

Semantic Web Services

Automated Matching, Composition and Selection

Tor Arne Kvaløy

Semantic Web Services

Automated Matching, Composition and Selection

Master of Science Thesis
Computational Science
University of Amsterdam

Tor Arne Kvaløy
September 2004

Supervised by
Dr. Erik Rongen (IBM)
Alfredo Tirado (UvA)
Prof. Peter Sloot (UvA)

Abstract

Web Services is a technology that enables platform and language neutral interaction on the web. Current standards like WSDL and SOAP provide, however, only syntactical interoperability, which constrains agents from automating discovery and composition of services. The Semantic Web provides standards for representing and processing computer interpretable information. Semantic Web Services is thus a synthesis of these two technologies where the service capabilities are given well-defined machine interpretable meaning in the form of ontologies. In this work we describe how semantics can be used to annotate Web Services so that discovery, composition and selection can be automated. We give extensive review of current literature and proposed standards, and develop the necessary algorithms and architectures for an implementation. A case study is presented where a domain ontology is defined and used to annotate two Semantic Grid Services. Automated matching, composition and selection are then demonstrated followed with an analyzes where we propose improvements in ontology design and capability matching.

Acknowledgements

In detachment lies the wisdom of uncertainty... in the wisdom of uncertainty lies the freedom from our past, from the known, which is the prison of past conditioning. And in our willingness to step into the unknown, the field of all possibilities, we surrender ourselves to the creative mind that orchestrates the dance of the universe.

– Deepak Chopra

Motivated by the beauty and mystery of life and nature, I came to Amsterdam to study these processes from a computational point of view. I have, however, during these two years not only gained an understanding of scientific research in this direction, but a significant better understanding of life as whole, with its intricate facets and infinite possibilities.

The path of working on this thesis was indeed created along the way, and several people contributed in making this journey challenging and enjoyable. I will therefore show my appreciation to some of you that inspired and helped me professionally or personally.

First and foremost I am very grateful to Dr. Erik Rongen (IBM), who initially brought semantics to my attention, but which I abandoned for four months, for then to come back to. Those four months studying virtual organization, profiling, various grid technologies and agent computing were long and daring. The relief was thus great when he again asked me about semantics and I realized that the Semantic Web was the missing piece in combining Agent Computing with Web Services. He is one of few that actually understand the content of this thesis, and that is due to his eagerness to comprehend new ideas – even though it means getting headache. I found the meetings with him always inspiring and I have learned a lot from working with him. Thanks for being so patient and for helping me to materialize my thoughts.

My special appreciation to IBM's Chief Innovation Officer and head of Centre for Advanced Studies Mr. Djeevan Schiferli, who gave me the unique chance to express my creativity in a company environment. I bow to your constant enthusiasm for innovation and for so empathically listening to my philosophical outbursts and contemplations.

I am grateful to Alfredo Tirado-Ramos (PhD. Student, UvA) for backing me up along the way, helping me to focus and making me understand the Grid Services in the VLE project.

Many thanks to Prof Peter Slood for organizing the Master of Computational Science program, and for introducing me to the VLE project and relevant PhD students.

Thanks to Sander van Splunter (PhD. Student, Vrije Universiteit Amsterdam) for initial talks on Semantic Web Services and composition.

The IBM Centre for Advanced Studies (formerly known as the ST!FT Innovation Team) has been a great place to work on my thesis, and I believe this is due to the friendly and open minded environment that the team members are creating. In addition to Erik and Djeevan I am therefore thankful to Jasper, Ellis and Johan for talking English and for being such a fine group of people.

What would Centre for Advanced Studies be without brilliant students? Even worse, what would work be without coffee breaks? My very special super appreciation to the kids that are gone save the Dutch economy; Ian, Joeron, Arvid, Manfred. Thanks for good talks, loads of power-balling and great fun.

Finally, I am expressing love to my family in Norway; to my dear mum that is always supporting me, and to my sisters that so sincerely care about me.

Tor Arne, Amsterdam, September 2004

Table of Contents

Abstract	4
Acknowledgements	5
Table of Contents	6
1. Introduction	8
1.1 Motivation	8
1.2 Research Questions	9
1.3 Organization	9
2. A Brief Review of Web Services, Agents and the Semantic Web	10
2.1 Web Services	10
Introduction	10
XML and XML Schema	11
SOAP	12
WSDL – Web Services Definition Language	12
2.2 Agents Explained in Comparison to Objects	14
2.3 The Semantic Web	15
Introduction	15
Knowledge Representation and Inference	16
OWL - Web Ontology Language	19
3. State of the Art in Semantic Web Services	21
3.1 Introduction to Semantic Web Services	21
Service Discovery	23
Service Composition	25
Service Interaction	25
3.2 Motivation for Semantic Web Services	26
3.3 OWL-S: Semantic Markup for Web Services	27
3.4 Capability Matching	30
3.5 Available Software	32
RACER	32
Pellet OWL Reasoner	32
OWL-S API	32
SNOBASE	33
4. Algorithms, Design and Implementation	34
4.1 Architecture	34
4.2 Service Capabilities in OWL-S	35
Semantic Data Structures	35
4.3 Algorithms and Design	36
4.4 Matching of Input and Output Parameters	37
4.5 Composition	38
4.6 Composition Selection	42
5. Case Study – Interactive Simulated Vascular Reconstruction	44
5.1 Introduction	44
Scenario	44
Benefits of Semantic Grid Services	46

5.2 Domain Ontology	46
Definition of Semantic Data Structures	46
5.3 Annotation and Matching of Capabilities	47
5.4 Composition	48
5.5 Composition Selection	48
6. Analyses and Discussion	50
6.1 Domain Ontologies with High Complexity	50
Revised Design of OWL-S	51
6.2 Scalability of Semantic Matching Architectures	52
6.3 Agreement on Domain Ontologies	53
6.4 Mapping of Domain Ontologies	53
6.5 The Order of Capability Parameters	54
6.6 Preconditions and Effects	54
6.7 User Defined Matching, Composition and Selection Criteria	55
7. Conclusions	56
7.1 Summary	56
7.2 Conclusion	57
7.3 Contributions	58
Ideas developed that were later found in literature	58
7.4 Further Work	59
Appendix A – Glossary of Terms	61
Appendix B – Abbreviations	62
Appendix C – Screenshot of the Case Study demo	63
References	64

1. Introduction

This chapter presents the motivation for this work by introducing the limitations of current standards for service discovery and composition. The research questions are then introduced followed with the structure of the work.

1.1 Motivation

Somewhere I have a service that does exactly what I want...

– Bob Sutor, IBM

The Web is evolving from being a static source of information to a highly dynamic network where resources are shared and information is generated on demand. Business strategies like B2B (Business-to-Business integration) are currently being developed to take advantage of the Web in order to allow organizations adapt faster to changing markets, and to optimize the market profits through strategic collaboration and partnership [18]. The web is equally important for scientific communities that require optimal usage of computational resources and shared databases [10].

A general requirement of Web technologies is interoperability between different systems running on different types of hardware. Web Services enables platform independent and language neutral interaction on the Web, because it relies on open XML based standards like SOAP [8] and WSDL [7] that are agreements on how information is exchanged.

Web Services are self-describing, self-contained units of application logic that are either used directly by a client or composed into a workflow that is executed as a whole. Integrating Web Services consists basically of finding the right services and then combining them into a flow that accomplishes a defined goal. Integrating and maintaining such systems is not hard when the system consists of few services and the number of available services is low, but when the number of available services increases and the system consist of a large number of services, then it becomes more difficult to locate the right service and to administrate the system.

Another issue with infrastructures is the need for dynamic and flexible interoperability, especially in a fast changing and unreliable environment like the Internet. A system consisting of several services should for example not stop because of one failing service, but should automatically try to replace the failing service with a similar one.

It is for these reasons desirable to automate tasks in order to make the infrastructure more autonomous, flexible and dynamic [15], and the first step would be to automate service discovery and composition.

Universal Description, Discovery, and Integration (UDDI) is a standard for locating services and provides functionalities to search for a service based on a description in the WSDL format [9] [12]. The search finds services that are compatible in terms of binding protocols and interfaces. But since interfaces are expressed in XML, there is only an agreement on the format of the data that is exchanged and not the content. The client and the service are able to interact with syntactical operability, but the interpretation of the information can be different. In same vain will two services that do different things, but with the equal interfaces not be distinguished. For automated discovery the meaning of the data needs to be taken into account. This standard is therefore not suited for automated discovery.

Business Process Execution Language for Web Services (BPEL4WS) is a standard for representing compositions of Web Services where the workflow and bindings are known a priori [47]. This is sufficient for static systems that are predefined, but fail to address the need for composition on demand where for example non-functional or changing information is taken into account [20]. Moreover, since compositions in BPEL4WS are based on WSDL, they have the same limitation of only guaranteeing syntactical similarities between services. The standard is therefore not tackling the need for automated composition.

In order to automate discovery and composition the services need to incorporate information with computer interpretable meaning. The Semantic Web is an effort by the W3C consortium that defines standards for representing and processing information that is computer interpretable, and which enables computers to classify and reason about information [29].

Semantic Web Services is an emerging technology that combines the Semantic Web with Web Services in an attempt to create an infrastructure that can be more automated [4]. In this work we are assessing the state of Semantic Web Services by investigating how the technology works and what the benefits are.

1.2 Research Questions

The following three research questions form the basis of the thesis.

How do Semantic Web Services work?

This question asks from a theoretical point of view what it means to semantically describe Web Services and how it is done. Furthermore, how Semantic Web Services are matched and composed based on the semantics.

What architectures, algorithms and software are required?

This question asks what is needed to make Semantic Web Services work. This means identifying software, standards, algorithms and architectures that are available.

What are the benefits of extending Web Services with semantics?

In this question we ask what the motivations are for using semantics and how it can improve service discovery and composition. We are especially interested in what the benefits Semantic Web Services have over Web Services.

1.3 Organization

Chapter 2 provides the necessary background for Semantic Web Services by introducing Web Services, Agents and the Semantic Web. This forms the basis for chapter 3 where the theory of Semantic Web Services is introduced and explained. Chapter 4 shows how the theory can be realized in an implementation by introducing the necessary algorithms and architectures. In chapter 5 the theory and implementation are tested in a real scenario where a domain ontology is defined and used to describe services that are matched and composed. In chapter 6 we analyze and discuss various aspects of Semantic Web Services, and chapter 7 concludes with summary, conclusions and further work.

2. A Brief Review of Web Services, Agents and the Semantic Web

This chapter intends to give the necessary background for understanding the next chapter on the State of the Art in Semantic Web Services. Web Services with related standards are briefly introduced. Then the idea of autonomous agents is explored with a comparison to object oriented technologies. And finally, theory and practical issues of the Semantic Web is elaborated on with detailed examples and discussion.

2.1 Web Services

Introduction

Web Services is a technology to access applications via a network. From its name it reasonable to assume that the technology is related the World Wide Web. But how exactly are they related? The choice of calling the technology a “Service” can also not be accidentally. What is a service and how does it relate to applications? These questions are here addressed.

Web “Services”

A service is defined as a well-defined, loosely coupled application that receives requests from a client [31]. For the service being well-defined it means that the service provides and contains sufficient information for clients to start interacting with the service. WSDL (Web Services Definition Language) is the standard for describing Web Services so that they will be well-defined – also called self-described [7]. Loose coupling of services is a software design approach where dependences between services are clearly defined avoiding ad hoc fuzzy dependences. Services are self-contained software modules that are deployed once and then are universally available. This contrasts Component Based Software Architectures where components are tightly coupled and deployed within a specific application context [31].

The concept of services lead to the development of a distributed-system paradigm called Service Oriented Architectures (SOA) [18]. The idea of SOA is to break down monolithic application into suites of services that are available across enterprises – making Business to Business integration easier. Figure 1 shows two enterprises sharing a service.

Services can then be thought of as building blocks that are equally used by clients as well as by other services. Application specific scenarios are made by organizing the services into sequences and hierarchies, as shown in Figure 1. The figure shows that Service A uses Service C, while Service B uses Service C and Service D.

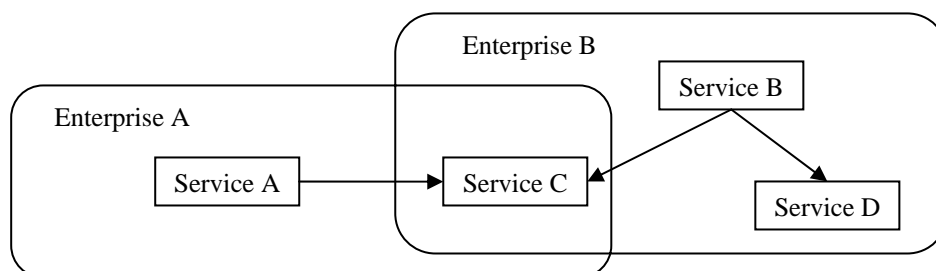


Figure 1: Service usage across enterprises. Arrows depict usage/dependences.

“Web” Services

Web Services relate to the Web in that the technology is defined with protocols standardized by the World Wide Web Consortium (W3C). Web Services are specified with XML based protocols recommended by W3C – an organization that its goal is to lead the Web to its full potential by developing common protocols that promote evolution and ensure interoperability. Different interest groups from universities and industry collaborate through this organization in creating standards that reflect the needs and thus will have a high ratio of acceptance.

The protocols used by Web Services are based on current standards for communicating on the Web. As shown in Figure 2, Web Services are defined with XML based protocols like WSDL and SOAP. WSDL describes the capabilities of the service while SOAP envelopes information so that it can be communicated on the network. SOAP defines so-called bindings which are mappings to underlying communication protocols e.g. HTTP. Web Services differs therefore from other technologies – like Corba IDL and Java RMI – in that it uses standard protocols that make optimal use of existing Internet infrastructure [31].

One important advantage of leveraging existing infrastructure – such as well tested protocols like HTTP – is that organizations avoid opening up their systems – firewalls – to “unknown protocols” making their system vulnerable. Security is therefore an important motivation for using existing technologies.

The protocols used by Web Services are XML-based. Interaction with Web Services is therefore platform independent and language neutral. What architecture or language that implements the service does not need to be known by the client. As long as services and clients comply with agreed standards any application can interact with each other.

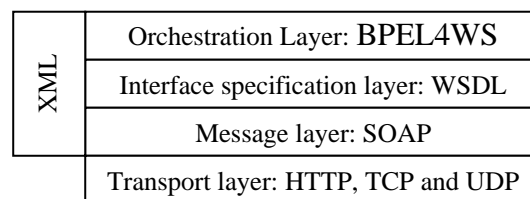


Figure 2: Web Services architecture layers.

XML and XML Schema

Extensible Markup Language (XML) is a general purpose specification of how data can be described in a data file. Descriptions are oriented around the use of markups, which “... encodes a description of the document's storage layout and logical structure” [5]. Information described in XML consists of data annotated with markups, where the markups describe the meaning of the data.

XML does not define markups itself. The markups, and thus the document's structure are explicitly defined by a XML Schema [6]. An XML document together with an XML Scheme provides a self-contained description of information. This makes XML a very powerful tool for carrying and exchanging information.

XML standardizes how applications read and store data. Data is described with markups specified with XML Schemas, making the data neutral so that any XML parsers can interpret the structure of XML documents. How this information is used and represented in the application is application specific. XML is therefore introducing a distinction between data description and data representation.

XML and XML Schema define how information is described in a syntactical sense. XML defines how the markups are represented and XML Schema defines how markups are

structured. The markups can have some meaning given by the developer. The developer knows for example how various markups are related to each other. When an application parses and interprets an XML document it needs to have been programmed to understand syntactical structure of the information. Applications are therefore hardcoded to understand XML documents. If relations were embedded with the markups then computers could have been able to relate various markups and possibly learn markups that it was not programmed to understand. XML Scheme can only relate markups on a syntactical level. We will later in this chapter see how markups can be given semantic meaning that computers use for inference purposes.

SOAP

SOAP is a protocol for sending information between applications, e.g. between clients and Web Services. The protocol specifies a format for how information is packaged. When information is packaged into this format, it can be unpackaged by applications that understand the format. SOAP is thus a messaging protocol that packages information into an envelope as shown in Figure 3.

SOAP messages can be exchanged using a variety of underlying protocols. A specification of how the messages are communicated between two applications is called SOAP binding. The most used one is not surprisingly HTTP.

For SOAP to be a general purpose protocol it needs capabilities for enveloping non-XML based data. SOAP solution to this is to use a defined Data Model together with an Encoding Scheme. The Data Model specifies how data can be represented as a graph, so that the data can be included in the SOAP Body. For data to be transferable on the wire the Data Model has to be serialized, and equally when received, deserialized. This mapping between Data Model and transferable information is done with Encoding Schemes. Data Models together with Encoding schemes must therefore be used to represent and transfer non-XML based information.

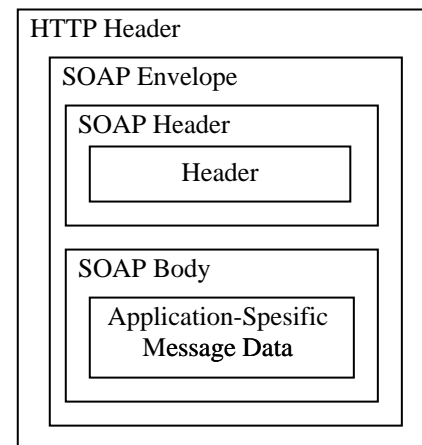


Figure 3: SOAP envelope layers.

WSDL – Web Services Definition Language

WSDL is a language to describe Web Services so they can be interacted with. For clients to interact with Web Services they need to know their location, what functionalities they provide and what protocols they support. A document describing this is called a Service Interface and – as shown in Figure 4 – is used by both clients and services. The service implements the interface while the client interacts with an API that is generated from the interface description.

