaking and process-aware composition, which deal with behavioral semantics.

[2] The distinct terms *service* and *Web service* are used in WSMO [28], and correspond to Preist's [93] *abstract service* and *concrete service*.

Finally, selection is supported by a ranking mechanism that sorts the discovered matching offers; the actual selection happens either by presenting the user with the ranked list of offers, or trivially by automatically choosing the highest-ranked offer. The following are the definitions of these tasks:

Definition 7.10 (selection) Selection is the process of choosing a single concrete offer that will fulfill the user's goal.
Inputs: goal, set of offers
Outputs: a single selected offer

Definition 7.11 (ranking) Ranking is the part of the selection process that can be automated; it adds ordering to the list of offers.
Inputs: goal (user preferences), set of offers
Outputs: sorted list of offers

The decomposition of discovery and selection into the above subtasks is an abstract separation of concerns; it does not imply a strict separation of the tasks in an implementation. For instance, some nonfunctional filtering can be done before and during offer discovery, and ranking can already sort known offers while offer discovery is running, presenting the user with preliminary results to increase the interactivity of the system.

The remaining sections of this chapter describe the steps outlined above in more detail, presenting concrete algorithms that may be used for realizing semantic automation of the various discovery and selection tasks. Most of the content consists of adaptations of existing algorithms to our lightweight semantic annotations, only offer discovery (Section 7.3) is novel and part of the contribution of this thesis.

7.2 Functional Web Service Matchmaking

As discussed in Chapter 2, Klusch [61] presents the most recent survey of semantic service discovery approaches. Among logic-based approaches, Klusch investigates what kinds and parts of service semantics are considered for matching in the various approaches, especially pointing out how various approaches use different combinations of the descriptions of service inputs, outputs, preconditions and effects (together known as IOPE). Among the matchmakers that use all the four semantic aspects, Klusch cites the work of Keller et al. [54][3], which we adopt here for the purpose of demonstrating matchmaking with WSMO-Lite.

Note that in the terms of [28, 54], Keller et al. address *Web service discovery* as opposed to *service discovery* (as discussed above in Section 7.1.2), while we use the term *functional matchmaking* here.

The approach of Keller et al. is based on a simple concept of modeling Web services and goals as sets of relevant objects: a service can deliver certain objects, and a goal requests them. We use Keller et al.'s naming of these sets: a Web service $\mathcal{W}$ is represented through a set named $R_{\mathcal{W}}$, and a goal $\mathcal{G}$ is represented through a set named $R_{\mathcal{G}}$.

[3]We use a newer publication by Keller et al. in the same line of work as that cited by Klusch.

Keller et al. distinguish two *modeling intentions*: existential and universal. In *existential* modeling, the sets overdescribe the services and goals — a goal $\mathcal{G}$ will be satisfied if any object(s) from $R_{\mathcal{G}}$ is delivered; and a service $\mathcal{W}$ can only really deliver some objects from $R_{\mathcal{W}}$. For example, a goal set can describe all room reservations in a given city at given dates, but the intention is to get only one reservation; and a service set can describe all the possible room reservations at a given hotel, but only some rooms will be available at some dates. In *universal* modeling, the sets describe the service or the goal exactly: the goal requires all the objects from $R_{\mathcal{G}}$ to be delivered, and the service can deliver all the objects from $R_{\mathcal{W}}$. While universal modeling is potentially more accurate, Keller et al. list no plausible use cases where the required level of effort would be desirable or practical. Therefore, we restrict our discussion only to the existential modeling intention.

In order for $\mathcal{W}$ and $\mathcal{G}$ to be considered a match, the sets $R_{\mathcal{W}}$ and $R_{\mathcal{G}}$ have to be interrelated. There are four possible set-theoretic relations, with an inherent match-degree ranking among them (the list goes from the best match to the worst):

- *Exact match of equal sets* ($R_{\mathcal{G}} = R_{\mathcal{W}}$): the service may be able to deliver all the objects requested by the goal, and it cannot deliver any other, irrelevant objects. This is the closest match between a goal and a service. For example, a goal requests accommodation in Rome, and a service offers exactly accommodation in Rome.

- *Web service subset of goal* ($R_{\mathcal{G}} \supseteq R_{\mathcal{W}}$): the service can only deliver some of the objects requested by the goal; it cannot deliver irrelevant objects. For example, if the goal requests accommodation in Rome, a particular service may only cover budget hotels in this city. The service description indicates that the service has limitations with respect to the goal (only budget hotels), but it can be acceptable to the client (whose goal does not specify the client's demands on accommodation quality).

- *Goal subset of Web service* ($R_{\mathcal{G}} \subseteq R_{\mathcal{W}}$): the service may be able to deliver all objects requested by the goal, but it may also (or even only) deliver irrelevant objects. For example, a service offering accommodation in Italy may have a limited coverage of hotels in Rome. In this case we merely have a possible match.

- *Non-empty intersection between service and goal* ($R_{\mathcal{G}} \cap R_{\mathcal{W}} \neq \emptyset$): the service may be able to deliver some of the requested objects, but it may also deliver irrelevant objects. For example, a Marriott hotel reservation service is clearly limited with respect to the hotel (it only books Marriott hotels), but it still is a possible match because the description does not say whether or not there are any Hilton hotels in Rome.

The four match degrees may be used to rank the discovered services. For example, performing negotiation with better-matching services first can quicker lead to useful results, which the semantic client system can asynchronously display to the user.

WSMO-Lite provides two distinct mechanisms for describing the functionality of Web services: the lightweight but coarse-grained *functional classification*

that deals with taxonomies of functionality, and the more expressive *capability* described with preconditions and effects. Both types can be interpreted as descriptions of sets of objects. Keller et al. [54] only deal with capabilities; we extend their approach to discovery using functional classifications.

In principle, matching a goal against a set of known services means evaluating the match between the goal and each individual service separately. In practice, matchmaking algorithms may be amenable to optimizations such as the use of search indices which work on the whole corpus of known service descriptions. For the sake of clarity, we present the matching algorithms without any such optimizations.

In the subsections below, we detail the concrete matchmaking algorithms that follow the set-based approach of Keller et al.: in Section 7.2.1, we discuss the extension of the work of Keller et al. for matchmaking with functional classifications, in Section 7.2.2, we summarize matchmaking with capabilities from [54], and then in Section 7.2.3 we combine the two separate approaches.

7.2.1 Matchmaking with Functional Classifications

Functional classification is the simpler one of the two mechanisms WSMO-Lite provides for functional description of Web services. Using SAWSDL model references, a Web service s can be associated with one or multiple categories $(c_1^s, \ldots, c_n^s)$ from one or multiple classification ontologies. For the purpose of the discovery algorithms here, we interpret those categories as specifying the sets of objects that can be delivered by Web services, as discussed above.

Multiple categories are treated in conjunction — a service belongs to all the functionality categories with which it is associated. In other words, the service is described by the intersection of the functionality categories. Effectively, we can say the service is associated with a single functional category $R_{\mathcal{W}}$:

$$R_{\mathcal{W}} = \bigcap_{i=1\ldots n} c_i^s$$

For selecting Web services, a user goal must specify a category of interest, $R_{\mathcal{G}}$, so that the matchmaking algorithm can return services associated with matching categories that are related with the goal category through the subclass relationships that make up the functionality classifications. Below, we first discuss how goals may describe the category $R_{\mathcal{G}}$, and then we proceed to the discussion of how goal categories are matched with service categories, and how the subclass relationships between functionality categories are taken into account in a concrete matchmaking algorithm.

Describing the Goal Category

There are numerous levels of detail on which a goal can specify the desired category, starting with a single concrete category, and extending towards increasingly complex combinations of categories:

1. **A single concrete category:** the goal specifies an existing functionality category with its identifier. This kind of goal is useful if there is a category that well covers the intent of the user.

2. **An intersection of multiple categories:** the goal specifies a set of existing categories $c_1^g, \ldots, c_m^g$ expecting that the matching services are associated with all the given categories. The desired category $R_\mathcal{G}$ is the intersection of the specified categories:

$$R_\mathcal{G} = \bigcap_{i=1\ldots m} c_i^g$$

Specifying the goal category as an intersection is useful for instance when the user requests services with multiple functionalities, such as a combined *flight ticket booking* and *hotel reservation* service.

3. **A union of multiple categories:** the goal specifies a set of existing categories $c_1^g, \ldots, c_m^g$ expecting that the matching services are associated with any one or more of the given categories. The desired category $R_\mathcal{G}$ is the union of the specified categories:

$$R_\mathcal{G} = \bigcup_{i=1\ldots m} c_i^g$$

Specifying the goal category as a union is useful for instance when there are multiple functionality classifications that cover the same functionalities, and services are likely to be described using one or the other classification. For instance, a goal might ask for all services associated either with the eCl@ss 7.0 category 25-12-13-90 "Hotel, guesthouse, pension (travel management, unclassified)"[4] or with the Wikipedia category Hotels[5], if eCl@ss and Wikipedia categories are used as WSMO-Lite functionality classifications. In this case, the goal uses a union of categories to locally (in the scope of this one goal) mediate between two classifications that differently capture the category of *accommodation booking services*.

4. **More complex expressions:** the goal specifies an expression that defines $R_\mathcal{G}$ as an arbitrary combination of concrete categories. The expression can be in any suitable language that can describe set operators or class membership rules. For instance, a goal might express a combination of the two examples above: the desired category covers services that do *accommodation booking* (category a, expressed concretely as the aforementioned eCl@ss category, here a_e, or the Wikipedia category a_w) and *flight ticket booking* (f, also in either of the two classifications, f_e and f_w):

$$R_\mathcal{G} = (a_e \cup a_w) \cap (f_e \cup f_w)$$

The expressivity of the goal language may grow by including further features such as negation (the desired services must not be in a certain category) etc.

For the purpose of illustrating matchmaking with WSMO-Lite, we focus on the second mentioned approach, specifying the goal category as the intersection

[4] http://www.eclass-online.com/system/suche/index.html?eversion=7.0&lang=en&ssuche_x=x&su=25121390

[5] http://en.wikipedia.org/wiki/Category:Hotels

of multiple concrete categories, which is symmetrical to how we describe services, and sufficient for the use cases presented throughout this thesis. Goals with category union would be able to incorporate local mediation between functionality classifications; however, mediation can also be done through mapping ontologies used by the matchmaking reasoner.

Matching Service and Goal Categories

WSMO-Lite recognizes only one type of relationship between functionality categories: the *subcategory* (subset) relationship. Categories and their subcategory relationships form directed acyclic graphs called category hierarchies. To determine the match degrees between goals and Web services, we use the category hierarchies:

- $R_{\mathcal{G}} = R_{\mathcal{W}}$: the service exactly matches the goal if it is both a subset and a super-set of the goal,[6] as defined below.
- $R_{\mathcal{G}} \supseteq R_{\mathcal{W}}$: the service is a subset of the goal if the service is associated with a subcategory of each of the goal categories.
- $R_{\mathcal{G}} \subseteq R_{\mathcal{W}}$: the goal is a subset of the service if the goal is associated with a subcategory of each of the service categories.[7]
- $R_{\mathcal{G}} \cap R_{\mathcal{W}} \neq \emptyset$: the service intersects the goal if any of the above is true, but also if we can find any one category that is a subcategory of all of the categories associated with both the goal and the service.[8]

Figure 7.2 shows the matchmaking algorithm that embodies our adaptation of the work of Keller et al. to WSMO-Lite functional classifications; it formalizes the conditions for each of the match degrees.

The algorithm employs subsumption reasoning, which can in some languages be reduced to the well-known and tractable problem of graph reachability (for instance, in WSML-Quark [122]). In other words, discovery with functional classifications can provide good performance at the cost of expressivity — the level of detail practically possible in service and goal descriptions.

7.2.2 Matchmaking with Preconditions and Effects

In addition to the coarse-grained functionality classifications discussed above, WSMO-Lite supports fine-grained service description with logical expressions that capture the precondition and effect (together, the *capability*) of the service. Preconditions and effects can also be used to model services as sets of objects, as shown in [55] (referenced from [54] as a concrete realization of set-based

[6] Note that this is a one-way implication, not an exclusive *iff*: for example when using two overlapping classifications for which we do not have a formal mapping, it is possible that the service category is an exact match of the goal category but they use different terms so the matchmaker has no means of verifying the match.

[7] Note that some taxonomies, such as the eCl@ss classification, only use leaf categories to classify objects. Such taxonomies eliminate the possibility of a service being a proper superset of any goal.

[8] We assume here that any explicitly defined functionality category can be seen as a **non-empty** set of objects that some service can deliver.

Algorithm: matchmaker for WSMO-Lite functional classifications
Inputs: set S of known services annotated with functionality categories,
goal categories $c^g_1, \ldots, c^g_m$
set C of all known WSMO-Lite functionality categories
Result: the set of tuples $\langle$*service, match degree from* $\{=, \supseteq, \subseteq, \cap\}\rangle$

```
1  M := ∅
2  for each s ∈ S        (associated with categories c^s_1, ..., c^s_{n_s})
3  |  δ := "–"          (match degree, initially "–" for "no match")
4  |  if ∀i ∈ {1...n_s}  ∃j ∈ {1...m} : c^g_j ⊆ c^s_i
5  |  |  δ := "⊆"
6  |  if ∀j ∈ {1...m}  ∃i ∈ {1...n_s} : c^g_j ⊇ c^s_i
7  |  |  if δ = "⊆"
8  |  |  |  δ := "="
9  |  |  else
10 |  |  |  δ := "⊇"
11 |  if ∃c ∈ C : (∀i ∈ {1...n_s} : c ⊆ c^s_i)  ∧  (∀j ∈ {1...m} : c ⊆ c^g_j)
12 |  |  δ := "∩"
13 |  if δ ≠ "–"
14 |  |  M := M ∪ {⟨s, δ⟩}
15 return M
```

Figure 7.2: Matchmaker algorithm for WSMO-Lite functional classifications

discovery in formal logics). Here, we provide an adaptation of their approach to the terminology of WSMO-Lite.

In [55], Keller et al. propose two ways of expressing the sets of objects that represent Web services and user goals:

- *Simple semantic descriptions:* the sets $R_\mathcal{W}$ and $R_\mathcal{G}$ are defined using first-order formulas $\phi(x)$ and $\psi(x)$[9] with one free variable each:

$$\begin{aligned} \mathcal{W}: &\quad R_\mathcal{W} = \{x \mid \phi(x)\} \\ \mathcal{G}: &\quad R_\mathcal{G} = \{x \mid \psi(x)\} \end{aligned}$$

- *Rich semantic descriptions:* the above is extended with a notion of *input data* that influences the set of objects delivered by the service. The service description expresses a precondition $\phi^{pre}(i_1 \ldots i_n)$ that defines valid inputs, and an effect $\phi^{eff}(x, i_1 \ldots i_n)$ that describes how the set of objects delivered by the service follows from the inputs. The precondition and the effect are combined together in a single expression $\phi(x, i_1 \ldots i_n)$. The service is not considered to deliver anything meaningful if the precondition cannot be fulfilled by a given set of concrete inputs. In the description, we denote $R_{\mathcal{W}_\mathcal{G}}$ the set of objects returned by the service for a given goal with concrete input data, which is captured in the goal description as a set of data instances $D_\mathcal{G}$:

$$\begin{aligned} \mathcal{W}: &\quad R_{\mathcal{W}_\mathcal{G}} = \{x \mid \exists\, i_1 \ldots i_n \in D_\mathcal{G} : \phi(x, i_1 \ldots i_n)\} \\ &\quad \phi(x, i_1 \ldots i_n) \leftrightarrow \phi^{pre}(i_1 \ldots i_n) \wedge \phi^{eff}(x, i_1 \ldots i_n) \\ \mathcal{G}: &\quad R_\mathcal{G} = \{x \mid \psi(x)\} \\ &\quad D_\mathcal{G} = \{i_1, \ldots, i_m\} \end{aligned}$$

[9] Note that in contrast to [55], we swap the use of the symbols ϕ and ψ, for consistency with the preceding chapters.

Simple semantic descriptions:

Set relationship	Proof obligation
$R_{\mathcal{G}} = R_{\mathcal{W}}$	$\mathcal{W}, \mathcal{G}, \mathcal{O} \models \forall x : (\psi(x) \leftrightarrow \phi(x))$
$R_{\mathcal{G}} \subseteq R_{\mathcal{W}}$	$\mathcal{W}, \mathcal{G}, \mathcal{O} \models \forall x : (\psi(x) \rightarrow \phi(x))$
$R_{\mathcal{G}} \supseteq R_{\mathcal{W}}$	$\mathcal{W}, \mathcal{G}, \mathcal{O} \models \forall x : (\psi(x) \leftarrow \phi(x))$
$R_{\mathcal{G}} \cap R_{\mathcal{W}} \neq \emptyset$	$\mathcal{W}, \mathcal{G}, \mathcal{O} \models \exists x : (\psi(x) \ \wedge \ \phi(x))$

Rich semantic descriptions:

Set relationship	Proof obligation
$R_{\mathcal{G}} = R_{\mathcal{W}_{\mathcal{G}}}$	$\mathcal{W}, \mathcal{G}, \mathcal{O} \models \exists\, i_1, \ldots, i_n : (\forall x : (\psi(x) \leftrightarrow \phi(x, i_1 \ldots i_n)))$
$R_{\mathcal{G}} \subseteq R_{\mathcal{W}_{\mathcal{G}}}$	$\mathcal{W}, \mathcal{G}, \mathcal{O} \models \exists\, i_1, \ldots, i_n : (\forall x : (\psi(x) \rightarrow \phi(x, i_1 \ldots i_n)))$
$R_{\mathcal{G}} \supseteq R_{\mathcal{W}_{\mathcal{G}}}$	$\mathcal{W}, \mathcal{G}, \mathcal{O} \models \exists\, i_1, \ldots, i_n : (\forall x : (\psi(x) \leftarrow \phi(x, i_1 \ldots i_n)))$
$R_{\mathcal{G}} \cap R_{\mathcal{W}_{\mathcal{G}}} \neq \emptyset$	$\mathcal{W}, \mathcal{G}, \mathcal{O} \models \exists\, i_1, \ldots, i_n : (\exists x : (\psi(x) \ \wedge \ \phi(x, i_1 \ldots i_n)))$

Table 7.1: Proof obligations for the four types of set-theoretic relationship between the goal and the service object sets (from [55])

In WSMO-Lite service descriptions, the formula $\phi(x)$ or $\phi^{\mathit{eff}}(x, i_1 \ldots i_n)$ is captured in a suitable logical language such as WSML [136] as the *effect* of a Web service capability K (kappa, see Section 4.3.2), and the formula $\phi^{\mathit{pre}}(i_1 \ldots i_n)$ is similarly captured as the capability's *precondition*.

In order to evaluate the various types of match, an automated reasoner can be employed to prove logical relationships between the formulas. In Table 7.1, we summarize the proof obligations defined in [55]. All of the proof obligations are of the form

$$\mathcal{W}, \mathcal{G}, \mathcal{O} \models \mathit{expression}$$

where $\mathcal{W}$ is the definition of the Web service, $\mathcal{G}$ is the definition of the goal, $\mathcal{O}$ is a set of ontologies to which both descriptions refer, and *expression* is the actual formula that combines $\psi(x)$ with $\phi(x)$ or $\phi(x, i_1 \ldots i_n)$. Note that for the rich semantic descriptions, we compare the set of desired objects $R_{\mathcal{G}}$ with the runtime-computed set of objects $R_{\mathcal{W}_{\mathcal{G}}}$ that the service provides for the given goal data.

A discovery algorithm straightforwardly follows from the table: the algorithm would simply use a reasoner to evaluate the proof obligations for the given goal and for each known service, returning the matching services along with their match degrees.

Modeling with rich semantic descriptions is backward-compatible with the simple semantic descriptions: if there is no precondition, and the effect is independent of the input data $i_1 \ldots i_n$, the proof obligations in the lower half of Table 7.1 reduce to those from the upper half of the table. This compatibility makes it possible for simple descriptions to be used together with rich descriptions in a single system. It also enables gradual adoption of semantic complexity — a system can first only support the simple descriptions, and later adopt rich descriptions when the involved service providers and users become familiar with the logics languages and with the ontologies involved in the system; the older simple descriptions will still be evaluated correctly.

7.2.3 Combining Functional Classifications and Capabilities

So far, we have adapted the work of Keller et al. to WSMO-Lite preconditions and effects (Section 7.2.2), and we have extended it to deal with WSMO-Lite functionality classifications (Section 7.2.1). Here, we discuss how these two mechanisms can usefully be combined.

