Implementation Environments for Semantic Web Services

Rien Broekstra

Thesis for a Master of Science degree in Telematics from the University of Twente, Enschede, The Netherlands

Graduation committee: Dr. L. Ferreira Pires
Dr. ir. M. van Sinderen
L.O. Bonino da Silva Santos MSc.

Date: December 29, 2007
Abstract

The Semantic Web is a vision in which data on the World Wide Web is annotated with machine-processable semantics, in order to enable agents to understand and process information that can only be processed by humans at present. The Semantic Web vision can be extended to the domain of Web Services. Semantically annotated Web Services are expected, amongst other things, to offer easier B2B integration, and better automation service discovery and composition. Several initiatives are under way to standardize and realize these Semantic Web Services.

This thesis evaluates the wsmx semantic web services implementation environment for support for activities required for successful use of Semantic Web Services, and proposes an extension of the implementation environment to enable it to support automated composition of services.

We find that wsmx is able to perform automated web service discovery, but have doubts about scalability of the implementation. Furthermore, the environment is not able to provide composition of services. In addition, extension and improvement of the environment has proven to be cumbersome for a number of practical reasons.
Acknowledgement

The work reported in this thesis is related to the Freeband A-MUSE project [http://a-muse.freeband.nl] in which the University of Twente participates. Freeband is sponsored by the Dutch government under contract BSIK 03025.
## Contents

1 Introduction
   1.1 Motivation .................................................. 1
   1.2 Objectives .................................................. 3
   1.3 Approach ................................................... 3
   1.4 Structure .................................................. 4

2 Background
   2.1 Services .................................................. 5
   2.2 Discovery .................................................. 6
   2.3 Mediation .................................................. 8
   2.4 Selection (Contract agreement) ......................... 9
   2.5 Delivery .................................................. 9
   2.6 Composition ............................................. 10
   2.7 Available standard initiatives ....................... 11
      2.7.1 OWL-S ............................................... 11
      2.7.2 SWSF ............................................... 12
      2.7.3 WSDL-S .............................................. 14
      2.7.4 WSMO ............................................... 14
   2.8 Standards comparison ................................ 15

3 The Web Service Modeling Ontology
   3.1 Background ............................................... 17
   3.2 Overview .................................................. 18
   3.3 Top level elements .................................... 19
      3.3.1 Ontologies .......................................... 19
      3.3.2 Web Services ....................................... 21
      3.3.3 Goals ............................................... 21
      3.3.4 Mediators .......................................... 21
   3.4 The WSMO language: WSML ............................ 23
   3.5 Implementation environments ....................... 24
      3.5.1 IRS-III ............................................. 25
      3.5.2 WSMX ............................................... 27
   3.6 Comparison ............................................. 27
Chapter 1

Introduction

This chapter presents the motivation, the objectives and the structure of this thesis. It introduces the concepts of service discovery and composition, elaborates on the importance of the incorporation of semantic machine-processable service descriptions into the Web Services infrastructure, and motivates why it is desirable to investigate the current state of development of implementation environments which support these kinds of service descriptions.

The chapter is further structured as follows: Section 1.1 presents some background on semantically-enabled services and motivates this work; Section 1.2 lists the objectives of this thesis; Section 1.3 briefly outlines the approach used in this thesis’ development; Section 1.4 gives the structure of the thesis by presenting an overview of the chapters.

1.1 Motivation

“A distributed system is one in which the failure of a computer you didn’t even know existed can render your own computer unusable.”
– Leslie Lamport

The topic of distributed systems is a broad one in computer science. A lot of different kinds of distributed systems exist, varying from small, highly engineered systems designed for fault tolerance (for example, the controller of a Space Shuttle), to large open systems designed for scalability or availability (for instance, the Domain Name System [31] [32]).

The growing presence of the Internet has opened the door for a new type of distributed application. In this type of application, most or all of the application’s functionality is provided by services offered from all over the Internet. The application itself is nothing more than a description of how the different services should be combined in order to get the desired result.

This new type of distributed applications poses a lot of new challenges. For example, since the different parts of the application are implemented by different entities, the parts should be independent of platforms and development-technologies. Also, since the availability
of the parts might not be guaranteed and since entities may change their part of the application without notice, loose coupling of the application parts is also desirable.

To facilitate these requirements, the Service-Oriented Architecture (SOA) paradigm has been introduced. SOA is a paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains. It provides a uniform means to offer, discover, interact with and use capabilities to produce desired effects consistent with measurable preconditions and expectations \[27\].

SOA proponents foresee a complete change in application design and development. What is currently achieved by monolithic, inflexible applications, may be achieved by distributed applications, composed, possibly on demand, from other services that are in turn composed from services \[28\].

The industry adopted SOAP/WSDL-based Web Services as the basic technology for the interoperability in a service-oriented fashion. Web Services provide an abstraction from actual service implementations, and provide a syntactic basis for interoperation between applications and services. However, syntactic interoperability alone is not enough.

A service requester and provider must share an understanding of terms. Different services provided by different providers may use a different term for the same concept, or may even use the same term for a completely different concept. In order for a requester to successfully use a service, the provider and the requester must agree on the service properties on the level of concepts, rather than on a syntactical level.

Additionally, service provider and requester must agree on behavioural aspects of the service. For example, a service provider may impose an order of message exchange. An airline ticket booking service might want to know a customer’s name and credit card number first, before accepting a request to book a flight. A service requester requires information of this prescribed message order, also called Service Choreography, in order to be able to make proper use of the service.

Service requesters need to be able to find a service which suits their requirements. These requirements may be functional (what should the service do for me?), and non-functional (how much does it cost? how long does it take?). Finding a suitable service is often referred to as Service Discovery.

Not all requirements of a service requester can be fulfilled by using one single service. However, a combination of different services might be able to fulfill a requirement. To this end, services need to be composed to create new services. This is referred to as Service Composition.

With current web service technology, the task of discovery and composition must be done manually: services must be looked up in registries and selected based on their textual description, and the meaning of messages and the order of message exchange must be deduced from their names or from (informal) service documentation. All these descriptions and documentation are likely to be ambiguous and/or incomplete. This makes this task error-prone.

As more and more services are being provided by more and more providers, the task of manually selecting and orchestrating services into a coherent application becomes cum-
1.3. Approach

bersome. Therefore, facilities are needed to automate these procedures. In order for a
computer to meaningfully discover and compose services, it needs the aforementioned un-
derstanding of terms and behaviour of the services. A way to achieve this understanding
is the enrichment of services with machine-processable semantics, creating semantically-
enabled Web Services, or Semantic Web Services (sws).

Several initiatives are currently under way to realize Semantic Web Services. These vary
from a lightweight extension to add some semantic information (Web Service Description
Language with Semantics (wsdl-s) [2]), to ontologies which represent web services-related
knowledge (Web Ontology Language and Services (owl-s) [3]), to complete conceptual
models describing several aspects of semantic web services (Web Service Modeling Ontol-
ogy (wsmo) [36], Semantic Web Services Framework (swsf) [6]).

The aforementioned standards and their theoretical capabilities have been extensively
documented by the initiators themselves, and by prospective users of semantic web service
technology (for example [44], [45], [26]). However, little documentation exists on the
current state of development of semantic web service implementation environments.

The evaluation of actual capabilities of sws implementation environments is expected
to give a realistic insight on the state of development of the semantic web. Such an
evaluation may identify area’s in which the technology is lagging behind the theory and
the standards they implement and as such will provide opportunities for further research
and development in the sws field.

1.2 Objectives

The main objective of this research is to investigate the level of support the current
implementation environments offer with respect to automated service discovery and com-
position.

To this end, this thesis selects a suitable semantic web service standards initiative based on
third-party evaluations, and selects a representative implementation environment which
implements that standard. The selection of the implementation environment is based on
the available features of the environment as claimed by their developers.

1.3 Approach

Our approach has included the following steps:

- Survey of third party state-of-the-art reports, in order to make a selection of a suit-
able semantic web service standard initiative, capable of automatic service discovery
  and composition. This resulted in the selection of WSMO.

- Definition of criteria for selecting a representative implementation environment.
  Since evaluating all available Semantic Web Service implementation environments is
beyond the scope of this research, only one promising environment has been chosen and considered in detail.

- Motivation of the selection of the Web Service Execution Environment (WSMX) implementation environment based on these criteria.

- Evaluation of WSMX by analyzing project documentation and source code in order to determine the available functionality. This revealed the lack of composition support in the environment.

- Deployment of WSMX in order to test the execution of discovery scenario’s.

- Architectural design of an extension of WSMX to enable automated web service composition, including a test of a third party planning/composition component.

1.4 Structure

The structure of this thesis reflects the issues dealt with during the thesis’ development:

- Chapter 2 introduces basic concepts and terminology used throughout the rest of this thesis, discusses the current state-of-the-art in web service formalisms, and motivates why WSMO is currently the most suitable semantic web service technology for this thesis.

- Chapter 3 elaborates on WSMO’s design and architecture, describes the available WSMO-compliant implementation environments and motivates the selection of WSMX as implementation environment to be evaluated.

- Chapter 4 reports on the state of development of WSMX, in particular in the area of automated discovery and composition.

- Chapter 5 describes our proposed extension to WSMX to enable it to perform automated web service composition.

- Finally, Chapter 6 presents the final conclusions and identifies topics for future work.
Chapter 2

Background

This chapter presents the background of this theses. It presents basic service-related concepts, addresses the activities expected to be supported by an implementation environment for semantic web services, presents a brief overview of SWS standard initiatives and underpins why the WSMO standard is used in the remainder of the thesis.

The chapter is further structured as follows: Section 2.1 introduces the concept of a service; Sections 2.2 to 2.6 elaborate on the activities of Service Discovery, Mediation, Selection, Delivery and Composition, respectively; Section 2.7 introduces relevant standard initiatives; Section 2.8 presents a comparison of features and motivates the choice of WSMO.

2.1 Services

With the introduction of the service oriented architecture, the term “service” has become ambiguous. Inspired by Preist et al. [37], we will define three different types of services, and present how they are related. The definitions and their relations are depicted in figure 2.1.

At the top level, there is the Business service. A business service provides something of value to a customer, in exchange for a payment to the supplier. Such a service may be completely unrelated to any computer technology. Examples of business services are a pizza delivery service and a housekeeping service.

Then there is the Technical service. A technical service is some program which can send and receive messages, and which may in turn provide a business service. Examples are online bookstores or ticket-booking agencies, which make their business services accessible via a technical service.