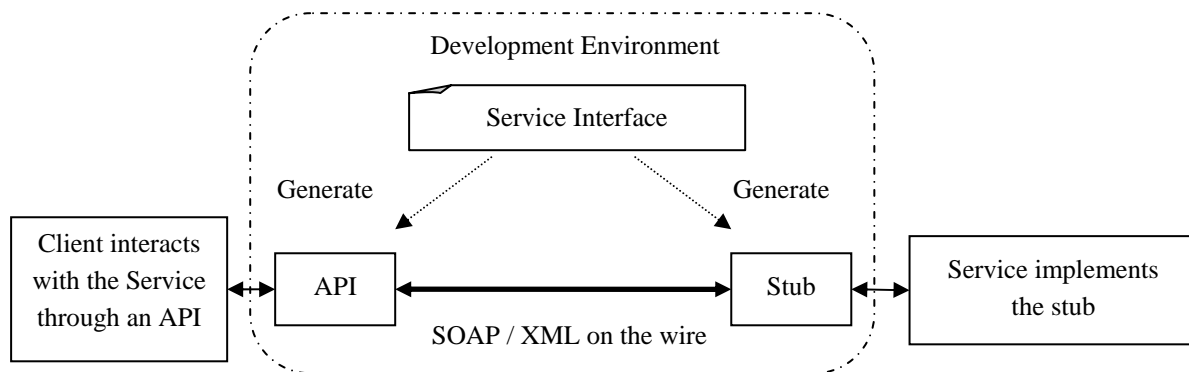


Figure 4: Conceptual illustration of how Web Services technology works.

Web Services are similar to java programs in that they define functions (operations) with parameters. Figure 5 shows a Service Interface for a translation service that takes one word as input and returns a word that is translated into another language. Lines 12-15 show the WSDL code for an operation with two messages. The message TranslationRequest is input to the service while TranslationResponse is a message that is returned. Lines 3-9 show the definitions of the messages where each of them has only one message part. Each message part – can be compared to function parameters – is given a name and an abstract data type. The abstract data types are either XML Schema datatypes or WSDL Complex datatypes provided with encoding schemes.

The partType in the Service Interface can be thought of as the basic functionalities of service due to that it consists of all operations the service defines. This information has to be communicated with some underlying protocol. Lines 18-31 show how the portType is binded to the SOAP messaging protocol. The binding can be thought of as a mapping between the structure used by WSDL to describe functionalities and how the information is actually communicated. WSDL messages are for example mapped into SOAP messages where the SOAP body is explicitly informed about the WSDL message parts (lines 24 and 28).

WSDL is also used to describe what is known as Service Implementations which holds information like where the Service Interface can be found. This can be used for service discovery purposes like in UDDI (Universal Discovery Description and Integration) where the service discovery facility stores information about Web Services under various business categories. The Service Implementations are stored in fields called BindingTemplates and point to Service Interfaces that are stored in so-called TModels. WSDL can therefore be used to create a complete description of Web Services' location and the interfaces.

The description is however only understandable by humans. A computer could for example not interpret what the service in Figure 5 does. This is because the operations, parameters and their data values are described with XML and thus has limited expressiveness for semantics. Take line 4 and 8 as an example. What a computer understands from these parameters is that they are of type string. For Web Services to be autonomously interacted with they need to be described so that their meaning can be derived by computer programs.

```
1 <wsl:definitions xmlns:wsl="http://schemas.xmlsoap.org/wsl/" ...">
2
3   <wsl:message name="TranslationRequest">
4     <wsl:part name="WordToTranslate" type="xsd:string"/>
5   </wsl:message>
6
7   <wsl:message name="TranslationResponse">
8     <wsl:part name="TranslatedWord" type="xsd:string"/>
9   </wsl:message>
10
11  <wsl:portType name="TranslationPortType">
12    <wsl:operation name="TranslationOperation" parameterOrder="WordToTranslate">
13      <wsl:input message="TranslationRequest"/>
14      <wsl:output message="TranslationResponse"/>
15    </wsl:operation>
16  </wsl:portType>
17
18  <wsl:binding name="TranslationSoapBinding" type="TranslationPortType">
19    <wslsoap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
20    <wslsoap:operation name="TranslationOperation">
21      <wslsoap:operation soapAction="">
22        <wsl:input name="TranslationRequest">
23          <wslsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
24            parts="WordToTranslate" use="encoded"/>
25        </wsl:input>
26        <wsl:output name="TranslationResponse">
27          <wslsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
28            parts="TranslatedWord" use="encoded"/>
29        </wsl:output>
30      </wsl:operation>
31    </wslsoap:operation>
32  </wsl:binding>
33 </wsl:definitions>
```

Figure 5: WSDL Service Interface of Translation Service example.

2.2 Agents Explained in Comparison to Objects

Agent technology introduces new ways of representation and interaction in distributed systems. Resources are represented as intelligent entities, and interaction is asynchronous and dynamic. To explain these new concepts it is useful to relate them to something that is well studied and understood. Agents are therefore in this section explained in comparison to object based technology.

Agents differ from objects in that they

- Are autonomous;
- Interact with other agents through messaging;
- Can be proactive.

The first difference comes from the very essence of agent technology and is that agents are autonomous. Autonomous means that they are in charge of their own sequence of execution, or as defined in literature, “capable of exercising choice over their actions and interactions.” [1]. Each agent has its own event loop where the sequence of execution is determined by an internal model. This model represents the core logic of the agent and is typically based on desires and goals. An object has no such control of the sequence of execution. Objects simply provide a set of methods that are executed in the sequence they are called. These methods might involve execution of several internal methods and other objects. However, the initial act of calling the method is not determined by the object itself, but by an external entity.

This leads to the second fundamental difference between objects and agents. It is that objects interact through method invocations, while agents interact through messaging. Messaging is asynchronous communication between agents and is necessary for autonomous behaviour. The reason for this is that interpreting the message should be a decision made by the agent itself. This gives the agents the ability to for example discriminate message on meta-information like sender, ontology or language. When and how the agents deal with an incoming message depends on the state the agent is in. The agents might for example be busy and would prefer to queue the message for later usage.

To exemplify the difference further it is useful to distinguish reactive and proactive behaviour. Reactive behaviour is simply to act on request. Both agents and objects are typically reactive, but objects have a stronger form in that they provide a set of functionalities as methods that are invoked by an external entity. Agents can be reactive, but because of their inherent autonomy and the way of interacting through messaging, they have a more flexible form. Based on their event loop they can decide when to react. Objects can, however, be made more flexible by introducing threads and using asynchronous method calls. This way of introducing parallelism is common practise to in many distributed systems [2].

Objects and agents are very different when it comes to being proactive. Proactive means to have the ability to initiate actions on its own, for example according to an internal model. A requirement for this is therefore to be autonomous. Objects are not autonomous and can therefore not be proactive.

Messaging can be seen as a requirement for autonomous behaviour, while the proactive ability can be seen on as feature of autonomous behaviour. Another requirement is that agents have their own thread of execution. They can however be suspended and check-pointed for reasons like backup, security and mobility, but a general requirement is that the autonomous core is experiencing a sequential execution.

2.3 The Semantic Web

The first step is putting data on the Web in a form that machines can naturally understand... This creates what I call a Semantic Web – a web of data that can be processed directly or indirectly by machines.

– Tim Berners-Lee [45]

Introduction

Formulated by the creator of the World Wide Web Tim Berners-Lee, the Semantic Web is about “bringing the web to its full potential” [32]. The web currently contains around 3 billion static documents, which are accessed by over 500 millions users. While these numbers are increasing at a staggering rate, the task of dealing with the information is getting harder [29]. The Semantic Web is an effort to develop technologies so that the value of information can scale with the increase of information, thus bringing the web to its full potential.

The Semantic Web’s approach of bringing this about is to make information “understandable” by computers. Information must therefore be described in such a way that computers can interpret it and derive its meaning. This will enable computers to work more intelligently with the information; for example assisting humans in classifying, filtering and searching information.

Computer understandable information is information annotated with semantics. Annotations can therefore be thought of as metadata that describes the meaning – the semantics – of the information. The annotations themselves have to be defined so that

computers can interpret and reason with them. A collection of annotations where their meaning is described is called an ontology. Figure 6 shows the idea of using an ontology to annotate some text in a document. Ontologies can however be used to annotate any form of data.

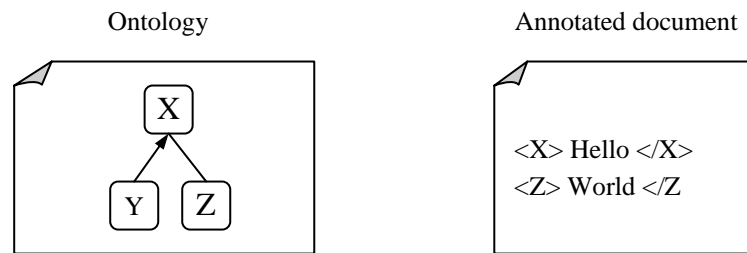


Figure 6: An ontology used to annotate some text in a document.

For ontologies to represent knowledge of a domain, they need to be expressed with language that can convey the necessary complexity of the domain. Description Logic (DL) is a knowledge formalism that describes the abstract world with concepts and relations. These basic constructs can be used to build up advanced hierarchies and graphs with restrictions on various levels.

Knowledge has to be represented so that logic can be applied. In special it is important to be able to compare and derive similarities between annotations. DL has advanced capabilities for this. One powerful feature in DL is subsumption, which checks whether or not a concept contains another concept. The advantages of annotations are closely related to what can be derived from the representation. Using a language with advanced capabilities for reasoning is therefore of great importance.

Applications interpreting the semantics of a document need to have access to the ontologies that define the semantics. When a document is annotated with semantics it includes information about where the annotations are described. The ontologies describing the annotations must therefore be available and readable so that applications can derive their meaning. For example in context of the Semantic Web – which is a distributed system – the ontologies have to be network accessible. They are therefore defined with URIs which are unique network identifications.

For ontologies to be used in distributed systems and across systems there has to be agreements on how knowledge is represented and reasoned with. These standards should be general and – at the same time – advanced enough to capture the wide needs of different interest groups. They should also specify syntax and formats for representation. Recently developed open standards for knowledge representation are RDF and OWL [3] [30].

Knowledge Representation and Inference

Ontologies represent knowledge of a particular domain. An ontology is defined by R. Studer as “an explicit and formal specification of a conceptualization” [29]. Conceptualization here means a view of a domain or an abstract world, and this is represented using some representation formalism.

One family of knowledge formalisms is DL. It represents knowledge of a domain by first defining a terminology of concepts and relations, and then initiating individuals in the domain by using the terminology [24]. Concepts and their relations can be thought of as a definition of physical laws of the world, and for individuals to populate the world, they have

to confirm to these laws. Individuals that confirm to the laws of a concepts is said to be a valid member of the concept.

DL is a highly expressive language with a vast number of possibilities for representing complex knowledge. A detailed discussion of DL is beyond the scope of this introduction. It is, however, important for the rest of the thesis that the reader has a basic understanding of knowledge representation. An introduction to the most important features of knowledge representation and reasoning is here given. For a detailed discussion the interested reader is referred to [24] [30] [29].

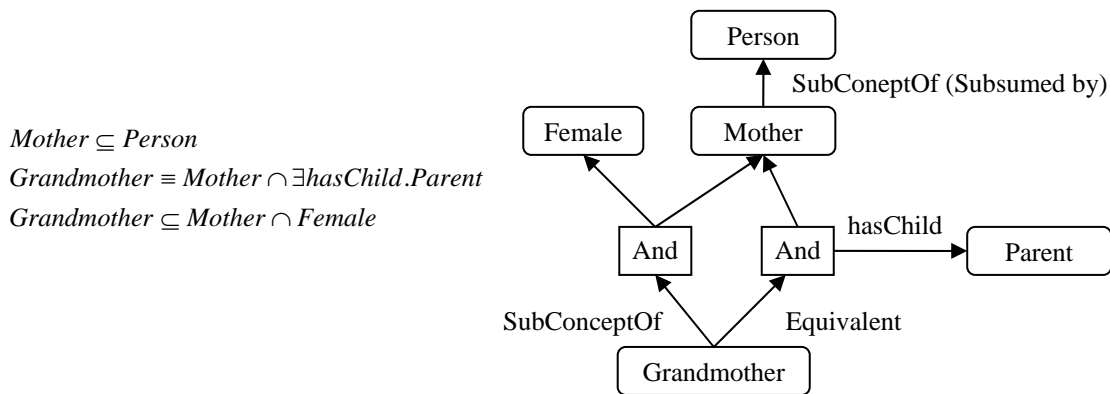


Figure 7: Example of family ontology.

Figure 7 shows a family ontology defined in DL notation accompanied with “free-style” visualization. The notation used for visualization does not confirm to any specific representation standard, but is merely provided to make the conceptualization more intuitive. The ontology is constructed to represent one perception of family relations and does not necessarily correspond to the real world.

The ontology contains five concepts; Female, Mother, Person, Parent and Grandmother. Concepts are related to each other with roles (also called properties). The ontology shows for example the hasChild property which connects Mother to a Parent. Concepts and roles are the main building blocks for representing abstract things and their relations in the domain, respectively, and are called constructors. Table 1 describes some of the constructors provided in DL [23][33][30].

DL Syntax ¹	Description	OWL Syntax
$C_1 \cap \dots \cap C_n$	Intersection	intersectionOf
$C_1 \cup \dots \cup C_n$	Union	unionOf
$\neg C$	Negation of arbitrary concept	complemenOf
$\{a_1, \dots, a_n\}$	Enumeration of individuals	one of
$\forall R.C$	Value restriction	allValuesFrom
$\exists R.C$	Existential qualification of concept	someValueFrom
$\exists R.\{a\}$	Existential qualification of individual	hasValue
$\geq_n R.C$	Qualified at least cardinality restriction	minCardinality
$\leq_n R.C$	Qualified at most cardinality restriction	maxCardinality
$\equiv_n R.C$	Qualified exactly cardinality restriction	Cardinality

Table 1: Some Description Logic constructors with their corresponding OWL syntax.

Constructors are furthermore related to each other with axioms. This is used to give concepts and roles characteristics, which typically are restrictions. The most general axioms are listed in Table 2.

DL Syntax	Description	OWL Syntax
$C_1 \subseteq C_2$	Concept subsumption – C_1 subsumes C_2	subClassOf
$C_1 \equiv C_2$	Equivalent concepts	equivalentClass
$P_1 \subseteq P_2$	Concept subsumption – P_1 subsumes P_2	subPropertyOf
$P_1 \equiv P_2$	Equivalent properties	equivalentProperty
$C_1 \subseteq \neg C_2$	Disjoint concepts	disjointWith
$\{a_1\} \equiv \{a_2\}$	Equal individuals	sameAs
$\{a_1\} \subseteq \neg\{a_2\}$	Different individuals	differentFrom
$P_1 \equiv \neg P_2$	Inverse properties	inverseOf

Table 2: Some Description Logic axioms with their corresponding OWL syntax.

Grandmother is defined as being equal to Mother that has a child of type Parent. The sign “ \equiv ”, in DL, defines equality while the sign “ \exists ” defines existential qualification for the relation, saying that there has to exist a “hasChild” relation with concept Parent.

Equality differs significantly from subsumption which is represented with the sign “ \subseteq ”. The difference lies in the conditions for valid concept membership. Grandmother is defined as being subsumed by Female and Mother. The subsumed relation expresses that it is *necessary* to be a valid member the Female and Mother to be a member of Grandmother - so membership of Grandmother means also membership of Female and Mother. But it is not *sufficient* to be a Female and a Mother to be a Grandmother. It is, however, necessary and sufficient to be Mother with a child of concept Parent to be a Grandmother. Equality and subsumption differ therefore in their direction of mapping. Equality defines a bidirectional mapping, while subsumption defines a one way mapping.

¹ Some of the symbols used in DL are not available in the environment this document is typeset in. This includes intersection, union and subsumption. The most similar symbols from set theory are in these cases used.

Ontologies can be used by agents to derive the meaning of information. Effectively, it means to classify the information as something that is known to the agent. As an example consider an agent that is interested in inferring the gender of some information. The available information describes a Mother with a hasChild relation to concept of type Parent. Using the ontology in Figure 7 an inference engine can see that there is equality between this information and Grandmother. Grandmother is furthermore subsumed by Female, thus member of that concept. Relations and axioms in the ontology have then been used to infer that a Mother with a Parent child is Female.

OWL - Web Ontology Language

Being a W3C Recommendation since 10 February 2004, OWL is the latest standard for knowledge representation [30]. OWL uses XML styled syntax and is based RDF (Resource Description Framework) and RDF Schema. Figure 8 illustrates these dependences.

OWL	Full
	DL
	Lite
RDF Schema	
RDF	
XML Schema	
XML	

Figure 8: Dependences between standards.

XML provides the basic definition on how information is described in a document using markups and namespaces. XML Schema is a language for defining and validating the structure of XML documents and includes definition of datatypes [5] [6].

RDF introduces semantics by providing constructs for describing resources and relations. RDF Schema extends this with a vocabulary for properties and classes that can be organized in hierarchies [3].

OWL adds to these standards a richer vocabulary for describing properties and classes, such as relations between classes, cardinality, equality, characteristics of properties and enumerated classes [29].

OWL is furthermore divided into three increasingly expressive sublanguages. OWL Lite is the most basic one providing constructs for users that need simple classification hierarchies. OWL DL provides full expressiveness while retaining computational completeness meaning that all conclusions are guaranteed to be computable, and decidability meaning that all computations will finish in finite time. OWL Full provides full expressiveness and syntactical freedom, but with no computational guarantee.

OWL Syntax and Constructs

The OWL language has a large number of constructs and syntactical possibilities. The reader is here introduced to some of these with an OWL representation of the family ontology from Figure 7. The code is shown in Figure 9, and is here presented with a walkthrough.

Concepts in OWL are called classes. Lines 9-11 define classes Person and Mother, where Mother is specified as a subclass of Person.

Roles in OWL are represented as binary properties that relate two classes to each other. OWL defines two types of properties; ObjectProperty and DataProperty. ObjectProperty relates an instance of class to an instance of another class, while DataProperty relates an instance of a class to an XML Schema datatype. The latter one is thus used to represent data in the ontology, while the former simply defines a relation between classes.