To combine functional classification discovery with capability discovery, functional classification descriptions can be straightforwardly translated to capability descriptions as follows:

$$\mathcal{W}: \qquad R_{\mathcal{W}} = \bigcap_{i=1\ldots n} c_i^s \quad \Longrightarrow \quad \phi(x) \;=\; (\forall i = 1 \ldots n : x \in c_i^s)$$

Equally, goal descriptions can also be translated in this manner. The translation would enable capability discovery to use functional classification annotations. However, due to the difference in nature of the taxonomies used for functional classifications and the ontologies used for fine-grained logical expressions, we do not expect that creating mappings between them would be economical; without mappings between the two worlds, the above translation could not result in new matches (better matchmaking recall). However, unifying functional classifications and capabilities is not the only way of combining the two matchmaking approaches.

Alternatively, the two approaches can be combined in an efficient two-stage discovery process, where functional category matching precedes precondition and effect evaluation. Functionality classifications can be expected to be coarse-grained, with each category expressing the consensus of a community. On the other hand, logical preconditions and effects provide the expressivity to describe services and goals in great detail, however, at the cost of decreased performance, due to the computational complexity of logical reasoning and proof. In combination, discovery over a large Web service registry can perform an efficient first step using functional classifications, and then evaluate the preconditions and effects only on the services with matching categories. Thus, WSMO-Lite offers improved scalability to large Web service registry sizes over existing approaches such as OWL-S and WSMO, where discovery traditionally always performs computationally intensive reasoning.

When used alone, capability descriptions need to be detailed and yet comprehensive to describe the service's functionality unambiguously. In combination with functionality classifications, a functional category specifies the broad functionality of the given service, therefore the preconditions and effects in the capability description must only express desired additional details. Effectively, the logical expressions that make up the capability descriptions may be simpler, easing the task of authoring the semantic service descriptions.

The two-stage combination of the two discovery approaches (based on functionality classifications and on formal capabilities) requires also that the user goal contain both kinds of data: a requested functionality category and a description of the desired effect (plus input data if rich capability descriptions are used). Also here, the logical expression that defines the desired effect is simplified because it operates in the context of the requested functionality category specified by the goal.

In summary, combining matchmaking through functionality classifications and through logical capabilities has benefits both in performance and in complexity of service descriptions.

Notably, Stollberg [116] achieves similar effects on discovery scalability and description complexity by enhancing WSMO discovery [54] with the use of goal templates and Web service templates, going through all three steps of the *heuristic classification* problem-solving method, as discussed in [28]. The templates are presumably defined through a similar process as our functionality classifications — by reaching consensus on a useful subdivision of a domain. Goal templates can be matched against Web service templates ahead of the time of concrete discovery, so that a concrete goal need only be matched against the concrete Web services that are described using Web service templates that match the template of the goal.

As Stollberg's work focuses on the performance benefits from pre-matching of templates, it does not pursue the difference between authoring logical capability descriptions and creating hierarchies of functionality templates. WSMO-Lite has functionality classifications separated from logical capabilities, stressing the differences in granularity and degree of collaboration expected to author the two different kinds of semantic descriptions.

7.3 Service Contracting, Offer Discovery

As described at the beginning of this chapter, offer discovery follows the task of service discovery, and its results go into filtering, ranking and selection. Service discovery returns a set of services that can potentially fulfill a user's goal. Offer discovery interacts with these services (or the service providers, if taken more generally) in order to find out any concrete offers that are relevant to the goal; the result of offer discovery is the set of available offers. This set is then subject to filtering and ranking according to the user's constraints and preferences, in order to select the best offer. In the end, the selected offer may be *consumed*, i.e., the client invokes the service that provided this offer and, after successful invocation, it will get the offered product or functionality.

Traditionally, offer discovery is done by code that is specific for a given service. For illustration, e-commerce data aggregation applications such as expedia.com need to have special code for any partner interface with which they interact. Having distinct code for multiple partnering services has negative impact on the costs of maintenance and evolution of the system. The main benefit of using semantics for offer discovery is that a semantic offer discovery mechanism need not really understand the full semantics of any particular web service interfaces (apart from the inputs and outputs).

In this section, we present an opportunistic approach to offer discovery that is based on data annotations and *safe* interactions. The material presented here is a significantly extended version of [63].

Note that offer discovery is an optional step, only applicable if the client's goal is sufficiently specific. In our hotel room reservation example, if the goal defines the concrete place and the dates, offer discovery can find concrete rates available in concrete hotels bookable through the discovered services. However, we can alternatively imagine the client looking for one hotel-booking service to include in a company travel reservation system (presumably so that they can

negotiate a beneficial exclusive contract), in which case offer discovery does not apply. In the absence of offer discovery, the following semantic automation steps (nonfunctional filtering and ranking, selection) can treat each discovered service as a single offer, so that we do not have to define two variants of each algorithm (such as *service ranking* and *offer ranking*).

This section continues in seven subsections. Section 7.3.1 formally defines what we mean by *offer* and *offer discovery*. Section 7.3.2 discusses the WSMO-Lite semantics we use for implementing offer discovery. In Section 7.3.3, we specify what data a user goal needs to contain to support offer discovery. In Section 7.3.4, we present a more fine-grained example hotel reservation service and a goal, which we then use in Section 7.3.5 to detail and explain our offer discovery algorithm. In Section 7.3.6, we discuss the particular AI planning approaches used in our offer discovery algorithm. Finally, in Section 7.3.7, we offer several concluding remarks on this offer discovery approach.

7.3.1 Analysis and Definition of *Offer* and *Offer Discovery*

In order to be able to talk about offer discovery, we need to specify what we mean by the term *offer*. From the point of view of contracting, an offer is a contract proposed by the service provider to the client, who can evaluate and accept or reject it.[10] With Web services, there is no widely-accepted specific interface that would explicitly talk about contracts or offers and their acceptance or rejection by the client. Instead, some operations of a Web service may be used to inquire for information about what the service offers, e.g. for finding out hotel room availabilities for given dates (see the example in Figure 6.3, on page 97). The client may reject an offer by simply ignoring that data, and it may accept it by calling the operations that deliver the actual functionality of the service; e.g. booking a room in a given hotel for the given dates.

Therefore, for the purposes of defining offer discovery, we distinguish two kinds of Web service operations: *inquiry operations* and *execution operations*, defined as follows:

Definition 7.12 (inquiry operation) An inquiry operation of a Web service is such an operation whose sole function is to query the current state of the Web service, without causing any side-effects.

For a given goal, the applicable offers of a service depend on the goal data, the functionality of the service, and on the current state of the service. The functionality is captured in the semantic description used for service discovery, therefore offer discovery needs to query the state of the service, using the data from the goal. The example hotel reservation service would have inquiry operations such as search(date,city) and getHotelDetails(hotel).

Definition 7.13 (execution operation) An execution operation of a Web service is such an operation whose invocation causes such side-effects that are considered part of the service functionality.

[10] Naturally, an offer may only be valid for a limited period of time; this consideration, however, is not yet included in our work.

Execution operations are those that are involved in consuming offers. The hotel reservation service would have execution operations such as reserve(rate, creditCard,guestInfo) and cancelReservation(reservation).

We denote the set of all operations of a given service as O, the set of all inquiry operations O_q, and the set of all execution operations O_x:

$$O_q \subseteq O, \qquad O_x \subseteq O, \qquad O_q \cap O_x = \emptyset$$

Apart from inquiry operations and execution operations, a Web service may other operations that are not relevant for offer discovery, such as management operations for administration purposes. In other words, we do not intend to present here an exhaustive categorization of Web service operations.

Web service operations have input and output messages, denoted as $in(o)$ and $out(o)$ ($o \in O$). With semantic annotations (cf. Table 4.2, page 63) based on an ontology $\Omega^I = (C, R, E, I)$, a service description can specify the type(s) of data that are contained in those messages, $T(in(o)) \subseteq C \cup R$ and $T(out(o)) \subseteq C \cup R$.

Inquiry operations, being intended for information retrieval, must have a non-empty set of output types:

$$\forall o \in O_q : T(out(o)) \neq \emptyset \tag{7.1}$$

For a given goal, service invocation may only use a subset of the available execution operations, which we call the *intended execution operations* $O'_x \subseteq O_x$. For instance, to book a hotel room, the invocation would only execute the reserve operation; the cancelReservation operation would naturally not be used.

In order for the client to be able to invoke the service's intended execution operations, it must provide data for input messages of the operations.

Definition 7.14 (needed execution input types) A type whose instance the client must provide for a successful invocation of the service's execution operations is called a needed execution input type.

We mark T_x the set of all the needed execution input types for a given goal and service. Because some execution operation input data can come from the output data of other execution operations, we approximate the needed execution input types as the input messages of the intended execution operations minus the types associated with intended execution operations' output messages:

$$T_x = \bigcup_{o \in O'_x} T(in(o)) \;\setminus\; \bigcup_{o \in O'_x} T(out(o))$$

This is a straightforward way of computing an approximate set of the needed execution input types, used below for defining the structure of an offer, without the necessity to analyze the ordering of the execution operations during an actual invocation, especially since the ordering may actually depend on the invocation data. The approximation may prove to be a limitation for Web services with conversational interfaces, but the otherwise necessary analysis of the invocation-time execution operation ordering is beyond the scope of this thesis.

In our example hotel booking scenario, the intended execution operation is reserve(rate,creditCard,guestInfo), therefore the needed execution input types are

hotel rate (which includes here the given set of dates), credit card information, and guest information.

Some of the input data values come from the user goal, in our case the guest and credit card data. The other values must be filled in offer discovery, so that the goal and the offer combined together contain data for each of the needed execution input types.

In addition to data for the needed execution input types, offers can also specify extra information, which is especially useful for ranking. In our example, every rate offered by a hotel reservation service is associated with a specific hotel, and detailed information about the hotel, such as its star rating or concrete location in the requested city, is useful for comparing the rates. Strictly speaking, the client does not need to know the star rating and the concrete location of a hotel to reserve a room there, however, the usefulness of this information for offer ranking is obvious, therefore offer discovery should also find the extra information.

Finally, offer discovery generally deals with multiple services, therefore each offer also needs to be associated with the specific service that offers it. This ensures that the offer is a self-contained construct for the further SWS automation steps: the invocation component knows what service to invoke; and in filtering and ranking the client may express constraints or preferences directly on the services, for instance by building trust with particular providers that delivered good value in the past.

Combining all the aspects discussed above, we formalize the structure of an offer:

Definition 7.15 (offer) An offer ϕ is a tuple that contains an identifier s of a service, and two sets of data values: execution data D^x required for invoking the service and consuming this offer, and extra information D^e that helps the client to filter and to rank the offers:

$$\phi = \langle s, D^x, D^e \rangle$$

For a service described using an ontology $\Omega^I = (C, R, E, I)$, all explicit instance data is understood to be part of E. Therefore,

$$D^x \subseteq E, \quad D^e \subseteq E$$

If the execution data D^x of an offer contains instances for all the needed execution types T_x, an offer is said to be *complete*:

$$\forall t \in T_x : \exists\, x \in D^x : t(x) \tag{7.2}$$

In Equation 7.2, we use the notation $t(x)$ to capture that x is an instance of the type t. Since the types come from the classes C and relations R of an ontology $\Omega^I = (C, R, E, I)$, they are predicates, so we use the predicate notation. For n-ary predicates (members of R), x would be an n-tuple that represents the n parameters of the predicate.

Only a complete offer can be used for service invocation to achieve the execution of the offered functionality. Invocation only uses the service identifier and the execution data of an offer. This allows us to establish an equivalence relationship for complete offers: two complete offers are equivalent iff their service

identifiers and their execution data sets are the same:

$$\phi_1 = \langle s_1, D_1^x, D_1^e \rangle, \phi_2 = \langle s_2, D_2^x, D_2^e \rangle : \ \phi_1 \equiv \phi_2 \Leftrightarrow D_1^x = D_2^x \wedge s_1 = s_2$$

This offer equivalence relation disregards the extra information D^e. If we had two complete offers that would vary only in the extra parameters, the service could not know which of the two offers the client intends to consume. Hence, the offer discovery process must assure that it does not produce different but equivalent complete offers.

Expanding on Definition 7.8 (page 124), we finally define offer discovery:

Definition 7.16 (offer discovery) Offer discovery is a function (named *DiscO* below) which maps a set S of discovered Web services into a set Φ of non-equivalent (Eq. 7.5) complete offers from the provided services (Eq. 7.6), taking into account the user goal $\mathcal{G}$.

$$DiscO(S, \mathcal{G}) = \bigcup_{s \in S} DiscO^1(s, \mathcal{G}) \tag{7.3}$$

$$DiscO^1(s, \mathcal{G}) = \Phi \tag{7.4}$$

$$\forall \phi_1, \phi_2 \in \Phi : \ \phi_1 \equiv \phi_2 \Leftrightarrow \phi_1 = \phi_2 \tag{7.5}$$

$$\forall \phi \in DiscO^1(s, \mathcal{G}), \phi = \langle s', D^x, D^e \rangle : \ s' = s \tag{7.6}$$

While we define offer discovery here to deal with a single discovered service at a time ($DiscO^1$), it is possible that a more sophisticated offer discovery approach can negotiate with multiple services in parallel and pitch them one against another in order to get better deals. For example, a retailer can promise to match any competitor's price, so the offer discovery process would need to get the competitors' offers first and then use them to get matching counteroffers from the retailer. Even though this kind of multi-way negotiation is sometimes possible in the real world, we have not seen a single example of a Web service with such capabilities, therefore we leave this as a possible extension to be revisited in the future.

While we choose to define offer discovery to produce only complete offers, it may be worth in the future to also investigate offer discovery approaches that may deal with incomplete offers. An example objective in this direction would be to use the needed execution input types missing in an incomplete offer to prompt the user for further data.

Finally, in case offer discovery is not actually performed (since the step is optional), every discovered service is treated as a single offer with no data beside the service. Then the function *DiscO* is replaced by a trivial version $DiscO^0$:

$$DiscO^0(S) = \bigcup_{s \in S} \{\langle s, \emptyset, \emptyset \rangle\} \tag{7.7}$$

7.3.2 WSMO-Lite Annotations for Offer Discovery

Offer discovery, as defined above, needs to know what operations are inquiry operations (and how to invoke them automatically), what operations are execution operations relevant to the goal (the *intended execution operations*), and what the inputs and outputs of all these operations are. WSMO-Lite annotations can capture this information, as detailed below.

Identifying inquiry operations: inquiry operations are those operations that only provide information and do not have any significant side-effects. In other words, they are what the Web architecture [3] calls "safe interactions," as described in Section 3.2.4. Information retrieval is the canonical example of a safe interaction: the client may query a service about the availability of hotel rooms, yet by issuing the query the client makes no commitment to book the room. Safe operations are available both in WS–∗ and RESTful services (cf. Sections 5.2 and 6.5). As part of the behavioral semantics of the service, WSMO-Lite uses the WSDL 2.0 category wsdlx:SafeInteraction to denote the safety of an operation.

At present, we treat all safe informations with non-empty output messages (cf. Equation 7.1) as offer-inquiry operations. Even if such an operation is not intended as an inquiry operation, if the offer discovery process determines it should be invoked, the operation's safety guarantees there is no harm in proceeding,[11] although the invocation may not actually return relevant information about the service offers.

Identifying intended execution operations: any non-safe operation can be seen as a potential execution operation. If a given service has a single non-safe operation, it can be presumed to be the intended execution operation. If there are multiple non-safe operations, the set of intended execution operations can be specified directly in the user goal.

Identifying inputs and outputs: Capturing the information semantics of Web services, WSMO-Lite annotates operation input and output messages with pointers to ontology entities, such as RDFS classes. These annotations determine, for instance, the needed execution input types, used in forming the offers and in distinguishing between execution data D^x and extra information D^e.

During offer discovery, the client invokes the inquiry operations, therefore it also needs the lowering annotations on their input messages, and lifting annotations on the output messages. These annotations are embodied directly in SAWSDL `loweringSchemaMapping` and `liftingSchemaMapping` attributes, and in their `lifting` and `lowering` MicroWSMO equivalents.

7.3.3 Goals for Offer Discovery

Offer discovery, like functional service matchmaking, is driven by a user goal, and therefore it also poses requirements on the information contained in the goal structure. An offer discovery goal must specify the intended execution operations O'_x, and it must provide instance data $D_\mathcal{G}$ for invoking the inquiry operations, in order to start the offer discovery process:

$$\begin{aligned} \mathcal{G}: \quad & O'_x \subseteq O_x \\ & D_\mathcal{G} = \{i_1, \ldots, i_n\} \end{aligned}$$

[11] No harm apart from the wasted processing and network resources, whose evaluation is out of scope of this thesis.

7.3.4 Example Service and Goal for Offer Discovery

To support our explanation of the algorithm in the following subsection, we provide here a more detailed example of a hotel reservation service and a matching goal. The service interface is tailored here to illustrate features of the offer discovery algorithm. The service has the following operations:

- *safe* listCities() returns the list of cities (in an ontology, instances of the class City) where this service has hotels.
- *safe* listHotels(City) returns the list of hotels (instances of the class Hotel) in a given city.
- *safe* listRates(Hotel, ReservationRequest) returns the list of concrete rates (instances of Rate) available at the given hotel, specific for the number of guests and for the dates that are together captured in the ReservationRequest input.
- *safe* getHotelDetails(Hotel) returns detailed information about the given hotel.
- reserve(Rate, CreditCard) reserves the room and returns a reservation confirmation, assuming the given rate can be reserved using the given credit card.

The reserve() operation is an execution operation, and all the other operations are inquiry operations. The types of the two parameters of reserve() are the *needed execution input types*: Rate and CreditCard.

The user goal is to reserve a hotel room in Rome (an instance of City) for two people, for August 2–8, 2011 (as specified in an instance of ReservationRequest), using the client's credit card (naturally, an instance of CreditCard). In effect, the goal has three instances in D_G. The goal need not specify that reserve() is the intended execution operation, since it is the only candidate.

7.3.5 Offer Discovery Algorithm

Before we can present our actual offer discovery algorithm, we need to define several auxiliary functions:

plan(initial state, goal state, operations)**:** The offer discovery algorithm uses *AI planning* (cf. [79]) to select a sequence of operations that return offer information. Section 7.3.6 discusses the particular planning algorithm we use.

invoke(service, operation, knowledge base)**:** In order to query the service about its offers, the client must invoke inquiry operations. The function *invoke* represents an invocation of a given operation on a given service, using a given knowledge base for the input data. The result is a set of instances $D^{out} = \{i_1 \dots i_m\}$. If the operation executes successfully, the response data should contain instances for all the output types of the operation:

$$\forall t \in T(out(o)) : \exists x \in D^{out} : t(x)$$

Function: $DiscO^1(s, \mathcal{G})$
Inputs: service s with inquiry operations O_q and execution operations O_x;
user goal $\mathcal{G}$ with intended execution operations O'_x and instance data $D_\mathcal{G}$,
T_x are the needed execution input types for O'_x
Result: the set of offers provided by s

1 $P := plan(D_\mathcal{G}, T_x, O_q) \quad (P = [o_1, \ldots, o_n],\ o_i \in O_q)$
2 $\phi_1 := \langle s,\ D^x = \{x \mid x \in D_\mathcal{G} \wedge \exists\, t \in T_x : t(x)\},\ D^e = \emptyset \rangle$
3 $\Phi := \{\phi_1\}$
4 **for** $i := 1 \ldots n$
5 $\quad \Phi' := \emptyset$
6 $\quad$ **for each** $\phi \in \Phi \quad (\phi = \langle s, D^x_\phi, D^e_\phi \rangle)$
7 $\quad\quad KB := D^x_\phi \cup D^e_\phi \cup D_\mathcal{G}$
8 $\quad\quad D^{out} := invoke(s, o_i, KB)$
9 $\quad\quad$ **if** $D^{out} \neq \emptyset$ **then** $\Phi' := \Phi' \cup addResponseDataToOffer(\phi, D^{out}, T_x)$
10 $\quad \Phi := \Phi'$
11 $\Phi := gatherExtraInformation(\Phi, s, \mathcal{G}, P)$
12 **while** $\exists\, \phi_1, \phi_2 \in \Phi : \phi_1 \neq \phi_2 \wedge \phi_1 \equiv \phi_2 \quad (\phi_1 = \langle s, D^x_{\phi_1}, D^e_{\phi_1} \rangle,\ \phi_2 = \langle s, D^x_{\phi_2}, D^e_{\phi_2} \rangle)$
13 $\quad \Phi := \Phi \backslash \{\phi_1, \phi_2\}$
14 $\quad \phi_1 := \langle s, D^x_{\phi_1}, D^e_{\phi_1} \cup D^e_{\phi_2} \rangle$
15 $\quad \Phi := \Phi \cup \{\phi_1\}$
16 **return** Φ

Figure 7.3: Offer Discovery Algorithm

addResponseDataToOffer(...) and *gatherExtraInformation*(...)**:** In the interest of brevity of the main algorithm, we have extracted two parts of the offer discovery logic into these two functions, discussed further in this section.

