Scalable Semantic Web Service Discovery for Goal-driven Service-Oriented Architectures

Dissertation

by
Michael Stollberg

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

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

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

Innsbruck, March 17, 2008

LEOPOLD-FRANZENS UNIVERSITÄT INNSBRUCK

Date: **17 March 2008**

Author:	**Michael Stollberg**
Title:	**Scalable Semantic Web Service Discovery for Goal-driven Service-Oriented Architectures**
Department:	**Semantic Technology Institute, Faculty of Mathematics, Computer Science, and Physics**
Degree:	**Dr. rer. nat.**

Contents

List of Figures

List of Tables

Abstract

The concept of Service-Oriented Architectures (SOA) is the latest design paradigm for IT systems. The idea is to use Web services as the basic blocks, which provide programmatic access to computational facilities over the Internet. The aim is to exploit the potential of the World Wide Web as an infrastructure for computation, help to reduce the development and maintenance costs of IT systems, and also to tackle the integration problem within and in between collaborating organizations.

The realization of sophisticated SOA technologies is a massive challenge. The initial Web service technology stack around WSDL, SOAP, and UDDI facilitates the technical provision and usage of Web services. However, it limits the detection and usability analysis of suitable Web services for a particular client request to manual inspection. To overcome these deficiencies, the emerging concept of Semantic Web services (SWS) develops inference-based techniques for the automated discovery, composition, and execution of Web services on the basis of exhaustive semantic annotations. This also addresses the problem of semantic interoperability by using ontologies as the underlying data model and by applying reasoning techniques developed in the context of Semantic Web.

One of the central operations in SOA environments is the detection of suitable Web services for solving a given request, commonly referred to as Web service discovery. This is usually performed as the first processing step that finds potential candidates out of the available Web services. Most techniques – in particular in the area of SWS – primarily consider functional aspects for the discovery task. The usability of the discovered Web services is then further inspected in subsequent processing steps that consider non-functional aspects such as the quality-of-service or behavioral compatibility. Next to the automation of the discovery tasks, recent approaches for advanced SWS technologies further envision to integrate automated Web service discovery engines as a heavily used software component. Examples for this are approaches for dynamic Web service composition or for semantically enabled business process management wherein the actual Web services shall be detected dynamically at runtime in order to achieve higher flexibility and better maintainability.

From this application purpose, two central requirements arise for automated Web service discovery engines: (1) a high retrieval accuracy in order to perform the discovery task with an appropriate quality, and (2) a high computational performance in order to serve as a operationally reliable software component, in particular within larger search spaces of available Web services that can be expected in real-world applications. The former can most adequately be achieved by semantic matchmaking techniques that work on sufficiently rich descriptions, which can achieve a better accuracy than other techniques. For the latter, it is necessary to reduce the time and the computational costs for the discovery task.

The present work addresses this challenge by developing a goal-based, semantically enabled Web service discovery technique along with a caching mechanism for enhancing the computational performance. In consequence, the thesis consists of three consecutive parts and aims at advancing the state-of-the-art in respective SWS technology developments:

1. a refined goal model for Semantic Web services for facilitating problem-oriented Web service usage in SOA systems: the client merely specifies the objective to be achieved as a goal that abstracts from technical details, and the system automatically discovers, composes, and executes suitable Web services for solving this;

2. a formally defined approach for semantically enabled Web service discovery that separates design time and runtime operations and warrants high retrieval performance by matchmaking of ontology-based functional descriptions of goals and Web services;

3. a caching mechanism for Web service discovery that captures the relevant knowledge of design time discovery runs and effectively exploits this in order to enhance the computational performance of Web service discovery at runtime.

The goal model defines goals as formal descriptions of the objectives that clients want to achieve by using Web services, and specifies how these are used and processed within SWS environments. A central aspect is the distinction of *goal templates* as generic and reusable objective descriptions that are stored in the system, and *goal instances* that describe concrete client requests and are defined by instantiating a goal template with concrete input values. This allows us to separate design time and runtime operations for the discovery task. The suitable Web services for goal templates are discovered at design time. The result is captured in a specialized knowledge structure which serves as the heart of the caching technique. At runtime, a client – either a human or a machine – formulates the concrete objective to be achieved in terms of a goal instance. As the time critical and expectably most frequent operation in real-world SOA applications, the discovery of suitable Web services for goal instances at runtime is optimized by exploiting the captured knowledge.