The property defined in lines 15-18 connects Mother to Parent with a “hasChild” relation. Properties can be specified with domain and range, which can be thought of as *from* and *to* for the property, respectively. They specify that the property applies to instances of Mother and that the values of the property have to be instances of Parent.

```

1 <?xml version="1.0"?>
2 <rdf:RDF
3   xmlns="http://mySite.com/myOntology#"
4   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
5   xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
6   xmlns:owl="http://www.w3.org/2002/07/owl#"
7   xml:base="http://mySite.com/myOntology">
8
9   <owl:Class rdf:ID="Person"/>
10
11  <owl:Class rdf:ID="Mother">
12    <rdfs:subClassOf rdf:resource="#Person"/>
13  </owl:Class>
14
15  <owl:ObjectProperty rdf:ID="hasChild">
16    <rdfs:domain rdf:resource="#Mother"/>
17    <rdfs:range rdf:resource="#Parent"/>
18  </owl:ObjectProperty>
19
20  <owl:Class rdf:ID="Grandmother">
21
22    <rdfs:subClassOf>
23      <owl:Class>
24        <owl:intersectionOf rdf:parseType="Collection">
25          <owl:Class rdf:ID="Female"/>
26          <owl:Class rdf:about="#Mother"/>
27        </owl:intersectionOf>
28      </owl:Class>
29    </rdfs:subClassOf>
30
31    <owl:equivalentClass>
32      <owl:Class>
33        <owl:intersectionOf rdf:parseType="Collection">
34
35          <owl:Class rdf:about="#Mother"/>
36
37          <owl:Restriction>
38            <owl:onProperty>
39              <owl:ObjectProperty rdf:about="#hasChild"/>
40            </owl:onProperty>
41            <owl:someValuesFrom>
42              <owl:Class rdf:ID="Parent"/>
43            </owl:someValuesFrom>
44          </owl:Restriction>
45
46          </owl:intersectionOf>
47        </owl:Class>
48      </owl:equivalentClass>
49    </owl:Class>
50
51    <Parent rdf:ID="Anny"/>
52    <Mother rdf:ID="Ingeborg">
53      <hasChild rdf:resource="#Anny"/>
54    </Mother>
55  </rdf:RDF>

```

Figure 9: Family ontology represented in OWL DL.

3. State of the Art in Semantic Web Services

Semantic Web Services are Web Services with semantically annotated capabilities [21]. This chapter introduces Semantic Web Services by first explaining the limitations of current Web Services and then how they can be overcome by using semantics in their capability descriptions.

Embracing Semantic Web Services implies that new standards have to be developed, and before such a significant decision is made, it is important to assess what the benefits are over current technologies. The motivations for Semantic Web Services are therefore discussed.

OWL-S is then introduced which is an emerging standard for Semantic Web Services that provides necessary framework for describing service capabilities and how services are composed and invoked.

The Semantic Web Services effort is in its early stage and is still to be clearly defined, but even though the idea of Semantic Web Services has only been around for a couple of years – the first publications appeared in the beginning of the millennium – it has gained a significant attention and support from academia and industry, including IBM, HP and Nokia [4]. The following sections cover the latest in this early work with extended explanations and examples.

3.1 Introduction to Semantic Web Services

The idea of Semantic Web Services is here introduced gradually by first looking into how current Web Services are described, and then how these services can be extended with semantics. A translation service is used through the text to make the explanation clear.

The capabilities of Web Services are described with Web Service Definition Language (WSDL) [7] by defining operations and parameters that the service supports. This is done by naming operations and parameters, and then associating the parameters with abstract types which can be thought of as data types. A translation service – named `TranslateNorwegianToDutch`² – illustrating this, is shown in Figure 10. The service specifies two parameters, one input parameter named `WordToTranslate` of type string, and one output parameter named `TranslatedWord`, also of type string. To invoke the service the word to translate given and the translated word is returned.

TranslateNorwegianToDutch		
	Abstract Type	Parameter Name
Input	String	WordToTranslate
Output	String	TranslatedWord

Figure 10: An example service annotated on a syntactical level.

² WSDL distinguishes service names and operations, but for simplicity in this example they are considered as the same.

The problem with this service description is that agents can not derive what the service does. An agent interpreting the parameters of the service can not derive their meaning by simply looking at them. For agents they are only parameters names – named variables that are used to contain information. Agents can not derive the content of these parameters, because they can not read the parameter names like humans can. What agents can infer from a service description is that `WordToTranslate` and `TranslatedWord` are input and output parameters, respectively, and that they are of type string.

Agents can only interpret information that conforms to a syntactical structure. They are programmed to derive the meaning from formats like WSDL, which is an agreement on how service descriptions are interpreted. The service description can thus be seen on as a structured syntactical description. A human developer who wants to use a service in a client program needs to read the syntax of service description, interpret it, and then write the client program conforming to the syntax of service. Agents can not read text like humans – they can understand the structure of the service descriptions but not the content [22].

Semantic Web Services are described so that agents can interpret their capabilities. Autonomous agents should be able to look into the service description to determine if the service provides the desired capabilities and if the agent is able to use the service. The service description must thus be extended with computer interpretable information called semantics. The parameters or the service names must be described in such a way that agents can find out what they mean. This is achieved by defining vocabularies – organized in ontologies – that are used to annotate service capabilities. For agents to interpret service descriptions they will have to use these ontologies – which are shared conceptualizations of domains [29].

TranslationService			
	Abstract Type	Parameter Name	Semantic
Input	String	WordToTranslate	NorwegianWordToTranslate
Output	String	TranslatedWord	DutchTranslatedWord

Figure 11: An example service annotated with semantics.

Figure 11 shows the same service as in Figure 10, but now annotated with semantics. An extra column is introduced with semantic annotation of the parameters. Notice that the semantic names are more descriptive than the parameter names, which is to emphasize that the names are referring to concepts defined in ontologies. An agent trying to interpret this service description will have access to the ontology where the semantic annotations are defined, and by using an inference engine, the agent can infer similarities between the semantics used to describe the service and the semantics that are known by the agent. The meaning of service descriptions is therefore indirectly derived by an inference engine.

Another thing to notice is that the service name has been changed from `TranslateNorwegianToDutch` to `TranslationService`. This is to point out that what the service does is implicitly described through semantics of the parameters. The input and output parameters represents an *information transformation* undertaken by the service. Research is currently being done on how the state of services can be represented as preconditions and effects [35]. Preconditions would represent what is necessary for using the service, and effects would represent the consequences of using the service. Preconditions and effects are therefore

specifying the *state transformation* of the service [14]. This work will focus on describing information transformation of Semantic Web Services.

Service Discovery

Transactions between Web Services typically involve three parties; provider, requester and mediator [22]. The provider is the server that offers services, the requester is the client or agent that needs extra functionalities and the mediator helps the client in discovering and possibly composing services. Figure 12 shows a scenario with these three parties [34].

Service Discovery is to locate services with certain capabilities and is provided by the mediator that has a list of available services. The mediator has an interface with functionalities to search through advertised services and is used by agents to look for services with capabilities that are needed. Service discovery is thus about helping clients in finding services that best serve their needs.

The initial step for providers is to advertise their services to the mediator. Information about these services – called *advertised service capabilities* – is stored in a registry until the providers decide to stop advertising their services.

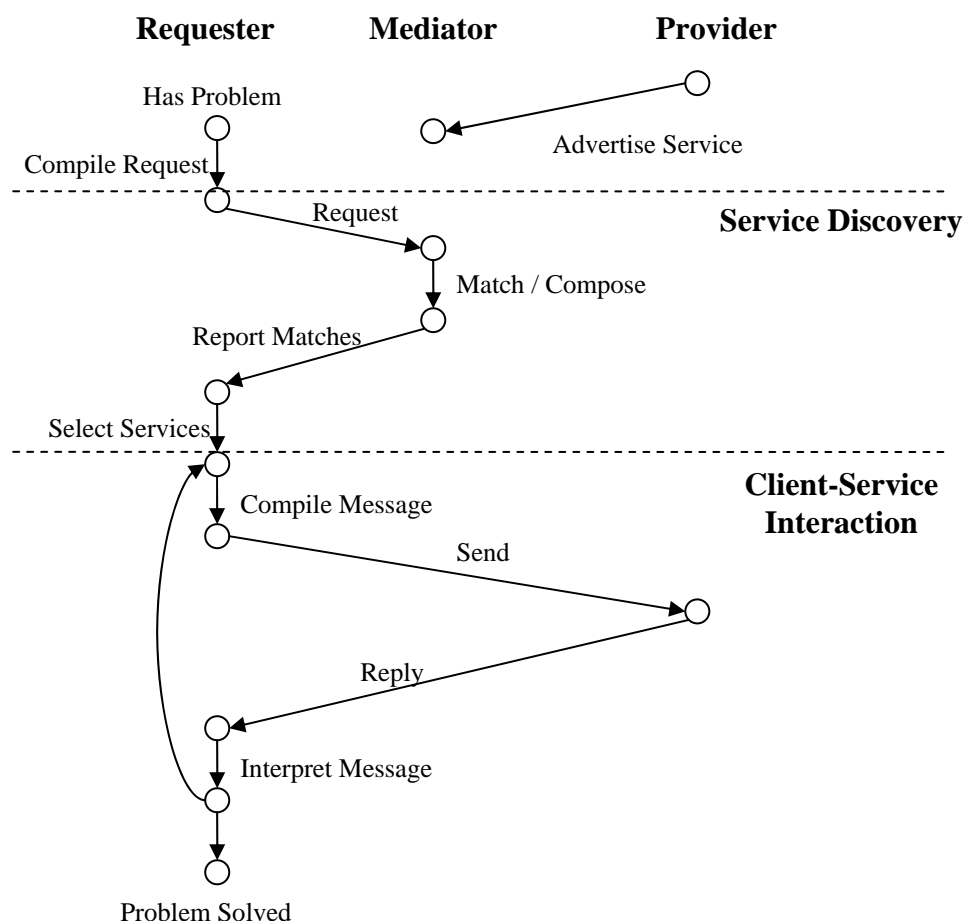


Figure 12: Illustration of service selection and interaction involving requester, mediator and provider.

At the requester's side a description of the functionalities that are needed is created. This description is called *requested service capabilities* and is communicated to the mediator

as request for capabilities that are needed. Figure 13 shows how ontologies are used to describe requested and advertised capabilities.

The mediator compares the requested capabilities with the advertised capabilities to determine if they are sufficiently similar. Sufficiently similar can mean that the advertised service satisfies the requested capabilities. Requested and advertised capabilities must thus be formulated so that they can be compared.

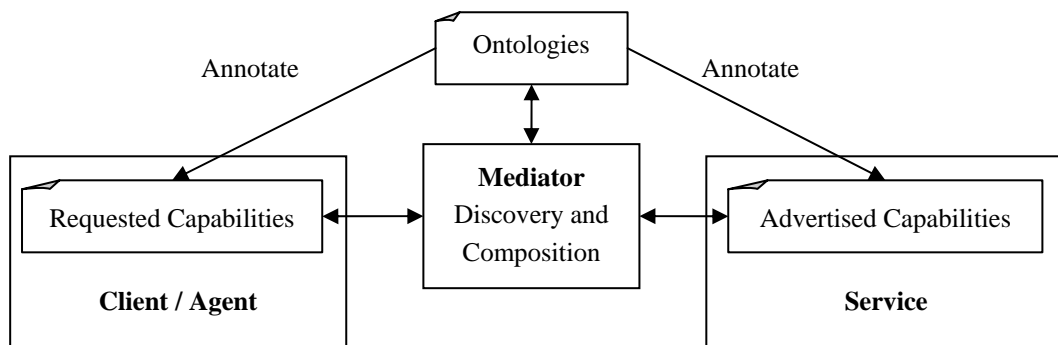


Figure 13: Illustration of how ontologies are used for annotation of requested and advertised capabilities, and by the Mediator.

The comparison done by the mediator is based on ontologies that the capabilities are annotated with. Inference logic on these ontologies reveals various degrees of similarities which are used by the mediator to select the advertised service that best match with the requested capabilities. These matches are then reported back to the client who makes a decision on which service to use.

Had the service capabilities only been described with WSDL, then comparison would have been on a syntactical level. A thesaurus service with input parameter of string and output parameter of string would for example have matched with the translation service defined in Figure 10. This is because the comparison is done on a syntactical level and not on semantic level. The meaning of the parameters is not taken into account.

Service Composition

It might be possible that none of the advertised services satisfy the requested service capabilities. Take for example a request for service that can translate from Norwegian to Sanskrit, and which does not exist. There are however advertised services that can translate from Norwegian to English and from English to Sanskrit. These services can be used in sequence to get the desired outcome of translation from Norwegian to Sanskrit and is called composition. Automated composition is possible because the services are annotated with semantics, and is achieved by resolving dependences between services and organizing them in an executable sequence. An illustration of service composition is shown in Figure 17.

Compositions using semantic descriptions can be automated with AI-reasoning [36][37], or it can be done interactively with users [38]. Independent on the method of compositions the workflow needs to be expressed in language that is executable. Some of these languages like SCUFL (Simple Conceptual Unified Flow Language) and BPEL4WS (Business Process Execution Language for Web Services) describe workflow statically without taking semantics into account, while OWL-S is a language that describes workflow on a semantic level [14][4].

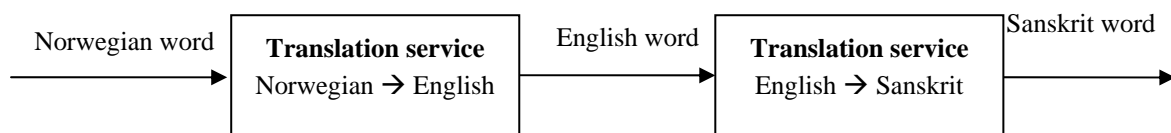


Figure 14: Composition of two translation services.

Service Interaction

Once services are selected the client can start interacting with them. Messages are then created using the agreed format and sent to services that reply with messages back to the client. These messages have to confirm to an agreed protocol that both parties can interpret. Service capabilities should therefore contain information about supported protocols. To avoid getting to this stage where the client intends to interact with the service and realizing that no common protocol exists, information about supported protocols should be taken into account during service discovery. The mediator could for example only return services that are compatible with the client.

The semantics used during service discovery does not necessarily need to be used for interaction. The semantically annotated parameters could be converted into data structures that are communicated. For example the translation service in Figure 11 the abstract types could be used. However for leveraging the Semantic Web the messages should be structured with semantics. Using ontologies messages can be structured with both data and meta-data. Figure 15 shows an example of a message asking for translation of the word “koselig”³ from Norwegian to Dutch.

³ “Koselig” is Norwegian and means “Cozy” in English, and “Gezellig” in Dutch.

```
<translate>
  <fromLanguage resource="translationOntology#Norwegian"/>
  <toLanguage resource="translationOntology#Dutch"/>
  <wordToTranslate> koselig </wordToTranslate>
</translate>
```

Figure 15: An example of a message to the Translation Service. The languages are specified by pointing to concepts in a domain ontology.

3.2 Motivation for Semantic Web Services

Increasing Number of Web Services

There are important trends in distributed systems that affect the importance of service discovery. The most noticeable trend is that the number of services increases rapidly. This is due to the popularity of Web Services and the fact that the service-oriented paradigm encourages loosely coupled independent services.

A result of the latter is that services are looked up on demand when they are needed and the effect of this might be that services are looked up more often. Service discovery is thus becoming more and more important in distributed systems.

Limitations of the UDDI

The most prominent technology for web service discovery is the Universal Description Discovery and Integration (UDDI). This technology is expected to be the internet standard for service discovery due to its strong industry backing [21].

UDDI is designed to be used by humans. This becomes clear from reading how services are typically found and invoked in the technical white paper [9]. It involves manual searching or browsing based on business descriptions, manual selection of service descriptions, and then finally construction of the client program based on the service description. Due to the fact that the intended users of UDDI are humans, it is natural that automated usage is not straight forward. It is however interesting to investigate what limitations it has to enable automated usage.

UDDI provides functionalities – an API – to search for services using keywords. These keywords are matched with text used in the descriptions of the services. It is very hard for agents to work with keywords because it involves some degree of language understanding. Keyword based search is thus not enabling autonomous discovery.

UDDI provides also functionalities to search for services based on the web service description given in the WSDL format. The comparison between the requested service and the searched service is then done on a syntactical level. This has some limitation; first, the syntactical match does not make sure that the service provides the requested service; second, since the agent can not look into the textual description provided for humans, it can not distinguish between two services providing the same syntactical description. This way of searching for services is therefore also not desirable for automated usage.

UDDI does not define a relationship between service descriptions in their so-called TModels. Different services with the same capabilities can thus be categorized in different business categories. Finding the service that best fit the agent's needs is thus difficult.

The TModel – which provides the main structure for the service description – is however general enough to contain semantic information. UDDI is simply not intended for

automated usage, and lacks therefore what is necessary for agent interaction. What is needed is a guideline for how to semantically annotate services and an API that supports search based on semantics. A lot of research activities are currently going on in how to extend UDDI with semantic functionalities [12] [16] [14].

Advantages of using Semantics in Service Discovery

There are several advantages of using semantics in service discovery.

The accuracy of the service discovery will improve with semantics. Semantics provide the expressiveness needed to make detailed descriptions of advertised service capabilities and capability requests. Matchmaking using semantics is more accurate than a keyword-based search, because the direct similarities are found using inference logic, while the keyword-based searches can be vague and inaccurate.

Service discovery based on semantics will enable a more dynamic coupling between clients and services. The exact services do not need be known in advance and the client can more easily change services that are used. This is an improvement towards the service oriented paradigm, where services are loosely coupled.

Dynamic coupling of services based on semantics will furthermore enable new technologies and applications to emerge. More resilient and flexible systems can for example be made by smart service mediators that use semantics to provide functionality transparency. This is useful in dynamic service environments or for systems that need high operability. Functionality transparency means that a client using a set of services through a service mediator does not know which or where these services are. The required functionality is just provided to the client. Failing or disappearing services can therefore be replaced unknowingly by the client.

Semantics can also improve the administration of service registers provided for manual browsing. Services can be automatically categorized and classified based on their semantics, making administration easier and registries more up to date.

3.3 OWL-S: Semantic Markup for Web Services

For semantic services to be useful in an open environment like the Internet there must be a general agreement on how they are represented. It is therefore a need for a standardized framework that provides the necessary languages and guidelines to describe services and their capabilities. Such a framework needs to be general enough so that it can be used to represent a variety of services, and at the same time restrict the usage to avoid ambiguous representation and unnecessary complexity.