Finally there is a Web Service. A Web Service is an implementation of a technical service, which uses SOAP/WSDL messages for communication between service requester and provider.

In the remainder of this thesis, the term service refers to a technical service. Practical experiments will be carried out using web services, them being a prominent implementation
2.2 Discovery

In service context, Service Discovery is the act of locating a machine-processable description of a resource that can be reached through a service that may have been previously unknown and that meets certain functional criteria [7]. As such, it is a key activity within the service oriented architecture paradigm.

In essence, the discovery activity comprises of matching the requirements of a service requester with the properties of a service provider (See figure 2.2). This will yield a list of candidate provided services, of which the properties fulfill the requirements of the requester.

Figure 2.2: Discovery matches requirements against provided service properties
Both the requester’s service requirements, and the provider’s service properties may be described in Service Descriptions. Services can be described in many different languages, which vary in expressiveness and precision [18]. A service description may only contain syntactical information, but it may also contain semantic information including functional requirements and properties, pre- and post-conditions, non-functional requirements and properties, etc. Examples of service description languages are wsdl [9] and wsml [15].

Service descriptions are retrieved from some service repository. This repository holds a number of descriptions, and allows the discovery process to retrieve and inspect these descriptions. A current example of a service repository standard is UDDI, which can be used to hold descriptions of SOAP/WSDL web services descriptions.

The following scenario illustrates the steps needed to discover a service, as envisioned by Keller et al. [22] (see figure 2.3):

- John wants to buy the book “Ender’s Game” by Orson Scott Card. John’s goal is to have the book on his desk, so ultimately he is looking for a business service that delivers this particular book to his house.

  John must describe his goal in terms of requirements that he needs to be fulfilled. These requirements can be used in the discovery process. In this case, John’s requirement may be “deliver Ender’s Game”, or - more general - “deliver book”. How general or specific the requirement must be depends on how specific the service providers specify their provided service properties. In this example, the requirement will be formulated as “deliver book”.

- The next step is to find start a discovery process which will find matching candidate
Mediation services to the requirements formulated in step one. This should yield a list of candidate services which can fulfill “deliver book”.

- Finally, it must be checked whether the candidate services for “deliver book” can also fulfill the requirement “deliver Ender’s Game”. John may do this manually, but this step may also be incorporated in the discovery process.

It should be noted that performing the discovery in 3 steps as described in this example is in fact a design decision, which imposes constraints on the service providers (Every service must provide a method to discover more concrete services). Although it is technically possible for a service provider to put every specific service it can deliver into a service description in order to perform the discovery in one step, this is not desirable. Doing so would lead to an enormous growth in the amount of service descriptions, leading to a much more time consuming discovery step. Keller et al. therefore argue that this model of heuristic classification [11] should be applied to discovery. We will use this method of discovery throughout the rest of this thesis. In short:

1. Goal discovery. This step translates the concrete goal of a user into requirements. This must be done manually by the user.

2. Abstract service discovery. This step returns services that can fulfill the generic goals derived in the first step.

3. Concrete service discovery. The resulting services are now consulted to see if the user’s concrete goal can be fulfilled by the services returned by the abstract service discovery step.

2.3 Mediation

In a distributed environment such as the internet, it is quite possible that people, programs, or services will have communication problems due to the fact that they speak a different language, or expect different behaviour from each other. This results in interoperability problems. A generic solution to these kind of interoperability problems is to introduce mediation. In semantic web services context, mediation can solve interoperability problems on several levels:

In order for a service requester to be able to reason about what a provided service offers, agreement must be reached between requester and provider on the terminology used for concepts. This terminology is often defined in ontologies. In the context of information technology, an ontology is a “formal explicit specification of a shared conceptualization” [20]. As such, an ontology defines concepts and relations between those concepts. This definition of concepts and relations can be used to make sure different entities (i.e. service requesters and providers) agree on the meaning of the terms they use.

If a provided service and a service request are use the same ontology, it is possible for the requester to reason about whether the provider is able to fulfill its request. However,
in case their ontologies do not overlap (i.e. there are no shared concepts/relations), or even contradict each other (concepts/relations are defined differently), a problem arises: the provider may be able to fulfill the requester’s request, but the requester can not know that, because it may not understand, or even worse, may misunderstand the provider’s terminology. For the same reason, a requester may deduce that a provider is able to fulfill the requirements, whereas it is not. This is obviously not desirable.

To overcome this type of miscommunication, Data level mediation can be introduced. A data level mediator is in essence an interpreter which translates between different terminologies used by provider and requester in order to enable meaningful communication between them.

Similarly, requester and provider may not agree on how to behave to each other. For example, the expected order of messages may differ, or requester and provider may have different timing constraints. Communication or Process level mediators solve these kind of behavioural interoperability problems between service requesters and providers.

A last type of mediation is Δ-mediation. Unlike the previous three types of mediation, Δ-mediation is not aimed to solve interoperability problems, but it is a way to model logical differences between goals and web services, reducing the need for storage of a lot of similar goal and webservice descriptions.

2.4 Selection (Contract agreement)

Once the discovery activity has yielded a number of candidate services, the service of choice has to be selected. In its simplest form, the selection process just picks a service from the list of candidates. However, a user (or agent, on the user’s behalf) may have additional requirements or preferences. For example, in case a user is selecting a suitable crate shipping service, he/she may prefer the lowest price, or the fastest delivery time or maybe the best insurance.

More advanced service requesters and providers may even engage in a contract negotiation process, in which the details of the delivered services are determined. This may result in a contract with an explicit representation of the functional and non-functional requirements which will be fulfilled by the agreed service.

2.5 Delivery

After the service of choice has been selected, the service can be delivered. Two issues have to be resolved before actual delivery can take place. First a connection between the service requester and provider has to be established. This connection is referred to as the Service Binding. To establish the connection, the requester needs all technical details to connect to the service, such as network protocols, addresses, etc. This is referred to as the Service Grounding.

Second, a service requester needs to conform itself to the behaviour of the service provider. The constraints imposed on the service requesters by the provider’s behaviour are called
2.6 Composition

Occasionally, it is necessary to combine the result of two or more web services, in order to achieve some goal. For example, a business that sells computer hardware might want to expose a web service that allows customers to find and buy hardware. This service may in turn act as a service requester to the business’ stock service, logistics service and billing service.

Also, occasionally the requirements of a service requester can only partially (or not at all) be fulfilled by the available services in the service repository. In this case, some combination of services may be able to fulfill the requirements.

Such a combination of services is referred to as a Composite Service. A composite service behaves as a regular service towards its requesters, with its own service description, binding and choreography. An illustration of a composite service is depicted in figure 2.4. A composite service encompasses a number of component services, and a coordinator, which orchestrates the communication with the component services.

Service composition is currently an active research subject. A great deal of approaches are being investigated, varying from manual process modeling (for example, WS-BPEL [4]) to approaches for fully automated composition and execution of services (For example, web service composition by HTN planning [40]).
WS-BPEL has been adopted by the industry as a web service composition standard, with several production-ready implementations (ActiveBPEL, Oracle BPEL process manager). However, automatic, run-time compositions are a debating and hot research issue, but there are still no industry standards that prescribe how to accomplish and support this form of composition [43].

Several approaches to automatic service composition have been recently investigated, but the problem appears to have no trivial solution. Some approaches that use some kind of AI planning technique to generate an execution plan or process model from scratch according to services inputs, outputs, postconditions and effects (IOPE) work, but scalable algorithms to synthesize the required control constructs are not easy to provide [41].

On top of this, classical AI planning techniques are not directly applicable to the problem of web service composition, for a number of reasons [41] [35]:

The web services infrastructure is unreliable: services in the repository may no longer be available, or there might be (temporary) problems to reach the service. Also, services might not return the expected results. Therefore, the employed planning technique must be able to deal with non-determinism. Furthermore, it might also be necessary to remake a part of the plan if an essential service in the plan is not working as expected.

Also, the messages and objects manipulated by web services can be much more complex then the simple objects used in a classical AI planning domain. Objects may also be created by services at runtime; most classical planners do not support this.

### 2.7 Available standard initiatives

In order to determine the relevant semantic web service initiatives, various state-of-the-art reports ([26] [45] [18]) and surveys of relevant technologies for semantic web services ([14] [5]) were consulted. Based on these documentes we found four initiatives relevant to our research. These are described in more detail in the following subsections.

#### 2.7.1 OWL-S

OWL-S is an ontology designed to support the description of web services. The goal of this ontology is to enable automatic web service discovery, invocation, composition and monitoring of the execution of web services [43].

According to [43], OWL-S provides three types of knowledge about a web services (see Figure 2.5):

- The *Service Profile* describes what a service requires from its users, and what it provides to them.
- The *Service Model* describes how a service behaves to its users.
- The *Service grounding* is a description how to access the service in terms of messages, protocols, etc.
The most important element for the discovery of services, and the composition of them is the Service Profile. Essential components of the profile are the specification of what functionality the service provides and the specification of the conditions that must be satisfied in order for the service to produce a successful result. In addition, the profile specifies what conditions result from the service, including the expected and unexpected results of the service activity.

The OWL-S Profile represents two aspects of the functionality of the service: the information transformation (represented by inputs and outputs) and change in the state of the world caused by the execution of the service (represented by preconditions and effects).

The Service Profile supplies sufficient information for reasoning software to discover services, and generate compositions of services. In addition to this, OWL-S also allows for compositions of services to be modeled within the ontology. Service can be modeled as processes, which might in turn be composed of other processes. The process ontology of OWL-S is depicted in Figure 2.6.

Processes might be either atomic or composite. Atomic processes correspond to the actions a service can perform by engaging it in a single interaction; composite processes correspond to actions that require multi-step protocols and/or multiple server actions. They are decomposable into other (non-composite or composite) processes. This decomposition can be specified using a number of different control constructs.

### 2.7.2 SWSF

The Semantic Web Services Framework is an effort to realize the Semantic Web Services vision as described by [29] by adding semantics to web service descriptions. Battle et al. envision that richer semantics can support various activities, such as greater automation of service selection and invocation; automated translation of message content...
2.7. Available standard initiatives

between heterogeneous interoperating services; automated or semi-automated approaches to service composition; more comprehensive approaches to service monitoring and recovery from failure; and better automation of verification, simulation, configuration, supply chain management, contracting, and negotiation for services \[5\].

The SWSF initiative consists of 2 major components:

1. The Semantic Web Services Ontology (SWSO), which is a conceptual model - including a formal characterization - by which Web Services can be described.