Our offer discovery algorithm is shown in Figure 7.3. In short, it starts by creating a plan for executing the inquiry operations, which is then executed, resulting in offer data.

On line 1, the algorithm tries to find a sequence of inquiry operations that will provide the needed execution input types. This is accomplished using AI planning (discussed below in Section 7.3.6), with the initial state being the goal instance data $D_\mathcal{G}$. The planning goal state is specified by all the needed execution input types T_x, and the available operations are all the inquiry operations O_q.

In our example, the goal has data for a City (Rome), a ReservationRequest (2 persons, August 2–8, 2011) and a CreditCard; the plan needs to result in a Rate and a CreditCard (the needed execution input types; the CreditCard instance is already in the goal); and the service provides the four inquiry operations. The resulting plan, straightforwardly, is listHotels→listRates.

The algorithm assumes that the goal instance data does not satisfy all the needed execution input types. If it did, the resulting plan would be empty and the algorithm would only attempt to gather the extra information, as described below.

Alternatively, if the instance data indeed does not satisfy the needed execution input types but no suitable plan can be found, our algorithm will fail because it can produce no complete offers. Such a situation can occur either when the service does not provide suitable offer inquiry operations, or when the goal does not have suitable input data.

After making the offer discovery plan, the algorithm creates an initial offer (line 2), using data from the goal: any instances from the goal that satisfy any of the needed execution input types become execution data of the initial offer. In the beginning, the initial offer is the only one that the offer discovery algorithm knows about (line 3). This offer is also most likely incomplete, or else the inquiry plan prepared above is empty.

In our example, the initial offer only contains the credit card in the execution data, and it needs an instance of the class Rate to be complete.

Now the algorithm can start invoking the planned inquiry operations (lines 4–10). Every operation is invoked for each currently known offer (lines 6–9); the first operation listHotels is invoked only for the initial offer. The request message for the operation invocation is created from the offer's execution data and extra information, together with the instance data from the goal (line 7).

The service's response D^{out} adds data to the current incomplete offer. In fact, the response may contain a list of data describing multiple offers based on the current one. On line 9, the function *addResponseDataToOffer*, detailed below, combines this response data with the current offer, resulting in one or more new offers. For example, our hotel service's listHotels(City) operation will return a list of the hotels in the given city (Rome, as specified in the goal instance data). Combined with the initial offer, the list translates into a set of incomplete offers, one for each hotel.

If the operation invocation fails (returns no data), we interpret it as there being no offers matching the input data. In our example, the service may have no hotels in a given city, or a given hotel is fully booked at the given dates. In such a situation, the current offer is eliminated from further consideration: it is not added to the set Φ' of new offers on which the next operation in the plan will be invoked.

After all the planned operations are invoked, all the offers in Φ are expected to be complete. However, a complete offer is only guaranteed to satisfy all the needed execution input types, but it does not necessarily contain all the interesting information that the service can provide. For example, after invoking listHotels and listRates, we presumably have information about the prices of the offered rooms, but we do not necessarily know anything about the hotels. For that, we want to invoke the operation getHotelDetails. This is done in the function *gatherExtraInformation* (line 11), described further below.

The algorithm, as described so far, does not guarantee that it results in non-equivalent offers (see Equation 7.5, page 137). No equivalent offers can be created in our example hotel scenario, because the offered rates are assumed to be hotel-specific. In a different scenario, however, we can demonstrate how the algorithm can arrive at two different equivalent offers: let us say the user wants a wall clock, and we are discovering the offers of an online store service. In this hypothetical case, there are two subcategories of wall clocks: analog and digital. A novelty clock that has both an analog face and a digital display could be in both categories, and would constitute two equivalent offers: the execution data would be the same (the clock's product ID), but the extra information would differ (one offer would have come through the analog clock subcategory, the other through digital).

Therefore, at the end (lines 12–15), the algorithm combines all equivalent offers by merging their extra information sets D^e (the service identifier and the execution data are the same in equivalent offers).

Combining response data with the current offer

An operation's response can contain data about a list of offers. Since we do not have any explicit offer mark-up in the data, we must recognize different offers by other characteristics of the data. Our approach is based on three assumptions on the structure and function of the inquiry operations:

1. For the successful invocation of any inquiry or execution operation o, we assume that for each input type $t \in T(in(o))$, the request (input) message (a set of instances D^{in}) need only contain a single instance x of that type:

$$\forall t \in T(in(o)),\ \forall x_1, x_2 \in D^{in} : (t(x_1) \wedge t(x_2)) \ \Rightarrow\ x_1 = x_2$$

   If a list of values is needed, it should be encapsulated in a single container, making explicit the relation between the multiple values.

2. We assume that there is only one type $t \in C \cup R$ (of the ontology $\Omega^I = (C, R, E, I)$) that has multiple instances in the response (output) data D^{out} of an invoked inquiry operation:

$$\forall t_1, t_2 \in C \cup R,\ \forall x_1, x_2, x_3, x_4 \in D^{out} :$$
$$\big(t_1(x_1), t_1(x_2), x_1 \neq x_2, t_2(x_3), t_2(x_4), x_3 \neq x_4\big) \ \Rightarrow\ t_1 = t_2$$

3. We also assume that an inquiry operation will not return new instances for types already satisfied in the current offer:

$$\neg\exists\, x \in D^{out} : \big(\exists\, t \in C \cup R,\ \exists\, x' \in D^x_\phi \cup D^e_\phi :\ x' \neq x \wedge t(x) \wedge t(x')\big)$$

With these assumptions, if the response of an inquiry operation contains multiple instances of some type, these instances describe distinct offers.

The first assumption allows us to treat the multiple instances of the same type as describing different offers, because they cannot be used together in a single invocation. It also implies that no offer will contain two different instances of the same type. The second assumption ensures that we only have one type with multiple instances; otherwise we wouldn't know what instance of the first type describes the same offer as some instance of the second type. Finally, the third assumption guarantees that no data in the response of an inquiry operation will conflict with existing data in the current offer.

As part of future work, we plan to investigate how constraining these assumptions are on real-world Web service descriptions, and how we could relax the assumptions while keeping automated offer discovery possible.

Building on these assumptions, Figure 7.4 shows the formalized algorithm that implements the function *addResponseDataToOffer*.

The *addResponseDataToOffer* algorithm works like this: if a type t exists that has multiple instances in the response data (line 3), the algorithm extracts the set D^* of those instances (line 4). These instances describe the different offers.

The other instances in the response data (the set $D^\diamond$) are then split into execution instances ($D^\diamond_x$) and extra instances ($D^\diamond_e$), depending on whether they are instances of a needed execution input type or not (lines 5–7). All these instances apply to all the (potential) multiple offers.

Function: *addResponseDataToOffer*(ϕ, D^{out}, T_x)
Inputs: offer $\phi = \langle s, D^x_\phi, D^e_\phi \rangle$,
response data set D^{out} from inquiry operation invocation,
the needed execution input types T_x
Result: the set of offers resulting from merging D^{out} with ϕ

1 $\Phi := \emptyset$
2 $D^* := \emptyset$ *(a set of multiple response data instances with the same type)*
3 **if** $\exists\, t : \big(\exists\, x_1, x_2 \in D^{out} : x_1 \neq x_2 \wedge t(x_1) \wedge t(x_2)\big)$ **then**
4 $\quad D^* := \{x \mid x \in D^{out} \wedge t(x)\}$
5 $D^\diamond := D^{out} \backslash D^*$
6 $D^\diamond_x := \{x \mid x \in D^\diamond \wedge \exists\, t' \in T_x : t'(x)\}$
7 $D^\diamond_e := D^\diamond \backslash D^\diamond_x$
8 **if** $D^* = \emptyset$ **then**
9 $\quad \phi' := \langle s,\ D^x_\phi \cup D^\diamond_x,\ D^e_\phi \cup D^\diamond_e \rangle$
10 $\quad \Phi := \{\phi'\}$
11 **for each** $x \in D^*$
12 $\quad$ **if** $\exists\, t \in T_x : t(x)$ **then**
13 $\quad\quad \phi' := \langle s,\ D^x_\phi \cup D^\diamond_x \cup \{x\},\ D^e_\phi \cup D^\diamond_e \rangle$
14 $\quad$ **else**
15 $\quad\quad \phi' := \langle s,\ D^x_\phi \cup D^\diamond_x,\ D^e_\phi \cup D^\diamond_e \cup \{x\} \rangle$
16 $\quad \Phi := \Phi \cup \{\phi'\}$
17 **return** Φ

Figure 7.4: Supporting Function *addResponseDataToOffer*

If there is no type that has multiple instances in the response data, the result is a new offer enriched with the response instance sets $D^\diamond_x$ and $D^\diamond_e$ (lines 8–10). Otherwise, each of the instances in D^* spawns a new offer that contains this instance, along with all the instances from the sets $D^\diamond_x$ and $D^\diamond_e$ (lines 11–16). The instance from D^* is added to the offer's execution data (line 13) or to the extra information (line 15), depending on whether or not it is an instance of a needed execution input type.

Gathering extra information for offer ranking

After invoking the planned inquiry operations, the main algorithm tries to gather further extra parameters for all the known offers (line 11). This is necessary because the planned operations only satisfy the execution parameters, so it would never invoke an operation such as getHotelDetails, because the resulting hotel information is not an execution parameter, and it is not an input to any other inquiry operations that would return execution parameters.

Figure 7.5 shows the formalized algorithm that implements the function *gatherExtraInformation*. In short, this algorithm tries the remaining inquiry operations (those not involved in the offer discovery plan in the main algorithm), and invokes those that can return new extra information about the known offers.

At first, on line 1, the algorithm selects the inquiry operations O'_q that aren't part of the main offer discovery plan and that do not return any execution data, which would likely conflict with the already-complete offers.

The algorithm then deals separately with each of the input offers (lines 3–13). For each offer ϕ, the algorithm starts with the full set of inquiry operations selected above, and then it selects and invokes any operation o for which the

Function: *gatherExtraInformation*$(\Phi, s, \mathcal{G}, P)$
Inputs: a set Φ of complete offers; service s with inquiry operations O_q;
user goal $\mathcal{G}$ with intended execution operations O'_x and instance data $D_\mathcal{G}$,
T_x are the needed execution input types for O'_x;
a plan P of operations used to acquire the complete offers Φ
Result: the set of updated offers

1 $O'_q := \{o \mid o \in O_q \ \wedge \ o \notin P \ \wedge \ \big(T(out(o)) \cap T_x\big) = \emptyset\}$
2 $\Phi' := \emptyset$
3 **for each** $\phi \in \Phi \quad (\phi = \langle s, D^x_\phi, D^e_\phi \rangle)$
4 $\quad KB := D^x_\phi \cup D^e_\phi \cup D_\mathcal{G}$
5 $\quad O^*_q := O'_q$
6 $\quad$ **while** $\exists\, o \in O^*_q : \big(\forall t \in T(in(o)) : \exists\, x \in KB : t(x)\big) \wedge$
7 $\qquad\qquad \big(\exists\, t \in T(out(o)) : \neg \exists\, x \in KB) : t(x)\big)$
8 $\quad\quad O^*_q := O^*_q \backslash \{o\}$
9 $\quad\quad D^{out} := invoke(s, o, KB)$
10 $\quad\quad \phi := \langle s, D^x_\phi, D^e_\phi \cup D^{out} \rangle$
11 $\quad\quad D^e_\phi := D^e_\phi \cup D^{out}$
12 $\quad\quad KB := D^x_\phi \cup D^e_\phi \cup D_\mathcal{G} \cup D^{out}$
13 $\quad \Phi' := \Phi' \cup \{\phi\}$
14 **return** Φ'

Figure 7.5: Supporting Function *gatherExtraInformation*

offer contains enough input data (line 6), and which can return some extra information that the offer does not yet have (line 7). The invoked operation is then removed from further consideration for this offer (line 8), and the current offer is updated with the new extra information (lines 10–12).

This algorithm may invoke operations whose outputs are not useful for ranking, wasting network and processing resources. Future improvements to the algorithm could add knowledge of types known to the system's ranking component(s), so that only useful data is targeted.

7.3.6 Discussion on Planning for Offer Discovery

Automated planning, developed in the research field of Artificial Intelligence (cf. [106]), is a process of finding a sequence of actions that takes a system from a specified initial state to a specified goal state. In offer discovery, we use planning to find a sequence of operations that will give us information about the available offers.

Planning problems are specified through an initial state, a goal state, and the set of available actions. In our case, the initial state consists of the instances available in the user goal[12] ($D_\mathcal{G}$), the goal state is that all the *needed execution input types* T_x are satisfied by the known data (we want to get *complete* offers), and the actions are all the inquiry operations O_q of the given service.

For planning, actions are specified with preconditions and effects, which may even be complex logical expressions. In WSMO-Lite, each inquiry operation o is specified through its input and output types sets $T(in(o))$ and $T(out(o))$.

[12] Note that the word "goal" is overloaded here: there is the SWS automation goal from the user (which we call *user goal*), and the planning goal state (called *goal state* for short). The user goal constitutes the initial state of the offer discovery planning; it has no direct relation the planning goal state.

The planning precondition of an offer inquiry operation is that the current state S_i contains instances for all its input types, and the effect of a successful invocation is that in the following state S_{i+1} there are new instances of all the output instances:

$$\begin{aligned} \textit{precondition}: &\quad \forall t \in T(in(o)) : \exists\, x \in S_i : t(x) \\ \textit{effect}: &\quad \forall t \in T(out(o)) : \exists\, x \in S_{i+1} : t(x) \end{aligned}$$

Since the planning is done before the actual invocation of the operations, we do not in fact deal with data instances; instead we can simply represent a state as the set of classes for which instances are assumed to be known, treating each of the classes as an atomic proposition.

In effect, we can capture our planning problem as a STRIPS instance (cf. [31]) $\langle S_0, S_g, A\rangle$, where S_0 is the initial state, S_g is the goal state, and A is the set of actions. In STRIPS, each action is specified as a triple $\langle p^+, e^+, e^-\rangle$, where p^+ is the precondition (a set of propositions that must be true) of the action, and e^+, e^- together specify the effect: e^+ is the *add list*, a set of propositions that will become true after the action, while e^- is the *delete list*, a set of propositions that will become false.

The initial state S_0 of our offer discovery planning problem would formally be the union of all the sets of types (from an ontology $\Omega^I = (C, R, E, I)$) of all the instances available in the goal. The goal state is simply the set T_x of the needed execution input types:

$$\begin{aligned} S_0 &:= \{t \mid t \in C \cup R \ \wedge\ \exists\, x \in D_{\mathcal{G}} : t(x)\} \\ S_g &:= T_x \end{aligned}$$

For each inquiry operation $o \in O_q$, we create a STRIPS action a as follows:

$$\begin{aligned} p_a^+ &:= T(in(o)) \\ e_a^+ &:= T(out(o)) \cup \bigcup_{t \in T(out(o))} \{\bar{t} \mid \bar{t} \in C \cup R \ \wedge\ t \subseteq \bar{t}\} \\ e_a^- &:= \emptyset \\ a &:= \langle p_a^+, e_a^+, e_a^-\rangle \end{aligned}$$

The precondition of the action consists of the input types of the respective inquiry operation. The add list of the action, however, is not only the operation's set of output types, but also all the super-classes (or super-relations) of those types. Note that the initial state S_0, shown above, contains all the types of the user goal instances, which implicitly also includes any super-classes or super-relations of those types. This allows us to include subsumption relationships from the underlying ontology, even if we do not support any other reasoning.

The actions in offer discovery planning have empty delete lists e^- — we cannot unlearn something. In our current offer discovery approach, we disregard the possibility that the state of the service may change during offer discovery, and that a subsequent inquiry operation would be able to express that. See also the discussion of replanning at the end of this section.

To solve the STRIPS instance, we use the GRAPHPLAN algorithm [14]. Since there cannot be any conflicts in the planning graph (due to the fact that we do

not have delete lists), the algorithm runs very efficiently and finds the shortest partial-order plan.

Static planning with inputs and outputs, as we have described, has proven to be a very efficient method of planning the offer discovery process. However, it also incurs limitations, such as the following:

- Related to our third assumption on the structure and function of inquiry operations (cf. page 142), the plan cannot use an operation that returns the same type of data that it has as its input. For instance, a retail service may structure its products into a hierarchy of categories, and it may have operations such as listProducts(Category) and listSubcategories(Category). The second operation would never be called because it returns instances of a type that we already must have in order to be able to invoke the operation.

- If the invocation of a planned operation fails, the main offer discovery algorithm simply eliminates the offer, assuming there are no complete offers with the data presented by this offer. While we listed earlier some cases where this would be true, it is also possible that an alternate inquiry operation, one that wasn't selected for the plan, would return useful offer data.

- The plan only uses the semantic service description in a limited way (only considering the input and output types as atomic propositions), therefore it cannot react to conditions that fall outside the description. For example, if a service requires a ProductID for ordering a product, and it has an operation listProducts that returns the type ProductDescription which contains the ID, the planner as described above would fail to find a plan, because it does not know that with a ProductDescription it also gets a ProductID.

Some of these limitations may be avoided if we employ *replanning* or other dynamic approaches (cf. [106]) to react to an environment that is not completely predictable. For example, if an operation fails, a new plan can be created that does not use this operation; or if unexpected data comes, a new plan may be able to use it to get better offer information.

Also, a planner might be able to use more complex reasoning than only subsumption, enhancing the algorithm's understanding of the inquiry operations. However, semantic planning is an ongoing research field outside of the scope of this thesis.

7.3.7 Offer Discovery Conclusions

The offer discovery algorithm described above is intentionally designed to require the minimal amount of semantics. In particular, if the goal specifies the intended execution operations, the only semantic annotations necessary for running offer discovery are operation safety (identifying the inquiry operations) and the inputs and outputs of all the operations. In the spirit of lightweight semantic annotation, the less semantics is needed, the easier it is for service providers to create the semantic descriptions.

We have described offer discovery independently of the other semantic automation tasks. Integration of the algorithm with other components of the SEE can result in useful improvements: for instance, if offer discovery is closer connected with ranking, the list of supported data types in the ranking component could inform the gathering of extra information. Cooperation of offer discovery with automated invocation could provide automatic recognition of the intended execution operations. And through closer integration with functional matchmaking, we could also better evaluate which of the offers returned by the service for the user's input data actually fit the user's goal, if captured using logical expressions.

We have also mentioned earlier that the algorithm might be improved by incorporating sensing and replanning aspects, where the offer discovery plan could react to the actual run-time behavior of the service. Here, we would be applying *nonclassical planning*, which includes approaches for partially observable or stochastic environments, as described for instance in Chapters 12 and 17 of [106].

Finally, while offer discovery communicates with third-party services, we have ignored the considerations of trust and security, which are, in general, well outside the scope of this thesis. For example, credit card information should be considered sensitive and as such, it should not be sent to arbitrary services on the public internet, even if the annotations of a service seem to indicate that sending credit card information would lead to information about offers.

A simple approach to tackling this problem could make a trust line after *service selection*: the SEE would not communicate any sensitive information to any services before the user selects a service (and a concrete offer) to execute. The offer discovery algorithm presented here would need to split the goal instance data into sensitive and public, using both for satisfying the needed execution input types, but only the public instance data would be used for invoking inquiry operations.

7.4 Nonfunctional Filtering and Ranking

As discussed in Section 7.1.2, after functional matchmaking and offer discovery come the steps of filtering (Definition 7.9) and ranking (Definition 7.11) based on nonfunctional properties of services and offers, with respect to the user's constraints and preferences. Services can be filtered and ranked according to their nonfunctional properties (NFPs), and if offer discovery is performed, the resulting offers can be filtered/ranked based both on the service NFPs and the actual offer data. The resulting ranked list will let the user (or the system) select the best-suitable service/offer to use.

In the following subsections we first analyze selected existing works on nonfunctional ranking and filtering, in order to choose a particular approach that is simple yet powerful enough to demonstrate the suitability of WSMO-Lite for these tasks. Then we define how this approach models nonfunctional properties and goals, and finally we describe the actual ranking and filtering algorithm.

7.4.1 Analysis and Definition of Nonfunctional Filtering and Ranking

In Chapter 2, we have discussed the main literature on service ranking. Based on the extensive analysis of nonfunctional service properties provided by O'Sullivan et al. in [81], Toma et al. proposed in [118] a set of NFP ontologies[13] and a multi-criteria ranking approach that we adapt here to WSMO-Lite.