This need for a framework has led to the development of OWL-S, which is a language to describe web services [4]. The language, specified as an ontology, intends to provide the necessary expressiveness to enable:

- automatic web service discovery;
- automatic web service invocation; and
- automatic web service composition and interoperation.

OWL-S, formerly known as DAML-S (DARPA Agent Markup Language), is developed by the OWL Services Coalition which consists of representatives from universities and industry. The specification is developing fast and is getting lots of attention due to its promise of improving middleware protocols by utilizing the Semantic Web. The following description is based on version 1.0.

In line with the Semantic Web, the OWL-S language itself is defined by a semantic language; OWL. OWL-S is thus an ontology that provides the necessary concepts and relations to describe the general capabilities of web services. This includes representation of the information transformation with inputs and outputs, and state transformation like preconditions and effects.

OWL-S provides moreover a high level description of web services as shown in Figure 16. This top level ontology describes how a Resource is related to a Service, and subsequently how the Service is related to the Profile, the Service Model and the service Grounding. In short, the Profile contains information about what the service does, the service Model describes how the service works, and the grounding describes how the service is accessed. Together, these three concepts are designed to give a total picture of the capabilities of a service.

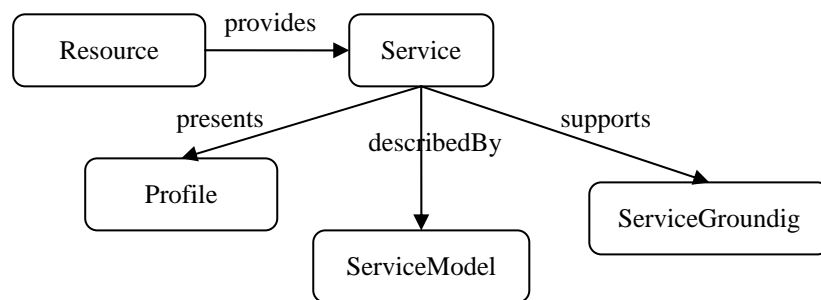


Figure 16: Top level ontology of OWL-S.

Profile

The Profile describes what the service does. An agent investigating the service capabilities would first look into the profile to see if the service provides what is needed. The profile contains the following information:

- Service name, contacts and description.
- Functionality description.
- Taxonomy classification and quality description.

The service name can be used as identification (specified as a string thus not necessary unique), while contact and service descriptions are text provided for humans. Functionality description enables the profile to expose the inputs, outputs, effects and preconditions of the service. The Profile does not provide constructs for describing these, but defines how to refer to functionalities defined in the Service Model. Finally, the Profile contains information to classify the service in various taxonomies and to describe the quality of the service.

Service Model

The Service Model describes how the service works. This is done by expressing the transformation that is undertaken by the service. The two types of transformations are data transformation with the inputs and the outputs and the state transformation with preconditions and effects.

The Service Model intends to provide full expressiveness to model various types of processes, process relations and process control. The following types of processes are defined:

- Atomic Process.
- Composite Process.

- Simple Process.

The Atomic Process is executable in a single step. It is thus a single shot process taking a set of inputs and returns a set of outputs. The atomic process is the smallest building block that does not contain visible sub processes.

The Composite Process is decomposable and means that the process can consist of sub-processes. Ordering and control of the processes is specified with constructs like Sequence, If-Then-Else, Unordered, Choice and Split.

The Simple Process is not executable. It provides an abstract view of a process or a composite set of processes.

Service Grounding

The Grounding describes how to access the service. This involves specifying how to invoke the service using protocols and message formats. Both Profile and Service Model are considered as abstract representations of the service capabilities. The role of the Grounding is to turn these abstract representations into a concrete form that can be used for interaction.

OWL-S relies on WSDL for service interaction. The developers of OWL-S want to use current standards for interaction [13]. OWL-S is therefore complementing WSDL by providing semantics to describe service capabilities, and the grounding is used to map between these two. Figure 17 illustrates that OWL-S is layer on top of WSDL.

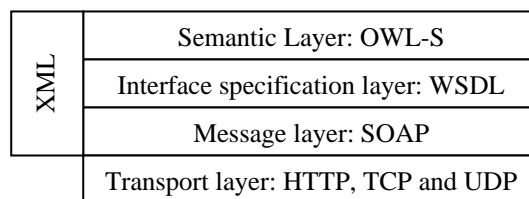


Figure 17: Infrastructure layers used by OWL-S.

Messages communicated with underlying protocols like WSDL and SOAP are constructed with abstract types. Each input and output parameter in the Service Model has to describe the data representation by providing such an abstract type with the property called parameterType.

OWL-S allows the data representation to be specified in both OWL and XML Schema, but WSDL supports only XML Schema representation. This is solved by having the abstract data types defined in the Process Model, and then extending WSDL with information that can refer to the Process Model. An example of how WSDL is extended with information to annotate the translation service is shown in Figure 18. The attribute “owl-s-wsdl:owl-s-parameter” has replaced the attribute “type” in the message part.

Extending WSDL with information as explained in the previous paragraph provides a mapping from WSDL to semantics in the Process Model. OWL-S gives a second way of mapping WSDL to the Process Model through the WSDLGrounding, where pairs between input/outputs and WSDL messages are specified [13].

```
<message name="translateRequest">
  <part name="wordToTranslate" owl-s-wsdl:owl-s-parameter="ontology.owl#wordToTranslate"/>
</message>

<message name="translateResponse">
  <part name="translatedWord" owl-s-wsdl:owl-s-parameter="ontology.owl#translatedWord"/>
</message>

<portType name="translateService">
  <operation name="translate">
    <input message="translateRequest"/>
    <output message="translateResponse"/>
  </operation>
</portType>
```

Figure 18: An example showing how OWL-S extends WSDL with an attribute.

3.4 Capability Matching

Capability matching means to compare the requested service description with the advertised service descriptions. The goal of this comparison is to obtain information on how similar they are. This degree of similarity is used to determine if the advertised service satisfies the requested capabilities. The following description of capability matching is based on Paolucci *et al.* (2002) [11].

Comparing the requested service description with the advertised service descriptions takes all the inputs and the outputs into account. The inputs of the requested service description are compared to the inputs of the advertised service descriptions. And the outputs of the requested service description are compared to the outputs of the advertised service descriptions.

In order to find the service that best matches the requested capabilities it is necessary distinguish degrees of similarities. Using DL comparing methods four different degrees of similarities between concepts can be identified:

- **Exact** – two concepts are semantically equal.
- **PlugIn** – either advertised output contains more than the requested output or the requested input contains more than the advertised input.
- **Subsumes** – inverse of PlugIn – either the requested outputs contains more than what is provided by the advertised outputs or the advertised input needs more than what is requested as input.
- **Fail** – none of the above.

These matching types have different effects on inputs and outputs. Understanding this is quite intricate. I intend therefore to give a detailed explanation in form of an example where the effects of the matching types are clarified. The example consists of comparing the parameters of a service request with the parameters of an advertised service. The services are annotated with the three concepts shown in Figure 19, and are listed in Figure 20.

Exact match means simply that the comparing concepts are equal. This is true for input 1 and output 1 in the example.

PlugIn is used to classify that the output of the advertised service is more general than what is requested. The output can be annotated as dogs and cats, while what is requested is only dogs. The output of service providing dogs and cats can then be plugged in place of the output that is requested and the extra information that is obtained – cats in this case – can be ignored.

Exact match is considered as stronger match than PlugIn because an exact match is closer to what is requested. This holds even though PlugIn matches provide the same – and more than – exact matches, and is based on the definition that a concept representing something general is less specialized than a concept that represents something less general, which is the same as saying that a service providing only dogs is more specialized than a service that provides dogs and cats. This is true in terms of specialization, but not necessary in terms of quality. A service providing dogs is not necessary having more types of dogs or better quality of dogs than a service that provides dogs and cats.

In the example we see that advertised output 2 provides concept A, while what is requested is B. The set in Figure 19 shows that concept A contains B. A is therefore a valid substitution, where A can be plugged in place of B. The output provides more than what is requested.

For the input it is the other way around. The advertised input needs to be more specific than the requested input. This is because the input has to be satisfied. In the example we see that advertised input 2 needs B, while what is requested is A. The request is therefore providing something more general than what is needed and satisfies the input as a PlugIn

The third match – subsumes – means partial fulfillment. This means that the advertised service is not enough to fulfill the requested functionalities. The advertised input 3 require A, but the requested service provides only B, which is a subpart of A. The input of the advertised service is therefore not satisfied.

Output 3 shows how the advertised output B subsumes the requested output. The output of the advertised service is therefore not enough to satisfy the requested output.

Partial fulfillment can however be useful in composition of services. Two services that provide complementary outputs can for example be combined to satisfy some requested output. Services can in the same way be combined to satisfy inputs.

The fourth match – fail – means that the comparing concepts do not have anything in common, they are disjointed.

Li and Harrocks (2003) [23] have identified a concept match that is weaker than subsumes but stronger than fail. This is intersection, where a part of the requested service parameter overlaps a part of the advertised parameter. This is not shown in the example. Intersection can in the similar way as subsumes be used for composition.

Distinguishing these types of matches is necessary for finding the most similar service(s). This discrete scale is used to give each advertised service a degree of match, which makes it possible to order the advertised services according to their similarity of the requested service.



Figure 19: Ontology represented with concepts, and various ways of describing subsumes relations.

Requested Service	Advertised Service	Type of Match
Input 1: A	Input 1: A	Exact
Input 2: A	Input 2: B	PlugIn
Input 3: B	Input 3: A	Subsumes
Input 4: A	Input 4: C	Fail
Output 1: A	Output 1: A	Exact
Output 2: B	Output 2: A	PlugIn
Output 3: A	Output 3: B	Subsumes
Output 4: A	Output 4: C	Fail

Figure 20: Classification of various types of concept matches.

3.5 Available Software

RACER

RACER is a software system for representation and reasoning with DL [26]. It uses highly optimized tableau calculus – a mathematical method – for reason with concepts and individuals. It supports subsumption, consistency checking and determination of sub-concepts and super-concepts. These functionalities are provided as a set of commands.

RACER is a standalone service which is accessed through HTTP using either the DIG format or JRacer. The latter is java interface that provides an API for the RACER commands. The reasoner supports ontologies expressed in the following languages; DAML+OIL, OWL, RDF and RDF Schema. These ontologies can be accessed either locally or from URIs.

RACER can be used in non-commercial research free of charge. To use the software in commercial products licenses are necessary. The use of RACER is intuitive and it comes with an excellent user manual.

Pellet OWL Reasoner

Pellet is an open-source Java based OWL DL reasoner [25]. It is based on the tableaux algorithms developed for expressive DLs. It supports all the OWL DL constructs including the ones about nominals, namely owl:oneOf and owl:hasValue.

OWL-S API

Mindswap, Maryland Information and Network Dynamics Lab Semantic Web Agents Project, is developing a library for accessing OWL-S ontologies [27]. The library provides a java API that can be used to interact with OWL-S annotated web service descriptions.

The library is under development and seems to follow the progress of done by the OWL-S Coalition.

SNOBASE

IBM Ontology Management System (also known as SNOBASE) is a framework for loading ontologies from files and via the Internet and for locally creating, modifying, querying, and storing ontologies [28]. It provides a java interface for reasoning with ontologies specified in languages such as RDF, RDF Schema, DAML+OIL, and OWL.

4. Algorithms, Design and Implementation

This chapter covers the necessary algorithms and design approaches for realizing the theory of Semantic Web Services in an implementation of a Matchmaker. The architecture of Matchmaker is first introduced, followed by the design approach taken in annotating services with OWL-S. The design of the Matchmaker and the developed algorithms are then explained in detail. The algorithms for matching of capabilities are inspired by Paolucci *et al.* [11].

4.1 Architecture

Figure 21 shows the main components of the Matchmaker including external sources of data and functionalities. Semantic matching and composition is provided by the Matchmaker through the method `findServiceCompositions`. The Matchmaker maintains a list of advertised Semantic Web Services with the methods `AddService` and `RemoveService`.

The semantic service capabilities of available Semantic Web Services are accessed through the OWL-S library which parses the OWL documents and provides a number of functionalities for retrieving input and output parameters.

Semantic reasoning is provided by Racer Inference Engine which is a standalone server is accessible via a Racer Java library [26]. Domain ontologies are loaded into the inference engine with the `owlReadDocument` method and then used for comparing concepts for equality and subsumption.

For demonstration purposes an interactive Java Servlet is developed (not shown in the figure). It enables users to experiment with matching and composition of services where their capabilities are graphically displayed with inputs and outputs.

The Matchmaker contains the algorithms developed and explained in this chapter, while the other components (OWL-S Lib, Racer Java Lib and Racer Inference Engine) are implementations provided by their respective developers. The Matchmaker is implemented as a Java library, but could equally have been provided as a Web Service.

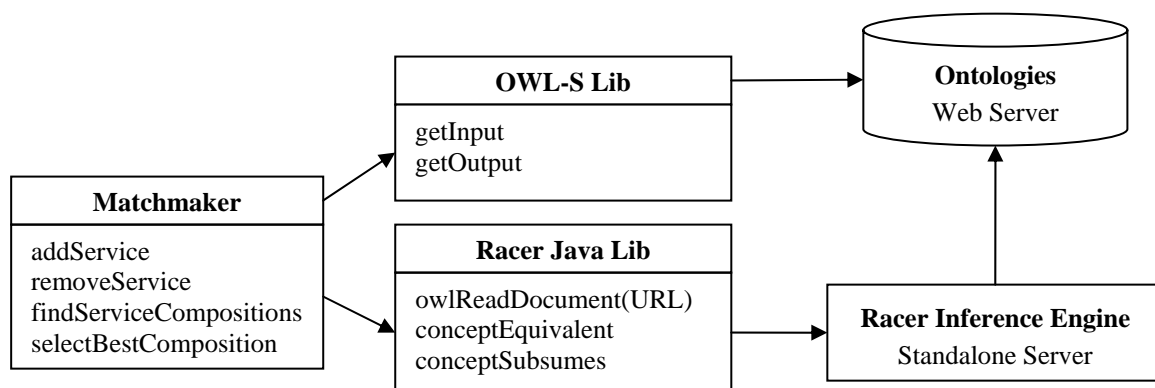


Figure 21: Main components for the Matchmaker.

4.2 Service Capabilities in OWL-S

Advertised service capabilities and requested service capabilities are annotated using the OWL-S ontology, where capabilities consist of a set of parameters. These parameters contain semantics that specify what information is needed for communicating with the service, and can thus be thought of as constraints on the information the parameters can contain. Figure 22 shows the design approach taken in creating instances of the parameters and then relating them to concepts in domain ontologies. `myProcess` is an instance of `AtomicProcess`, and is provided with `hasInput` and `hasOutput` properties that point to instances of `Input` and `Output` concepts, respectively [4].

Semantic Data Structures

Each of the parameters (`Input` and `Output`) has a property, `parameterType`, which links the parameter to an OWL concept defined in a domain ontology. As we demonstrate in the next chapter, such a concept is a semantic specification of the parameter, where data and semantics are mixed. This is achieved by defining properties that connect the concept to data values or to other concepts, where in the latter case the concepts can have their own properties. As explained while introducing the Semantic Web in chapter 2, restrictions on properties are enforced with definition of axioms.

These concepts with their properties can be thought of as specifications of structured semantically described data. To our best knowledge there exists no adequate name for such a semantic structure, thus we coin the name *Semantic Data Structure* (SDS). This neologism is made because of two main reasons. First, Semantic Data Structures used in annotating service capabilities should be distinguished from domain ontologies. A Semantic Data Structure is not necessarily the domain ontology, but is part of a domain ontology, because domain ontologies can contain many SDSs. The second reason is due to their importance in describing the parameters of service capabilities. The parameters are represented by Semantic Data Structures, and thus form the fundament for semantic matching and composition. Semantic Data Structures are thus specially designed for representing parameters.

Semantic Data Structures specify the structure and semantics of information that is communicated as service parameters. Information that is communicated through messages has to conform to these structures. It is therefore natural that Semantic Data Structures are defined as OWL classes, and where the information communicated with messages are instances of these classes.

A Semantic Data Structure can consist of webs of concepts, but each SDS has one concept that is the core of the structure. This concept is the center of the ontology, because all concepts relate back to this concept through axioms. Semantic Data Structures consist therefore of one main concept that we call the *core concept*.

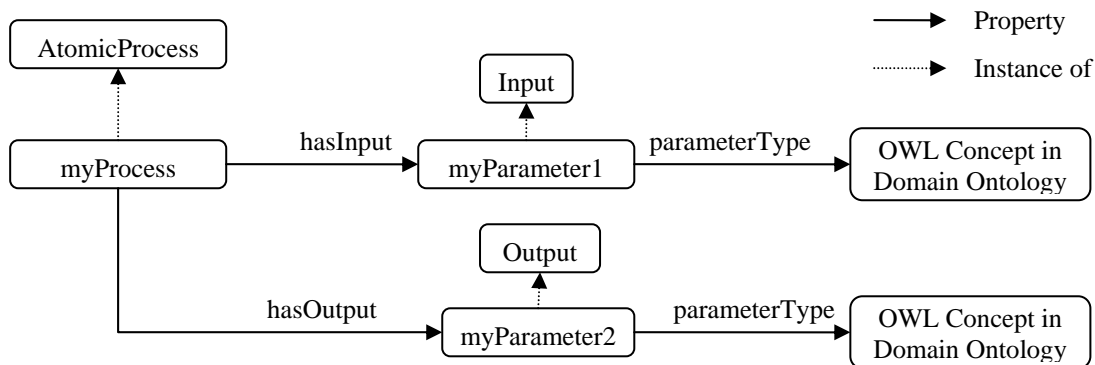


Figure 22: Service capabilities in OWL-S.

4.3 Algorithms and Design

Algorithms developed in this work match parameters as Exact, PlugIn and Fail (see chapter 3). This is because service with capabilities that match as Exact and PlugIn can be used directly. Exact matches of capabilities are perfect matches, and in case of PlugIn matches of input parameters, the client provides more information than what is needed, and in case of PlugIn matches of the outputs, the service returns more information back to the client than what is requested.