2. The Semantic Web Services Language (SWSL), which is the language by which SWSO concepts can be described.

The SWSF initiative has taken the OWL-S ontology as a starting point for their work. As a result, their ontology (called FLOWS) is structured similarly to OWL-S: it has similar top-level components:

- **Service Descriptors**, which contains information about the service (used for discovery).
- **Process Models**, which define the service behaviour.
- **Grounding**, which defines the mapping from abstract services onto concrete services, including the technical details to reach these concrete services.
SWSF is a quite extensive standard, and has also been quite recently released. For this reason, tool support is very limited, and no implementation environments are available yet which implement this standard.

### 2.7.3 WSDL-S

WSDL-S is an evolutionary and backwards compatible extension of the existing Web services standards and descriptions language, which augments the expressivity of WSDL with semantics in an arbitrary semantic representation language [2]. WSDL-S designers believe that extension of an industry standard, retaining backwards compatibility, is a more practical approach for widespread adoption than introducing a complete new language for web service representation.

As a result, WSDL-S can be easily implemented in existing web service environments, but the drawback is that WSDL-S in itself does not provide support for the activities described in the previous sections: WSDL-S provides a means to supply semantic information, but actual semantic functionality has to be provided by additional components, which are not part of the WSDL-S initiative.

### 2.7.4 WSMO

WSMO refines and extends the Web Service Modeling Framework described in [19]. The ultimate goal of WSMO is to provide a standard which enables automation of service discovery, selection, composition, etc.

WSMO designers foresee that the Internet will be transformed from an information repository into a world-wide system for distributed web computing. In accordance with this vision, Bruijn et al. [12] acknowledge that a suitable framework for Semantic Web services needs to integrate basic web principles, principles defined for the Semantic Web, as well as design principles for distributed, service-oriented computing on the Web.

The following principles are the foundation of WSMO’s design:

- **Strict decoupling**: the open nature of the Internet implies that resources may appear and disappear at random. Therefore, all resources in WSMO are defined completely separated from others, to prevent interdependence between resources.

- **Centrality of mediation**: The open nature of the internet also implies that there are all sorts of interoperability problems to be solved in order to enable meaningful communication and interaction. Mediators are a top level element in WSMO, enabling interoperability problems to be tackled at an architectural level.

- **Compliance with web technology**: WSMO uses URI as resource identifiers, and employs XML and other W3C recommended technologies.

- **Ontology-based**: Being a widely accepted method of knowledge representation, ontologies have been identified as the central enabling technology for the semantic web.
2.8 Standards comparison

<table>
<thead>
<tr>
<th></th>
<th>OWL-S</th>
<th>SWSF</th>
<th>WSDL-S</th>
<th>WSMO</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fadlisyah et al.</td>
<td>Limited</td>
<td>Yes</td>
<td>Limited</td>
<td>Yes</td>
</tr>
<tr>
<td>Kolter et al.</td>
<td>Yes</td>
<td>-</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Vrijkorte</td>
<td>Limited</td>
<td>Limited</td>
<td>-</td>
<td>Limited</td>
</tr>
</tbody>
</table>

Table 2.1: Web service formalism automated discovery capabilities according to [26], [45] and [18] (- = not evaluated)

<table>
<thead>
<tr>
<th></th>
<th>OWL-S</th>
<th>SWSF</th>
<th>WSDL-S</th>
<th>WSMO</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fadlisyah et al.</td>
<td>Limited</td>
<td>Limited</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Kolter et al.</td>
<td>Yes</td>
<td>-</td>
<td>No</td>
<td>-</td>
</tr>
<tr>
<td>Vrijkorte</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 2.2: Web service formalism automated composition capabilities according to [26], [45] and [18] (- = not evaluated)

The WSMO conceptual model has four top-level elements:

- **Ontologies**, which can be used to define terminology and its meaning, enabling meaningful interaction between other elements.

- **Goals**, which represent the desire of a user in terms of capabilities requested of a web service.

- **Web Services**, which represent web services in terms of their capabilities.

- **Mediators**, which can be designed to solve interoperability problems between other elements.

The conceptual model can be formally described by a language family called the Web Service Modeling Language (WSML).

2.8 Standards comparison

In order to choose a suitable formalism, a number of state-of-the-art reports concerning semantic web technology were consulted. Kolter et al. [26] present a practical analysis of strengths and weaknesses of OWL-S, WSMO, WSDL-S and BPEL4WS, and available frameworks/toolsets for these initiatives. The analysis was done in the context of applicability in an e-government scenario. Vrijkorte [45] assesses the usefulness of OWL-S, WSMO and SWSF, including practical applicability and tool support. Fadlisyah et al. [18] discuss the support of OWL-S, WSMO, SWSF and WSDL-S with respect to service interoperability, with a focus on semantic web services.

Relevant conclusions from these documents related to automated discovery and composition have been summarized in Tables 2.1 and 2.2.
We concluded from these evaluations, that OWL-S, SWSF and WSMO have some support for both automatic discovery and automatic composition. WSMO is the only initiative which has an explicit notion of mediation. Furthermore, WSMO is the only standard for which there exist several implementation environments which aim to support the complete standard. For these reasons WSMO is used as our semantic web technology throughout the rest of this thesis.
Chapter 3

The Web Service Modeling Ontology

This chapter elaborates on the Web Service Modeling Ontology (WSMO). This will give the reader insight in how WSMO designers plan on solving the challenges in semantic web services.

The chapter is further structured as follows: Section 3.1 places WSMO in its context; section 3.2 presents an overview of WSMO; Section 3.3 elaborates on the top level elements ontologies, goals, web services and mediators; Section 3.4 briefly introduces the language used to describe WSMO elements; Section 3.5 presents the implementation environments available that implement the WSMO standard.

3.1 Background

WSMO is being developed and designed by the Digital Enterprise Research Institute (DERI). DERI’s mission is to exploit semantics for people, organisations and systems to collaborate and interoperate on a global scale. WSMO is one of their projects, and it aims at describing all relevant aspects related to services with the ultimate goal of automating the tasks of discovery, selection, composition, etc. DERI believes this will be a major step forward in e-commerce and business integration.

The basis of WSMO lies in the Web Service Modeling Framework (WSMF) [19]. WSMF provides a modeling framework for the development and the description of services. This modeling framework is based on two complementary principles:

1. Strong de-coupling of all components that comprise an application. This avoids any interdependence between components, and is therefore suited to model an application which consists of components (services) over which the application designer has no direct control.

2. Strong de-coupling is complemented by a strong mediation service, which is aimed to enable meaningful communication between all de-coupled components. With these
3.2. Overview

In order to position WSMO and its elements, we will use the MetaObject Facility (MOF) [34]. MOF is a metamodeling architecture originally developed to define the Unified Modeling Language (UML) [33]. MOF can be used to specify an architecture in different layers, in increasing level of concreteness. Figure 3.1 illustrates this with a simple 4 layer model, which is a UML model of a video repository:

- The m0-layer, or information layer, comprises of physical objects. In this case the only physical object is the dvd of the movie Office Space.
- The m1-layer, or model layer, consists of a model of the real world.
- The m2 layer is the meta model, or the language to describe the model in the m1 layer. In our example, the M2 layer is the UML.
- The m3 layer is the meta-meta model, which is the language to describe the meta-model. This is the MOF.

WSMO can be positioned in a similar fashion. In terms of the four MOF layers, WSMO itself constitutes the meta-model (m2) layer. Ontology, web service, goal and mediator specifi-
3.3. Top level elements

WSMO, like WSMF, identifies the four top-level elements ontology, goal, web service and mediator, as the main concepts which have to be described in order to define semantic web services.

- Ontologies provide the terminology, used by other WSMO elements to describe the relevant aspects of the domains of discourse, and also give this terminology a formally defined meaning.
- Web Services represent computational entities able to provide access to services.
- Goals describe aspects related to user desires with respect to functionality requested of a service.
- Mediators describe elements that handle interoperability problems between different WSMO elements.

The relations between these elements are depicted in Figure 3.3.

3.3.1 Ontologies

In the context of information technology, an ontology is a “formal explicit specification of a shared conceptualization” [20]. More specifically, an ontology defines concepts and
relations between those concepts. As such, it can be used to make sure different entities (for example, Web Services) share a common terminology.

To illustrate how an ontology is used in this context, imagine an airline ticket booking service provider. Such a provider may, for example, describe their service as a service which sells a flight with a certain departure location and a certain destination location, or perhaps as a trip which takes someone from A to B.

Although the terms 'destination' and 'to' are different, they refer to the same concept: the place where the plane is going to land. An ontology is used to offer a consensus on the terms which are used to designate certain concepts. Stating that a travel agent and an airline use the same ontology, essentially means that the travel agent and the airline can understand each other, because they are sharing the same conceptualization.

Ontologies in WSMO consist of concepts, relations, functions and instances. Concepts are the basic elements of the terminology (i.e. 'destination'), instances are concrete instances of these concepts (i.e. 'Amsterdam'), relations model interdependencies between the specified concepts (i.e. 'A flight is a specific kind of trip') and functions are special relations, which can describe more complex relations between concepts. (For example a function that determines the approximate trip duration, given a certain trip distance and vehicle).

The creation and maintenance of an ontology for a broad problem domain can be challenging. Therefore, WSMO is designed to make modular design and use of ontologies possible: ontologies can be imported, allowing the construction of one large ontology for a specific target from smaller, more domain specific ontologies. Importing ontologies right away requires that there are no conflicts between the imported ontologies.

Even when conflicts do exist between ontologies, these ontologies can - in principle - be used together, by employing one or more ontology mediators.
3.3. Top level elements

3.3.2 Web Services

In WSMO, a Web Service is defined by specifying what the web service can do. This definition is given in terms of the capabilities of the web service. A capability is in turn defined by preconditions and postconditions, specifying the information space of the web service before and after execution of the web service, and by assumptions and effects, describing the state of the world before and after execution.

Continuing the airline service provider example, a web service describing a flight from Amsterdam to Moscow will have an assumption like ‘the customer is in Amsterdam’, and an effect ‘the customer is in Moscow’. Pre- and postconditions are similar descriptions, but they specify constraints on information which the service needs as input (for example, the customer’s passport number), and will give as output (for example, a reservation number).

Apart from the capabilities of the web service, a web service definition also includes the web service’s interface. The interface describes how the capabilities of a web service can be fulfilled, by describing the service’s choreography.

3.3.3 Goals

The counterpart of a Web Service in WSMO is a Goal. A goal defines an objective which a user seeks to accomplish, and it is described in the same terms as a web service, namely the capability and the interface which the user desires. Thus, essentially, a user goal is represented as the description of a web service which would potentially fulfill his desire.