Where common NFP filtering and ranking approaches such as [94, 70] consider only numerical or keyword values for nonfunctional properties, Toma argues that such approaches are inflexible and that an ontological representation with the help of logical expressions allows *semantic ranking* to provide more accurate results. In Toma's approach, the value of a nonfunctional property of a service may depend on the concrete goal data: the service NFP description includes logical expressions that compute concrete NFP values at run-time. For example, given a package described in the goal data supplied by the user, the NFP expressions can compute the actual price and the expected duration of shipping the package. Formally, Toma treats the computed NFP values as properties of the goal data, in context of a given service.

[118] only deals with nonfunctional ranking (based on user preferences), it does not deal with nonfunctional filtering on user constraints, therefore we provide a straightforward extension that adds filtering functionality. Compared to ranking, nonfunctional filtering is a simpler problem where the client specifies a set of constraints that can be decided over each discovered service through direct expression evaluation.

There are also approaches to dealing with nonfunctional properties such as [148] that evaluate client's requirements on a whole composition of services (cf. Section 7.6), not independently on each service, as is done in [118]. [148] uses simple numerical values for nonfunctional properties, but it can easily be combined with Toma's semantic approach, therefore our choice of the approach of [118] is sufficient to demonstrate that WSMO-Lite supports powerful nonfunctional filtering and ranking.

7.4.2 Modeling Nonfunctional Properties with WSMO-Lite

In WSMO-Lite, nonfunctional properties are associated with a service through model references pointing to instances in a nonfunctional-semantics ontology $\Omega^N = (C, R, E, I)$, as shown in Table 4.2 (page 63). The instances can either carry an actual fixed literal value, or a logical expression used to compute the value at run-time. In RDF, each nonfunctional property instance is of type wl:NonfunctionalParameter; data-type predicates on the instance are concrete fixed nonfunctional values; and logical expressions are attached to the instance with the predicate rdfs:isDefinedBy.

At run-time, we treat the NFP values (whether the literal values fixed on a service or the values computed from goal data) as *extra parameters* of concrete offers; these values are the same for all the offers of the particular service, and affect the comparison of offers from different services.

[118] does not provide a strong formalization of their NFP model and ranking

[13] The NFP ontologies are available at `http://www.wsmo.org/ontologies/nfp/`

approach, therefore we offer here a formalization consistent with the semantic service model from Section 4.3.2, and with the offer model from Section 7.3.

Taking the aggregate of all the nonfunctional property instances attached as annotations to a single service s, we can represent the nonfunctional properties of this service as a tuple $N_s = \langle V_s, L_s \rangle$ where V_s is a set of actual NFP values, and L_s is a set of logical expressions. NFP values in V_s are instances[14] $\pi(x_\pi)$, where $\pi \in R$ is an NFP property type (e.g. *actual price* or *expected duration* for a delivery service, modeled in the NFP ontology), and x_π is the concrete numeric value of the given property. The value can either be fixed (expressed in RDF as an explicit triple), or computed based on a given user's goal by the logical expressions in L_s.

To summarize, the concrete form for nonfunctional description of a service with the approach adapted from [118] is formalized as follows:

$$\begin{aligned} \mathcal{W}: \quad N_s &= \langle V_s, L_s \rangle \\ V_s &= \{\pi(x_\pi) \mid \pi \in R,\ \pi(x_\pi) \in E\} \\ L_s &= \{l_1, \ldots, l_n\} \\ \Phi_s &= \{\phi \mid \phi = \langle s, D^x, D^e \rangle \ \wedge\ D^e \supset V_s\} \end{aligned}$$

Note that the definition also captures how the nonfunctional properties become part of the extra parameters of all the service's offers.

7.4.3 Goals for NFP Filtering and Ranking

For nonfunctional filtering and ranking, the user goal must specify the input data used to calculate run-time NFP values, along with the user's constraints and preferences. The constraints define the acceptable ranges of nonfunctional values, while the preferences set the relative importance of different nonfunctional parameters.

Formally, the input data is a set of instances denoted $D_\mathcal{G}$. Client preferences, in accordance with [118] but using our notation, are expressed as a set $P_\mathcal{G}$ of tuples $\langle \pi, r_\pi \rangle$, where π is an NFP property and r_π is a number between -1 and 1 that defines the *relevance* of the value of x_π for service ranking. Positive relevance expresses a desire for higher values of π, whereas negative relevance means that lower values are preferred.[15] The absolute value $|r_\pi|$ expresses the importance of π relative to other properties.

Since [118] does not handle nonfunctional filtering, we extend the approach with a straightforward formalization of user constraints, expressed as a set $C_\mathcal{G}$ of logical expressions $l_1, \ldots, l_m$ that are evaluated over concrete offers.

In summary, a goal for nonfunctional filtering and ranking is described as follows:

$$\begin{aligned} \mathcal{G}: \quad P_\mathcal{G} &= \{p \mid p = \langle \pi, r_\pi \rangle,\ \pi \in R \ \wedge\ -1 \leq r_\pi \leq 1\} \\ D_\mathcal{G} &= \{i_1, \ldots, i_n\} \\ C_\mathcal{G} &= \{l_1, \ldots, l_m\} \end{aligned}$$

[14] We use the predicate notation same as in Equation 7.2.

[15] Toma et al. only allow relevance (which they call *importance*) between 0 and 1, and they add a global parameter to the goal which controls the ordering of the results; their approach, however, does not allow the user to prefer lower values of one property and higher values of another.

7.4.4 NFP Filtering and Ranking Algorithm

Figure 7.6 shows the ranking algorithm, reflecting the adaptations we made to the approach of [118]. It works in two steps: first it computes all the NFP values for all the available services (lines 1–8), filtering out those services that do not satisfy the goal constraints, and then it computes the ranking value of each service, based on the goal preferences (lines 9–15). In the end, a higher ranking value means a better match of the service to the goal.

Algorithm: Web service ranking with WSMO-Lite nonfunctional properties
Inputs: set Φ of all known offers,
sets $D_\mathcal{G}$ with the input data, $C_\mathcal{G}$ with goal constraints,
and $P_\mathcal{G}$ of goal preferences.
Result: a set of filtered offers along with their ranking values

1 $\Phi' := \emptyset$ *(the filtered offers, including their computed NFP values)*
2 $V := \emptyset$ *(all the NFP values, used for normalization)*
3 **for each** $\phi \in \Phi : \phi = \langle s, D^x, D^e \rangle$
4 $\quad V'_s := evaluate(L_s, D_\mathcal{G})$
5 $\quad V := V \cup V_s \cup V'_s$
6 $\quad \phi' := \langle s, D^x, D^e \cup V_s \cup V'_s \rangle$
7 $\quad$ **if** $check(C_\mathcal{G}, \phi')$
8 $\quad\quad \Phi' := \Phi' \cup \{\phi'\}$
9 $\Phi^* := \emptyset$ *(set of offers and their ranking values)*
10 **for each** $\phi \in \Phi' : \phi = \langle s, D^x, D^e \rangle$
11 $\quad r_\phi := 0$
12 $\quad$ **for each** $\pi : \ \pi(x_\pi) \in D^e, \ \langle \pi, r_\pi \rangle \in P_\mathcal{G}$
13 $\quad\quad m_\pi := max(\{\, |x| \mid \pi(x) \in V \,\})$
14 $\quad\quad r_\phi := r_\phi + r_\pi \frac{x_\pi}{m_\pi}$
15 $\quad \Phi^* := \Phi^* \cup \{\langle \phi, r_\phi \rangle\}$
16 **return** Φ^*

Figure 7.6: NFP-based Web service ranking algorithm for WSMO-Lite

On line 4, the function *evaluate* encapsulates reasoning over the logical expressions in the service's nonfunctional properties to compute the NFP values. The result of the function is a set of NFP values with the same structure as V_s defined above.

On line 7, the function *check* encapsulates reasoning over the logical expressions in the goal constraints, returning true if all the expressions in $C_\mathcal{G}$ were satisfied by the nonfunctional properties of the offer ϕ'. If so, the offer is added to the filtered set Φ' of all the offers that satisfy the goal constraints.

For each such offer, lines 11–14 compute its ranking value as a sum of the normalized value $\frac{x_\pi}{m_\pi}$ of every nonfunctional property π (m_π is the maximum absolute value of the property π among the known offers) multiplied by the relevance r_π from the goal preferences.

After the services have been ranked, one should be selected for use. The selection process is discussed in the following section.

7.5 Final Service/Offer Selection

After functional service discovery finds suitable services for a goal, and offer discovery optionally obtains applicable offers, nonfunctional filtering and rank-

ing sorts the services or offers based on the client's constraints and preferences. Ultimately, one or more of the services or offers may be selected for invocation, to actually use a service, or to accept an offer. In this section, we discuss considerations around the selection from the ranked list of services or offers in various scenarios that would employ semantic discovery.

Note that the following text only deals with offers; if offer discovery is not performed, each discovered service can be seen as a single trivial offer, and repeating the words "services or offers" in the text would harm its readability.

Offer selection can be automatic (simply the best-ranked offer), or manual, giving the ranked list of offers for review to a human user.

Automatic selection is suitable in settings with high-quality data: the service registry must contain reliable (complete, correct, up-to-date) service descriptions, and the client goal must be detailed enough to avoid functional false-positives, and to express a sufficient degree of nonfunctional constraints and preferences that the top-ranked offers are always acceptable. For example, an enterprise may maintain an internal vetted registry of the services of contracted business partners, classifying them in strictly non-overlapping functional categories, and describing fully their relevant nonfunctional properties such as price. An ordering system within such an enterprise may automatically select the cheapest service for a given category of supplies that are needed at the moment.

Even though automated selection will always choose the best-ranked offer, the remainder of the ranked offers list may still be useful, mainly for failover: if the selected offer fails (because of a service or network failure, or other runtime factors), the system may simply move to the next-best offer. By keeping the ranked offers list, such a system may avoid repeating the potentially expensive discovery and ranking steps.

On the other hand, **manual selection** is suitable when it is desirable to have a human review the ranked list of offers to verify which one(s), if any, should be invoked. There are a number of possible reasons why human review could be desirable: i) the data in the service registry may not be completely reliable or trustworthy; ii) the client goal may not fully capture the user's needs and preferences; or iii) a human user may simply not intend immediate invocation, instead they may be browsing the offers to see what is available.

The first two cases both boil down to formal insufficiencies in the inputs to discovery and ranking. In the first case, the service descriptions may contain errors or omissions, and in the second case, the client goal may be underspecified. The focus of our work on the use of lightweight semantics implies that service providers, and clients, should not be required to define every minute detail of their services and goals in a formal logic. In effect, a manual review of the ranked list of offers is balanced against the ease of use of the lightweight semantic system: time and effort is saved on describing the services and formulating the goals, the matchmaking and ranking engine works faster because it does not have to deal with high expressivity, and in the end, some time is spent on manual selection. Therefore, manual offer selection naturally complements automation based on lightweight semantics.

In cases when the service invocation to accept an offer is *safe*[16], the first entry (or the first few entries) of the ranked list can be enriched with the results

[16] A safe interaction in terms of Web architecture.

of invocation. This kind of enrichment is common in Web search: the search engine Google, when asked for "weather in Rome", returns a list of Web sites about weather and Rome, but the first result is immediately the current and forecast weather. Similarly in SWS discovery, if the first offer/service in the ranked list is invoked prior to returning the list to the client for manual selection, the invocation results may be immediately useful for the user. In effect, such opportunistic invocation increases the usage efficiency of the system, at the cost of performing potentially unnecessary (but safe) invocations.

In summary, even though selection seems a trivial task after discovery and ranking, SWS automation systems should support a manual human review of the ranked list of discovered services and offers. Involving the user in key points of an automated process improves the transparency and thus the perceived trustworthiness of the system.

7.6 Service Composition

As defined at the beginning of this chapter, Web service composition is the process of combining existing services in such a way that they provide a desired functionality; it is used especially in cases when the whole desired functionality is not offered by any single available service. The result of a service composition process is a *composite service*.

As discussed in Chapter 2, there are many different approaches to automated service composition. For the tractable functional-level composition, [68] represents approaches that match services into a sequence based on their inputs and outputs, and [42] is an example of more expressive approaches that use the preconditions and effects of Web services. Process-level composition approaches such as [91] take into account the behavioral interfaces of the composed services, treating services as processes rather than atomic functions.

The result of Web service composition may simply be a linear sequence of services (e.g. [42]), or it can be a non-linear composition with parallel and/or conditional branches ([68, 91]).

In this section, we adapt to WSMO-Lite the composition approach from Hoffmann et al. [42]. This chosen approach illustrates how WSMO-Lite can support powerful composition with service preconditions and effects, without delving into the complexity of process-level composition. While the algorithm produces linear compositions, it is a property of the algorithm — WSMO-Lite can just as well support more complex composition approaches.

In the following subsections, we first define the formalism we use for the adapted approach, and then we show the actual composition algorithm.

7.6.1 Composition Formalism

The composition algorithm of Hoffmann et al. is defined using a formalism that is independent of the underlying SWS technology; here we show how it can be used with WSMO-Lite. We only show a subset of the formalism that is required to explain the high-level functioning of the algorithm adapted to WSMO-Lite.

In the formalism, Web services are represented with their preconditions and effects. In WSMO-Lite, these are captured as a capability K (kappa, see Section 4.3.2). Further, client goals are defined by Hoffmann et al. through precon-

ditions and effects, where the precondition serves only as the supply of initial constants. Therefore, we capture a goal directly as the desired effect and the set of initial constants:

$$\begin{aligned} \mathcal{W}: \quad & K = (\Sigma, \phi^{pre}, \phi^{eff}) \\ \mathcal{G}: \quad & \psi^{eff}(x_1, \ldots, x_n) \\ & D_{\mathcal{G}} = \{i_1, \ldots, i_m\} \end{aligned}$$

where the effect logical expression $\psi^{eff}(x_1, \ldots, x_n)$ describes the desired goal models, and $D_{\mathcal{G}}$ represents the initial set of constants for the algorithm.

Note that the description of the Web services and goals for composition here is compatible with the descriptions for functional matchmaking with preconditions and effects (Section 7.2.2): descriptions tailored for fine-grained discovery can be used for automatic composition as well.

The composition algorithm of Hoffmann et al. follows Winslett's *possible models* approach [128] to define the semantics of updates that occur when a Web service is applied to a given expected state. The algorithm is built on a notion of *beliefs*: a belief is captured as a set of *models* that are considered possible; i.e., at each point in a composition, our uncertainty about the true state of the execution is expressed in terms of the set of models that may be possible. An *initial belief* b_0 is created from the background ontology and the constants in $D_{\mathcal{G}}$. A *solved belief* is such a belief whose all models fulfill the desired goal effect.

Given a belief b and a service s, the result of *applying* s in b is a new belief (a set of models), denoted $apply(b, s)$. Each of the new models captures one possible way the old belief can be updated to reflect the service's effect ϕ^{eff}. Creating new models that satisfy a given logical expression is called *update reasoning*, and it is a known hard problem (cf. [41]). Hoffmann et al. use approximate reasoning with Horn theories, which they show to be tractable. The detailed formal definition of the function $apply(m, s)$ can be found in [42]; the gist of their approximate reasoning is that the algorithm computes an under-approximation and an over-approximation of the statements that hold after applying service s in model m. These two approximations are then used to check for solutions, as we discuss below.

7.6.2 Composition Algorithm

Figure 7.7 outlines the composition algorithm, with updates from [42] to adopt WSMO-Lite terminology. The overall structure of the algorithm is typical for state-space search algorithms (see [106]). The inputs are the known services and the goal, and a successful output is a sequence of service applications that solves the goal. The algorithm searches in a space of beliefs that correspond to states in a typical AI planning search. The initial belief b_0, created on line 1, combines the background ontology with the goal data.

The algorithm works with a so-called *open-list* O, which contains all the beliefs that have yet to be processed; initially, that is only the belief b_0 (see line 3). In the open-list, each belief is kept in a 4-tuple $\langle b, h, H, p \rangle$, where b is the belief itself, p is the path that leads to this belief (a sequence of Web service applications), and h and H are additional values returned by a *heuristic function* that can help guide the search. The value of h is an estimate on how many Web

Algorithm: forward-search Web service composition for WSMO-Lite
Inputs: set S of known services annotated with capabilities,
$\psi^{\mathit{eff}}(x_1, \ldots, x_n)$ defining the goal models,
$D_{\mathcal{G}}$ with the initial constants.
Result: a list of Web services to be applied in a sequential composition

```
1  b_0 := initialBelief(D_G)
2  (h, H) := heuristicFunction(b_0)
3  O := (⟨b_0, h, H, ()⟩)
4  while O ≠ ∅
5  |  ⟨b, h, H, p⟩ := selectAndRemoveBest(O)
6  |  if isSolved(b) then return p
7  |  for each s ∈ H do
8  |  |  b' := apply(b, s)
9  |  |  if b' is undefined then next s
10 |  |  (h', H') := heuristicFunction(b')
11 |  |  add(O, ⟨b', h', H', add(p, s)⟩)
12 abort "no solution exists"
```

Figure 7.7: Web service composition algorithm for WSMO-Lite

services still need to be applied to b in order to obtain a solution (in other words, how close to the solution the belief appears to be), and H is a subset of all the available Web services that the heuristic function deems applicable to b. The heuristic value h guides the algorithm to the most appropriate beliefs in the search graph, and the set H prunes the search graph by dropping unwanted services.

The main loop on lines 4–11 executes until the open-list is empty, or until the solution is found. At every step, it selects the *best* next belief for processing (as indicated by the heuristics) and removes it from the open-list (line 5). As presented, the algorithm is a "greedy best-first search" [106], but it can be changed effortlessly to other search algorithms, such as A* which is commonly very efficient in finding a solution.

Having selected the next belief, we compare it on line 6 with the goal, using the function *isSolved*. If the goal effect is satisfied in the belief, we have found a solution and the algorithm ends. With the tractable approximate update reasoning used by Hoffmann et al., a solution is guaranteed if the goal effect is satisfied in the under-approximated view on the current belief, and a solution is only potentially found if the goal effect is satisfied in the over-approximated belief. In an iterative system, the search algorithm could return all the *potential* solutions it encounters while continuing to search for a *guaranteed* solution, increasing the responsiveness of the user interface.

If the solution has not been reached yet, we *expand* the currently-selected belief: we generate new beliefs by applying the available Web services (in the loop on lines 7–11). Applying the Web service s on the belief b (line 8) leads to a changed belief b', which is added to the open-list, along with its heuristic values and the updated path (lines 10 and 11). If the Web service s is not applicable to the belief b (either its precondition does not apply, or its effect leads to a contradiction), we simply skip this service and move to the next one (line 9).

If the open-list becomes empty, the algorithm has explored the entire search space without finding a solution, and it ends (line 12).

After the algorithm finds a composition solution, whether a potential one found with the over-approximated reasoning, or a solution guaranteed by the under-approximated reasoning, the composition can be presented to the user, who may need to fill in details of data or process mediation (e.g. [20, 78]). If multiple potential solutions are found, it can be useful to rank them according to the nonfunctional properties of the constituent services, using a ranking algorithm such as the one presented in Section 7.4, adapted to aggregate the NFPs of multiple services in a composition, as discussed in Section 2.1.4.

For further details on the approximate reasoning and on possible heuristics for the composition algorithm, we refer the reader to [42, 43].

Chapter 8

Implementations

The preceding chapter of this thesis showed how semantic Web service descriptions can be used for automating service discovery and various other tasks involved in the use of Web services. In this chapter, we discuss various kinds of implementations that are directly pertinent to the evaluation of the languages proposed in Part II of this thesis, including an implementation of several semantic discovery techniques.

Section 8.1 introduces the SOA4All Studio, an example of a comprehensive Semantic Execution Environment. As a selection of the Studio's components, Section 8.2 discusses parsers for the languages proposed by this thesis, Section 8.3 describes a service registry for WSMO-Lite semantic descriptions, and Section 8.4 shows two service description editors.

8.1 Semantic Execution Environment

WSMO-Lite is intended to support automation of the use of Web services, thus the main implementation is in tools that realize automation algorithms such as those presented in the preceding chapter. The Semantic Execution Environment Technical Committee[1] (SEE TC) at OASIS, the leading standardization body for WS–∗ Web services, sees such tools as parts of a Semantic Execution Environment (SEE), which employs semantic technologies to support users in achieving their goals with Web services.

The draft Reference Architecture for Semantic Execution Environments [56] adapts the view of a Semantically-enabled Service Oriented Architecture (SESA) from [126], shown here in Figure 8.1. The main contribution of this thesis, the languages presented in Chapters 4–6, fits in the formal languages component at the base of the SEE. Chapter 7 shows algorithms that mainly support the discovery, adaptation and composition components.