To match capabilities for Subsumes (inverse of PlugIn) is easy, but because the parameter, represented as a Semantic Data Structure, only fulfills the request capabilities partly, the service can not be used directly. The missing parts in the Semantic Data Structures have to be identified and complemented by other services.

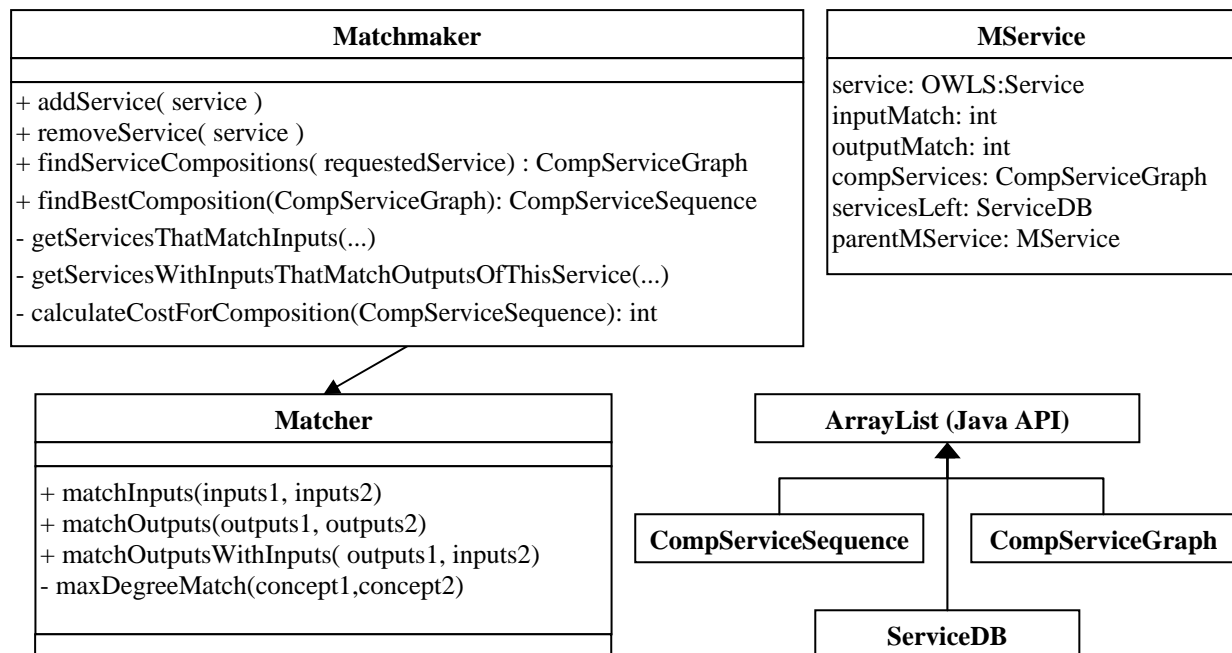


Figure 23: Class diagram emphasizing algorithms in the Matchmaker.

The design of the Matchmaker is shown in Figure 23. The main class provides four public methods; `addService` to add service capabilities to the registry, `removeService` to remove the service from the registry, `findServiceComposition` to get all possible compositions for given requested service capabilities, and `findBestComposition` which takes compositions as input and finds the best composition. Compositions are stored in graph structures called `CompServiceGraph`, and single compositions are stored as sequences using `CompServiceSequence`.

Services are represented with the class `MService`, which contain information about the overall best input and out match, and information used for service composition. The class `Matcher` contains low-level methods for matching parameters and deals with interaction between the Matchmaker and the matching engine.

4.4 Matching of Input and Output Parameters

Semantic Data Structures are compared and the best match is found with the method `maxDegreeMatch` shown in Figure 24. The method first verifies that the Semantic Data Structures exist, and then checks them for Exact match (semantic equality) and then `PlugIn` match (semantic inverse subsumption).

The order of the parameters provided to this method is critical for it to work as expected. That is because the parameters, `concept1` and `concept2`, are compared for subsumption, and thus where the parameter order (`par1`, `par2`) has the inverse result as the parameter order (`par2`, `par1`). When this method is used by method `matchInputs` and `matchOutputsWithInputs` the order should be (`reqParameter`, `advParameter`), and when used by method `matchOutputs` the order of the parameters should be (`advParameter`, `reqParameter`).

```

int maxDegreeMatch( concept1, concept2 ):
    mEngine.reset()
    if(!mEngine.conceptExists(concept1) or !mEngine.conceptExists(concept2))
        return fail

    if(mEngine.conceptEquivalent(concept1, concept2))
        return exact
    else if(mEngine.conceptSubsumes(concept2, concept1))
        return plugin

    return fail

```

Figure 24: Matching concepts (`maxDegreeMatchOfConcepts`).

`MatchInputs`, shown in Figure 25, checks if all the requested inputs (`reqInputList`) are provided in the inputs of the advertised service (`advInputList`). The algorithm maps `reqInputs` to `advInputs` by iterating over `reqInputs`, and selects the best matched input parameters from `advInputs`. Exact match is defined as being a better match than `PlugIn`.

The parameters are mapped in a one-to-one fashion so that each requested input match to one and only one advertised input. The weakest match of all the best matches is found, and represents the overall match between two service capabilities. The order of parameters is therefore not taken into account.

The methods `matchOutputs` and `matchOutputsWithInputs`, algorithm for comparing outputs of two service capabilities and algorithm for comparing outputs with inputs, are designed with same approach as `matchInputs`.

```

Int matchInputs( reqInputList, advInputList ):
    globalDegreeMatch = exact

    for all reqInput in reqInputList
        bestMatch = default as fail
        for all advInput in advInputList
            match = maxDegreeMatch( reqInput, advInput )
            if(match > bestMatch)
                bestMatch = match;
                bestMatchInput = advInput
        reqInputList.remove( bestMatchInput )
    if(bestMatch == fail)
        return fail
    if(bestMatch < globalDegreeMatch)
        globalDegreeMatch = bestMatch
    return globalDegreeMatch

```

Figure 25: Matching input parameters (`matchInputs`).

The algorithm `getServicesThatMatchInputs` creates a list of advertised services that satisfy the requested inputs (Figure 26). The requested inputs are compared to the inputs of all the advertised service capabilities using `matchInputs`. Services that match as valid inputs – greater than fail – are recorded together with a list of the remaining advertised services, which is information used for composition.

```

CompServiceGraph getServicesThatMatchInputs(reqService, advServices):
    CompServiceGraph inputMatches

    for all advService in advServices
        MService service
        service.inputMatch = matchInputs(reqService.inputs, advService.inputs)
        if(service.inputMatch > fail)
            service.servicesLeft = copy of advServices
            service.remove( advService )
            inputMatches.add( service )
    return inputMatches

```

Figure 26: Gathering services that satisfy requested inputs (`getServicesThatMatchInputs`).

4.5 Composition

A key functionality is to automate composition of services from a client's definition of the initial and final states. These states are semantic descriptions of information, and composition

means to find a sequence of combined services that provides an information transformation from the initial state to the final state. Constructing such compositions is achieved by resolving the dependences between services based on their inputs and outputs. A service that, for example, has inputs that match with the requested initial inputs can have outputs that match with the inputs of more than one service. Compositions can therefore be thought of as branches and thus be represented as a graph, where nodes correspond to services and edges represent information communicated between services.

Figure 27 illustrates compositions as a graph where nodes A and B are services that match with the requested initial inputs. The outputs of these services are then subsequently matched with inputs of the remaining unused advertised services, where each service-node keeps a list of the remaining unused services for this branch (attribute `servicesLeft` in `MService`). Branches of nodes are exhaustively expanded with all possible compositions, but without allowing the same service to be used more than once, and thus avoiding infinite loops. In the graph we see that services A and B have inputs that match with the requested inputs X. Service C has inputs that match with the outputs of service A, and service D has inputs that match with outputs of service B.

Services that are inserted into the graph have their outputs both matched with the requested final outputs and with the inputs of the remaining unused available services for that branch node. The rationale for this is that they represent different compositions, and even though one of the compositions has reached the final state, the other composition can for some reason be preferred. As an example, consider Figure 27 where service A branches off into two nodes of the same service C. The reason for this is that the output parameter Y' of service C matches as a PlugIn with Y, and is therefore a match with the requested final output, and – at the same time – match with the inputs of the unused advertised service E. In this way two different compositions are created.

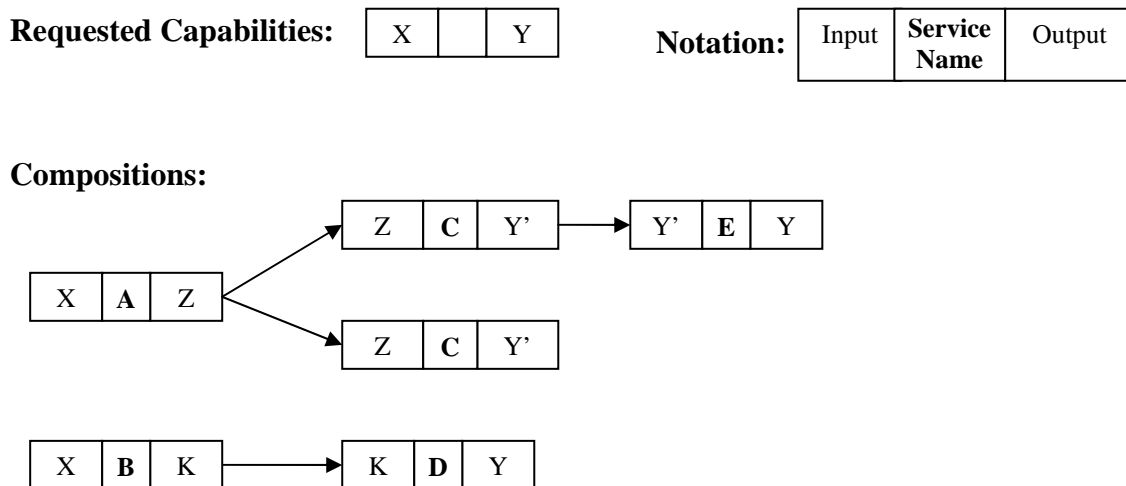


Figure 27: Compositions represented as graphs.

The algorithm `getServicesWithInputsThatMatchOutputsOfThisService`, shown in Figure 28, is similar to `getServicesThatMatchInputs`, but instead of matching inputs with inputs, the outputs are matched with inputs, so that a chain of services can be made. The algorithm uses `matchOutputsWithInputs` which is constructed in the same way as `matchInputs` (Figure 25).

```
CompServiceGraph getServicesWithInputsThatMatchOutputsOfThisService( outputService ):
    CompServiceGraph matchedServices

    For all inputService in outputService.servicesLeft
        MService service
        service.outputMatch =
            matchOutputsWithInputs(outputService.outputs, inputService.inputs)
        if(service.outputMatch > fail)
            service.servicesLeft = copy of outputServices.servicesLeft
            service.servicesLeft.remove( inputService )
            service.parentService = outputService
            inputMatches.add( service )
    return matchedServices
```

Figure 28: Finding services with inputs that match outputs of a given service.

The developed algorithm for service composition is shown in Figure 29. A stack is used to create the graph where root-nodes are pushed onto the stack and then subsequently dealt with. The main while-loop continues as long as there are nodes to process, and for each loop a service is taken off the stack and matched against the requested outputs and with the remaining unused services – the latter with `getServicesWithInputsThatMatchOutputsOfThisService`. If the current service matches with the requested service and at the same time matches with the inputs of one or more advertised services, then a sibling in the graph is made by creating copy of the service-node. The algorithm identifies whether the current service is a root-node, in that case the copy is added to the `compGraph`, or if the service has parents, then the attribute `parentMService` is used to insert a copy into the graph.

The parameter (`compGraph`) – which contains the services that matched with the requested inputs – is used as the base of the composition. The graph is constructed from this initial set of services by adding service-nodes. The method does therefore not return anything but void.

```

void buildCompositionGraph(CompServiceGraph compGraph, Service reqService):
    Stack compStack = all root nodes in compGraph

    while(compStack is not empty)
        MService topStackService = compStack.pop()
        topStackService.outputMath = matchOutputs(outputs of topStackService,
                                                    outputs of reqService)

        CompServiceGraph matchedServices =
            getServicesWithInputsThatMatchOutputsOf(topStackService)

        if(topStackService.outputMatch > fail and matchedService is not empty)
            /** Matches with both reqOutputs and other advertised services */
            if(topStackService.parentMService != null)
                topStackService.parentMService.compServices.add(topStackService.clone());
            else
                compGraph.add(topStackService.clone());

        if(matchedServices is not empty)
            /** Add sub-nodes */
            topStackService.compServices = matchedServices
            compStack.addAll(matchedServices)

```

Figure 29: Creation of all possible compositions (buildCompositionGraph).

Figure 30 shows the main control flow of method findServiceCompositions. First a list of those advertised services that match with the requested inputs are found (getServicesThatMatchInputs). The method buildCompositionGraph is then used with this list to create a graph with all possible compositions of advertised services, and which is returned if it's not empty.

```

ServiceCompositionGraph findServiceCompositions(Service reqService):
    advServices = list of advertised services
    ServiceCompositionGraph advServicesInputMatch =
        getServicesThatMatchInputs(reqService, advServices)

    buildCompositionGraph(advServicesInputMatch, reqService)

    if(advServicesInputMatch is not empty)
        return advServicesInputMatch
    else
        throw exception "No compositions found"

```

Figure 30: Main control flow for creating all possible compositions (findServiceCompositions).

4.6 Composition Selection

Agents need a way to autonomously select one of the possible compositions. An algorithm for selection of the most optimal composition is therefore developed, and is shown in Figure 31. A depth first search is conducted in a recursive fashion where the cost of each branch, representing a valid composition, is calculated and compared to the best composition found. And if the current branch is better than the best composition found, meaning that the cost is lower, it simply replaces the best composition found.

A depth first search is used because all compositions should be considered. This can become a problem when the number of services gets high. For this reason a `maxDepth` is introduced, while an even better solution would be an algorithm that finds the overall first N best compositions. For this a breath first search has to be conducted, where the search stops after finding the first N complete compositions.

Valid compositions are those where the last service in the chain has outputs matched with requested outputs. The algorithm keeps track of the best composition found (`bestCompSeq`) and compares this with the current branch (`currentCompSeq`).

```
CompServiceSequence selectBestComposition(CompServiceGraph compGraph):
    CompServiceSequence currentCompSeq
    CompServiceSequence bestCompSeq

    For all service in compGraph /** These are root-nodes **/
        bestCompReq(service,currentCompSeq, bestCompSeq,1)
    return bestBestSeq

void bestCompReq(MService service,
    CompServiceSequence currentCompSeq
    CompServiceSequence bestCompSeq, int depth):

    if(depth > maxDepth) return void
    currentCompSeq.add(service)
    if(service.compServices.isEmpty() and service.outputMatch > fail)
        /** Leaf node and valid composition **/
        int currentCompCost = calculateCostForComposition(currentCompSeq)
        int bestCompCost = calculateCostForComposition(bestCompSeq)
        if(bestCompSeq.isEmpty() or (currentCompCost < bestCompCost ))
            bestCompSeq.clear()
            bestCompSeq.addAll(currentCompSeq)
        else if(service.compServices.size() > 0)
            For all service in service.compServices
                bestCompReq(compGraph,currentCompSeq, bestCompSeq,depth+1)
            currentCompSeq.remove(service)
```

Figure 31: Finding the best composition (findBestComposition).

The cost of each composition is calculated by a separate method called `calculateCostForComposition` and is shown in Figure 32. The cost depends on the number of involving services, where few services are favored, and the quality of matches between services in the composition.

This algorithm traverses the service composition, starting with service that matches with the requested inputs and ends with the service that matches with the requested outputs, and sums the costs of each match. The cost depends on the type of match, where Exact match costs 1, and PlugIn match costs 2. Finding the best service composition is thus not only a matter of finding the shortest path in the graph, but depends on the types of matches in the composition.

```
int calculateCostForComposition(CompServiceSequence compSeq):
    int cost = 0
    int exactMatchCost = 1
    int pluginMatchCost = 2

    For each service in compSeq
        if(service.inputMatch == exact)
            cost += exactMatchCost
        else if(service.inputMatch == plugin)
            cost += pluginMatchCost

        if(service.outputMatch == exact)
            cost += exactMatchCost
        else if(service.outputMatch == plugin)
            cost += pluginMatchCost

    return cost
```

Figure 32: Computing the cost of a single composition (calculateCostForComposition).

5. Case Study – Interactive Simulated Vascular Reconstruction

This chapter presents a case study where two Grid Services are developed into Semantic Grid Services by applying the theory of Semantic Web Services. The purpose of this study is to validate the theory, algorithms and implementation developed in previous chapters.

The scenario is first introduced with a description of the involved Grid Services. A domain ontology is then developed from the description of these services, and then used to annotate their service capabilities. The domain ontology is then used to describe the requested capabilities that the client or agent needs. Matching, composition and service selection based on these service capabilities are then discussed.

5.1 Introduction

Problems where blood vessels are weakened or cluttered are called vascular disorders, and lead to reduced or blocked blood circulation, causing in many cases stroke, and eventually death⁴. Medical data acquired by e.g. Magnetic Resonance Angiography (MRA) may be used by specialists to detect these disorders, and treatment consists of reconstructing defected vessels or adding bypasses so that the blood flow can normalize. The best treatment is however not obvious. University of Amsterdam is therefore researching and developing systems that allow surgeons to see the results of an operation a priori [10].

The Grid-based Virtual Laboratory AMsterdam (VLAM-G) is “a science portal for remote experiment control and collaborative, Grid-based distributed analyses in applied sciences, using cross-institutional integration of heterogeneous information and resources” [41].

Grid Services [39] in this project are being developed to enable pre-treatment planning of vascular disorders, and includes interactive model-manipulation of virtual blood vessels in 3D and on-demand simulation of blood flow circulation. These services are used within a Problem Solving Environment [49].