3.3.4 Mediators

Mediators in WSMO handle a variety of interoperability problems between WSMO elements. They are expected to link reusable vocabularies and interaction styles, facilitating the integration of services offered by different service providers [13].

WSMO defines four types of mediators, which offer an increasing amount of mediation support: OO Mediators offer data level mediation; GG Mediators use OO Mediators for data level mediation, and offer Δ-mediation, WG Mediators extends this by offering mediation on the communication level, and WW Mediators solve heterogeneity problems between web services in case of web service composition.

OO Mediators

In WSMO, all aspects related to Semantic Web Services have to be semantically described using ontologies. A creator of such descriptions has the choice of creating their own ontology or to reuse already existing ones. As the ontologies are “shared conceptualizations”, reusing ontologies should be the first choice in any modelling process. As a result, it might be necessary in some occasions to refer to multiple ontologies that can describe overlapping domains using different concepts and even conflicting models.
Mocan et al. [30] argue that an OO Mediator should offer one of the following pieces of functionality:

- Instance transformation based on a set of mappings identified between the input ontologies.
- Query rewriting from the terms of one ontology in terms of the target ontology. This might also include the task of merging the instances of two mediated ontologies to suppress duplicates.
- Make available the rules to convert one ontology to the other.

**GG Mediators**

Similarly to ontologies, goal and web service description in WSMO are envisioned to be reusable. Mocan et al. argue that it is not efficient to store all goals and webservice descriptions separately, but propose to infer some of these by combining some goals and web services with $\Delta$-relations, which define the logical differences between goals and web services. This allows one to infer goals from a much smaller repository, leading to more efficient resource management.

For example: Consider two Goals: G1 defines “buy product”, and the target Goal G2 defines “buy ticket”, whereby “ticket” is sub-concept of “product” in the used domain ontology. Now consider a Web Service WS1 for purchasing train tickets as a sub-class of ‘tickets’. Given G1 and WS1, the $\Delta$-relation in a respective GG Mediator specifies “all products that are not train tickets” as the explicit logical difference between G1 and WS1. Having also given the $\Delta$-relation between G1 and G2 as “all products that are not tickets”, we can determine the $\Delta$-relation between G2 and WS1 on the basis of the known $\Delta$-relations between G1, G2, and G1, WS1 [36]

**WG Mediators**

In addition to the functionality of the OO and GG Mediators, WG Mediators resolve mismatches between a Web Service and a Goal on the communication level. That is, resolving problems in the message exchange between service requesters and providers.

Fensel et al. [19] identify two cases where message exchange might fail.

- Resolvable message mismatch. This case appears when the two partners use different exchange patterns, but several transformations can be performed in order to resolve the mismatches For example when one partner sends more than one concept in a single message, but the other one expects them separately. In this case the mediator can split the initial message and send the concepts one by one.

- Unresolvable message mismatch. In this case, one of the partners expects a message that the other one does not intend to send (for example, an acknowledgement). Unless the mediator can provide this message, the communication reaches a dead-end (one of the partners is waiting indefinitely).
3.4. The WSMO language: WSML

According to the architecture in [10] a WG Mediator may resolve the following message mismatches:

- Stopping an unexpected message - If one of the partners sends a message that the other one does not want to receive, the mediator should just retain and store it. This message can be sent later, if needed, or it can just be deleted after the communication ends.

- Inversing the order of messages - If one of the partners sends the messages in a different order than the other partner expects, the messages that are not yet expected will be stored and sent when needed.

- Splitting a message - If one of the partners sends multiple sets information in a single message that the other one expects to receive in different messages, the information can be split and sent in a sequence of separate messages.

- Combining messages - If one of the partners expects a single message, containing information sent by the other one in multiple messages, the information can be combined into a single message.

- Sending a dummy acknowledgement - If one of the partners expects an acknowledgement for a certain message and the other partner does not intend to send it, even if it receives the message, an acknowledgment can be automatically generated and sent to the partner which requires it.

WW Mediators

WWMediators, have the task of solving heterogeneity problems between Web Services. This heterogeneity may occur when a Web Service needs to combine (invoke) multiple Web Services in order to achieve its capability. A WWMediator operates between the Orchestration of this Web Service and the choreographies of the invoked Web Services.

However, WW Mediators have not been defined yet, since there are other prerequisites in the WSMO standard (specifically, WSMO orchestration) that have not been completely defined yet.

3.4 The WSMO language: WSML

In order to formally described all elements defined in WSMO, the Web Service Modeling Language (WSML) [15] was designed. WSML is in fact a family of languages, consisting of WSML-Core, WSML-DL, WSML-Flight, WSML-Rule and WSML-Full. These language variants are expected to be used in different circumstances and therefore differ in terms of logical expressiveness and the use language paradigms.

These variants are offered, to allow users to make a trade-off between the provided expressivity and the implied complexity on a per-application basis. Figure 3.4 illustrate the
relations of the respective variants: The basic language WSML-Core is extended in two directions, namely, Description Logics (WSML-DL) and Logic Programming (WSML-Flight, WSML-Rule). WSML-Rule and WSML-DL are both extended to a full First-Order Logic with nonmonotonic extensions (WSML-Full), which unifies both paradigms.

All WSML variants are specified in terms of a human-readable syntax with keywords similar to the elements of the WSMO conceptual model. Furthermore, XML and RDF exchange syntaxes are provided, as well as a mapping between WSML ontologies and OWL ontologies for interoperability with OWL-based applications.

Figure 3.5 shows a simple web service description expressed in WSML-Flight. The web service provides airline tickets between any european city.

The web service description uses the ontology travelDomainOntology from the namespace urn:wsmx.example.org/testcase1/ontologies#. Its capability is defined by a postcondition which states that the end result is a ticket (which is defined in the ontology as something encompassing a departure location (from), arrival location (to) and a vehicle type (vehicle)). Furthermore, the postcondition defines that both from and to must be member of EuropeanCity, and that the vehicle must be an Airplane.

For a complete language reference of WSML we refer to 15.

3.5 Implementation environments

At present, two implementation environments are available that are WSMO-compliant, and that support most of the activities described in Chapter 2, the Internet Reasoning Service
wsmlVariant _"http://www.wsmo.org/wsml/wsml-syntax/wsml-flight"

namespace _"urn:wsmx.example.org/testcase1/webservices#",
    dO _"urn:wsmx.example.org/testcase1/ontologies#",
    dc _"http://purl.org/dc/elements/1.1#"

webService flyeuropeTicketService

   nfp
       dc#description hasValue "Ticket booking service for flyeurope airline tickets"
   endnfp

   importsOntology dO#travelDomainOntology

   capability flyeuropeTicketServiceCapability

   postcondition
       definedBy
           ?ticket[dO#from hasValue ?from,
                   dO#to hasValue ?to,
                   dO#vehicle hasValue ?vehicle] memberOf dO#Ticket
           and ?from memberOf dO#EuropeanCity
           and ?to memberOf dO#EuropeanCity
           and ?vehicle memberOf dO#Airplane.

Figure 3.5: Example web service description in WSML-flight

3 (IRS-III [25]), and the Web Service Execution Environment (WSMX [21]).

3.5.1 IRS-III

IRS-III (Internet Reasoning Service 3) is a Semantic Web Services framework developed by the Knowledge Media Institute, which allows applications to semantically describe and execute Web Services. IRS supports the provision of semantic reasoning services within the context of the Semantic Web.

According to [17], IRS-III has four features, that distinguishes it from other Semantic Web Service-related efforts.

1. IRS-III supports “one-click publishing” of LISP and Java programs. This feature transforms arbitrary programs into web services, by automatically generating appropriate wrappers.

2. The WSMO goal and web service concepts have been extended, allowing the direct invocation of Web Services by supplying a goal. Domingue et al. call this feature capability-driven service execution.
3. The framework is programmable in the sense that IRS-III components of can be replaced by user-defined semantic web services.

4. IRS-III services are web service compatible. Standard web services can be trivially published through the reasoning service, and any IRS-III service automatically appears as a standard web service to other web service infrastructures.

IRS-III is composed by the main following components: the IRS-III Server, the IRS-III Publisher, IRS-III Editors and the IRS-III Client, which communicate through a SOAP-based protocol. The collaboration of these components is shown in Figure 3.6.

The IRS-III Server is based on an http-server, which has been extended with a soap handler. Separate modules handle SOAP-based requests from the browser, the publishing platforms and the invocation clients. Messages result in a combination of queries or changes to the entities stored within the WSMO library.

The IRS-III Browser and Editors provide simple interfaces enabling the creation and edition of WSMO based descriptions. The Browser provides multiple visualizations of the semantic descriptions, which can be navigated by direct manipulation. The editor visualises related WSMO components into single multiple tabbed windows. (For example, a web service window contains the associated interface, capability, choreography and orchestration definitions).

Publishing with IRS-III entails associating a deployed web service with a WSMO web service description. When a web service is published in IRS-III, all of the information necessary to call the service (the host, port and path) is stored within a publisher-information class associated with the web service’s interface. Additionally, updates are made to the
appropriate publishing platform. IRS-III contains publishing platforms to support the publishing of standalone Java and Lisp code and of web services. More detail on the IRS-III can be found in [16].

3.5.2 WSMX

The Web Service Execution Environment (WSMX) is the WSMO reference implementation. It is a software system aimed to be a semantic implementation environment capable to consume semantic messages, discover semantically described web services, and to invoke and compose these services for the end-user’s benefit. As such, it enables the creation and execution of Semantic Web Services based on WSMO. Service providers can use it to register and offer their services and requesters can use it to dynamically discover and invoke relevant services [21]. WSMX has the following features:

- Storage service to store goal, web service, ontology and mediator descriptions, as well as other data.
- Service discovery, matchmaking and selection, based on reasoning on web services and goals.
- Data and process mediators
- Communication managers, allowing service enactment with prescribed choreography.
- Allows communication via SOAP. Other communication protocols can be supported by providing an appropriate adapter plugin.
- Extendable and modifyable due to modular design.

Information on the architecture can be found in chapter 4. More elaborate information is presented in [46].

3.6 Comparison

Both IRS-III and WSMX seem suitable candidates for the purpose of this research. There are three compelling arguments to choose WSMX, however.

1. The modular design of WSMX should allow for the addition of new features.
2. WSMX is the WSMO reference implementation, which guarantees that the framework will not deviate significantly from the proposed standards.
3. The source IRS-III has not been published by the authors. It might be possible to get a license to work with the IRS-III source for the purpose of evaluation, but given the other arguments in favor of WSMX this was not attempted.