WSMO-Lite is the cornerstone of a semantic execution environment called SOA4All Studio [66], developed through cooperation of several project partners in the research project SOA4All.[2] On top of a service registry (the storage component at the base of the SEE), it implements the components for discov-

[1] `http://oasis-open.org/committees/tc_home.php?wg_abbrev=semantic-ex`

[2] `http://soa4all.eu`, a prototype of the Studio is accessible at `http://coconut.tie.nl:8080/dashboard`

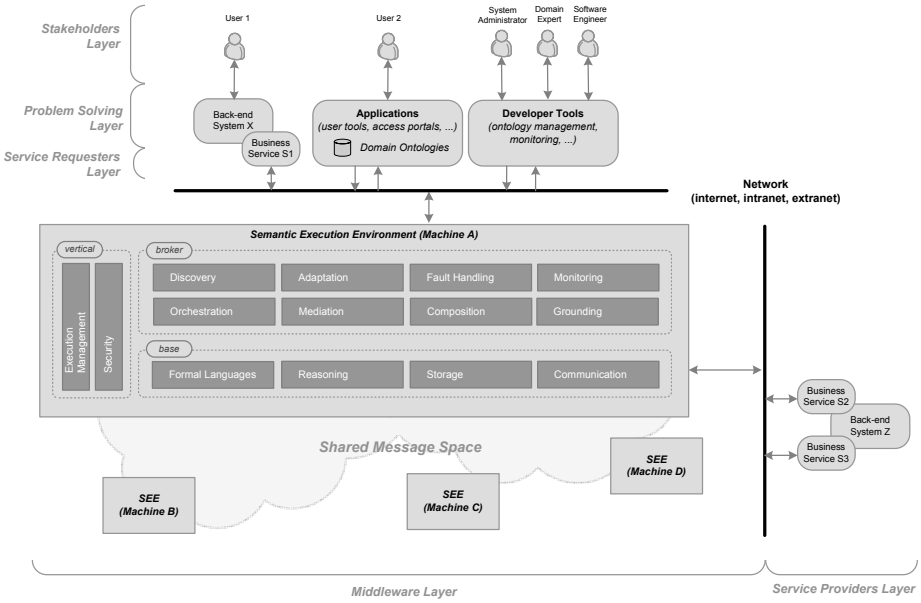


Figure 8.1: A global view of a Semantically-enabled Service Oriented Architecture, Figure 1 from [126]

ery, adaptation (ranking), composition and monitoring. The Studio provides a number of user and developer tools, including editors for semantic service descriptions, and discovery, ranking, assisted composition, monitoring and analysis tools. Figure 8.2, adopted from [66], shows the Studio as a front-end of the whole SOA4All platform.

In the following sections, we focus on several critical components and tools:

- Section 8.2 presents parsers that read semantic service descriptions in WSDL/SAWSDL or HTML/hRESTS and turn them into RDF data that follows the service model presented in Section 4.4 (page 62); the RDF form is then used for processing in other components and tools.
- Section 8.3 describes a service registry for WSMO-Lite semantic descriptions, along with its high-level discovery API.
- Section 8.4 shows two tools that support the process of creating semantic service descriptions, covering WS–∗ services on one side and RESTful APIs on the other.

Our direct contributions to the implementations: The author of this thesis has been directly involved in two of these implementations: the parser for HTML/hRESTS descriptions (wholly developed by the author), and the service registry (where the author mainly implemented the service discovery API and algorithms). The remainder of the implementations was developed by the author's research partners.

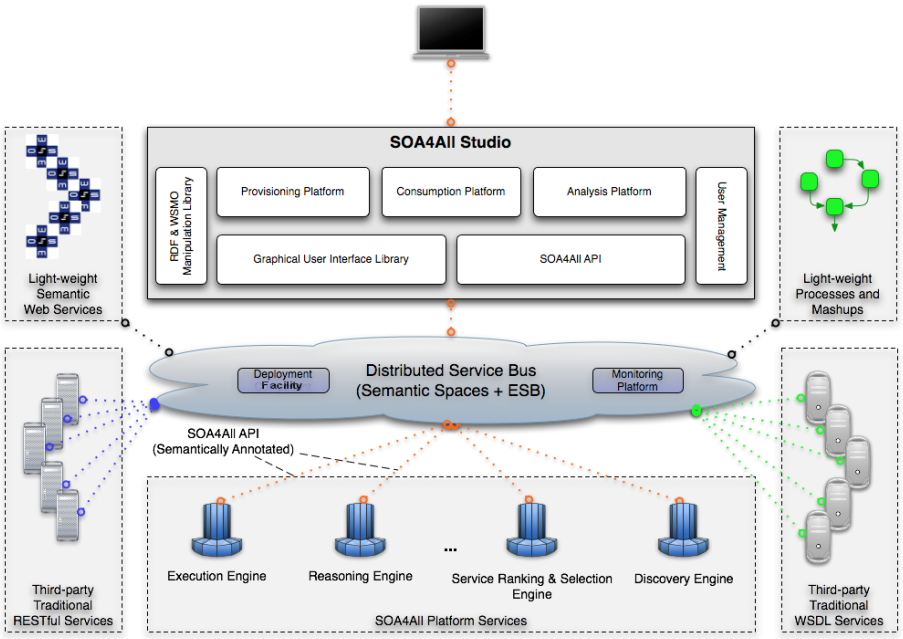


Figure 8.2: SOA4All Architecture, taken from [66]

8.2 Service Description Parsers

Our lightweight semantic Web service description languages build on underlying structured and semi-structured data formats, in particular XML (WSDL) and HTML (hRESTS). As we propose semantic processing to happen on RDF data, the underlying descriptions must be *parsed* into the RDF form. As discussed in Section 4.4, the parsed RDF graph should be treated as a read-only view on the underlying description.

The general role of a *parser* is to load a document as a stream of characters and to return a structured model of the information contained in the document. In our case, we need a parser for SAWSDL data in WSDL documents, and another for MicroWSMO/hRESTS data in HTML documents. The following two subsections detail these two parsers.

8.2.1 SAWSDL Parsers

In the context of WS–∗ languages, there are a number of pieces of software that can be called WSDL and SAWSDL parsers (e.g. WSDL4J[3] and SAWSDL4J[4], EasyWSDL and EasySAWSDL[5]); their purpose is to simplify programmatic manipulation of WSDL and SAWSDL documents. In the context of our work, though, we need a parser that reads SAWSDL documents and produces the RDF view along the service model from Section 4.4.

[3] `http://sourceforge.net/projects/wsdl4j/`
[4] `http://sawsdl4j.sourceforge.net/`
[5] Both available at `http://easywsdl.ow2.org/`

The SOA4All project has produced such a WSMO-Lite oriented SAWSDL parser, based on the EasySAWSDL library. The core functionality is a straightforward implementation of the mapping between the WSDL component model and the WSMO-Lite service model, defined in Table 5.2 (page 80). Importantly, to support all the semantic description deployment options (see Section 5.3), the parser should include in its output any RDF statements embedded in the source WSDL document.

Figure 5.2 (page 82) shows a consolidated example of a SAWSDL description and its mapping to RDF, performed by the SOA4All parser.

8.2.2 hRESTS/MicroWSMO Parser

As hRESTS and MicroWSMO are microformats, the mapping of hRESTS/MicroWSMO descriptions into RDF can be implemented through GRDDL [35], a mechanism for extracting RDF data from Web pages, particularly suitable for dealing with microformats. With GRDDL, the Web page is processed by one or more XSLT transformations that output RDF triples; the result is an RDF view on the content of the page.

We have provided a combined GRDDL XSLT transformation[6] that handles both hRESTS and its MicroWSMO annotations. This transformation can be used directly to translate hRESTS/MicroWSMO service descriptions into RDF, for instance when a description is submitted to a registry, as described in Section 8.3.

Furthermore, in accordance with GRDDL, an XHTML document that contains hRESTS (and MicroWSMO) data can itself link to the XSLT transformation from its header metadata:

```
<head profile="http://www.w3.org/2003/g/data-view">
  <link rel="transformation"
        href="http://cms-wg.sti2.org/TR/d12/v0.1/20081202/xslt/hrests.xslt" />
  ... further metadata, especially page title ...
</head>
```

This header enables Web browsers, crawlers and other tools to extract the RDF form of the service description data, even if the tools are not specifically aware of the hRESTS and MicroWSMO microformats. With this GRDDL header, hRESTS/MicroWSMO service descriptions can become part of the Web of linked data [11].

In Chapter 6, Listing 6.5 (page 105) shows the RDF data extracted from an example MicroWSMO description from Listing 6.4, using our GRDDL XSLT transformation.

8.3 Service Registry, Discovery API

The SOA4All Studio is backed by iServe [89], a semantic service description registry[7] built to support WSMO-Lite. Beside using the semantic RDF model internally, iServe also publishes all its data in an open manner according to the Linked Data principles [13, 88].

[6] At `http://cms-wg.sti2.org/TR/d12/v0.1/20081202/xslt/hrests.xslt`

[7] Accessible at `http://iserve.kmi.open.ac.uk/`

The registry is fully based on the WSMO-Lite service model, supporting service descriptions in WSDL/SAWSDL and in hRESTS/MicroWSMO. Support for the RDFa syntax alternative to hRESTS, described in Section 6.2.4, is planned. Additionally, iServe is also capable of importing OWL-S service descriptions and treating them as WSMO-Lite descriptions. This importing capability is particularly used for evaluation with the OWL-S retrieval test collection[8], as shown in Chapter 9.

iServe provides a browser GUI for human usage, and a set of RESTful APIs for programmatic access. The browser GUI, a Web-browser-based application, is shown in Figure 8.3, with service categorizations on the left, a list of services in the top-right part, and the details of a selected service in the bottom-right part. It supports simple browsing and searching of services, and direct queries with SPARQL [114].

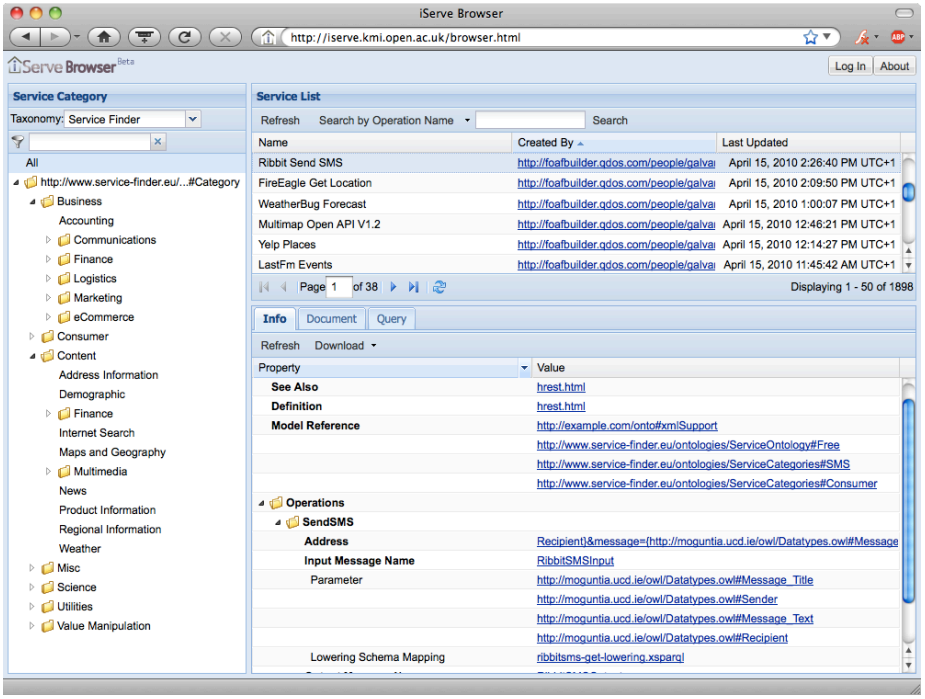


Figure 8.3: A screenshot of the iServe browser GUI, taken from [65]

The RESTful APIs provide support for accessing and submitting service annotations and service documentation, and high-level discovery capabilities. The following subsection provides further details on the iServe discovery API.

8.3.1 iServe discovery API

In line with the general RESTfulness of the registry, discovery functionality is made available through a Web API; as such, it deserves a more detailed description here. Currently, iServe implements three types of discovery: i) discovery with functionality classifications, ii) matching of input and output signatures,

[8] `http://projects.semwebcentral.org/projects/owls-tc/`

and iii) statistical similarity-based approximate matchmaking. The first two types only take into account direct logical relationships between semantic concepts, whereas the third uses information retrieval techniques that avoid strict logical false negatives.

The functionality-classification-based discovery implements the algorithm described in Section 7.2.1, using functionality taxonomies in RDFS and on SKOS. The statistical text-similarity matchmaker is based on iMatcher [59]. Chapter 7 only describes the first type of discovery because the adaptation of the other two to WSMO-Lite is straightforward.

The discovery API offered by iServe is structured as follows:

`/data/disco/func-rdfs?class=`C_1`&class=`C_2`&...`
: uses RDFS functional classification annotations and returns those services that are related to all the functional categories C_i (which are URIs of RDFS classes).

`/data/disco/func-skos?concept=`C_1`&concept=`C_2`&...`
: same as above, using SKOS concepts instead of RDFS classes.

`/data/disco/io-rdfs?i=`C_1^I`&i=`C_2^I`&o=`C_1^O`&...`
: uses ontology annotations of inputs and outputs and returns services for which the client has suitable input data (C_i^I) and which provide the outputs requested by the client (C_i^O).

`/data/disco/imatch?strategy=levenshtein&label=`L
: returns all services ranked according to string similarity of the service label with the string L.

In the spirit of using Web standards, the API represents discovery results as Atom feeds [4], with the entries representing matching services, sorted by matching degree. The Atom feed format was chosen for several reasons: it is a standard generic container format with wide support in software libraries and products, and it defines strong metadata properties (such as titles, identities and update times) that make feed readers a meaningful standalone software category. With Atom, iServe discovery queries can, for example, be syndicated and manipulated in generic systems such as Yahoo! Pipes[9], or end users can watch for new interesting services by registering iServe discovery queries in their feed readers.

The common representation of discovery results as Atom feeds can be exploited for supporting arbitrary combinations of discovery approaches through list operations on the results of separate discovery queries. iServe includes three Atom feed combinators:

1. Union: the resulting feed contains the entries of all the constituent feeds. For discovery queries, the union of results is equivalent to the *or* (disjunction) operator: a service is returned if it matches any of the given queries.

2. Intersection: results in a feed with only the entries that are present in all the constituent feeds. This is equivalent to the *and* (conjunction) operator for discovery queries.

[9] `http://pipes.yahoo.com`

3. Subtraction: results in a feed with the entries of the first feed that are not in any other provided feed. In discovery, this enables the *and not* operator: it can return services that match one query but not another.

All these combinators are part of iServe's RESTful API, and they take feed URIs as parameters. To illustrate the use of the discovery API, including the Atom combinators, the following URI would discover proximity search services that take as inputs a raw address (*proximity search* and *raw address* are terms in an ontology used to annotate a set of geography services present in iServe):

```
http://iserve.kmi.open.ac.uk/data/atom/intersection?
  f=/data/disco/func-rdfs?class=
    http://iserve.kmi.open.ac.uk/2010/05/s3eval/func.rdfs%2523ProximitySearch
 &f=/data/disco/io-rdfs?i=
    http://iserve.kmi.open.ac.uk/2010/05/s3eval/data.rdfs%2523RawAddress
```

The example contains altogether five URIs: the location of the intersection combinator, the location of the RDFS functional classification discovery service (note that the URI is relative to the atom combinator URI), the identifier of a class of proximity search services, the location of the RDFS input/output matchmaker and the identifier of the concept of a raw address. Note that the nesting of URIs requires careful percent-encoding of special characters: for instance the hash sign '#' is encoded as '%23' and the percent-sign is encoded again as '%25' because the URI with the hash sign is nested in two others.

The separation of the individual discovery algorithms from the mechanism by which they are combined supports easy extensibility: new discovery algorithms can be added to iServe independently (as plug-ins) and then usefully combined with the algorithms that are already there.

8.4 Semantic Description Editors

To ease the acquisition of semantic service descriptions, SOA4All Studio supports service providers (or interested third parties) in creating semantic descriptions for Web services, both WS–∗ and RESTful.

Describing Web services semantically is a knowledge-intensive task that we cannot fully automate, but tools can support it in several ways: i) by suggesting appropriate ontologies, ii) by verifying some consistency and completeness criteria (see Sections 5.4 and 6.7), and also iii) by guiding a user through the steps of a semantic service description methodology. For instance, the OASIS SEE Technical Committee is considering a proposed methodology called *MEMOS* (A Methodology for Modeling Services [57]), which should be directly applicable to WSMO-Lite as well as other SWS frameworks.

The SOA4All Studio contains two separate semantic description editors, called SWEET and SOUR:

- **SWEET** ("Semantic Web sErvices Editing Tool" [71]) is an editor for hRESTS and MicroWSMO. It is a Web-browser-based application[10] that supports the creation of semantic descriptions of Web APIs. SWEET takes as input an HTML Web page describing a Web API, and it allows

[10]SWEET is accessible at `http://sweet.kmi.open.ac.uk/`

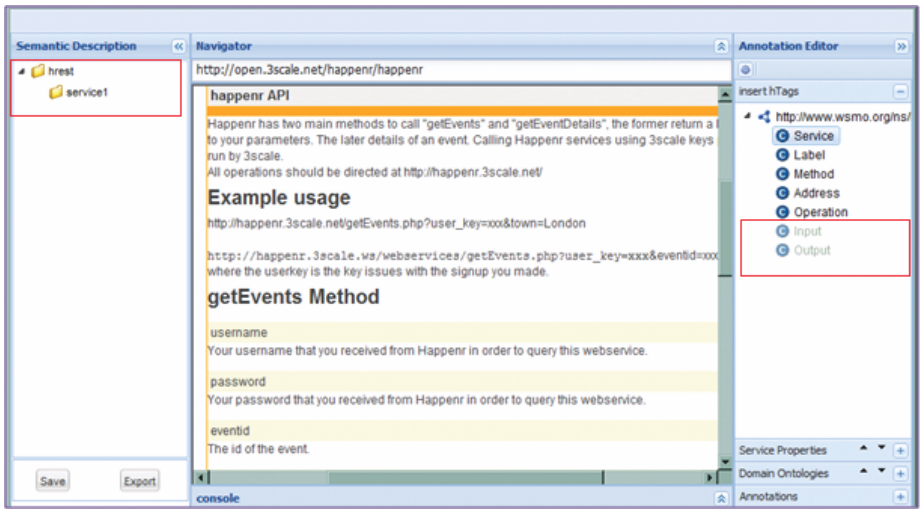


Figure 8.4: The user interface of SWEET (taken from [71])

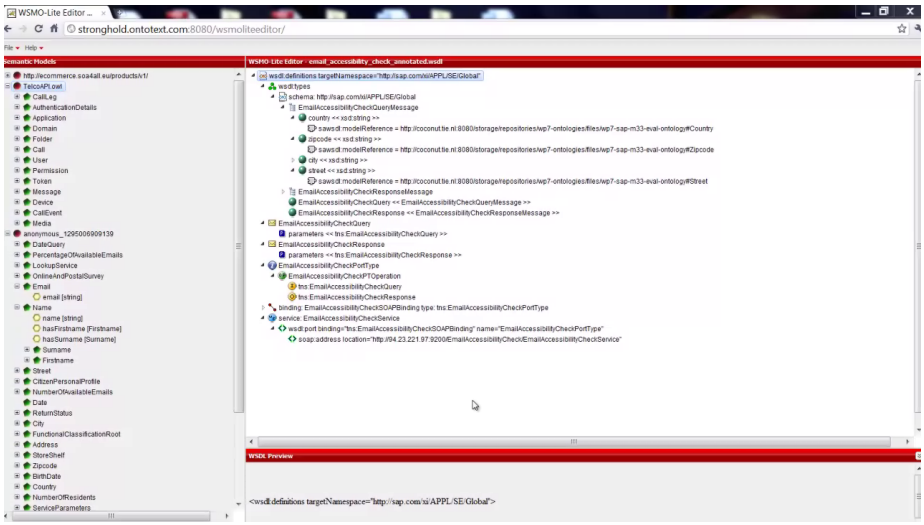


Figure 8.5: The user interface of SOUR

the user to mark up the service structure (with hRESTS) and to annotate it with semantic information (with MicroWSMO).