Grid Services are with OGSi and later WS-Resources considered as Web Services with extra functionalities for statefulness including service creation and termination [39] [40]. Semantic annotation of service capabilities discussed in this work has been accomplished by extending the abstract service interfaces (WSDL) with semantics from ontologies. And since Grid Services uses the same abstract interfaces as Web Services, we maintain that semantic description of Web Services applies equally to Grid Services. Hence, we define Semantic Grid Services as Grid Services with semantically annotated capabilities.

Scenario

For our case study, the problem solving environment (PSE) involves the usage of two local components and two Semantic Grid Services. Dataflow between components is shown in Figure 33 and description of the Grid Service parameters is shown in Figure 35. The Model Manipulation component is an interactive 3D environment where the user works with a model of the virtual blood vessels acquired from the MRA scan. The user (surgeon), for example,

⁴ In memory of the author's father that passed away due to stroke in 1994.

inserts or modifies a bypass around the region of interest (defected area), and then submits the data of artery's geometry (ArteryLBMGrid) to the Blood flow Simulation Service that does the necessary calculations for the simulation of the blood stream. The data of the artery is in a specific format used by the PSE's implementation of the Lattice Boltzmann Method for fluid dynamics [50]. In addition to data of the artery the service needs a configuration of the specific simulation that is requested and includes information like blood flow boundary conditions and simulation time length.

The output of the Blood flow Simulation Service is visualization of the blood flow in a visualization-specific data format. The Grid Visualization Kernel (GVK) [51] is a service that takes this Visualization Data as input, and converts it to 3D polygons in the VTK format, here called VTKPolyData. The data is then sent to the local rendering component where it is displayed to the user. And, which repeats the procedure until the desired effects are obtained.

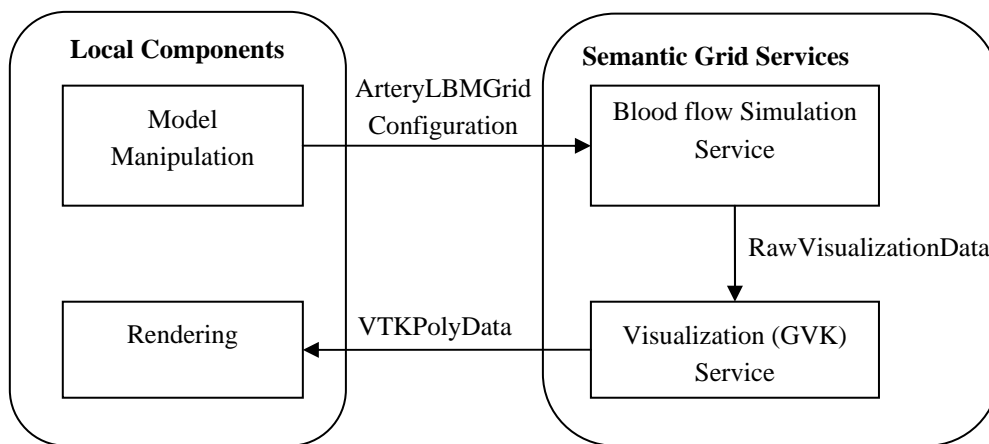


Figure 33 Dataflow between components and Grid Services in case study scenario.

The two Semantic Grid Services – Blood flow Simulation and Visualization Service – are annotated with a domain ontology developed in the next section. Formulation of the requested service capabilities should ideally have been part of the Problem Solving Environment, but is currently done manually by selecting concepts from domain ontologies. The Matchmaker – developed in the previous chapter – is then used to locate and compose the services based on the requested capabilities. This results in a workflow description that is used by the Problem Solving Environment to invoke the services.

Flow Simulator Service	
Input	Data Model of Artery
Input	Configuration
Output	Raw Visualization Data

Visualization Service	
Input	Raw Visualization Data
Output	Data in VTK format

Figure 34: Description of the two Semantic Grid Services.

Benefits of Semantic Grid Services

The first benefit of using semantics in this scenario is that Grid Services are discovered based on capabilities, and since the developer or the user has configured the system to use these specific capabilities, the system can automatically integrate the discovered services into the application. The second benefit is that Semantic Grid Services can be composed automatically based on their capabilities and then the best composition is autonomously selected. The overall benefit is that the process of finding, composing and integrating services become easier, because it is largely automated.

5.2 Domain Ontology

Figure 35 shows a domain ontology that is defined to annotate the capabilities of the Simulation and Visualization Services. The ontology is developed in the context of the Virtual Lab Environment with information provided by domain experts. The ontology is constructed to demonstrate specific features in semantic matching and composition, and is therefore incomplete.

Concepts in the ontology are organized in a hierarchy where VLE is the top concept that most of the other concepts are subclasses from. Two categories are defined as concepts Simulator and Visualization, and are subclasses of VLE. The concept LBM is in the category of simulations, and is therefore defined as subclass of Simulation. LBM is furthermore subdivided into LightLBM, which is light version of the implementation, and FullLBM, which is a fictitious full version.

Definition of Semantic Data Structures

LightLBM and FullLBM are different services that take different configurations as inputs. FullLBM requires two properties in the configuration; Viscosity and ReynoldsNumber, while LightLBM only needs Viscosity. In the ontology this is solved by defining a concept ConfigLight (categorized as LightLBM) with properties Viscosity, and then defining another concept ConfigFull that is subclass of ConfigLight, and thus derives the property Viscosity from ConfigLight. The concept ConfigFull defines furthermore a property ReynoldsNumber, which means that two properties need to be provided for this concept – Semantic Data Structure – to be used.

ConfigFull and ConfigLight are example of concepts in the ontology that are designed to represent capability parameters. What is important to realize is that these concepts are semantic structures that contain data – Semantic Data Structures. The content of the parameters are described with semantics, where the semantics specify both meaning and constrains of the data the parameters can contain. A parameter annotated with the concept ConfigLight specifies, for example, that the property Viscosity has to be provided, with a data value of type Integer. The previous chapter offers an extensive discussion on the Semantic Data Structures.

RawVisData (subClassOf LightLBM) is defined as equivalent as RawVisualizationData (subClassOf GVK) and means that they can be used interchangeable. This equality is in this case introduced to show an example of how a mapping is defined and, at the same time, show the flexibility of ontologies. Because of this equality, it is not necessary for the concept RawVisData to define a property LocatedAt, since it is derived from RawVisualizationData.

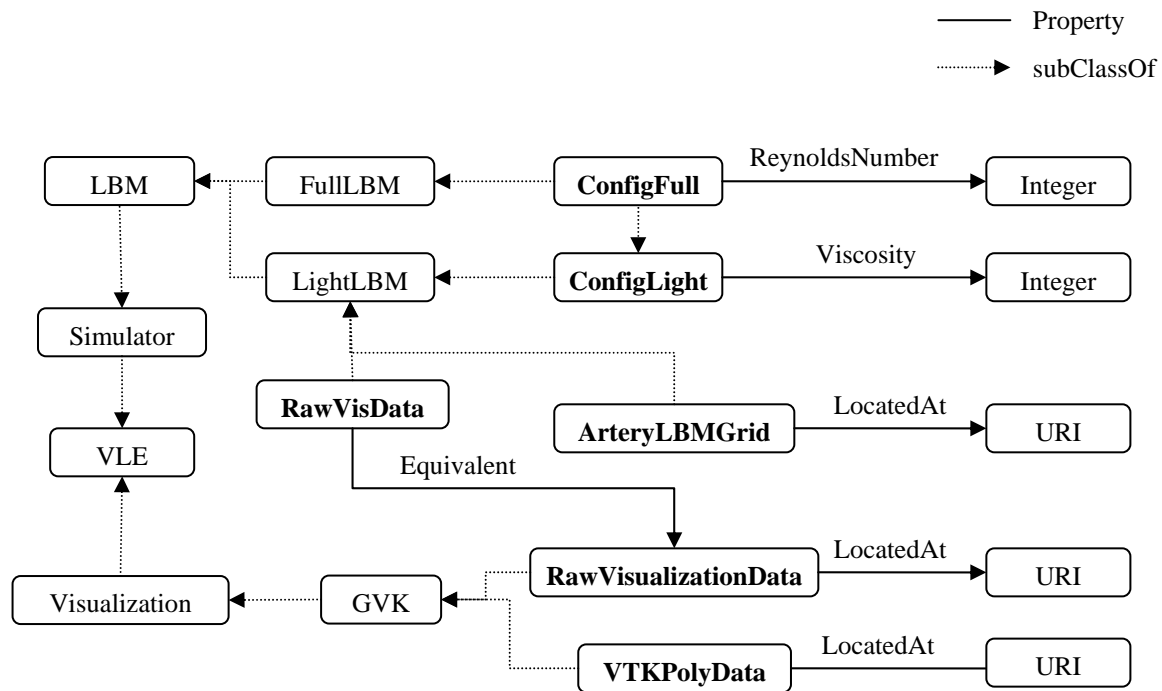


Figure 35: A simplified Domain Ontology. Core concepts of Semantic Data Structures are in bold.

5.3 Annotation and Matching of Capabilities

Concepts in the domain ontology are used to annotate requested and advertised service capabilities as shown in Table 3.

Requested Simulator Service		Advertised Simulator Service		Type of Match
Input	ArteryLBMGrid	Input	ArteryLBMGrid	Exact
Input	ConfigFull	Input	ConfigLight	PlugIn
Output	RawVisualizationData	Output	RawVisualizationData	Exact

Table 3: Requested and advertised Flow Simulation service capabilities with semantic matching of their parameters.

The table shows that ArteryLBMGrid and RawVisualizationData match as exact between requested capabilities and advertised capabilities. This is no surprise because the same concepts are used in request and advertised parameter.

ConfigFull compared to ConfigLight matches as PlugIn, because ConfigFull subsumes ConfigLight. ConfigFull is a subclass of ConfigLight, and thus contains everything that ConfigLight contains. If the inputs of the service request are thought of as what the client can provide to the service, then the case is that the client can provide more information than what is necessary for using the service. The client requests a service with capabilities that taking configuration with Viscosity and ReynoldsNumber. However, the service only needs Viscosity as part of its configuration parameter. Thus using this service means that the ReynoldNumber will be discarded – not used by the service even though it is provided.

Consider the requested and advertised capabilities of the Visualization service in Table 4. Both parameters match as exact, even though RawVisData and RawVisualizationData are different concepts in the domain ontology. The parameter matches as exact because the concepts are defined as equal, which is also possible across different ontology documents.

Requested Visualization Service		Advertised Visualization Service		Type of Match
Input	RawVisData	Input	RawVisualizationData	Exact
Output	VTKPolyData	Output	VTKPolyData	Exact

Table 4: Requested and advertised capabilities of the Visualization service with semantic matching of their parameters.

5.4 Composition

Table 5 shows a set of requested capabilities annotated with the domain ontology. What is requested is a service that takes ArteryLBMGrid and ConfigLight as input, and expects VTKPolyData in return. Five different compositions that satisfy these requested capabilities are displayed in Figure 36.

The compositions consist of two simulation services (FlowSim 1 and FlowSim 2), two visualization services (Vis 1 and Vis 2) and one service that provide both; simulation and visualization (FlowSimAndVis). Composition number 1 consists of first FlowSim 1 then Vis 1, another one is FlowSim 1 then Vis 2.

Requested Capabilities	
Input	ArteryLBMGrid
Input	ConfigFull
Output	VTKPolyData

Table 5: Requested capabilities.

5.5 Composition Selection

The Matchmaker provides a method – findBestComposition – that can be used by agents to find the most optimal composition. Figure 36 shows five compositions where each has an associated cost that reflects the quality the composition. The cost increases with the number of services taking part in the composition and the type of matches between each service, thus lower cost for a composition means higher quality.

For composition 1: Inputs of FlowSim 1 matches as PlugIn with requested inputs, hence has the cost 2. Output of FlowSim 1 matches as Exact with inputs of Vis 1 and thus adds the cost 1, and finally output of Vis 1 matches as exact with requested outputs and adds cost 1, which makes the total cost 4. For composition 3: All matches are exact, thus the total cost for the composition is 3.

The algorithm for finding the most optimal composition, developed in the previous chapter, selects the first composition found with the lowest cost. It is therefore not possible to say which one – of the three compositions with cost 3 – that will be selected.

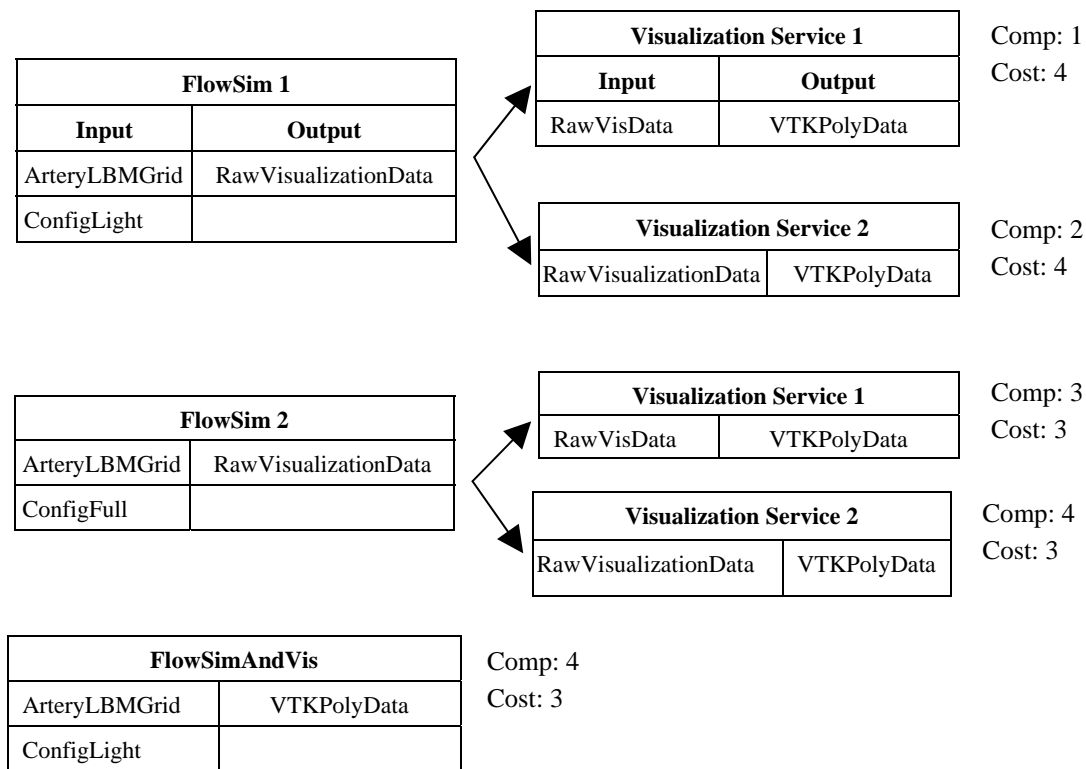


Figure 36: Five possible compositions.

6. Analyses and Discussion

Based on experience with theory, implementation, and the case study, we provide in this chapter an analysis where we discuss various aspects of Semantic Web Services.

6.1 Domain Ontologies with High Complexity

Semantic Data Structures are defined in domain ontologies to annotate advertised service capabilities, and then used by developers to annotate requested service capabilities⁵. The latter is either done manually or with an aiding tool where the developer for example browses the ontology and selects appropriate concepts.

Each parameter in the capabilities points to one *core concept*. Parameters should be able to contain more than a simple data value; hence with axioms, the concept can include a large number of properties, connecting the concept to data values and to other concepts through properties, thus creating a complex Semantic Data Structure of concepts and relations. When the complexity of these structures rise it becomes increasingly more difficult to determine which concepts that are designed as core concepts to represent parameters. This, because it might not be clear from looking into the ontology which concept that is designed as the core concept; there might simply be so many. But a more important reason is that tools, using semantic reasoning, can not tell concepts and core concepts apart. Developers can therefore not with certainty determine which concepts in an ontology that should be used in annotating requested service capabilities.

The matching techniques discussed in chapter 3 and 4 can only find similarities (Exact and PlugIn) between parameters annotated with the same core concept⁶. For example a parameter annotated with F in the domain ontology shown in Figure 37, would not match with a parameter annotated as B. It is therefore critical that core concepts, developed to annotate service capabilities, are used to annotate requested capabilities. If not, similarities can not be detected.

Ontologies should therefore contain information that distinguishes core concept from other concepts. The solution we propose is to define a concept, that works as a top-level concept, and which all core concepts derive from. And since this information is part of the ontology, semantic reasoning can be used to tell core concepts from the rest.

Instead of annotating requested capabilities with *any* concept from the ontology, only concepts that derive from a certain common concept can be used. This will improve the process of semantic annotation and the likelihood of matching.

⁵ Already defined ontologies can also be used in annotating other advertised service capabilities.

⁶ Two concepts are considered the same if they are related with an equality (see VLE Domain Ontology).

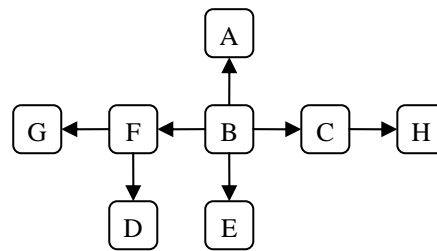


Figure 37: Example ontology. A parameter annotated as F would not match a parameter annotated as B.

Revised Design of OWL-S

OWL-S connects parameter (in the OWL-S ontology) to concepts (in the domain ontology) by having each parameter specifying the property `parameterType` (illustrated in chapter 4). As of version 1.0, the range of this property is unspecified, meaning that the value of this property can be anything. The rationale for this – as explained in chapter 3 – is to allow the Semantic Data Structures to be specified as both XML Schemas and OWL concepts. From a Semantic Web Services’ point of view we are only interested in annotating capabilities with semantics, which makes XML Schemas, as core concepts, obsolete. In other words, OWL-S should not allow parameters to be purely represented as XML Schemas.

One solution to the core concept problem is to define core concepts as subclasses of the concept `Parameter`. This is however not a good solution because concepts like `Input`, `ConditionalOutput` and `UnConditionalOutput` derive from this concept. And having the core concept subclass one of these would constrain core concept (in the domain ontology) to certain types of parameters like `Input` or `ConditionalOutput`.