For these reasons, WSMX is used as implementation environment throughout the rest of this thesis.
Chapter 4

The Web Service Execution Environment

In this chapter, the WSMX architecture and operation is presented, and the implementation of discovery and composition components is considered in detail. The goal of the chapter is to show the current state of development of the environment, and to provide insight in how WSMX solves certain challenges.

The chapter is structured as follows: in Section 4.1, the general architecture is outlined; Section 4.2 describes the usual WSMX execution flow; Sections 4.3 and 4.4 discuss the implementation of the Discovery component and the possibilities for automatic web service composition in WSMX.

4.1 Architecture

The complete WSMX package consists of a core, a number of data and communication protocol adapters, and a number of components that provide actual functionality to the system. The currently available components and adapters, and their relation are depicted in Figure 4.1.

4.1.1 Core

The WSMX core provides an environment for semantic web service execution, with a modular design, that facilitates the modification and extension of the WSMX environment. A running WSMX core (also called a WSMX instance) is managed and monitored using JMX management technologies [42].

The WSMX core can be interfaced through Adapters, which are described in detail below. The actual functionality of the system is achieved via WSMX components. The WSMX core documentation describes an information model along with interfaces for components required to implement this information model. The current component interfaces and the level of their implementation is briefly described below.
4.1.2 Adapters

An adapter facilitates communication between WSMX and something else. Adapters are responsible for transforming data from and to WSML.

Currently, three adapters have been implemented: the AxisAdapter, the HttpAdapter, and the SshAdapter. The HttpAdapter and the SshAdapter allow for the management of a WSMX instance, and support the invocation of WSMX component’s methods (for example, for testing purposes). The AxisAdapter enables access to WSMX functions via SOAP/HTTP and is as such more suited for production use.

SshAdapter

The ssh adapter enables basic management of a WSMX instance with a standard ssh client. It provides a simple command line interface, allowing the user to perform a limited amount of management functions.

HttpAdapter

The HTTP adapter provides a web-based management console, which enables control of WSMX via a standard web-browser. An example of interfacing with WSMX through the HTTP adapter is depicted in figure 4.2.
4.1. Architecture

Figure 4.2: The HTTP adapter.

AxisAdapter

The Axis adapter enables interfacing with WSMX using standard web service-technology (i.e., SOAP/HTTP). The adapter exposes WSMX functions as standard web services, which can be invoked by any web service-compliant service requester. As such, the AxisAdapter enables interoperability between WSMX and other web services.

4.1.3 Components

The documentation of WSMX components varies heavily in quality and availability. Official publications are clear, but are often outdated. Furthermore, for a lot of functionality there is no documentation publicly available yet. For this reason, the description of the current WSMX components used in this project has been taken from the WSMX API-documentation and directly from their implementation, by reading and interpreting the sourcecode.

Resource Manager

The Resource Manager is a storage component. It is responsible for storing all data used by WSMX, such as ontologies, web service descriptions and goals. The current implementation of the Resource Manager actually does not store data on a persistent medium, but it keeps it only in memory. Therefore, data has to be re-registered after a WSMX instance has been restarted.
Communication Manager

The Communication Manager is responsible for sending and receiving messages to entities external to WSMX. The Communication Manager consists of an Invoker and a Receiver interface. The Invoker interface sends messages to external entities, whereas messages from external entities enter a WSMX instance through the Receiver interface.

A Communication Manager does not perform any translation from WSML to other languages and vice versa. For translation purposes, the communication manager uses appropriate adapters.

Parser

The Parser is responsible for the syntactic validity checks of WSML documents. It determines whether a WSML document can be processed and converts the document to the internal representation used by WSMX, enabling other WSMX-components to use it.

Discovery

The Discovery component is responsible for web service discovery. It allows a user to search the WSMX service repository for web services that match a specified goal. The implementation of the Discovery component is considered in detail in Section 4.3.

Selector

The selector component has designed to allow the selection of the most appropriate web service in case the Discovery component has returned multiple matching service descriptions. There is no implementation of this component available yet.

Data Mediator

A Data Mediator provides support for instance transformation between ontologies. That is, instances expressed in terms of some source ontology are transformed in instances expressed in terms of the target ontology, according to some provided mapping between source and target ontology.

Data level mediators should be implemented by creating a web service which performs the actual mediation/instance transformation.

Process Mediator

A WSMX Process Mediator has the role of reconciling the public process heterogeneity that can appear during the invocation of web services. That is, ensuring that the public processes of the invoker and the invoked Web Service match. Since both the invoker and
the web service publish their public processes as choreographies, and the public processes are executed by sending/receiving messages, the ProcessMediator Component will deal with reconciliation of message exchange patterns based on choreography [16].

**ChoreographyEngine**

The choreographyengine keeps track of the state of a communication with a web service using a state machine. Web services and Goals with a choreography description can be registered with the engine. Further messages between requester and provider either cause the engine to update its state in case a message is valid within the registered choreography, or result in an error in case a message is not legal within the service’s choreography.

**Reasoner**

The reasoner component is responsible for reasoning about WSML messages. The current component is implemented as a wrapper around an external WSML reasoner. It is currently able to register ontologies, determine all super- or subconcepts of a given concept, determine whether there is a subsumption match between two concepts, and determine whether a given instance is an instance of a given concept.

**4.2 Operation**

To illustrate the roles the WSMX components play in the usage of WSMX, the basic WSMX execution flow is depicted in Figure 4.3. WSMX defines three modes of operation:

1. **Registration**: In registration mode service providers and requesters register their services, goals, ontologies and mediators with WSMX. When registering some instance, a WSML document is provided to the running WSMX instance via the Communication Manager. The document is then processed by the Parser, and if the document is valid, its internal representation is stored in WSMX’s persistent storage using the Resource Manager.

2. **Discovery**: After wsml documents have have been registered, WSMX can be used to perform discovery. Service requesters send a discovery request to some WSMX adapter, which in turn translates the message to WSML format. This WSML document is passed on to the Receiver component, then validated using the Parser, and finally put into the Discovery framework, which performs the actual discovery activity.

3. **Invocation**: In this mode, WSMX invokes one or more web services on behalf of a service requester. In this position, WSMX may for example provide mediation services.

In case of an invocation, a WSML document from the requester is parsed, and then sent back into the communication manager, which will in turn invoke a service at a
4.3 Discovery component

In this section, the Discovery component of WSMX is considered in more detail. This detailed description is necessary, since web service discovery is one of the prerequisites for web service composition. Shortcomings in the discovery system would immediately cause difficulties in automated composition, and the absence of a working discovery mechanism would make automated composition impossible.

The discovery component is currently shipped with four engines which provide different kinds of discovery:

1. A keyword-based discovery engine, which discovers services based on keyword matching against the service’s non-functional description fields.

2. A lightweight discovery engine, which provides semantic discovery on a service description level.

3. An instance-based discovery engine, which allows the discovery framework to actually invoke a web service for more fine-grained information about what the service can offer. This requires support on the service provider’s side for such inquiries.)
4.3. Discovery component

Figure 4.4: Discovery engines in the WSMX discovery framework

4. A QoS discovery engine, which is able to take certain QoS values into account while discovering. This too requires support on the service provider’s side for such inquiries.

Since the tasks these engines fulfill are on different level of granularity, the engines are supposed to be used in a particular order. This order is depicted in Figure 4.4.

First, the non-functional-properties (NFP’s) of the goal are checked for properties that prescribe that instance-based discovery should be performed. If this is the case, the instance-based discovery engine is invoked to generate a preliminary list of matches. If not, the preliminary list of matches will be generated by the keyword-based discovery engine. This engine acts as a coarse filter to limit the input to the lightweight discovery engine. Subsequently, the lightweight discovery engine is invoked to produce a more refined list of matches.

Subsequently, the framework checks the goal’s NFP’s for properties which indicate a QoS refinement should be done. If so, the preliminary list of matches is refined using the QoS discovery engine. Finally, the remaining matches are returned.

Lightweight discovery

The lightweight discovery engine performs the actual semantic discovery of services that have a WSML description. It uses a reasoner to detect exact and partial matches between the supplied goal and the available services, and returns a list of matches.
4.3. Discovery component

Keyword-based discovery

Semantic discovery currently demands a great deal of processing power. For example, the current implementation of the semantic discovery engine takes seconds to check for a match between one service and one goal. Doing semantic discovery through a service repository of hundreds or thousands of services would take far too much time. Therefore, the discovery framework contains a keyword-based discovery engine, which is used to quickly filter and rank large amounts of service descriptions based on keywords in the goal and service descriptions. Based on matches of keywords, this engine yields a number of results that can be further refined using a semantic discovery approach.

The drawback of this approach is that services that would match when compared on a semantic basis may be discarded by the keyword-based engine. Despite this drawback, Kilgariff et al. [23] argue that keyword discovery is a suitable method for filtering irrelevant services for a certain discovery operation.

Instance-based discovery

This discovery engine extends the lightweight discovery engine in the sense that it automatically invokes the discovered web services in order to retrieve more specific information on a candidate service. The services are invoked according to interface descriptions and contracts defined in the web service descriptions.

This is useful to enable the discovery process to take more fine-grained dynamic content supplied by the web services into account. For example, an online bookstore service that sells all kinds of different books may have an offering that varies heavily. By using instance based discovery, the discovery engine is able to discover exactly what titles the bookstore has in stock, while the bookstore does not have to maintain and update an extensive semantic description of all books available in their store.

QoS-based discovery

The QoS-based discovery engine is a third party component (incorporated in the WSMX distribution) that allows refinement of the list of discovered web services using QoS descriptions. It is able to discover services matching a user’s QoS requirements, to capture and update web service QoS parameters via user reports, to rank web services based on their QoS compliance, and to parallelize the discovery on multiple computing nodes for better scalability.

Consider a crate shipping service. A requester needs a crate shipped from Europe to the USA within 7 days. The QoS discovery engine can use the additional 7 day time constraint as a factor in the search, in order to find crate shipping services who claim to be able to ship the crate within 7 days.

In addition to this, users can report actual delivery times of the crate shipping services to the discovery engine. This is used by the discovery engine to rank matching services
4.4. Service composition component

by QoS compliance. Crate shipping services which violate their QoS once in a while will then be ranked lower than the more reliable shipping services.