SWEET is shown in Figure 8.4. Its user interface has three main parts: the central panel contains the HTML description of a selected Web API; the right-hand-side panel shows the allowed and recommended annotations, both for the hRESTS structure (shown) and for semantic properties; and the left-hand-side panel visualizes the current service model structure of the documentation.

SWEET assists users in locating appropriate annotations from among the existing ontological data on the Web, and it fosters ontology reuse, by integrating the ontology search engine Watson[11].

- **SOUR** [73] is an semantic annotation editor for WSDL and SAWSDL. It is a browser application[12] that supports adding (and manipulating) semantic annotations in WSDL and XML Schema descriptions, according to the distribution of the kinds of semantics defined in Table 5.1.

  SOUR is shown in Figure 8.5. Its user interface also has three main parts: the main panel on the right-hand side displays the hierarchical structure of a WSDL document, where the user can see and manipulate the semantic annotations; the bottom "WSDL Preview" panel shows the XML source code of the selected WSDL element, highlighting any semantic annotations; and finally the "Semantic Models" panel on the left shows loaded ontologies from which the user can select semantic elements to annotate the WSDL document.

The semantic description editors are complemented by a Grounding Editor [110], also available in the in the SOA4All Studio. The grounding editor has a drag-and-drop user interface that supports the creation of mappings between XML Schemas and ontologies, from which it can generate the appropriate XSLT lifting and lowering transformations for XML data.

In summary, Web Service and API providers or interested third parties can use SWEET and SOUR as a user-friendly way of preparing semantic service descriptions, enabling tool support for discovery and so on. While SOUR presents a common view of WSDL documents for annotation with SAWSDL, SWEET starts with Web API documentation in HTML and supports annotation with hRESTS and MicroWSMO. Both tools have integrated support for submitting to the iServe registry: when the user completes the semantic annotation of the HTML or WSDL description, the result can be published in iServe, or it can be saved locally.

[11] Watson Semantic Web Search, `http://watson.kmi.open.ac.uk`

[12] SOUR is accessible at `http://stronghold.ontotext.com:8080/wsmoliteeditor/`

Chapter 9

Evaluation

The main contribution of this thesis comprises the languages defined in Part II. In the preceding chapters, we have defined several SWS automation algorithms that use these languages, and we have detailed numerous tools and libraries that constitute implementations of the languages. Here, we evaluate the languages from several angles, aiming to demonstrate that they are viable, that they carry no adverse effects on system performance, and how they compare to main other existing SWS approaches.

In Section 9.1, we describe the methodology that guided our evaluation efforts; Section 9.2 discusses the viability of the languages, Section 9.3 focuses on their performance, and Section 9.4 compares them to the state of the art in Semantic Web Services.

9.1 Evaluation Methodology

Throughout this thesis, we have stressed the need for lightweight semantics for Web services, especially including RESTful APIs that have seen limited attention from Web service automation researchers. The languages we have presented directly address the need: WSMO-Lite covers service semantics, and hREST-S/MicroWSMO support RESTful APIs; these languages are the primary contribution of this thesis. In this chapter, we evaluate this contribution from three angles: viability (will it work?), performance (will it work effectively and efficiently?), and by comparing it to the state of the art (does it have advantages/disadvantages over other approaches?).

In effect, we are checking here our main **success criteria**, as stated in Chapter 1: i) that our service semantics ontology is sufficiently expressive to support the desired degree of automation (comprising service discovery, selection and composition), and ii) that the automation works equally well with RESTful services as it does with WS–∗ services.

To demonstrate viability (or fitness for purpose) of WSMO-Lite, we have adapted to it a number of automation algorithms, we have discussed implementations that are now relatively mature, and we have performed a practical exercise that included annotating a set of real-world services, and comparing of WSMO-Lite discovery in iServe with a set of discovery tools for other approaches (discussed below). All these points are put together in Section 9.2.

To evaluate the performance of WSMO-Lite, we have used a common goal-based evaluation method of comparing various systems on a benchmark task, where we have focused on discovery, a heavily-researched part of SWS automation that even runs an annual public contest. The results are shown and analyzed in Section 9.3.

Finally, we have also performed a goal-free evaluation of the WSMO-Lite languages by comparing them to the main existing SWS frameworks, OWL-S and WSMO, and to WSDL-S, the visionary lightweight framework that led to SAWSDL. We have uncovered some potential advantages and some potential disadvantages of WSMO-Lite, as discussed in Section 9.4.

Ultimately, the goals for the lightweight languages presented in this thesis are: i) to stimulate convergence of existing SWS technologies on top of SAWSDL (and to include RESTful APIs), and ii) eventually to make Semantic Web Service technologies easier to use and thus to foster their adoption. Our success on these goals can only be evaluated in time; for now, we can quote the W3C, which stated in the Team Comment to the WSMO-Lite submission [27], that "it is a useful addition to SAWSDL for annotations of existing services and the combination of both techniques can certainly be applied to a large number of semantic Web services use cases."

As of this writing, we know of two collections of WSMO-Lite service descriptions: the semantic service registry iServe knows about over 2000 service descriptions[1], and the WSMO-Lite Test Collection [17] contains over 1000 service descriptions, adopted from the test collections OWLS-TC and SAWSDL-TC.[2]

9.2 Fit-for-Purpose Evaluation

The viability (fitness for purpose) of the languages is demonstrated in a number of places throughout this thesis; let us summarize it here in the order of the life cycle of semantic service descriptions: descriptions are first created, then stored (published) somewhere, and finally processed for discovery and other automation purposes.

Creation of service descriptions: a major principle guiding our work has been to keep the languages lightweight, in order to ease the creation and authoring of WSMO-Lite descriptions. In Section 8.4, we have described two published editor (SWEET, SOUR) tools that support hRESTS/MicroWSMO and WSDL/SAWSDL, respectively.

Authoring WSDL descriptions is a well-known and settled area, with existing tools[3] ranging from syntax-highlighting XML editors to graphical editors based on the structure of WSDL. The tool SOUR provides simple drag-and-drop editing capabilities for extending WSDL files with SAWSDL annotations.

In contrast to the maturity of WSDL editing support, annotating the HTML documentation of RESTful services is still a research area with much space for

[1] From `http://iserve-dev.kmi.open.ac.uk/iserve/` in October 2012.

[2] `http://semwebcentral.org/projects/owls-tc/` and `http://semwebcentral.org/projects/sawsdl-tc/`

[3] To point out a few: `http://www.liquid-technologies.com/wsdl-editor.aspx`, `http://wiki.eclipse.org/index.php/Introduction_to_the_WSDL_Editor`, `http://www.oxygenxml.com/wsdl_editor.html`

further exploration. For example, some RESTful APIs have documentation that is not a close fit to the hRESTS structure of a list of operations with their separate inputs and outputs, in which case an editor tool such as SWEET will likely need to use the flexible RDFa form of hRESTS and MicroWSMO discussed in Section 6.2.4.

The development and use of the mentioned two editor tools, free of significant issues against our languages, has shown that creation of WSMO-Lite descriptions can be effectively supported for both major kinds of Web services. As discussed in the next section, for performance evaluation we have also annotated 50 existing Web services and achieved good discovery results with minimal annotation effort; this also indicates that creation of WSMO-Lite descriptions is well possible.

Publishing service descriptions: for the storage and publishing of WSMO-Lite service descriptions, Section 8.3 describes the public registry iServe. As of August 2011, the registry contains more than 2000 service descriptions.

iServe is used as the service registry in the EU projects SOA4All and NoTube [147]. Below, in Section 9.3, we show a performance evaluation of WSMO-Lite discovery, as implemented by iServe.

Processing service descriptions: Chapter 7 describes several semantic service automation algorithms directly adapted to the WSMO-Lite service model and semantics. The discovery algorithms in particular are implemented in the iServe registry and they are evaluated in more detail below, showing that they perform on par with state-of-the-art service matchmakers for other SWS frameworks. This gives us confidence that further SWS algorithms can be adapted to WSMO-Lite without loss of functionality, and that WSMO-Lite is therefore a viable SWS description approach.

We do not intend to imply that the adapted algorithms are the most efficient or the most user-friendly; they are a selection of common and proven approaches, on which we can easily demonstrate how such algorithms can be adapted to WSMO-Lite. By using known and tested approaches, we show how SWS research may converge on a technology like WSMO-Lite that is close to Web Services practitioners. Importantly, when adapting the algorithms to WSMO-Lite, we have not encountered significant challenges or obstacles, beyond the expected vocabulary alignments. In other words, WSMO-Lite is not leaving the existing body of research on SWS automation algorithms behind.

Note that WSMO-Lite is the only SWS approach that we know which supports offer discovery. The offer discovery algorithm shown in Section 7.3 uses only information semantics and operation safety annotations (not supported by other frameworks), and neither of these types of semantics is specific to offer discovery.

Since one of the main principles behind WSMO-Lite is modularity, it is likely that some service descriptions will only include those pieces of service semantics that are necessary to run the automation algorithms for which the descriptions were created — for example, we envision that Web service deployments will be likely to start by specifying only the functional classifications of services. Therefore, adapting different existing SWS automation algorithms and systems to WSMO-Lite cannot immediately result in complete interoperability

with diverse sources of semantic service descriptions. To illustrate, an input/output/precondition/effect matching algorithm for discovery cannot process service descriptions that only specify functional classification of services.

Still, adapting existing SWS algorithms and systems to WSMO-Lite gives researchers a common ground and a shared vocabulary, making it easier to identify and work out the differences between various approaches to the same SWS automation problem (such as discovery). WSMO-Lite has two effects on terminology that have to be accepted in an adapted algorithm: first, WSMO-Lite takes the SAWSDL point of view building bottom-up on the underlying technical descriptions, as opposed to top-down from a semantic model; and second, WSMO-Lite distinguishes the four types of semantics (functional, nonfunctional, behavioral and information) and it specifies the five RDFS classes to express them.

9.3 Performance Evaluation

In this section, we attempt to evaluate the languages presented by this thesis from the angle of performance.

Directly, both SAWSDL and hRESTS/MicroWSMO clearly have negligible overhead on document size and parsing complexity of service descriptions. Building on RDFS, the entry-level knowledge modeling standard for the Semantic Web, WSMO-Lite also does not impose any advanced reasoning requirements; however, RDFS is only the base line — WSMO-Lite does not restrict SWS algorithms from employing advanced expressivity and reasoning. In summary, the languages presented in this thesis present no significant overhead to any direct performance measures.

As an indirect performance evaluation, we can measure the effect of WSMO-Lite and of the underlying service description annotation languages on the performance of the SWS algorithms adapted to WSMO-Lite, especially when compared to their "native" versions. Here, we perform such evaluation on WSMO-Lite discovery, as implemented in iServe. First, we compare the performance of iServe's statistical and logics-based discovery with the performance of OWLS-iMatcher [59], a native statistical OWL-S matchmaker, showing good results across the board, and demonstrating that WSMO-Lite presents no processing overhead. Second, we test the logics-based discovery of iServe on a set of geography services, with results comparable to other state-of-the-art matchmaking solutions, with surprisingly little annotation effort.

Our discovery evaluation is performed within the framework of the 2009 S3 Contest on Semantic Service Selection,[4] which is the reference contest for evaluating service matchmakers. The S3 contest is structured in 3 different tracks. The first two tracks are specific for OWL-S and SAWSDL respectively, whereas the third track does not impose any formalism — participants are allowed to use any approach. We present here an evaluation of iServe WSMO-Lite-based discovery in all the three tracks of the contest.[5]

Firstly, we compared iServe's statistical matchmaker directly with its original "native" version, the OWLS-iMatcher (a high-scoring participant in the

[4] http://www-ags.dfki.uni-sb.de/~klusch/s3/

[5] Our evaluation was performed after the conclusion of the contest, using the publicly available test collections and results.

S3 context, Track 1). The comparison was carried out on the OWL-S test collection,[6] which was imported into the form of WSMO-Lite in iSERVE. Figure 9.1(a) shows on four similarity strategies that the discovery results are virtually the same (any local deviations can be explained by differences in the underlying ordering of services with the same similarity degree). In other words, the iMATCHER similarity matchmaking is not harmed by the translation of the service descriptions from OWL-S to WSMO-Lite. We have performed the same comparison with the SAWSDL iMATCHER as well (S3 contest, Track 2), with the same results — treating SAWSDL-TC service descriptions as WSMO-Lite ones does not harm iMATCHER similarity matching either.

Further improvements were brought to the discovery performance by combining similarity strategies with logics-based matchmaking algorithms. Effectively, the combination makes iSERVE a hybrid matchmaker (as discussed in [61]) and Figure 9.1(b) confirms that the hybrid approach generally performs better than pure logical or pure statistic approaches. Again, while iSERVE discovery is implemented over WSMO-Lite, the tests were done on the OWL-S (shown) and SAWSDL test collections imported into iSERVE.

The third track of the S3 contest[7] uses a subset of the Jena Geography Dataset (JGD) that consists of 50 Web services, with approximately half of them being Web APIs. The contestants in this track are free to use any service description and matchmaking approaches. The contestants first receive the services and create their descriptions, and only then they receive the queries, formulated in plain English, which the contestants express as queries in their discovery systems. The discovery results are then compared against up-to-now-secret relevance judgments (assigning relevant services to each query). In our evaluation, we proceeded exactly along the same steps.

Because of the lack of matchmakers able to deal with Web APIs, the contestants in this track typically use artificially-created "fake" WSDL descriptions to represent the Web APIs. We carried out the third track of the evaluation using the HTML documentation of 16 of these Web APIs annotated with hRESTS and MicroWSMO, in lieu of their fake WSDLs;[8] the use of native Web API descriptions (as opposed to the fake WSDLs) makes no difference to the discovery results over WSMO-Lite.

For matchmaking, we have only used logics-based discovery, combining functional classifications with straightforward input/output matching; the results indicate that iSERVE was able to achieve a performance level comparable to that of the best-in-class Web service matchmakers.

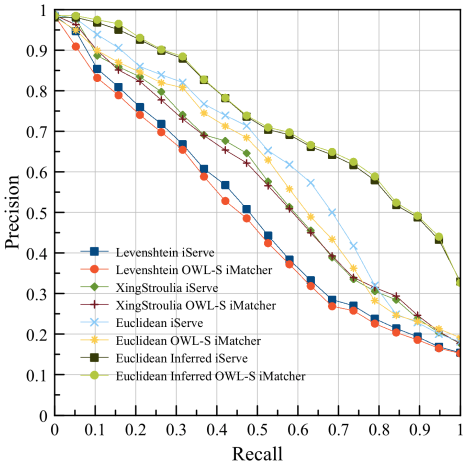
The graph actually indicates that iSERVE is the best, but due to the low number of services and the nature of the contest track, the performance is affected significantly by the quality of the hand-crafted service annotations and discovery queries. Before executing this evaluation, we had expected to perform worse with our first set of service annotations (which were intentionally created with minimum effort, in the spirit of lightweight descriptions) and then to refine the service descriptions and the discovery queries to achieve acceptable performance, measuring any extra refinement effort. However, the initial semantic descriptions and discovery queries led to the good results shown, hinting that

[6] `http://projects.semwebcentral.org/projects/owls-tc/`

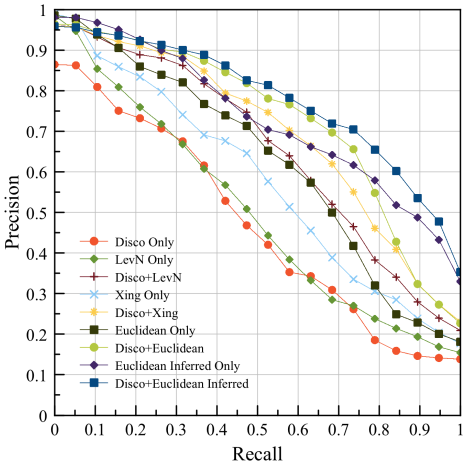
[7] `http://fusion.cs.uni-jena.de/professur/jgdeval/`

[8] The annotated service descriptions can be found at `http://iserve.kmi.open.ac.uk/2010/05/s3eval/services/`

(a) Comparing iMatcher integrated in iServe, working over OWL-S TC4 imported into WSMO-Lite, with the original OWLS-iMatcher



(b) Performance of various combinations of iServe discovery over OWL-S TC4



(c) Comparing iServe to participants of the third track of the S3 contest, results macro-averaged for Binary_7 ranking[7]

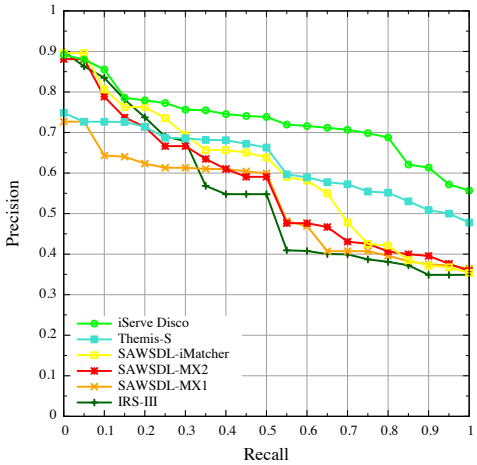


Figure 9.1: Precision/recall graphs of WSMO-Lite discovery evaluated in iServe

the lightweight service descriptions really require minimal effort with a good effect. Note that we have not performed a formal comparison of the efforts of creating service annotations in WSMO-Lite and the other S3 contestants; evaluating the effort required to create service descriptions and discovery queries for various matchmaking mechanisms is an open research topic in this area.[9]

Our evaluation not only shows that the logics-based discovery algorithms adapted to WSMO-Lite give competitive results, but it also establishes iServe as the first system (that we know of) which can transparently deal both with WS–∗ Web services and with RESTful APIs. It is also the first system that provides any advanced logics-based discovery capabilities for RESTful APIs at all. Thanks to WSMO-Lite, iServe is the only single system that can currently tackle all the three tracks of the S3 contest, whereas for example iMatcher was present in the context in two separate variants to support OWL-S and SAWSDL.

In effect, the evaluation shows that the languages proposed by this thesis support discovery performance comparable to the best existing systems; this gives us a good level of confidence that WSMO-Lite will work effectively and efficiently in other Web service automation tasks as well.

9.4 Comparison to the State of the Art

In the following two subsections, we compare WSMO-Lite with selected existing SWS approaches, namely OWL-S and WSMO, the two most mature and comprehensive SWS frameworks, and WSDL-S, the lightweight proposal that spawned SAWSDL.

9.4.1 Comparing WSMO-Lite to WSMO and OWL-S

The main differences between WSMO-Lite and the two major preceding frameworks lie especially in the scope of the approaches and in their relation to standards, but there are other notable differences. All of the differences are detailed in the following paragraphs, and summarized in Table 9.1. Where OWL-S differs from WSMO, WSMO-Lite happens to be closer to OWL-S; therefore we generally start by comparing WSMO-Lite to WSMO and then add a remark about OWL-S.

The foremost difference between WSMO-Lite and WSMO is that of **scope**: where WSMO is a comprehensive framework that covers all the areas of semantic descriptions around services (including user goals, ontologies and mediators), OWL-S and WSMO-Lite have a narrower scope that deals only with service descriptions. Both OWL-S and WSMO-Lite use RDFS and OWL for ontologies; and both delegate goals and mediators to the infrastructure.

Further, where WSMO supports detailed description of both the outer and the inner behavior of services in choreography and orchestration processes, both OWL-S and WSMO-Lite only describe the outward behavior of services; OWL-S uses an explicit process model, while WSMO-Lite captures the functionalities

[9] For example, there is an ongoing (albeit dormant at the time of this writing) SWS Challenge effort with such evaluation as its goal. Information about the SWS Challenge is available at `http://sws-challenge.org/wiki/index.php/Main_Page`

of the operations, leaving the choreography process implicit. In this regard, the scope of WSMO is deeper.

A major difference between WSMO, OWL-S and WSMO-Lite, one which may be affecting the adoption of these semantic technologies in service-oriented environments, lies in the **relation** of the technologies **to existing standards**.

On the side of the standards for Web services, WSMO-Lite builds directly on WSDL and SAWSDL, while both WSMO and OWL-S remain independent of Web service technologies, shielded by a layer called "grounding" that provides the necessary links to WSDL.

On the side of Semantic Web standards RDF and OWL, both WSMO-Lite and OWL-S use them directly, while WSMO provides its own ontology language WSML, with a mapping to the W3C Recommendations. Because until recently there was no Semantic Web standard for logical expressions, both OWL-S and WSMO-Lite have relied on third-party specifications, such as SWRL [44] or WSML [136]. Now that the W3C has standardized the Rule Interchange Format RIF [102], both OWL-S and WSMO-Lite may adopt this new standard, and WSMO may specify a mapping to WSML.