The solution we propose is to only allow concepts of a certain type to be used in the `parameterType` relation. This is achieved by introducing a new concept in the OWL-S ontology, call it `CoreParameter`, and then defining a range restriction for the `parameterType` property to this concept. In this way only concepts that subclass `CoreParameter` are valid concepts in annotating capabilities, and core concepts are distinguished from other concepts in the ontology. Figure 38 shows the revised design of OWL-S.

We can argue that restricting parameters to be only annotated with concepts that subclass a certain concept is a too strict constrain, e.g. for reusability of domain ontologies over time and across different framework. We believe this will not be a problem, because we are dealing with one simple concept that can easily be mapped to other concepts.

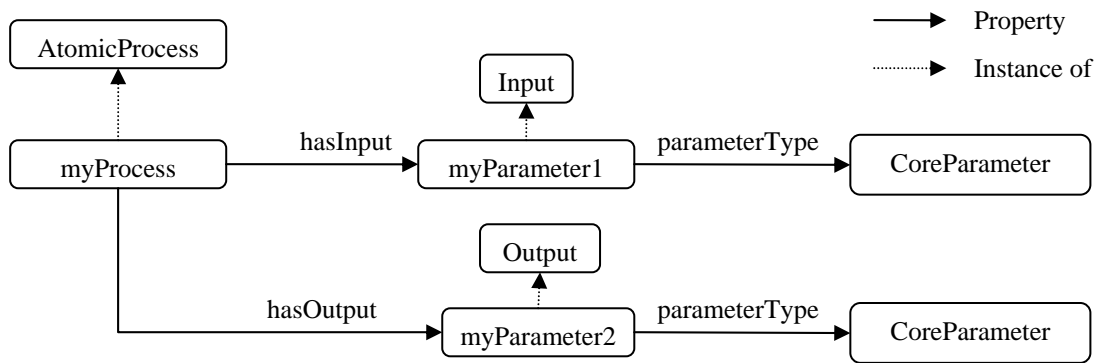


Figure 38: Proposed redesign of OWL-S.

6.2 Scalability of Semantic Matching Architectures

Semantic matching means to measure the similarity between two concepts defined in ontologies, and is used during service discovery and composition while comparing requested capabilities with advertised capabilities. An architecture for semantic matching, with emphasize on the semantic matching part, is illustrated in Figure 39, and shows that the process starts with a request for capabilities, where they are compared to the advertised service capabilities in the database.

The actual matching is done by a matching engine, which receives information about the two concepts that are to be compared, for example as two URIs. The matching engine loads the necessary ontology documents, either from a database or from the web, and conducts the requested comparison, by for example checking for equality or subsumption.

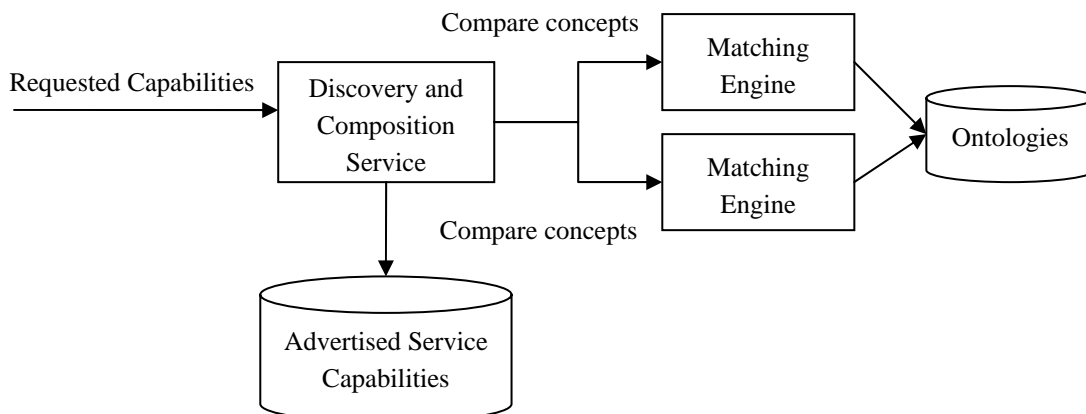


Figure 39: Design of a scalable matching architecture. Arrows depict requests.

Matching conducted by the matching engine – call them matching jobs – are independent in that they do not depend on each other, and the order they are compared do not matter. It is thus trivial to parallelize the usage of matching engines by invoking more than one matching engine at the same time. Figure 39 shows for example the use of two matching engines where each receives half of the jobs. Extra matching engines can thus be added to the

architecture, and a theoretical linear scalability is obtained. Job distribution is what the parallel computing community describes as embarrassingly parallel [16].

The bottlenecks for linear scalability are access to ontologies from the matching engines, and management of the jobs. Ontology access can be improved by caching ontologies closer to matching engine, and if possible, replicate the ontologies. Job management can be improved by parallelizing the Discovery and Composition Service, for example by introducing a dispatcher that distributes the capability requests in a round robin fashion. The architecture for semantic matching is thus scaleable with an increased load.

6.3 Agreement on Domain Ontologies

As discussed in chapter 2, with the expressiveness of semantic languages there are many possible design approaches for describing information. It is therefore most unlikely that two independent developers would come up with the same design of a domain ontology [42]. We know from previous discussions, in chapters 3-5, that semantic matching needs certain criteria in order to detect similarities between domain ontologies used in annotating requested and advertised capabilities. Two different ontologies will not match, even though they describe the same information, unless they share the same core concept.

For domain ontologies to be widely useful in an open environment like the internet, it is necessary that domain ontologies are agreed upon, e.g. through standardization organizations. This applies specially to all kinds of commonly used services like translation, booking and ordering services. It is, however, possible that this standardization will organize itself, in that the most popular ontologies will be used to annotate requested capabilities and newly advertised capabilities. This is obviously hard to predict, but we assume that some form of standardization process will be necessary for Semantic Web Services to succeed.

One can also start questioning about the usefulness of Semantic Web Services when we are dealing with such a limitation. We would have preferred a more flexible solution, where for example, the agent and the service could negotiate about the capabilities needed for interaction. This is, however, a complete different generation of application where advanced artificial intelligence techniques are required. Due to several advantages, as discussed in chapter 3, we believe that semantic matching of domain ontologies used in annotating service capabilities is the next natural step in the evolution of Web Services technology.

6.4 Mapping of Domain Ontologies

It is unavoidable that different ontologies describing the same capabilities are developed and used. In such cases it might be useful to provide a mapping between the ontologies so that they can be used interchangeable. This is accomplished by providing connections between each of the involving concepts, and the information about these mappings must be part of either one of the ontologies.

Mapping between ontologies must be defined with care so that contradictions are avoided. This can be especially difficult when ontologies are developed with different design approaches, in that there might not be any possible logical mapping between the involving concepts.

Defining mappings manually is error-prone, tedious and clearly not desirable in the long run on a web scale [43]. Several methods for automating this process is emerging where for example machine learning techniques and rules defined by domain experts are used [44].

Even though ontologies should in general not be changed after being published, it might happen. For example if the ontology contains mistakes or errors, and the owner decides to change the ontology instead of publishing a new one. In such cases the mapping might not valid any more.

Providing and maintaining mappings become increasingly even more difficult when the complexity of the ontologies rises. We are for these reasons skeptical to the use of mappings in defining domain ontology for service capabilities. We believe, however, that it would be improved with clear guidelines and design patterns that reflect best practice.

6.5 The Order of Capability Parameters

Services capabilities consist of input and output parameters. The algorithm for semantic matching, developed in chapter 3, maps the requested parameters to the advertised parameters in one-to-one fashion without taking the order into account. This works fine when either inputs or outputs consist of different concepts. But will not be an appropriate solution when, for example, two inputs of the service capabilities point to the same concept. This is because the client would not have a guarantee that the parameters are interpreted in the same order as they are given. Usage of this service is therefore undefined.

WSDL specifies the order parameters with the attribute `parameterOrder` [7]. But this is, to our best knowledge, not reflected as information in OWL-S. The matching algorithm presented in Paolucci *et al.* (2002) does not take the order into account [11]. The motivation for this is not elaborated on, but we assume that the reason is that by enforcing similar order of parameters in requested and advertised capabilities the chance of matching is decreasing. A comprise could be to enforce ordering of the parameters that refer to the same concept.

6.6 Preconditions and Effects

Semantic annotation of input and output parameters describes the information transformation the service undertakes. Due to the expressiveness of semantics (OWL), it reasonable to assume that most information transformations can be described with a clever ontology design.

There are, however, things that services undertake that are not directly related to the information transformation, but has to do with conditions and consequences for using the service.

The consequence (called effects in OWL-S) of using the Blood flow Simulation Service in chapter 5 could for example be an invoice of a certain amount. While a precondition for using the service could have been that the client is authorized by a third party, and thus possesses a certain authentication key.

An interesting approach to describe service capabilities is to combine information transformation with state information. A calculation service can for example be perfectly described with inputs and outputs. By combining information and state transformations a new avenue of design approaches emerges. For example, instead of including information about the multiplication in the inputs and output parameters of the service, the service could take two Integers as input, and then having the effect specified as Multiplied.

Mixing information and state transformations will have consequences for semantic matching – a discussion that is beyond the scope of this work. We can however state that without clear design guidelines, such a mix would complicate semantic matching and composition considerable.

6.7 User Defined Matching, Composition and Selection Criteria

The matching algorithms, developed in chapter 4, compares capabilities with a method based on DL where the similarities between parameters are established as Exact, PlugIn or Fail. The user should, however, have the opportunity to configure the matching preferences, like only allowing Exact matches, or perhaps specifying a different method, for example based on machine learning (techniques for mapping ontologies can in theory also map requested capabilities to advertised capabilities).

The user should also have influences on the composition process. The user could for example specify a more optimized approach by first composing services that match as Exact, and then if no Exact matches are found, compose with PlugIn matches.

The algorithm for automatically selecting a composition could be tuned with several parameters. This depends however on non-functional information that is available about the involving services, and could include: geographical location of the service, economical costs of using the services, previous experience of using the services, quality and popularity.

7. Conclusions

In this chapter we conclude the work with a short summary, followed with answers to the research questions stated in the first chapter, and then a short list of contributions, and finally suggested further work.

7.1 Summary

Web Services technology are XML based protocols that enable platform and language neutral interaction on the internet. The Semantic Web initiative defines standards (e.g. OWL) for how computer interpretable information is expressed and processed. Semantic Web Services is a synthesis of these technologies where the service capabilities are given computer interpretable meaning with semantics. This enables service discovery based on contents, and moreover automated composition and selection.

XML Schema constrains the structure of XML documents, whereas OWL allows the definition of an ontology (vocabulary) that is used in modeling information. The meaning of annotated information is derived from the ontology, which can be equality, subsumption and disjointedness of concepts. OWL provides furthermore consistency checking of ontologies and classification of instances.

Service capabilities consist of input and output parameters that describe the information transformation of the service. Each parameter points to a core concept – defined in an ontology – that is subsequently connected to data values and other concepts through axioms, effectively defining the data structure of the parameter. We coin the notation Semantic Data Structure to describe a group of concepts and relations that are needed to satisfy the constraints of a core concept.

Service discovery is provided by matching requested service capabilities, defined by the client, with advertised service capabilities. Motivated by [11] we have developed and implemented algorithms that find semantic similarities between capabilities by comparing the parameters for equivalence and subsumption.

We have also developed algorithms for creating workflow compositions based on the semantically described service capabilities, and algorithms for autonomously selecting the optimal composition, where the optimization function is based on the length of composition and the type of matches.

Theory and implementation are then validated with a case study, where a domain ontology is defined in context of medical Grid Services, and then used to annotate the capabilities of the two services. Matching, composition and selection of these services are then demonstrated in a scenario.

Through analyzes we point out that it becomes increasingly more difficult to identify core concepts in a domain ontology when the complexity of the ontology rises. The result of this might be that parameters in requested and advertised capabilities are annotated with different concepts in the ontology, even though they are meant to mean the same. And the problem with this is that they will not match semantically. For core concepts to be distinguished on a semantic level we propose that they are defined as subclasses of a certain predefined concept.

We show that the architectures for semantic matching are scalable through parallelization.

Mapping of ontologies is complicated and prone to errors, but unavoidable. In order to ease mapping, and to generally improve standardization, we stress the need for guidelines and design patterns for developing domain ontologies.

7.2 Conclusion

We conclude our work by answering the research questions stated in chapter 1.

How do Semantic Web Services work?

- The capabilities of Semantic Web Services, specified as input and output parameters, are annotated with Semantic Data Structures (concepts and relations) defined in ontologies. The input and output parameters describe the information transformation of the service.
- Ontologies are defined with semantic languages e.g. OWL which is a Semantic Web standard based on Description Logics. OWL is very expressive and can describe a wide variety Semantic Data Structures.
- Semantic service discovery is achieved by matching the capabilities of advertised services with a set of requested capabilities specified by the client. The semantics of the input and output parameters are then compared for equality (Exact match) and subsumption (PlugIn match).
- Composition of services means to create a workflow of services where outputs match with inputs of the next service in the sequence, and where inputs of the first service and outputs of the last service match with requested inputs and requested outputs, respectively.

What architectures, algorithms and software are required?

- The architecture for semantic service discovery is similar to UDDI in that a registry of available services is used to match search requests. The difference is that semantic service discovery compares requested capabilities with advertised service capabilities taking the semantics into account, and for this comparison needs a matching engine. The second difference is that ontologies used in annotating the capabilities are fetched from the internet. Based on architecture described in [22] [12], we have developed and implemented an architecture called Semantic Service Mediator.
- Racer is a matching engine that has proved to be flexible and well documented [26].
- Algorithms for matching capabilities are described in [11]. We have implemented this algorithm (and recommended an improvement), and developed algorithms for automated composition and selection.
- A standard for Semantic Web Services exists and is evolving rapidly (OWL-S) [22].
- Architectures for semantic matching and composition are scalable in that they are easily parallelized.

What are the benefits of extending Web Services with semantics?

- The accuracy of service discovery is improved with semantics because services are not only matched on a syntactical level (as UDDI), but on a semantic level. This becomes essential when the number of services rises, where e.g. thousands of services have the same syntactical interface.
- The client specified a set of requested capabilities that are matched with the capabilities of advertised services. A service that matches these capabilities can be instantly used because the client expects these capabilities. Opposed to UDDI, which

does not take the content into account, a human must manually verify that the service does what is requested before usage.

- Composition of services can be automated because the information transformation is described on a semantic level.

7.3 Contributions

While exploring Semantic Web Services we have made the following contributions:

- Grounded Semantic Web Services technologies into a realistic scenario where:
 - A domain ontology is defined.
 - Service capabilities are annotated with the domain ontology.
 - Demonstrated semantic matching, composition and autonomous selection.
- Identified the need to distinguish core concepts from other concepts in domain ontologies, especially when the ontology contains many concepts and properties. We propose that core concepts should be subclasses of a certain concept. For example in the framework of OWL-S, the core concept of Semantic Data Structures could subclass concept CoreParameter. And furthermore, to enforce that only core concepts are used to represent parameters, the property parameterType could be extended with a range restriction.
- Made it clear that capabilities should support more than one parameter that is annotated with the same core concept. The Semantic Web Service framework, e.g. OWL-S, should therefore contain information about ordering of the parameters like WSDL does with the attribute parameterOrder [7]. For parameters annotated with different concept the order is not important, but when matching capabilities with more than one parameter annotated with the same concept, they should be compared sequentially – in the order they are provided.
- Developed simple algorithms for automated composition.
- Identified the need for automated selection of a composition (among a set of compositions), and developed algorithms for finding the optimal composition based on various criteria.
- Shed light on an emerging field.

Ideas developed that were later found in literature

We have during this work developed two significant ideas that were later found in literature. The first idea was to extend Paolucci *et al*'s work [10] by combining matching, composition and invocation into a service called Semantic Service Mediator. This approach is very similar to what is proposed by among others IBM in Akkiraju *et al.* [12].

The second idea was to extend the message part of parameters in WSDL with an attribute that connected the parameter to a concept defined in an ontology. This was later found to be discussed in an online working document describing OWL-S grounding in relation to WSDL [12].

7.4 Further Work

The WS-Resource Framework makes Web Services stateful by introducing a large set of new capabilities [40]. It would have been interesting to see how semantics can reflect the state of such services. The semantics should represent the state of the service in such a way that agents could reason about the usage. An agent probing a service could for example reason that it is smarter to wait for the service to complete, or a listening agent could observe for certain state changes, and when triggered, inform other agents.

The state of services can also play an important role in dynamic composition and invocation. As an example, consider the composition graph in Figure 40, where A is invoked followed by either B or C. Usually the composition workflow is chosen a priori – before invocation starts. With services that change state, the decision of which service next to invoke can be made just before the invocation starts. As in this example, when service A has completed, the state of B and C could be taken into account in making the most optimal choice. This line of research should be pursued.

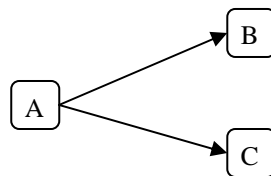


Figure 40: Dynamic choice of service depended on the current state.

The next important step is invocation of composed services. For this the compositions need to be expressed in a composition language that is executable, and the OWL-S composition process structure is the natural choice because the current implementation already uses the library for parsing, and converting the composed graphs into OWL-S compositions is trivial.

Investigation has to be made in how Semantic Data Structures are best communicated. For example whether SOAP-WSDL encoding schemes is flexible enough, or if semantic inference has to be used on this low level. Because of the importance of the Semantic Web and that OWL has become a W3C standard; we can assume that agreements of how parts of an ontologies – Semantic Data Structures – are communicated will be made in the near future.

Modeling of preconditions and effects of Semantic Web Services has to be further researched. Some work has already been done in experimenting [48] and the upcoming version of OWL-S is supposed to include a language for expressing the logic of preconditions and effects [49].

Semantic Web Services should also include non-functional information like cost of usage (variable and fixed), location, legal consequences and quality of service. Research has to be done in how this information is best represented, either as ontologies or simply as XML structures.