4.4 Service composition component

In the current state of development, there is no component available to provide (automated) service composition. There is no such component in the WSMX distribution, and one of the earlier papers on WSMX [21] explicitly state that a composition component is not yet being developed. Haller et al. state that the language to specify compositions is still under consideration in WSMO. They furthermore believe that adding composition support in a later stage should not be problematic due to WSMX’s component-based architecture.
4.4. Service composition component
Chapter 5

Implementing web service composition in WSMX

This chapter proposes an extension to wsmx to enable the framework to support web service composition.

The chapter is structured as follows: Section 5.1 proposes the capabilities that the extension must possess. Section 5.3 shows a high level architecture of the proposed extension, sections 5.4 and 5.5 elaborate on the planner and coordinator components.

5.1 Process models

The coordination of a composed service can be modeled as a process, which can be viewed as a system in which a sequence of transitions which change the state of the system. Depending on what transitions occur, the system will show some particular behaviour.

For the purpose of our service composition component, we consider two types of process models

1. A process model may describe states and transitions in terms of web services which need to be invoked, and their results. A simple example is depicted in figure 5.1. This process will invoke a specific service which will return an ISBN number of a book, given a title and an author. If one exists, it invokes an ordering service, which will order a book given a certain ISBN. As the model refers to concrete web services, it can be executed without the need for modification (provided all web services are available). Therefore, we will call this a concrete process model.

2. A similar process may be defined which contains goals instead of web services (see figure 5.2). For this process to be successfully executed, a discovery step must be executed to find a matching service for every goal which is encountered during the execution. As the model does not refer to concrete web services, we will call such a process model abstract.
5.2 Required capabilities

Implementing a composition component satisfying all prerequisites is a challenging and time-consuming task. Since the goal of the extension is to prove that the WSMX framework can be used for web service composition, only a basic composition component has been designed. Features such as transaction support and execution monitoring are omitted, as we believe that this functionality could be designed on top of the basic extension.

We define five activities which need to be supported in order to use both manual and automatic composition.

- **Syntactic validation of hand-crafted process models**
  This entails syntactically validating a process model in some process modeling language, and possibly converting them to some other representation which can be used by the coordinator.

- **Registration of validated process models**
  This entails storing the syntactically validated models, and making them available for later use. The stored process model then may be retrieved and used at some time in the future. This allows for the reuse of composed services.

- **Execution of concrete process models**
  This entails traversing the process model, invoking the services in the process model and making decisions in the process model based on the results of the invoked web services.
5.3. Architecture

The architecture of our extension to the WSMX framework has been designed with backwards compatibility, and minimal interference with the current architecture in mind. This led to a number of design choices:

- A composed service is, in essence, another web service with its own IOPE’s. Therefore the choice was made to embed the process model representing a composed service into a WSMO Web Service. In this way, the current execution flows for the services. Handling erroneous or unexpected output from the invoked services is also an important issue in the execution of a process model.

  - Execution of abstract process models
    Execution abstract process models resembles the execution of a concrete process model, but it requires additional discovery steps: For every needed service in the process model, the framework has to perform service discovery in order to find a suitable concrete service.

  - Generation of process models
    This entails generating a process model. This activity combines a number of services from a repository into a process model, which fulfills a supplied goal.
registration of WSMO elements and the flow for the discovery of WSMO web services can be used without modification.

- Generation and execution of a process model are separate tasks. Since the execution of a process model should be part of the invocation flow (it is after all the invocation of a web service from the requester’s perspective), and since the invocation flow should be altered minimally, we chose to leave process model generation out of the invocation flow altogether and add a complete new flow for model generation. This implies that a client has to make an explicit request for a service composition before it can invoke the composite service.

- There are many ways to implement coordination of the execution of a process model. The most straightforward way is to use one central coordinator that handles the communication with both the service requester and the composite services alone. Other, more distributed approaches are possible, and possibly better for various reasons. For example, parts of the execution of the process may be offloaded to a slave coordinator which is geographically (or latency-wise) closer to the services that need to be executed. Also, parallel tasks within a process may be delegated to other coordinators for the purpose of workload sharing.

However, since the complete WSMX framework acts as one central intermediary, it is a logical choice to implement the coordinator as a central component as well.

The proposed execution flow of WSMX’s invocation aspect is depicted in Figure 5.3. The change made to the original flow is the addition of a branch after the compiler component: if the Web Service invoked by the requester is not a composed service, the flow continues identically to the original execution flow. However, if it is a composed service, a new Coordinator component is activated, which coordinates the invocation of the composite services.

The proposed execution flow of the composition generation aspect is depicted in Figure 5.4. The client submits a goal, and receives a Web Service when the generation is complete. The actual model generation is performed by a new Planner component. This component retrieves all available Web Services from the Resource Manager, and generates a process model that leads to the fulfillment of the goal submitted by the requester. This model is stored in the repository, and a Web Service is returned to the requester. Subsequently, this Web Service can be invoked by the requester.

In order to make automatic service composition completely transparent to the requester, the composition generation can also be integrated into the discovery aspect. In this case, the framework should automatically generate and register a composition of services if the matchmaker cannot find a direct match. This is depicted in figure 5.5.

5.4 Plan-generation component

The plan-generation component is responsible for the generation of a process model from a specified WSML goal, and the web services provided in the WSMX service repository. The
component takes a WSML goal and a set of WSML services as input, and returns a new WSML service containing a process model that fulfills the input goal.

### 5.4.1 Planning algorithms

There is a research area dedicated to the automatic solving of planning problems. However, not all solutions from this area are equally applicable to service composition.

Although a complete analysis of suitable planning techniques is beyond the scope of this thesis, it is necessary to mention several methods to generate plans. Peer [35] provides a survey on formalization techniques and related planning techniques suitable for web service composition.

This paper introduces a number of requirements on a planner that are needed to perform useful web service composition. Amongst them are the ability to deal with incomplete information, the ability to dynamically introduce new objects in the planning phase, and the ability to cope with nondeterminism.
The planners under consideration in \cite{Peer2005} are implementations of well-established planning paradigms (State-Space based planning, Graph-based planning, Partial-Order Refinement planning, Satisfiability planning and Logic Programming), and implementations of more recent planning approaches that use control knowledge in the plan generation (Hierarchical Task Network planning (HTN), High-level Program Execution, Model Checking and Temporal Planning). Peer concludes that most classical planners are not able to generate a plan that fulfills the requirements for web service composition, but that HTN and program execution can deal with these problems.

Furthermore, several initiatives are under way to adapt or build planners for the explicit purpose of web service composition. For example, Sirin et al. \cite{Sirin2004} adapted the SHOP2 HTN planner to make compositions of web services described in OWL-S and execute these compositions. Klusch et al. \cite{Klusch2005} also uses a HTN planner to generate service composition plans of web services described in OWL-S. Albert et al. \cite{Albert2006} specifies and implements a prototype of a tool for goal-oriented composition for WSMO web services called GOBAC.

At present, GOBAC is the only planner/composer which employs WSMO web services. Unfortunately, it is not publicly available. SHOP2 and OWLS-Xplan\* are both publicly available.

Figure 5.4: Proposed flow for process model generation aspect.
5.4. Plan-generation component

5.4.2 Task decomposition

In order to be able to use one of the existing planning implementations, the planner component has been decomposed into three sequential tasks as depicted in Figure 5.6:

1. Translation from WSML to OWL-S.
2. The planning, which uses some algorithm to deduce a process model from the input goal and services described in OWL-S.
3. Translation of the process model generated by the planner back to WSML.

5.4.3 Language mappings

In order for the OWLS-Xplan* planner is used in the planner component, a language mapping is needed, either from WSML to PDDL, or from WSML to OWL-S. The formal mapping from WSMO to OWL-S is already described in [39]. Furthermore, a mapping tool to automate translation from WSML to OWL-S is planned for future versions of that document.
The generated process model must be embedded into a new WSML web service, and it should possibly converted to another language depending on the input required by the coordinator component.

### 5.4.4 Service composition example with OWLS-Xplan*

In order to show that generating a plan from a goal and a set of services is possible with current technologies, a test scenario was executed using the OWLS-Xplan* planner. We will describe this scenario in terms of the initial state of the world, the goals, and the available services in the next paragraphs, and show that OWLS-XPlan* can find a composition of services which fulfills the goal. The complete OWL-S source of this scenario is depicted in appendix A.

**Background**

In this scenario the planner must look for a composition of services, leading to the successful transportation of a patient from one hospital to another. To accomplish this, the patient should be transported to the nearest airport, transported by plane to the airport closest to his destination hospital, and transported from this airport to the hospital. A representation of the ontology for this problem domain is depicted in Figure 5.7.

There is a repository of service descriptions which operate on instances of (subclasses of) the concepts defined in the ontology. For the purpose of the composition example, some services require the availability a valid account (which can in turn be created by another service in the repository), or require the explicit registration of person details with the transport company. Some services offer normal transportation, others have the possibility of medical treatment during transportation. Which one will be used in the composition depends on the goal specification.

The services available to the composition algorithm are depicted in Table 5.1 (Concepts from the ontology are printed in italics).
5.4. Plan-generation component

Scenario

In the scenario a Patient called “Mikka” needs transport with medical treatment capabilities from the “Distrital Faro” hospital, to the “Eye and Ear Hospital”. He will be accompanied by a Person called “Hans Mustermann”. This information (and some other information irrelevant to this specific scenario) is formally described in the initial state as listed in Appendix A.1.3.

In the desired goal state, the following is accomplished:

- Found the nearest airport to the Distrital Faro, and marked it as the departure airport.
- Found the nearest airport to the Eye and Ear Hospital, and marked it as the arrival airport.
- Booked a medical vehicle transport from the Distrital Faro hospital to the departure airport for Mikka and Hans Mustermann.

Figure 5.7: The health-scallops domain ontology used for the composition example
5.4. Plan-generation component

<table>
<thead>
<tr>
<th>Service</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>BookFlight / BookFlight2</code></td>
<td>Allows users with a valid Account to book a Flight. It automatically charges the Creditcard specified at account creation time.</td>
</tr>
<tr>
<td><code>BookMedicalFlight / BookMedicalFlight2</code></td>
<td>Books a previously selected MedicalFlight for a Patient and an attendant (which is a Person).</td>
</tr>
<tr>
<td><code>FindNearestAirport</code></td>
<td>Finds the nearest Airport given an Address.</td>
</tr>
<tr>
<td><code>ProposeFlight / ProposeFlight2</code></td>
<td>Returns a Flight given a number of flight parameters such as the departure Airport and Time, the arrival Airport and Time, etc.</td>
</tr>
<tr>
<td><code>ProposeMedicalFlight / ProposeMedicalFlight2</code></td>
<td>Proposes a Flight with the capability of a specified medical treatment offered by the medical flight company.</td>
</tr>
<tr>
<td><code>RegisterPersonWithTransport / RegisterPersonWithMedicalTransport</code></td>
<td>Registers a Person with a previously requested VehicleTransport.</td>
</tr>
<tr>
<td><code>RequestTransport / RequestMedicalTransport</code></td>
<td>requests a VehicleTransport at a transport Company.</td>
</tr>
<tr>
<td><code>Create*Account</code></td>
<td>Services to create Accounts at the various companies.</td>
</tr>
</tbody>
</table>

Table 5.1: Available services for composition example

- Booked a medical vehicle transport from the the arrival airport to the Eye and Ear Hospital for Mikka and Hans Mustermann.