In both areas of standardization, WSMO-Lite is positioned close to the formal standards, and as such, it can easier be adopted in environments that already use the standard technologies.

The WSMO framework comes with a special **syntax**, described as "abstract" and "human-readable", and it also provides two further exchange formats, in XML and in RDF. The human-readable syntax is especially intended for advanced users who can author logical statements and service descriptions in a source form; other users are expected to use authoring tools. In contrast, both OWL-S and WSMO-Lite use RDF as their only representation format; the lack of a friendlier syntax may be seen as a disadvantage.

Other notable differences between WSMO-Lite and the two preceding frameworks lie in support for using functionality classifications, and in support for RESTful services.

As part of functional and behavioral semantics, WSMO-Lite supports the use of **functionality classifications**. WSMO focuses on expressing service functionality and behavior through logical expressions, therefore it has no explicit support for categorizing service and operation functionalities. In OWL-S, service classification used to be supported (only on services, not on operations); in

	WSMO	**OWL-S**	**WSMO-Lite**
Scope			
breadth	broad – 4 top-level areas	narrow – only services	
depth	deeper	shallower – no orchestration	
Relation to standards			
Web services	grounding in WSDL		directly on SAWSDL
Semantic Web	supplements RDF/OWL/RIF	use RDF/OWL/RIF	
syntax	"human-readable", RDF, XML	RDF	
Other differences			
svc. classification	not supported	deprecated	supported
RESTful services	not supported, but grounding possible		directly supported

Table 9.1: Summary of comparison of WSMO-Lite and WSMO

the newest version this support is deprecated and delegated to domain-specific extensions.

And finally, with the proposed hRESTS/MicroWSMO microformats, WSMO-Lite **supports** the semantic description of **RESTful services**, which are an increasingly important part of the Web. In contrast, both OWL-S and WSMO would need a grounding specification for RESTful services, and we know of no efforts in this direction.

In summary, by keeping a tight scope and a close relation to standards, we are positioning WSMO-Lite as a common base for convergence and adoption of SWS technologies. WSMO-Lite's direct support for RESTful services is a major advantage over preceding SWS approaches.

9.4.2 Comparing WSMO-Lite to WSDL-S

WSDL-S is a technology that was developed as an extension of WSDL, in order to bring the semantic descriptions closer to the underlying Web services technologies. Along with a generic modelReference construct that became the cornerstone of SAWSDL, WSDL-S also defined constructs for operation preconditions and effects and for WSDL interface categorization, neither of which were carried over into SAWSDL but are present in WSMO-Lite.

To facilitate semantic automation, WSDL-S allows expressing preconditions and effects on WSDL operations. WSMO-Lite also supports preconditions and effects, which are treated as functional annotations, supported both on operations, as part of the behavioral semantics of a service, and on the service itself, as part of its functional semantics. Describing high-level preconditions and effects on the service as a whole (supported in WSMO-Lite but not in WSDL-S) is useful for coarse-grained service discovery and composition, without delving into the details of the service's operations. In this way, the discovery or composition process may be able to overlook formal incompatibilities that SWS automation couldn't resolve, but which can be overcome through process mediation devised by a human engineer.

For coarse-grained service discovery, WSDL-S specifies a construct for attaching categorization information to WSDL interfaces. WSMO-Lite also supports categorizations, treating them as functionality classifications, allowed both on the service (interface) as functional semantics, and on the service's operations, where functional categories serve as parts of behavioral semantics of the service. Where WSDL-S expects the coarse-grained approach of categorizations to be useful only on the level of whole services, WSMO-Lite can also handle categories applicable to operations; for instance wsdlx:SafeInteraction is a category of operations that are *safe* for invocation, as defined in the architecture of the Web, and discussed here especially in Section 6.5.

Effectively, WSMO-Lite embraces all WSDL-S constructs, generalizing their use as part of the functional and behavioral semantics of Web services.

Chapter 10

Conclusions and Future Work

In this final chapter, we present a concise summary of our work on lightweight semantic Web service automation, and we lay down some plans for future work.

The Semantic Web of data is starting to be taken seriously outside the research community, as demonstrated for instance by the increasing numbers of data providers in the Linked Data project [13], especially including open government data sources.[1] However, the vision of the Semantic Web is not limited to data interoperability; it has also always included processing services, as implied in [12] and further discussed in [39].

Research on Semantic Web Services strives for automation of using and combining Web services, similarly to how Semantic Web research focuses on using and combining data. Numerous approaches to semantic Web service automation have been proposed. However, SWS research has been fragmented and detached from the lower-level WS–∗ specifications, which are likely some of the reasons for its limited adoption in industrial settings. The W3C has started consolidation of SWS approaches through standardization, with SAWSDL being the first step on this way, albeit a small one. SAWSDL directly addresses the issue of limited adoption by putting semantics close to the accepted standard Web services description language WSDL. But SAWSDL does not specify any actual service semantics.

This thesis defines **WSMO-Lite**, an ontology for service semantics that fits directly into SAWSDL annotations, covering functional, nonfunctional, behavioral and information semantics of Web services. To demonstrate the viability of this ontology, we have adapted to WSMO-Lite **several SWS automation algorithms**. WSMO-Lite is intentionally lightweight, in order to smoothen the learning curve for adopters of SWS technologies.

In addition to SAWSDL-based support of WS–∗ services, WSMO-Lite also supports **RESTful services**, which have so far been overlooked by SWS research. RESTful services are becoming an increasingly important component of Web applications. There is no widely accepted machine-oriented description language for RESTful services, therefore this thesis also proposes two microformats, **hRESTS** and **MicroWSMO**, which mirror WSDL and SAWSDL on top

[1] See `http://data.gov` and `http://data.gov.uk` as examples.

of human-oriented HTML documentation of RESTful services. With a minimal semantic service model that is an abstraction of WSDL and hRESTS, RESTful services can seamlessly be included in semantic processing with WSMO-Lite. Such seamless integration of RESTful and WS–∗ Web services will especially gain importance as the popularity of RESTful services increases in enterprise environments that have traditionally favored WS–∗ technologies.

To summarize the intended impact of WSMO-Lite:

- Before WSMO-Lite, Web service technologies (WSDL, RESTful) and SWS approaches were quite disconnected, with little adoption of semantics for services.
- WSMO-Lite brings a unifying approach for semantics of both WS–∗ and RESTful services. While lightweight, it supports powerful automation algorithms.
- WSMO-Lite simplifies the creation of semantic descriptions of Web services, and thus aims to increase the usage of Web services.
- Web service users and developers should benefit most from WSMO-Lite: the users can employ semantic technologies for dealing with services, and service developers and providers can easier describe their services and thus support their clients.

WSMO-Lite was submitted to the W3C for consideration towards standardization, and acknowledged as a *Member Submission* [27]. The W3C Team Comment on the submission stated that it "is a useful addition to SAWSDL for annotations of existing services and the combination of both techniques can certainly be applied to a large number of semantic Web services use cases."

10.1 Future work

The main task for future work is fostering adoption of SAWSDL and WSMO-Lite in industrial WS–∗ settings, and adoption of hRESTS and MicroWSMO in RESTful service-oriented systems.

Adoption is related to standardization, especially in the standards-heavy environment of service-oriented computing. We plan further efforts for community standardization of the two RESTful service description microformats; especially hRESTS, which is independent of semantics and can provide well-understood benefits of machine-readable interface definitions for application development. On top of the already-standard SAWSDL and the two microformats, standardization of SWS approaches can continue in the peace-meal fashion taken with SAWSDL.

Adoption is further linked to the existence (or lack) of domain ontologies — an enterprise will more likely consider annotating their services with semantics if there are established ontologies that cover those services. This is a chicken-and-egg problem, in that ontologies cannot become established before they gain some adoption, but they are not likely to be adopted if they aren't perceived as established. Therefore, there is a need for research efforts that will propose and evaluate domain ontologies.

Among other tasks for future work are the parts that we have left out of scope of this thesis: the creation of semantic descriptions, goal modeling, automated service invocation.

For creating semantic descriptions of existing services, there is potential for applications of process and data mining technologies, natural language processing. Similarly, new methodologies for creating services can include steps that would document the new services with semantic technologies, possibly including some model verification.

Goal modeling would become important in integrated semantic execution environments (SEEs). The algorithms adopted to WSMO-Lite in Chapter 7 all have requirements on the information in client goals, but we do not attempt to propose a unified model for client goals that would cover all these requirements.

Our work contains only basic support for semantics-driven automatic invocation of Web services, through programmatic lifting and lowering transformations that work on the whole request and response messages. There is certainly space for declarative approaches to lifting and lowering, especially if the inputs and outputs are described with fine granularity both on the semantic level and on the level of on-the-wire messaging. Our coarse-grained approach does not make lifting and lowering transformations reusable — it is uncommon that two services with similar semantics also share message schemas, therefore the data grounding transformations are specific to their services. Finer-grained and declarative approaches would increase the potential of reuse, lowering the cost of creating semantic descriptions that support invocation.

Finally, there is more space for future work on the implementations and tools around WSMO-Lite. For instance, beyond the implementations described in Chapter 8, there is only preliminary tooling for service composition, invocation, and for service description validation.

Bibliography

[1] Rama Akkiraju, Joel Farrell, John Miller, Meenakshi Nagarajan, Marc-Thomas Schmidt, Amit Sheth, and Kunal Verma. Web Service Semantics – WSDL-S, November 2005. W3C member submission, available at `http://www.w3.org/Submission/WSDL-S/`.

[2] Anthony Nadalin (editor) et al. Web Services Security Policy Language (WS-SecurityPolicy). Technical note, December 2002. Available at `http://msdn.microsoft.com/ws/2002/12/ws-security-policy/`.

[3] Architecture of the World Wide Web. Recommendation, W3C, December 2004. Available at `http://www.w3.org/TR/webarch/`.

[4] The Atom Syndication Format. RFC 4287, IETF, December 2005. Available at `http://www.rfc-editor.org/rfc/rfc4287.txt`.

[5] The Atom Publishing Protocol. RFC 5023, IETF, October 2007. Available at `http://www.rfc-editor.org/rfc/rfc5023.txt`.

[6] Ziv Baida, Jaap Gordijn, and Borys Omelayenko. A shared service terminology for online service provisioning. In *ICEC '04: Proceedings of the 6th international conference on Electronic commerce*, pages 1–10, New York, NY, USA, 2004. ACM Press.

[7] Claudio Bartolini and Chris Preist. A Framework for Automated Negotiation. Technical Report HPL-2001-90, HP Laboratories Bristol, 2001.

[8] The Base16, Base32, and Base64 Data Encodings. RFC 4648, IETF, October 2006. Available at `http://www.rfc-editor.org/rfc/rfc4648.txt`.

[9] Steve Battle and David Martin. W3C Workshop on Frameworks for Semantics in Web Services Summary Report, 2005. Available at `http://www.w3.org/2005/04/FSWS/workshop-report.html`.

[10] Carrie Beam and Arie Segev. Automated negotiations: A survey of the state of the art. *Wirtschaftsinformatik*, 39(3):263–268, 1997.

[11] Tim Berners-Lee, Yuhsin Chen, Lydia Chilton, Dan Connolly, Ruth Dhanaraj, James Hollenbach, Adam Lerer, and David Sheets. Tabulator: Exploring and analyzing linked data on the semantic web. In *In Proceedings of the 3rd International Semantic Web User Interaction Workshop*, Nov 2006. Co-located with the 5th International Semantic Web Conference (ISWC 2006), Athens, Georgia, USA.

[12] Tim Berners-Lee, James Hendler, and Ora Lassila. The Semantic Web. *Scientific American*, 284(5):34–43, 2001.

[13] Christian Bizer, Tom Heath, and Tim Berners-Lee. Linked Data - The Story So Far. *Intl Journal on Semantic Web and Information Systems, Spc. Issue on Linked Data*, 2009.

[14] Avrim L. Blum and Merrick L. Furst. Fast planning through planning graph analysis. *Artificial intelligence*, 90(1–2):281–300, 1997.

[15] Egon Börger and Robert F. Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.

[16] Luis Botelho, Alberto Fernández, Benedikt Fries, Matthias Klusch, Lino Pereira, Tiago Santos, Pedro Pais, and Matteo Vasirani. Service Discovery. In Michael Schumacher, Heikki Helin, and Heiko Schuldt, editors, *CASCOM: Intelligent Service Coordination in the Semantic Web*, chapter 10. Springer Birkhäuser, 2008.

[17] Liliana Cabral, Ning Li, and Jacek Kopecký. Building the WSMO-Lite test collection on the SEALS Platform. In *Second International Workshop on Evaluation of Semantic Technologies (IWEST 2012), co-located with the 9th Extended Sematic Web Conference 2012 (ESWC 2012)*, volume 843 of *Ceur Workshop Proceedings*, Heraklion, Greece, May 2012. Available at `http://oro.open.ac.uk/34170/`.

[18] Jorge Cardoso, John A. Miller, Amit Sheth, Jonathan Arnold, and Krys Kochut. Quality of service for workflows and web service processes. *Journal of Web Semantics*, 1(3):281–308, 2004.

[19] Henry Chesbrough and Jim Spohrer. A research manifesto for services science. *Commun. ACM*, 49(7):35–40, 2006.

[20] Emilia Cimpian, Adrian Mocan, and Michael Stollberg. Mediation Enabled Semantic Web Services Usage. In *The Semantic Web – ASWC 2006*, volume 4185 of *Lecture Notes in Computer Science, LNCS*, pages 459–473. Springer, 2006.

[21] DeWitt Clinton. OpenSearch 1.1 Draft 5. Working Draft, A9.com, March 2011. Available at `http://www.opensearch.org/Specifications/OpenSearch/1.1`.

[22] Jos de Bruijn, Christoph Bussler, John Domingue, Dieter Fensel, Martin Hepp, Uwe Keller, Michael Kifer, Birgitta König-Ries, Jacek Kopecký, Rubén Lara, Holger Lausen, Eyal Oren, Axel Polleres, Dumitru Roman, James Scicluna, and Michael Stollberg. Web Service Modeling Ontology (WSMO), June 2005. W3C member submission, available at `http://www.w3.org/Submission/WSMO/`.

[23] Jos de Bruijn, Dieter Fensel, and Holger Lausen. D34v0.1: The Web Compliance of WSML. Technical report, DERI, 2007. Available from: `http://www.wsmo.org/TR/d34/v0.1/`.

[24] Jos de Bruijn, Holger Lausen, Axel Polleres, and Dieter Fensel. The Web Service Modeling Language WSML: An Overview. In *Proceedings of the 3rd European Semantic Web Conference (ESWC 2006)*, volume 4011 of *Lecture Notes in Computer Science, LNCS*. Springer, 6 2006.

[25] Schahram Dustdar and Wolfgang Schreiner. A Survey on Web Services Composition. *International Journal of Web and Grid Services*, 1(1):1–30, 2005.

[26] Dieter Fensel and Christoph Bussler. The Web Service Modeling Framework WSMF. *Electronic Commerce Research and Applications*, 1(2):113–137, 2002.

[27] Dieter Fensel, Florian Fischer, Jacek Kopecký, Reto Krummenacher, Dave Lambert, and Tomas Vitvar. WSMO-Lite: Lightweight Semantic Descriptions for Services on the Web, August 2010. W3C member submission, available at `http://www.w3.org/Submission/WSMO-Lite/`.

[28] Dieter Fensel, Uwe Keller, Holger Lausen, Axel Polleres, and Ioan Toma. WWW or what is wrong with web service discovery. In *Proceedings of the W3C Workshop on Frameworks for Semantics in Web Services*, 2005. available at `http://www.w3.org/2005/04/FSWS/Submissions/50/WWW_or_What_is_Wrong_with_Web_service_Discovery.pdf`.

[29] Roberta Ferrario and Nicola Guarino. Towards an ontological foundation for services science. In *Future Internet – FIS 2008*, volume 5468 of *Lecture Notes in Computer Science, LNCS*, pages 152–169, 2009.

[30] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, Irvine, 2000. Chair: Richard N. Taylor.

[31] Richard E. Fikes and Nils J. Nilsson. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence*, 2:189–208, 1971.

[32] Jesse James Garrett. Ajax: A New Approach to Web Applications. Blog article, available at `http://www.adaptivepath.com/ideas/ajax-new-approach-web-applications`, February 2005.

[33] Karthik Gomadam, Ajith Ranabahu, and Amit Sheth. SA-REST: Semantic Annotation of Web Resources, 2010. W3C member submission, available at `http://www.w3.org/Submission/SA-REST/`.

[34] Asunción Gómez-Pérez, Mariano Fernández-López, and Oscar Corcho. *Ontological Engineering: With Examples from the Areas of Knowledge Management, E-Commerce and the Semantic Web*. Springer, 2004.

[35] Gleaning Resource Descriptions from Dialects of Languages (GRDDL). Recommendation, W3C, September 2007. Available at `http://www.w3.org/TR/grddl/`.

[36] Marc J. Hadley. Web Application Description Language (WADL). Technical report, Sun Microsystems, November 2006. Available at `https://wadl.dev.java.net/`.

[37] Armin Haller, Emilia Cimpian, Adrian Mocan, Eyal Oren, and Christoph Bussler. WSMX – A Semantic Service-Oriented Architecture. *International Conference on Web Services (ICWS 2005)*, July 2005.

[38] James Hendler. Agents and the Semantic Web. *IEEE Intelligent Systems*, 16(2):30–37, Mar–Apr 2001.

[39] James Hendler, Tim Berners-Lee, and Eric Miller. Integrating Applications on the Semantic Web. *Journal of the Institute of Electrical Engineers of Japan*, 122(10):676–680, October 2002. In Japanese; English reprint available at `http://www.w3.org/2002/07/swint`.

[40] Martin Hepp. Possible Ontologies: How Reality Constraints the Development of Relevant Ontologies. *IEEE Internet Computing*, 11(1):90–96, 2007.

[41] Andreas Herzig and Omar Rifi. Propositional belief base update and minimal change. *Artificial Intelligence*, 115(1):107–138, 1999.

[42] Jörg Hoffmann, Ingo Weber, James Scicluna, Tomasz Kaczmarek, and Anupriya Ankolekar. Combining Scalability and Expressivity in the Automatic Composition of Semantic Web Services. In *Proceedings of the 8th International Conference on Web Engineering (ICWE'08)*, Yorktown Heights, USA, July 2008.

[43] Jörg Hoffmann, Ingo Weber, James Scicluna, Tomasz Kaczmarek, and Anupriya Ankolekar. Combining Scalability and Expressivity in the Automatic Composition of Semantic Web Services. Technical report, DERI Innsbruck, February 2008. Available at `http://www.loria.fr/~hoffmanj/papers/tr-icwe08.pdf`.

[44] Ian Horrocks, Peter F. Patel-Schneider, Harold Boley, Said Tabet, Benjamin Grosof, and Mike Dean. SWRL: A Semantic Web Rule Language Combining OWL and RuleML. Technical report, Joint US/EU ad hoc Agent Markup Language Committee, 2003. Available at `http://www.daml.org/2003/11/swrl/`.

[45] HTML 4.01 Specification. Recommendation, W3C, December 1999. Available at `http://www.w3.org/TR/html401`.

[46] HTML Microdata. Working Draft, W3C, March 2012. Available at `http://www.w3.org/TR/microdata`.

[47] Microdata to RDF: Transformation from HTML+Microdata to RDF. Interest Group Note, W3C, Octorber 2012. Available at `http://www.w3.org/TR/microdata-rdf/`.

[48] HTML5: A vocabulary and associated APIs for HTML and XHTML. Working Draft, W3C, March 2012. Available at `http://www.w3.org/TR/html5`.

[49] Hypertext Transfer Protocol – HTTP/1.1. RFC 2616, IETF, June 1999. Available at `http://www.rfc-editor.org/rfc/rfc2616.txt`.

[50] Ching-Lai Hwang and Kwangsun Yoon. *Multiple attribute decision making: methods and applications: a state-of-the-art survey*. Lecture Notes in Economics and Mathematical Systems. Springer-Verlag New York, 1981.

[51] Michael C. Jaeger, Gregor Rojec-Goldmann, Christoph Liebetruth, Gero Mühl, and Kurt Geihs. Ranked Matching for Service Descriptions Using OWL-S. In *Kommunikation in verteilten Systemen (KiVS)*, pages 91–102. Springer, 2005.

[52] Java Message Service (JMS) API. JSR 914, Java Community Process, April 2002. Available at `http://www.oracle.com/technetwork/java/jms/index.html`.

[53] The application/json Media Type for JavaScript Object Notation (JSON). RFC 4627, IETF, July 2006. Available at `http://www.rfc-editor.org/rfc/rfc4627.txt`.