With Semantic Web Services a real web of services can be made. Since the input and output parameters can be matched there is possible to create a web of all possible connections between available services (Figure 41). This could be a complete composition graph of all services, which is both semantically searchable and viewable at different abstraction levels. We envision that this pre-created complete composition might be the solution when the number of services increases, because composing every time from scratch is time consuming

and not necessary. This would then be a World Wide Web of Semantic Web Services. The dynamics of such a web should be researched.

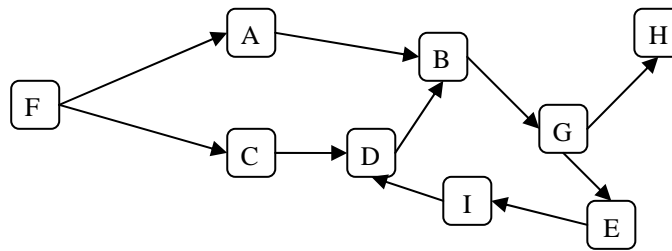


Figure 41: Web of Semantic Web Services, where the services are connected based on compatible capabilities.

Ideally we would like to have autonomous and pro-active agents that can reason and adapt to service capabilities. Taking Semantic Web Services to such a level would involve advanced artificial intelligence techniques. Before pursuing such a research we propose that scenarios reflecting real needs are defined, and then investigated which techniques that can be applied. However, as for the current state, OWL is very expressive and we believe that Semantic Web Services can be improved by clever domain ontologies combined with well-known reasoning techniques.

Appendix A – Glossary of Terms

Advertised Service Capabilities

Semantic description of a service that is available for usage.

Requested Service Capabilities

A description of a set of capabilities that are requested by a client. For Semantic Web Services these capabilities are described with semantics, or more accurately, annotated with concepts defined in ontologies.

Semantic Data Structure

A group of concepts and relations that are needed to satisfy the axioms (constrains) of a concept.

Semantic Web

“The Semantic Web is a mesh of information linked up in such a way as to be easily processable by machines, on a global scale. You can think of it as being an efficient way of representing data on the World Wide Web, or as a globally linked database.” [46]

Semantic Web Services

Web Services with semantically annotated capabilities.

Appendix B – Abbreviations

DAML-S	DARPA Agent Markup Language for Services
DL	Description Logic
DAML+OIL	Darpa Agent Markup Language with Ontology Inference Layer
HTTP	Hypertext Transfer Protocol
OWL	Web Ontology Language
OWL-S	Web Ontology Language for Services
RDF	Resource Description Framework
SDS	Semantic Data Structure
SOAP	Simple Object Access Protocol
UDDI	Universal Description, Discovery and Integration
W3C	World Wide Web Consortium
WSDL	Web Services Definition Language
XML	Extensive Markup Language

Appendix C – Screenshot of the Case Study demo

The following picture is screenshot of the application showing requested capabilities, advertised service capabilities, and five compositions as in the case study.

Automated Matching and Composition of Semantic Web Services

Requested Service Capabilities

ConfigSuper	SimThenVis_Req	VTKPolyData
ArteryLBMGrid		

Advertised Service Capabilities

- | | | |
|---------------|------------------------|------------|
| ConfigLight | FlowSimulation_Service | RawVisData |
| ArteryLBMGrid | | |
- | | | |
|----------------------|-----------------------|-------------|
| RawVisualizationData | Visualization_Service | VTKPolyData |
|----------------------|-----------------------|-------------|
- | | | |
|---------------|-----------------------------|------------|
| ArteryLBMGrid | FlowSimulationSuper_Service | RawVisData |
| ConfigSuper | | |
- | | | |
|------------|-------------|-------------|
| RawVisData | Vis_Service | VTKPolyData |
|------------|-------------|-------------|
- | | | |
|---------------|--------------------|-------------|
| ArteryLBMGrid | SimThenVis_Service | VTKPolyData |
| ConfigLight | | |

Deactive Advertised Services

Compositions Found

<table style="width: 100%; border-collapse: collapse;"> <tr> <td style="padding: 2px;">ConfigLight</td> <td style="padding: 2px;">FlowSimulation_Service</td> <td style="padding: 2px;">RawVisData</td> </tr> <tr> <td style="padding: 2px;">ArteryLBMGrid</td> <td></td> <td></td> </tr> <tr> <td colspan="3" style="padding: 2px;">(plugin match)</td> </tr> </table>	ConfigLight	FlowSimulation_Service	RawVisData	ArteryLBMGrid			(plugin match)			<table style="width: 100%; border-collapse: collapse;"> <tr> <td style="padding: 2px;">RawVisualizationData</td> <td style="padding: 2px;">Visualization_Service</td> <td style="padding: 2px;">VTKPolyData</td> </tr> <tr> <td colspan="3" style="padding: 2px;">(exact match)</td> </tr> </table>	RawVisualizationData	Visualization_Service	VTKPolyData	(exact match)				Cost: 4
ConfigLight	FlowSimulation_Service	RawVisData																
ArteryLBMGrid																		
(plugin match)																		
RawVisualizationData	Visualization_Service	VTKPolyData																
(exact match)																		
<table style="width: 100%; border-collapse: collapse;"> <tr> <td style="padding: 2px;">RawVisData</td> <td style="padding: 2px;">Vis_Service</td> <td style="padding: 2px;">VTKPolyData</td> </tr> <tr> <td colspan="3" style="padding: 2px;">(exact match)</td> </tr> </table>	RawVisData	Vis_Service	VTKPolyData	(exact match)			<table style="width: 100%; border-collapse: collapse;"> <tr> <td style="padding: 2px;">RawVisualizationData</td> <td style="padding: 2px;">Visualization_Service</td> <td style="padding: 2px;">VTKPolyData</td> </tr> <tr> <td colspan="3" style="padding: 2px;">(exact match)</td> </tr> </table>	RawVisualizationData	Visualization_Service	VTKPolyData	(exact match)				Cost: 4			
RawVisData	Vis_Service	VTKPolyData																
(exact match)																		
RawVisualizationData	Visualization_Service	VTKPolyData																
(exact match)																		
<table style="width: 100%; border-collapse: collapse;"> <tr> <td style="padding: 2px;">ArteryLBMGrid</td> <td style="padding: 2px;">FlowSimulationSuper_Service</td> <td style="padding: 2px;">RawVisData</td> </tr> <tr> <td style="padding: 2px;">ConfigSuper</td> <td></td> <td></td> </tr> <tr> <td colspan="3" style="padding: 2px;">(exact match)</td> </tr> </table>	ArteryLBMGrid	FlowSimulationSuper_Service	RawVisData	ConfigSuper			(exact match)			<table style="width: 100%; border-collapse: collapse;"> <tr> <td style="padding: 2px;">RawVisualizationData</td> <td style="padding: 2px;">Visualization_Service</td> <td style="padding: 2px;">VTKPolyData</td> </tr> <tr> <td colspan="3" style="padding: 2px;">(exact match)</td> </tr> </table>	RawVisualizationData	Visualization_Service	VTKPolyData	(exact match)				Cost: 3
ArteryLBMGrid	FlowSimulationSuper_Service	RawVisData																
ConfigSuper																		
(exact match)																		
RawVisualizationData	Visualization_Service	VTKPolyData																
(exact match)																		
<table style="width: 100%; border-collapse: collapse;"> <tr> <td style="padding: 2px;">RawVisData</td> <td style="padding: 2px;">Vis_Service</td> <td style="padding: 2px;">VTKPolyData</td> </tr> <tr> <td colspan="3" style="padding: 2px;">(exact match)</td> </tr> </table>	RawVisData	Vis_Service	VTKPolyData	(exact match)			<table style="width: 100%; border-collapse: collapse;"> <tr> <td style="padding: 2px;">RawVisData</td> <td style="padding: 2px;">Vis_Service</td> <td style="padding: 2px;">VTKPolyData</td> </tr> <tr> <td colspan="3" style="padding: 2px;">(exact match)</td> </tr> </table>	RawVisData	Vis_Service	VTKPolyData	(exact match)				Cost: 3			
RawVisData	Vis_Service	VTKPolyData																
(exact match)																		
RawVisData	Vis_Service	VTKPolyData																
(exact match)																		
<table style="width: 100%; border-collapse: collapse;"> <tr> <td style="padding: 2px;">ArteryLBMGrid</td> <td style="padding: 2px;">SimThenVis_Service</td> <td style="padding: 2px;">VTKPolyData</td> </tr> <tr> <td style="padding: 2px;">ConfigLight</td> <td></td> <td></td> </tr> <tr> <td colspan="3" style="padding: 2px;">(plugin match)</td> </tr> </table>	ArteryLBMGrid	SimThenVis_Service	VTKPolyData	ConfigLight			(plugin match)					Cost: 3						
ArteryLBMGrid	SimThenVis_Service	VTKPolyData																
ConfigLight																		
(plugin match)																		

References

- [1] **Agent Technology: Enabling Next Generation Computing, A Roadmap for Agent Based Computing.** AgentLink, 2003.
- [2] Tannenbaum, A., Steen, M. **Distributed Systems – Principles and Paradigms.** Prentice Hall, 2002.
- [3] **RDF Primer.** W3C Recommendation 10 February 2004
- [4] Dean, M. (ed). **OWL-S: Semantic Markup for Web Services.** Version 1.0, 2004.
- [5] **Extensible Markup Language (XML) 1.0 (Third Edition).** W3C Recommendation 04 February 2004.
- [6] **XML Schema Part 1: Structures.** W3C Recommendation 2 May 2001.
- [7] **Web Services Description Language (WSDL).** W3C Note 15 March 2001.
- [8] **SOAP Version 1.2 Part 0: Primer.** W3C Recommendation 24 June 2003.
- [9] **UDDI Technical White Paper.** September 6, 2000.
- [10] P.M.A. Sloot; A. Tirado-Ramos; A.G. Hoekstra and M. Bubak. **An Interactive Grid Environment for Non-Invasive Vascular Reconstruction.** 2nd International Workshop on Biomedical Computations on the Grid (BioGrid'04), in conjunction with Fourth IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid2004)
- [11] Massimo Paolucci, Takahiro Kawamura, Terry R. Payne, and Katia Sycara. **Semantic Matching of Web Services Capabilities.** The First International Semantic Web Conference (ISWC), Sardinia (Italy), June, 2002.
- [12] Rama Akkiraju, Richard Goodwin, Prashant Doshi, Sascha Roeder. **A Method for Semantically Enhancing the Service Discovery Capabilities of UDDI.** In the Proceedings of IJCAI Information Integration on the Web Workshop, Acapulco, Mexico, August 2003.
- [13] **Describing Web Services using OWL-S and WSDL.** DAML-S Coalition working document, October 2003.
- [14] Katia Sycara, Massimo Paolucci, Anupriya Ankolekar and Naveen Srinivasan. **Automated Discovery, Interaction and Composition of Semantic Web services.** Journal of Web Semantics, Volume 1, Issue 1, September 2003, pp. 27-46
- [15] Paul Horn. **Autonomic Computing: IBM's Perspective on the State of Information Technology.** IBM Corporation, October 15, 2001.
- [16] Pokraev, S., Koolwaaij, J. and M. Wibbels. **Extending UDDI with context-aware features based on semantic service descriptions.** ICWS'03: Proceedings of the International Conference on Web Services
- [17] A.G. Hoekstra. **Syllabus Architectuur en Parallel Rekenen Part 1: Introduction to Parallel Computing.** University of Amsterdam, Section for Computational Science, 2002.
- [18] M.W.A, Strating, P., Lankhorst, M.M., Doest, H. ter & Iacob, M.-E. **Service-Oriented Enterprise Architecture.** To appear in: Zoran Stojanovic & Ajantha Dahanayake (Eds.), Service-Oriented Software System Engineering: Challenges and Practices. IDEA Group, 2004
- [19] **A B2B Solution using WebSphere Business Integration V4.1 and WebSphere Business Connection V1.1.** Red Book, IBM, ISBN 0738453358
- [20] Evren Sirin, James Hendler, Bijan Parsia. **Semi-automatic Composition of Web Services using Semantic Descriptions.** Accepted to "Web Services: Modeling, Architecture and Infrastructure" workshop in conjunction with ICEIS2003, 2002.

-
- [21] Massimo Paolucci and Katia Sycara. **Autonomous Semantic Web Services**. The Zen of the Web. September-October 2003, Published by the IEEE Computer Society.
- [22] Massimo Paolucci, Katia Sycara, Takuya Nishimura, and Naveen Srinivasan. **Toward a Semantic Web e-commerce**. To appear in Proceedings of BIS2003.
- [23] Li, Lei and Horrocks, Ian. **A Software Framework for Matchmaking Based on Semantic Web Technology**. In Proceedings International WWW Conference, Budapest, Hungary. (2003)
- [24] F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. Patel-Schneider, editors. **The Description Logic Handbook**. Cambridge University Press, 2002.
- [25] **Pellet OWL Reasoner**. Mindswap, University of Maryland.
- [26] Volker Haarsley and Ralf Moller. **RACER User's Guide and Reference Manual** Version 1.7.7.
- [27] Evren Sirin. **OWL-S API 1.0**. Mindswap, University of Maryland.
- [28] **IBM Ontology Management System (SNOBASE)**. AlphaWorks, October 2003.
- [29] Grigoris Antoniou and Frank van Harmelen. **A Semantic Web Primer**. The MIT Press, 2004.
- [30] **OWL Web Ontology Language Overview**. W3C Recommendation 10 February 2004.
- [31] Gregor Hohpe. **Web Services: Pathway to a Service-Oriented Architecture?** ThoughtWorks, Inc., 2002.
- [32] Daconta, Obrst, Smith. **The Semantic Web: A Guide to the Future of XML, Web Services, and Knowledge Management**. Wiley Publishing, Inc. 2003.
- [33] Stefan Tang. **Matching of Web Service Specifications using DAML-S descriptions**. Thesis, Institut für Telekommunikationssysteme, Technische Universität Berlin, March, 2004.
- [34] Massimo Paolucci, Katia Sycara, and Takahiro Kawamura. **Delivering Semantic Web Services**. In Proceedings of the Twelve's World Wide Web Conference (WWW2003), Budapest, Hungary, May 2003, pp 111- 118
- [35] Dean, M. (ed.) **OWL-S: Semantic Markup for Web Services**. Version 1.1 Beta, 2004.
- [36] S. Narayanan and S. A. McIlraith. **Simulation, Verification and Automated Composition of Web Services**. Presented at Eleventh International World Wide Web Conference (WWW-11), Honolulu, Hawaii, 2002.
- [37] Richards, D. Splunter, S. van Brazier, F.M.T. Sabou, M. **Composing Web Services using an Agent Factory**. In Proceedings of AAMAS Workshop on Web Services and Agent-Based Engineering(WSABE), Melbourne, Australia , July , 2003
- [38] Jihie Kim and Yolanda Gil. **Towards Interactive Composition of Semantic Web Services**. 2004 AAI Spring Symposium Series, March, 2004
- [39] Tuecke, S., Czajkowski, K., Foster, I., Frey, J., Graham, S., Kesselman, C, Maguire T., Sandholm, T., Snelling, D., and Vanderbilt, P. **Open Grid Services Infrastructure (OGSI) Version 1.0**.
- [40] I. Foster (ed.). **Modeling Stateful Resources with Web Services v. 1.1**. March 5, 2004.
- [41] H. Afsarmanesh, R.G. Belleman, A.S.Z. Belloum, A. Benabdelkader, J.F.J. van den Brand, G.B. Eijkel, A. Frenkel, C. Garita, D.L. Groep, R.M.A. Heeren, Z.W. Hendrikse, L.O. Hertzberger, J.A. Kaandorp, E.C. Kaletas A1, V. Korkhov, C.T.A.M. de Laat, P.M.A. Sloot, D. Vasunin, A. Visser and H.H. Yakali. **VL-E: A Grid-based virtual laboratory**. Scientific Programming Journal Vol. 10, No. 2, page 173-181, 2002

-
- [42] Natalya F. Noy and Deborah L. McGuinness. **Ontology Development 101: A Guide to Creating Your First Ontology**. Stanford Knowledge Systems Laboratory Technical Report KSL-01-05 and Stanford Medical Informatics Technical Report SMI-2001-0880, March 2001.
- [43] AnHai Doan, Jayant Madhavan, Pedro Domingos, and Alon Halevy. **Learning to Match Ontologies on the Semantic Web**. WWW2002, May, 2002, Honolulu, Hawaii, USA.
- [44] Marc Ehrig and York Sure. **Ontology Mapping - An Integrated Approach**. Institute AIFB, University of Karlsruhe, April 21, 2004.
- [45] Tim Berners-Lee. **Weaving the Web**. Harper San Francisco, 1999.
- [46] Sean B. Palmer. **The Semantic Web: An Introduction**. 2001-09
- [47] Satish Thatte (ed.). **Business Process Execution Language for Web Services (BPEL4WS)** Version 1.1, 05 May 2003.
- [48] Mikko Laukkanen and Heikki Helin. **Composing Workflows of Semantic Web Services**. In AAMAS Workshop on Web Services and Agent-Based Engineering, 2003.
- [49] K.A. Iskra; R.G. Belleman; G.D. van Albada; J.Santosa; P.M.A. Sloot; H.E. Bal; H.J.W. Spoelder and M.Bubak. **The polder Computing Environment, a system for interactive distributed simulation, Concurrency and Computation**. Special Issue on Grid Computing Environments Vol. 14, pp.1313-1335. John Wiley and Sons, 2002.
- [50] A.M.M. Aroli; A.g. Hoekstra and P.M.A. Sloot. **Simulation of a Systolic Cycle in a Realistic Artery with the Lattice Boltzman BGK Method**. International Journal of Modern Physics B, vol. 17, nr 1 & 2pp.95-98. World Scientific Publishing Company, January 2003.
- [51] A. Tirado-Ramos; J.M. Ragas; D.P. Shamonin; H.Rosmanith and D.KranzlmueLLer. **Integration of Blood Flow Visualization on the Grid: the FlowFish/GVK Approach**. In Proceedings of the 2nd European Accross Grids Conference, ((Accepted) Nicosia, Cypres, January 2004.
- [52] The OWL Services Coalition. **OWL-S: Semantic Markup for Web Services**. Version 1.1 Beta, 2004.