- Booked a flight from the departure airport to the arrival airport for Mikka and Hans Mustermann.

The composition algorithm will have to perform other tasks not specified in the goal state, in order to fulfill prerequisites of the required services. For example, the goal state does not specify that accounts must be created, however this is necessary in order to be able to book a flight.

Result

OWLS-Xplan* is able to find a composition which fulfills the described goal. The output of the planner is depicted in Figure 5.8 A statechart of the generated plan is depicted in figure 5.9

It can be observed that the planner booked the medical vehicle transports and the medical flight, and also executed a number of services which were required by he preconditions of the booking services such as the creation of accounts at the transportations and flight companies.
5.5 Coordinator component

The coordinator component is responsible for executing a process model previously generated by the planner component, or supplied by a user. The coordinator must parse a process model in some modeling language, invoke the web services according to the process model and evaluate the results.

Process models may be expressed in BPEL, for which several production-ready coordinator solutions are available. These include open source solutions which may under certain conditions be transparently integrated into the composition extension. (for example ActiveBPEL [1]).

Alternatively, processes may be modeled as an OWL-s Process. The OWL-s Process ontology has the necessary control structures such as sequence, split, split-join, any-order and choice (see figure 2.6).

Implementing a state machine or some other type of application that parses and executes services defined as an OWL-s process is quite similar to, for example, the operation of a
Figure 5.9: A simplified statechart of the process model generated by OWLS-XPlan*. BPEL engine. Although not completely trivial, this is hardly an academic challenge.
Chapter 6

Conclusions

This chapter presents the main contributions of this thesis, draws relevant conclusions and identifies points of future work.

The chapter is further structured as follows: Section 6.1 presents our general conclusions and summarizes the contributions of this thesis. Section 6.2 identifies issues for further work.

6.1 General conclusions

We have investigated the level of support of current implementation environments with respect to automatic web service discovery and composition. Our efforts towards this investigation included: (i) a literature study of available web service formalisms designed with support for automatic service composition and discovery, leading to the selection of WSMO as a suitable formalism, (ii) a survey of available implementation environments implementing WSMO, leading to the selection of WSMX as suitable implementation environment, (iii) deployment of the WSMX and evaluating it for support of automatic web service discovery, (iv) architectural design of an extension to WSMX in order for the environment to support web service composition, and (v) investigation of how to implement the extension into the WSMX codebase.

All off the investigated web service formalisms OWL-S, SWSF, WSDL-S and WSMO have at least limited support for automated web service discovery, whereas only WSMO and OWL-S have full support for web service composition.

The investigated implementation environment WSMX has proven to be able to solve certain web service discovery scenarios based on semantic descriptions of web service and goals. Also some use cases concerning more complex discovery with QoS constraint have been solved using WSMX. We conclude that WSMX is able to perform automated webservice discovery, however we question the suitability of the current implementation for real world operation. The semantic discovery operation takes a considerable amount of time, even when operating on only a few web service descriptions. In its current implementation,
WSMX limits the number of services which can be used in a semantic discovery to only 5, which is clearly insufficient for real-world applications.

WSMX does currently not support automated web service composition. We have shown on an architectural level that an extension to WSMX which enables composition is possible. However, implementing this extension and integrating it in the WSMX codebase has proven to be cumbersome for a number of practical reasons:

**Prerequisite software not available**

Some prerequisites are still absent: Working planners which understand WSML are not available yet, mappings between WSML and other languages, which are needed in case a non WSMO-native planner is used, are incomplete or unavailable. And even the mappings which are defined have not been transformed to a mapping tool yet.

Since building new conversion tools is outside the scope of this research, any prototype would rely solely on mockup components which uses hard-coded plans.

**Poor implementation documentation**

Whereas documentation on WSMX architecture and underlying conceptual model is abundant, the documentation on the actual implementation is largely absent, and the little available documentation is largely outdated. It is therefore only possible to design the extension on an architectural level, but building and plugging in an actual prototype is virtually impossible due to lack of information how and where to plug in a component.

**Poor source tree management**

The problem with documentation is aggravated by poor source tree management. The source tree contains outdated components, mockup components, and components which contain quick one-time-fixes to accommodate a certain use case defined elsewhere. This alone is not a problem, but the fact that the versioning system used to manage the source tree does not reflect these things is. It does not contain branches where components tailored to one specific goal are kept, nor does it contain tags to be able to consistently download one working release. On top of this, broken (not compiling) code is committed to the versioning system on a regular basis. This complicates working with the code to a large extent.

### 6.2 Future work

Numerous issues have to be resolved before semantic web service technology and specifically automated discovery and composition is ready for use in an application/production environment. The following list presents topics which need further investigation:
• Work has to be done to make mappings between OWL-S, WSMO, and possibly other planning and process-modeling languages. Well defined mappings, with appropriate tooling to convert one language in the other, will allow for the integration of existing planners within the WSMX framework.

• Other formalisms and implementation environment need to be evaluated; these might be better suited for automated discovery and composition already than WSMX in its current state is.

• Other existing SWS-related components have to be investigated and tested for performance. A lot of SWS-software is reportedly able to solve specific use cases, but for actual production use a test of performance and applicability to other, more general scenarios might be in order.

• Vast optimization of the current discovery algorithm implementations is necessary to cope with large service repositories as envisioned by SWS proponents.
6.2. Future work
Bibliography


Appendix A

Source code

A.1 Patient transport scenario

In the next subsections the owl-s source code of the patient transport scenario used in Chapter 5 is depicted.

A.1.1 Domain ontology

```xml
<?xml version="1.0"?>
<rdf:RDF
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema#
    xmlns="http://127.0.0.1/health-scallops/ontology/Health-Scallops_Ontology.owl#
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#
    xmlns:owl="http://www.w3.org/2002/07/owl#
    xml:base="http://127.0.0.1/health-scallops/ontology/Health-Scallops_Ontology.owl">
    <owl:Ontology rdf:about=""/>
    <owl:Class rdf:about="#Airport">
        <owl:Restriction>
            <owl:allValuesFrom>
                <owl:Class rdf:ID="Airport"/>
            </owl:allValuesFrom>
            <owl:onProperty rdf:ID="hasDepartureLocation"/>
        </owl:Restriction>
        <owl:Restriction>
            <owl:allValuesFrom>
                <owl:Class rdf:ID="Airport"/>
            </owl:allValuesFrom>
            <owl:onProperty rdf:ID="hasDestinationLocation"/>
        </owl:Restriction>
    </owl:Class>
    <owl:Class rdf:ID="Flight">
        <owl:Restriction>
            <owl:allValuesFrom>
                <owl:Class rdf:ID="Airport"/>
            </owl:allValuesFrom>
            <owl:onProperty rdf:ID="hasDepartureLocation"/>
        </owl:Restriction>
    </owl:Class>
    <owl:Class rdf:ID="Transport">
        <owl:Restriction>
            <owl:allValuesFrom>
                <owl:Class rdf:ID="FlightParameters"/>
            </owl:allValuesFrom>
            <owl:onProperty rdf:ID="hasParameters"/>
        </owl:Restriction>
    </owl:Class>
</rdf:RDF>
```
A.1. Patient transport scenario

A.1.2 Initial world state

<?xml version="1.0" encoding="windows-1252"?>

<!-- Created with Protege (with OWL Plugin 2.1, Build 284) http://protege.stanford.edu -->