[54] Uwe Keller, Ruben Lara, Holger Lausen, and Dieter Fensel. Semantic Web Service Discovery in the WSMO Framework. In Jorge Cardoso, editor, *Semantic Web: Theory, Tools and Applications*. Idea Publishing Group, 2006.

[55] Uwe Keller, Ruben Lara, Axel Polleres, Ioan Toma, Michael Kifer, and Dieter Fensel. D5.1: WSMO Web Service Discovery. Technical report, DERI Innsbruck, 2004. Available from `http://www.wsmo.org/TR/d5/d5.1`.

[56] Mick Kerrigan and Barry Norton (eds.). Reference Architecture for Semantic Execution Environments. OASIS Technical Committee Initial Draft of the Semantic Execution Environment Tech. Committee (SEE TC), Jan 2009. Available at `http://oasis-open.org/committees/download.php/30675/Semantic%20Execution%20Environment%20Reference%20Architecture_20090113_BN.doc`.

[57] Mick Kerrigan, Barry Norton, and Elena Simperl. MEMOS: A Methodology for Modeling Services. In *Proceedings of the 5th International workshop on Semantic Business Process Management (SBPM)*, Crete, Greece, 2010. Co-located with the 7th European Semantic Web Conference (ESWC).

[58] Rohit Khare and Tantek Çelik. Microformats: a pragmatic path to the semantic web (Poster). *Proceedings of the 15th international conference on World Wide Web*, pages 865–866, 2006.

[59] Christoph Kiefer and Abraham Bernstein. The Creation and Evaluation of iSPARQL Strategies for Matchmaking. In *Proceedings of the 5th European Semantic Web Conference (ESWC)*, volume 5021 of *Lecture Notes in Computer Science, LNCS*. Springer, February 2008.

[60] Knowledge Interchange Format (KIF). Draft proposed American National Standard (dpANS) NCITS.T2/98-004, ANSI, 1998. Available at `http://logic.stanford.edu/kif/dpans.html`.

[61] Matthias Klusch. Semantic Web Service Coordination. In Michael Schumacher, Heikki Helin, and Heiko Schuldt, editors, *CASCOM: Intelligent Service Coordination in the Semantic Web*, chapter 4. Springer Birkhäuser, 2008.

[62] Jacek Kopecký, Matthew Moran, Tomas Vitvar, Dumitru Roman, and Adrian Mocan. WSMO Grounding. Available at `http://www.wsmo.org/TR/d24/d24.2/`, April 2007.

[63] Jacek Kopecký and Elena Simperl. Semantic Web Service Offer Discovery For E-commerce. In *Proceedings of the 10th International Conference on Electronic Commerce 2008, Innsbruck, Austria, August 19–22, 2008*, 2008.

[64] Jacek Kopecký, Tomas Vitvar, Carine Bournez, and Joel Farrell. SAWSDL: Semantic Annotations for WSDL and XML Schema. *IEEE Internet Computing*, 11(6):60–67, 2007.

[65] Jacek Kopecký, Tomas Vitvar, Carlos Pedrinaci, and Maria Maleshkova. RESTful Services with Lightweight Machine-readable Descriptions and Semantic Annotations. In Erik Wilde and Cesare Pautasso, editors, *REST: From Research to Practice*, chapter 22, pages 473–506. Springer, 2011.

[66] Reto Krummenacher, Barry Norton, Elena Simperl, and Carlos Pedrinaci. Soa4all: enabling web-scale service economies. In *Proceedings of the 2009 IEEE International Conference on Semantic Computing*, pages 535–542. IEEE, 2009.

[67] Ulrich Küster and Birgitta König-Ries. Supporting dynamics in service descriptions — the key to automatic service usage. In *Proceedings of the Fifth International Conference on Service Oriented Computing (ICSOC07)*, Vienna, Austria, September 2007.

[68] Freddy Lecue and Alain Leger. A formal model for semantic Web service composition. *Lecture Notes in Computer Science*, 4273:385–398, 2006.

[69] Wilfried Lemahieu. Web service description, advertising and discovery: WSDL and beyond. In M. Snoeck and J. Vandenbulcke, editors, *New Directions in Software Engineering*. Leuven University Press, 2001.

[70] Yutu Liu, Anne H.H. Ngu, and Liangzhao Zeng. QoS computation and policing in dynamic web service selection. In *Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*, pages 66–73. ACM New York, NY, USA, 2004.

[71] Maria Maleshkova, Carlos Pedrinaci, and John Domingue. Supporting the creation of semantic restful service descriptions. In *Workshop: Service Matchmaking and Resource Retrieval in the Semantic Web (SMR2) at 8th International Semantic Web Conference*, 2009.

[72] Maria Maleshkova, Carlos Pedrinaci, and John Domingue. Investigating Web APIs on the World Wide Web. In *Proceedings of he 8th IEEE European Conference on Web Services (ECOWS 2010)*, 2010. Available at `http://oro.open.ac.uk/24320/`.

[73] Maria Maleshkova, Guillermo Álvaro Rey, Alex Simov, Bruno Renie, and Dong Liu. Service Provisioning Platform Second Prototype, Aug 2010. Deliverable D2.1.4 of the project SOA4All, available at `http://soa4all.eu/file-upload.html?func=startdown&id=229`.

[74] 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, November 2004. W3C member submission, available at `http://www.w3.org/Submission/OWL-S/`.

[75] David Martin, Massimo Paolucci, and Matthias Wagner. Bringing Semantic Annotations to Web Services: OWL-S from the SAWSDL Perspective. In Karl Aberer, Key-Sun Choi, Natasha Noy, Dean Allemang, Kyung-Il Lee, Lyndon Nixon, Jennifer Golbeck, Peter Mika, Diana Maynard, Riichiro Mizoguchi, Guus Schreiber, and Philippe Cudré-Mauroux, editors, *The Semantic Web*, volume 4825 of *Lecture Notes in Computer Science, LNCS*, pages 340–352. Springer, 2007.

[76] Deborah L. McGuinness. Ontologies Come of Age. In Dieter Fensel, Jim Hendler, Henry Lieberman, and Wolfgang Wahlster, editors, *Spinning the Semantic Web: Bringing the World Wide Web to Its Full Potential*, pages 171–192. MIT Press, 2003.

[77] Sheila A. McIlraith, Tran Cao Son, and Honglei Zeng. Semantic Web Services. *IEEE Intelligent Systems*, 16(2):46–53, Mar–Apr 2001.

[78] Adrian Mocan and Emilia Cimpian. An ontology-based data mediation framework for semantic environments. *International Journal on Semantic Web & Information Systems*, 3(2):69–98, 2007.

[79] Dana Nau, Malik Ghallab, and Paolo Traverso. *Automated Planning: Theory & Practice*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

[80] Jörg Nitzsche, Tammo Van Lessen, and Frank Leymann. WSDL 2.0 message exchange patterns: limitations and opportunities. In *Third International Conference on Internet and Web Applications and Services, 2008. ICIW'08.*, pages 168–173. IEEE, 2008.

[81] Justin O'Sullivan, David Edmond, and Arthur H. M. ter Hofstede. Formal description of non-functional service properties. Technical Report FIT-TR-2005-01, Queensland University of Technology, 2005. Available from `http://service-description.com/resources.htm`.

[82] OWL Web Ontology Language Overview. Recommendation 10 February 2004, W3C, 2004. Available at `http://www.w3.org/TR/owl-features/`.

[83] OWL-S 1.1 Release. Technical report, OWL Services Coalition, November 2004. Available at `http://www.daml.org/services/owl-s/1.1/`.

[84] OWL 2 Web Ontology Language Document Overview. Recommendation 27 October 2009, W3C, 2009. Available at `http://www.w3.org/TR/owl2-overview/`.

[85] Massimo Paolucci, Matthias Wagner, and David Martin. Grounding OWL-S in SAWSDL. In Bernd J. Krämer, Kwei-Jay Lin, and Priya Narasimhan, editors, *ICSOC*, volume 4749 of *Lecture Notes in Computer Science, LNCS*, pages 416–421. Springer, 2007.

[86] Shamimabi Paurobally and Nicholas R. Jennings. Protocol engineering for web services conversations. *Engineering Applications of Artificial Intelligence*, 18(2):237–254, 2005. Special Issue on Agent-oriented Software Development.

[87] Shamimabi Paurobally, Valentina Tamma, and Michael Wooldrdige. A Framework for Web service negotiation. *ACM Trans. Auton. Adapt. Syst.*, 2, November 2007.

[88] Carlos Pedrinaci and John Domingue. Toward the Next Wave of Services: Linked Services for the Web of Data. *Journal of Universal Computer Science*, 16(13):1694–1719, Jul 2010. Available at `http://www.jucs.org/jucs_16_13/toward_the_next_wave`.

[89] Carlos Pedrinaci, Dong Liu, Maria Maleshkova, Dave Lambert, Jacek Kopecký, and John Domingue. iServe: a Linked Services Publishing Platform. In *Proceedings of 1st International Workshop on Ontology Repositories and Editors for the Semantic Web, ORES 2010, colocated with* 7^{th} *ESWC*, 2010.

[90] Charles Petrie, Axel Hochstein, and Michael Genesereth. Semantics for smart services. In Haluk Demirkan, James C. Spohrer, and Vikas Krishna, editors, *The science of service systems*, pages 91–105. Springer, New York, 2011.

[91] Marco Pistore, Pierluigi Roberti, and Paolo Traverso. Process-level Composition of Executable Web Services: "On-the-fly" Versus "Once-for-all" Composition. In *Proceedings of the Second European Semantic Web Conference (ESWC'05)*, volume 3532 of *Lecture Notes in Computer Science, LNCS*, pages 62–77. Springer-Verlag, 2005.

[92] Axel Polleres, Thomas Krennwallner, Nuno Lopes, Jacek Kopecký, and Stefan Decker. XSPARQL Language Specification, January 2009. W3C member submission, available at `http://www.w3.org/Submission/xsparql-language-specification/`.

[93] Chris Preist. A Conceptual Architecture for Semantic Web Services. In *Proceedings of the 3rd International Semantic Web Conference (ISWC 2004)*, volume 3298 of *Lecture Notes in Computer Science, LNCS*, pages 395–409. Springer, 2004.

[94] Shuping Ran. A model for web services discovery with QoS. *ACM SIGecom Exchanges*, 4(1):1–10, 2003.

[95] Resource Description Framework (RDF): Concepts and Abstract Syntax. Recommendation, W3C, February 2004. Available at `http://www.w3.org/TR/rdf-concepts/`.

[96] RDF Semantics. Recommendation, W3C, February 2004. Available at http://www.w3.org/TR/rdf-mt/.

[97] RDFa in XHTML: Syntax and Processing. Recommendation, W3C, October 2008. Available at http://www.w3.org/TR/rdfa-syntax/.

[98] RDF Vocabulary Description Language 1.0: RDF Schema. Recommendation, W3C, February 2004. Available at http://www.w3.org/TR/rdf-schema/.

[99] RDF/XML Syntax Specification (Revised). Recommendation, W3C, February 2004. Available at http://www.w3.org/TR/2004/REC-rdf-syntax-grammar-20040210/.

[100] Reference Model for Service Oriented Architecture 1.0. OASIS Standard, Oct 2006. Available at http://docs.oasis-open.org/soa-rm/v1.0/soa-rm.html.

[101] Leonard Richardson and Sam Ruby. *RESTful Web Services*. O'Reilly Media, May 2007.

[102] RIF Core Dialect. Recommendation, W3C, June 2010. Available at http://www.w3.org/TR/rif-core/.

[103] RIF Basic Logic Dialect. Recommendation, W3C, June 2010. Available at http://www.w3.org/TR/rif-bld/.

[104] RIF Overview. Recommendation, W3C, June 2010. Available at http://www.w3.org/TR/rif-overview/.

[105] Dumitru Roman, Uwe Keller, Holger Lausen, Jos de Bruijn, Ruben Lara, Michael Stollberg, Axel Polleres, Cristina Feier, Christop Bussler, and Dieter Fensel. Web Service Modeling Ontology. *Applied Ontology*, 1(1):77–106, 2005.

[106] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern approach (second edition)*. Prentice Hall, Englewood Cliffs, NJ, 2002.

[107] Semantic Annotations for WSDL and XML Schema. Recommendation, W3C, August 2007. Available at http://www.w3.org/TR/sawsdl/.

[108] Amit P. Sheth. Semantic Web Process Lifecycle: Role of Semantics in Annotation, Discovery, Composition and Orchestration. Invited Talk, Workshop on E-Services and the Semantic Web, at World Wide Web Conference. Available at http://lsdis.cs.uga.edu/lib/presentations/WWW2003-ESSW-invitedTalk-Sheth.pdf, 2003.

[109] Amit P. Sheth, Karthik Gomadam, and Jon Lathem. SA-REST: Semantically Interoperable and Easier-to-Use Services and Mashups. *IEEE Internet Computing*, 11(6):91–94, 2007.

[110] Alex Simov. WSMO Data Grounding Component, Aug 2009. Deliverable D3.4.4 of the project SOA4All, available at http://soa4all.eu/file-upload.html?func=startdown&id=137.

[111] Simple Knowledge Organization System. Recommendation, W3C, August 2009. Available at http://www.w3.org/TR/skos-reference/.

[112] Reid G. Smith. The Contract Net Protocol: High-Level Communication and Control in a Distributed Problem Solver. *Computers, IEEE Transactions on*, C-29(12):1104–1113, Dec 1980.

[113] SOAP Version 1.2 Part 1: Messaging Framework. Recommendation, W3C, June 2003. Available at http://www.w3.org/TR/2003/REC-soap12-part1-20030624/.

[114] SPARQL Query Language for RDF. Recommendation, W3C, January 2008. Available at http://www.w3.org/TR/rdf-sparql-query/.

[115] Stefen Staab and Rudi Studer. *Handbook on Ontologies*. Springer, 2004.

[116] Michael Stollberg. *Scalable Semantic Web Service Discovery for Goal-driven Service-Oriented Architectures*. PhD thesis, University of Innsbruck, 2008.

[117] Katia Sycara, Massimo Paolucci, Anupriya Ankolekar, and Naveen Srinivasan. Automated discovery, interaction and composition of Semantic Web services. *Web Semantics: Science, Services and Agents on the World Wide Web*, 1(1):27–46, 2003.

[118] Ioan Toma, Dumitru Roman, Dieter Fensel, Brahmananda Sapkota, and Juan Miguel Gómez. A multi-criteria service ranking approach based on non-functional properties rules evaluation. In Bernd J. Krämer, Kwei-Jay Lin, and Priya Narasimhan, editors, *ICSOC*, volume 4749 of *Lecture Notes in Computer Science, LNCS*, pages 435–441. Springer, 2007.

[119] David Trastour, Claudio Bartolini, and Javier Gonzalez-Castillo. A Semantic Web Approach to Service Description for Matchmaking of Services. Technical Report HPL-2001-183, HP Laboratories Bristol, 2001.

[120] UDDI Version 3.0.2. OASIS Standard, Oct 2004. Available at http://www.oasis-open.org/committees/uddi-spec/doc/spec/v3/uddi-v3.0.2-20041019.htm.

[121] User Datagram Protocol (UDP). RFC 768, IETF, August 1980. Available at http://www.rfc-editor.org/rfc/rfc768.txt.

[122] Gulay Unel, Uwe Keller, Florian Fischer, and Barry Bishop. Defining the features of the WSML-Quark language, March 2009. Deliverable D3.1.1 of the project SOA4All, available at http://soa4all.eu/file-upload.html?func=startdown&id=80.

[123] Uniform Resource Identifier (URI): Generic Syntax. RFC 3986, IETF, January 2005. Available at http://www.rfc-editor.org/rfc/rfc3986.txt.

[124] Kunal Verma, Rama Akkiraju, and Richard Goodwin. Semantic Matching of Web Service Policies. In *Proceedings of the Second International Workshop on Semantic and Dynamic Web Process (SDWP), in conjunction with ICWS 2005*, Orlando, Florida, USA, 2005.

[125] Tomas Vitvar, Jacek Kopecký, Jana Viskova, and Dieter Fensel. WSMO-Lite Annotations for Web Services. In *The Semantic Web: Research and Applications, 5th European Semantic Web Conference, ESWC 2008*, pages 674–689, Tenerife, Spain, 2008. Springer.

[126] Tomas Vitvar, Adrian Mocan, Mick Kerrigan, Michal Zaremba, Maciej Zaremba, Matthew Moran, Emilia Cimpian, Thomas Haselwanter, and Dieter Fensel. Semantically-enabled service oriented architecture : concepts, technology and application. *Service Oriented Computing and Applications*, 2(2), 2007.

[127] Tomas Vitvar, Maciej Zaremba, and Matthew Moran. Dynamic service discovery through meta-interactions with service providers. In Enrico Franconi, Michael Kifer, and Wolfgang May, editors, *ESWC*, volume 4519 of *Lecture Notes in Computer Science, LNCS*, pages 84–98. Springer, 2007.

[128] Marianne Winslett. Reasoning about actions using a possible models approach. In *Proc. AAAI'88*, 1988.

[129] Web Services Agreement Specification (WS-Agreement). Recommendation, Open Grid Forum, March 2007. Available at `http://www.ogf.org/documents/GFD.107.pdf`.

[130] Web Services Architecture. Working group note, W3C, February 2004. Available at `http://www.w3.org/TR/ws-arch`.

[131] Web Services Policy 1.5 – Framework. Recommendation, W3C, September 2007. Available at `http://www.w3.org/TR/ws-policy/`.

[132] Web Services Conversation Language (WSCL) 1.0. W3C Member Submission, Hewlett-Packard, March 2002. Available at `http://www.w3.org/TR/wscl10/`.

[133] Web Services Description Language (WSDL) Version 2.0. Recommendation, W3C, June 2007. Available at `http://www.w3.org/TR/wsdl20/`.

[134] Web Services Description Language (WSDL) Version 2.0: Adjuncts. Recommendation, W3C, June 2007. Available at `http://www.w3.org/TR/wsdl20-adjuncts/`.

[135] Web Services Description Language (WSDL) Version 2.0: Primer. Recommendation, W3C, June 2007. Available at `http://www.w3.org/TR/wsdl20-primer/`.

[136] The Web Service Modeling Language WSML. Technical report, WSMO Working Group, 2008. Available at `http://www.wsmo.org/TR/d16/d16.1/v1.0/`.

[137] XForms 1.0 (Third Edition). Recommendation, W3C, October 2007. Available at `http://www.w3.org/TR/xforms/`.

[138] Extensible Markup Language (XML) 1.0. Recommendation, W3C, February 2004. Available at `http://www.w3.org/TR/REC-xml`.

[139] XML Media Types. RFC 3023, IETF, January 2001. Available at http://www.rfc-editor.org/rfc/rfc3023.txt.

[140] XML Schema Part 1: Structures. Recommendation, W3C, October 2004. Available at http://www.w3.org/TR/xmlschema-1/.

[141] XML Path Language (XPath) Version 1.0. Recommendation, W3C, November 1999. Available at http://www.w3.org/TR/xpath.

[142] XML Path Language (XPath) 2.0 (Second Edition). Recommendation, W3C, December 2010. Available at http://www.w3.org/TR/xpath20.

[143] XQuery 1.0: An XML Query Language. Recommendation, W3C, June 2006. Available at http://www.w3.org/TR/xquery.

[144] XQuery 1.0 and XPath 2.0 Data Model (XDM) (Second Edition). Recommendation, W3C, December 2010. Available at http://www.w3.org/TR/xpath-datamodel.

[145] XQuery 1.0 and XPath 2.0 Functions and Operators (Second Edition). Recommendation, W3C, December 2010. Available at http://www.w3.org/TR/xpath-functions.

[146] XSL Transformations. Recommendation, W3C, November 1999. Available at http://www.w3.org/TR/xslt.

[147] Hong Qing Yu, Neil Benn, Stefan Dietze, Ronald Siebes, Carlos Pedrinaci, Dong Liu, David Lambert, and John Domingue. Two-staged approach for semantically annotating and brokering TV-related services. In *Proceedings of the IEEE International Conference on Web Services (ICWS)*, Miami, Florida, USA, 2010.

[148] Tao Yu, Yue Zhang, and Kwei-Jay Lin. Efficient Algorithms for Web Services Selection with End-to-End QoS Constraints. *ACM Transactions on the Web*, 1(1), May 2007. Available at http://doi.acm.org/10.1145/1232722.1232728.

[149] Maciej Zaremba, Tomas Vitvar, Matthew Moran, and Thomas Hasselwanter. WSMX Discovery for SWS Challenge. SWS Challenge Workshop, Athens, Georgia, USA, November 2006.