A.1.2 Initial world state
A.1. Patient transport scenario

<?xml version="1.0" encoding="windows-1252"?>
<owl:Ontology rdf:resource="http://127.0.0.1/health-scallops/ontology/Health-Scallops_Ontology.owl"/>
<owl:Ontology rdf:resource="http://t27.0.0.1/health-scallops/ontology/Health-Scallops_Ontology.owl"/>
<owl:Ontology rdf:resource="http://www.w3.org/2003/11/swrl#"/>
<owl:Ontology rdf:resource="http://www.w3.org/2002/07/owl#"/>
<owl:Ontology rdf:resource="http://www.w3.org/2000/01/rdf-schema#"/>
<owl:Ontology rdf:resource="http://www.w3.org/2005/01/rdf-syntax-ns#"/>
<owl:Ontology rdf:resource="http://www.w3.org/2001/XMLSchema#"/>
<owl:Ontology rdf:resource="http://www.w3.org/2001/XMLSchema-instance#"/>
<owl:Ontology rdf:resource="http://www.w3.org/2001/XMLSchema annon#"/>
<owl:Ontology rdf:resource="http://www.w3.org/2001/XMLSchema#"/>
<owl:Ontology rdf:resource="http://www.w3.org/2002/07/owl#"/>
<owl:Ontology rdf:resource="http://www.w3.org/2000/01/rdf-schema#"/>
<owl:Ontology rdf:resource="http://www.w3.org/2002/07/owl#"/>
<owl:Ontology rdf:resource="http://www.w3.org/2000/01/rdf-schema#"/>
<owl:Ontology rdf:resource="http://www.w3.org/2002/07/owl#"/>
<owl:Ontology rdf:resource="http://www.w3.org/2000/01/rdf-schema#"/>
<owl:Ontology rdf:resource="http://www.w3.org/2002/07/owl#"/>
<owl:Ontology rdf:resource="http://www.w3.org/2000/01/rdf-schema#"/>
<owl:Ontology rdf:resource="http://www.w3.org/2002/07/owl#"/>
<owl:Ontology rdf:resource="http://www.w3.org/2000/01/rdf-schema#"/>
<owl:Ontology rdf:resource="http://www.w3.org/2002/07/owl#"/>
<owl:Ontology rdf:resource="http://www.w3.org/2000/01/rdf-schema#"/>
<owl:Ontology rdf:resource="http://www.w3.org/2002/07/owl#"/>
<owl:Ontology rdf:resource="http://www.w3.org/2000/01/rdf-schema#"/>
<owl:Ontology rdf:resource="http://www.w3.org/2002/07/owl#"/>
<owl:Ontology rdf:resource="http://www.w3.org/2000/01/rdf-schema#"/>
<owl:Ontology rdf:resource="http://www.w3.org/2002/07/owl#"/>
<owl:Ontology rdf:resource="http://www.w3.org/2000/01/rdf-schema#"/>
<owl:Ontology rdf:resource="http://www.w3.org/2002/07/owl#"/>
<owl:Ontology rdf:resource="http://www.w3.org/2000/01/rdf-schema#"/>
<owl:Ontology rdf:resource="http://www.w3.org/2002/07/owl#"/>
<owl:Ontology rdf:resource="http://www.w3.org/2000/01/rdf-schema#"/>
<owl:Ontology rdf:resource="http://www.w3.org/2002/07/owl#"/>
<owl:Ontology rdf:resource="http://www.w3.org/2000/01/rdf-schema#"/>
<owl:Ontology rdf:resource="http://www.w3.org/2002/07/owl#"/>
<owl:Ontology rdf:resource="http://www.w3.org/2000/01/rdf-schema#"/>
<owl:Ontology rdf:resource="http://www.w3.org/2002/07/owl#"/>
<owl:Ontology rdf:resource="http://www.w3.org/2000/01/rdf-schema#"/>
<owl:Ontology rdf:resource="http://www.w3.org/2002/07/owl#"/>
<owl:Ontology rdf:resource="http://www.w3.org/2000/01/rdf-schema#"/>
<owl:Ontology rdf:resource="http://www.w3.org/2002/07/owl#"/>
<owl:Ontology rdf:resource="http://www.w3.org/2000/01/rdf-schema#"/>
<owl:Ontology rdf:resource="http://www.w3.org/2002/07/owl#"/>
<owl:Ontology rdf:resource="http://www.w3.org/2000/01/rdf-schema#"/>
<owl:Ontology rdf:resource="http://www.w3.org/2002/07/owl#"/>
<owl:Ontology rdf:resource="http://www.w3.org/2000/01/rdf-schema#"/>
<owl:Ontology rdf:resource="http://www.w3.org/2002/07/owl#"/>
<owl:Ontology rdf:resource="http://www.w3.org/2000/01/rdf-schema#"/>
<owl:Ontology rdf:resource="http://www.w3.org/2002/07/owl#"/>
<owl:Ontology rdf:resource="http://www.w3.org/2000/01/rdf-schema#"/>
<owl:Ontology rdf:resource="http://www.w3.org/2002/07/owl#"/>
<owl:Ontology rdf:resource="http://www.w3.org/2000/01/rdf-schema#"/>
<owl:Ontology rdf:resource="http://www.w3.org/2002/07/owl#"/>
<owl:Ontology rdf:resource="http://www.w3.org/2000/01/rdf-schema#"/>
<owl:Ontology rdf:resource="http://www.w3.org/2002/07/owl#"/>
<owl:Ontology rdf:resource="http://www.w3.org/2000/01/rdf-schema#"/>
<owl:Ontology rdf:resource="http://www.w3.org/2002/07/owl#"/>
<owl:Ontology rdf:resource="http://www.w3.org/2000/01/rdf-schema#"/>
<owl:Ontology rdf:resource="http://www.w3.org/2002/07/owl#"/>
<owl:Ontology rdf:resource="http://www.w3.org/2000/01/rdf-schema#"/>
<owl:Ontology rdf:resource="http://www.w3.org/2002/07/owl#"/>
A.1. Patient transport scenario

xmlns:grounding="http://www.daml.org/services/owl-s/1.1/Grounding.owl#"
xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns: schema="http://www.w3.org/2001/XMLSchema#
xmlns:j= "http://127.0.0.1/health-scallops/ontology/Health-Scallops_Ontology.owl#"
xmlns:prefix= "http://www.daml.org/services/owl-s/1.1/Profile.owl#"
<owl: Ontology>
<owl: imports rdf: resource= "http://127.0.0.1/health-scallops/ontology/Health-Scallops_Ontology.owl#"
<owl: Ontology>
<j: Hospital rdf:about= "http://www.owl-ontologies.com/Ontology1.owl#DepartureHospital"/>
<j: Address rdf: ID=" ArrivalAirportAddress"/>
<j: Hospital rdf: about= "http://www.owl-ontologies.com/Ontology1.owl#DestinationHospital"/>
<j: Airport rdf: ID=" DepartureAirport"/>
<j: hasAddress/>
</j: Airport>
<j: Patient rdf: about= "http://www.owl-ontologies.com/Ontology1.owl#Patient_0"/>
<j: VehicleTransport rdf: ID=" TransportToHospital"/>
<j: hasDepartureLocation>  
<j: Airport rdf: ID=" ArrivalAirport"/>
</j: hasDepartureLocation>
<j: hasDestinationLocation rdf: resource= "http://www.owl-ontologies.com/Ontology1.owl#DestinationHospital"/>
<j: hasParameters>
</j: hasParameters>
<j: isBookedFor>
<j: Person rdf: about= "http://www.owl-ontologies.com/Ontology1.owl#EMAAttendant"/>
<j: isBookedFor rdf: resource= "http://www.owl-ontologies.com/Ontology1.owl#Patient_0"/>
</j: isBookedFor>
</j: VehicleTransport>
<j: Flight rdf: ID=" Flight"/>
<j: hasDepartureLocation rdf: resource= "# DepartureAirport"/>
<j: hasDestinationLocation rdf: resource= "# ArrivalAirport"/>
<j: hasParameters>
</j: hasParameters>
<j: isBookedFor>
<j: Person rdf: about= "http://www.owl-ontologies.com/Ontology1.owl#EMAAttendant"/>
<j: isBookedFor rdf: resource= "http://www.owl-ontologies.com/Ontology1.owl#Patient_0"/>
</j: isBookedFor>
</j: Flight>
<j: VehicleTransport rdf: ID=" TransportToAirport"/>
<j: hasDepartureLocation rdf: resource= "http://www.owl-ontologies.com/Ontology1.owl#DepartureHospital"/>
<j: hasParameters>
</j: hasParameters>
<j: isBookedFor>
<j: Person rdf: about= "http://www.owl-ontologies.com/Ontology1.owl#EMAAttendant"/>
<j: isBookedFor rdf: resource= "http://www.owl-ontologies.com/Ontology1.owl#Patient_0"/>
</j: isBookedFor>
</j: VehicleTransport>
<j: Address rdf: about= "http://www.owl-ontologies.com/Ontology1.owl#DepartureHospitalAddress"/>
<j: hasNearestAirport rdf: resource= "# DepartureAirport"/>
</j: Address>
<j: Address rdf: about= "http://www.owl-ontologies.com/Ontology1.owl#DestinationHospitalAddress"/>
<j: hasNearestAirport rdf: resource= "# ArrivalAirport"/>
</j: Address>
</rdf: RDF>

A.1.4 Flight Company Ontology

<?xml version="1.0"?>
<rdf: RDF
xmlns= "http://127.0.0.1/health-scallops/ontology/FlightCompany_Ontology.owl#"
xmlns: rdf = "http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns: xsd = "http://www.w3.org/2001/XMLSchema#
xmlns: flight-company = "http://127.0.0.1/health-scallops/ontology/Health-Scallops_Ontology.owl#"
xmlns: flight-company-schema = "http://www.w3.org/2001/XMLSchema#
xmlns: flight-company = "http://127.0.0.1/health-scallops/ontology/Health-Scallops_Ontology.owl#"
xmlns: flight-company-schema = "http://www.w3.org/2001/XMLSchema#
xmlns: xmlbase = "http://127.0.0.1/health-scallops/ontology/FlightCompany_Ontology.owl#"
<owl: Ontology rdf: about= "http://127.0.0.1/health-scallops/ontology/FlightCompany_Ontology.owl#"
</owl: Ontology>
<owl:Class rdf:ID=" ProvidedFlight">
<rdf: subClassOf rdf: resource= "http://127.0.0.1/health-scallops/ontology/Health-Scallops_Ontology.owl#Flight"/>
</owl:Class>
<owl:Class rdf:ID=" ValidAccount">
<owl: superClass rdf: resource= "http://127.0.0.1/health-scallops/ontology/Health-Scallops_Ontology.owl#Account"/>
</owl:Class>
</rdf: RDF>

<!-- Created with Protege (with OWL Plugin 2.1, Build 284) http://protege.stanford.edu -->
A.1.5 Medical Flight Company Ontology

```xml
<?xml version="1.0"?>
<rdf:RDF
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:xmlns="http://www.w3.org/2001/XMLSchema#
    xmlns:health-scallops="http://127.0.0.1/health-scallops/ontology/Health-Scallops_Ontology.owl#"
    xmlns:daml="http://www.daml.org/2001/03/daml+owl#"
    xmlns:dc="http://purl.org/dc/elements/1.1/"
    xmlns:base="http://127.0.0.1/health-scallops/ontology/MedicalFlightCompany_Ontology.owl#"
>
<owl:Ontology rdf:about=""/>
<owl:Class rdf:ID="ProvidedFlight">
    <rdfs:subClassOf rdf:resource="http://127.0.0.1/health-scallops/ontology/Health-Scallops_Ontology.owl#Flight"/>
</owl:Class>
<owl:Class rdf:ID="ValidAccount">
    <rdfs:subClassOf rdf:resource="http://127.0.0.1/health-scallops/ontology/Health-Scallops_Ontology.owl#Account"/>
</owl:Class>
</rdf:RDF>
```

A.1.6 Medical Transport Company Ontology

```xml
<?xml version="1.0"?>
<rdf:RDF
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:xmlns="http://www.w3.org/2001/XMLSchema#
    xmlns:health-scallops="http://127.0.0.1/health-scallops/ontology/Health-Scallops_Ontology.owl#"
    xmlns:daml="http://www.daml.org/2001/03/daml+owl#"
    xmlns:dc="http://purl.org/dc/elements/1.1/"
    xmlns:base="http://127.0.0.1/health-scallops/ontology/MedicalTransportCompany_Ontology.owl#"
>
<owl:Ontology rdf:about=""/>
<owl:Class rdf:ID="ProvidedTransport">
    <rdfs:subClassOf rdf:resource="http://127.0.0.1/health-scallops/ontology/Health-Scallops_Ontology.owl#VehicleTransport"/>
</owl:Class>
<owl:Class rdf:ID="ValidAccount">
    <rdfs:subClassOf rdf:resource="http://127.0.0.1/health-scallops/ontology/Health-Scallops_Ontology.owl#Account"/>
</owl:Class>
</rdf:RDF>
```