The Web service discovery techniques developed in this work perform semantic matchmaking of sufficiently rich functional descriptions. For this, we define functional descriptions that precisely describe the start- and end-states of the possible executions of Web services, respectively solutions of goals in terms of preconditions and effects. These are specified on the basis of ontologies, and we define their formal semantics in a first-order logic framework. Upon this, we specify the necessary semantic matchmaking techniques for Web service discovery at both design time and runtime. We shall show that these achieve a higher precision and recall than most existing techniques for semantically enabled Web service discovery.

The caching mechanism is based on a directed acyclic graph that organizes the goal templates in a subsumption hierarchy with respect to the requested functionalities and captures the relevant knowledge on the usability of the available Web services from the design time discovery results. This provides a formally defined index for the efficient search of goals and Web services, and we specify the necessary algorithms for automatically generating and properly maintaining this graph whenever a goal template or a Web service is added, removed, or modified. The optimized discovery algorithms exploit this to enhance the computational performance by minimizing the relevant search space and the number of necessary matchmaking operations. This is a novel approach in the field of Semantic Web services that can achieve a better performance increase than existing optimization techniques and maintain a high retrieval accuracy for automated Web service discovery.

To evaluate the achievable performance increase, we compare our caching-enabled Web service discovery with other engines that do not apply any or merely less elaborated optimization techniques. In the shipment scenario that has been defined in the Semantic Web Services Challenge – a widely recognized initiative for the demonstration and comparison of SWS techniques – our prototype implementation shows significant improvements in the efficiency, scalability, and stability for performing the discovery task. To also assess the practical relevance of the developed technology, we examine its applicability in one of the largest existing SOA systems maintained by the US-based telecommunication provider *Verizon* as well as in prominent application scenarios for Semantic Web services. This reveals that the goal-based approach and semantically enabled discovery techniques can greatly increase the quality of SOA technology, and that optimized techniques appear to be necessary in order to warrant the operational reliability of automated discovery engines in real-world applications wherein larger numbers of available Web services can be expected.

The specifications throughout this work are by purpose kept on a generic level. We only formally define the central aspects, using classical first-order logic (FOL) as the specification language. The aim is to support the adaption of the developed techniques to several SWS frameworks as well as to other, not semantically enabled SOA technologies.

Acknowledgements

This work presents the research results achieved during my employment at the Digital Enterprise Research Institute (DERI) – now called the Semantic Technology Institute (STI) – that, under the lead of Univ.-Prof. Dr. Dieter Fensel, has been involved in the edge of research around the Semantic Web and Semantic Web services. Despite the very dynamic growth and changes of the institute, this has given me the chance to collaborate with several internationally renowned researchers in the field.

I would like to thank my advisor Univ.-Prof. Dr. Dieter Fensel for the supervision of this work, and also Univ.-Prof. Dr. Martin Hepp for fruitful discussion and support. I dedicate special thanks to my fellow PhD students for the continuous collaboration, in particular to Uwe Keller, Holger Lausen, and Rubén Lara. I also want to thank Dr. Jörg Hoffmann and Dr. Stijn Heymans for fruitful feedback. I was able to gain valuable insights from the collaboration and discussions with academic and industrial partners in international research projects, in particular within the EU FP 6 Integrated Project DIP (2004 – 2006). The discussions with Prof. Michael Genesereth during a four-month research exchange at the Stanford University in California, USA, helped to define the scientific scope and focus of the work. I also thank Dr. Michael Brodie, chief scientist at Verizon, for interesting discussions and the provision on non-confidential information about the Verizon SOA system. Moreover, the feedback received from several people at numerous conferences, workshops, and other events has helped to continuously improve the presented work.

Last but not least, I would like to thank my family for the thorough and all-embracing support that has allowed me to complete this thesis.

Chapter 1

Introduction

The aim of the present work is to develop a scalable, semantically enabled Web service discovery technique in order to meet the requirements that arise for the deployment of automated discovery engines in Service-Oriented Architectures.

To introduce into the work, this chapter explains the motivation and aim of the thesis, identifies the relevant research questions, and defines the scientific methodology for addressing these. Finally, we outline the structure of the document.

1.1 Motivation and Objectives

The idea of Service-Oriented Architectures (SOA) is to use Web services as the basic building blocks for IT systems [Erl, 2005]. A Web service supports the invocation and consumption of a computational facility via a standardized interface over the Internet, facilitating the usage of the World Wide Web as an infrastructure for computation [Alonso et al., 2004]. To overcome the deficiencies of the initial technology stack around WSDL, SOAP, and UDDI that limits the detection of suitable Web services to manual analysis, research on Semantic Web services (SWS) develops inference-based techniques for the automated discovery, composition, and execution of Web services [McIlraith et al., 2001; Fensel et al., 2006].

One of the central operations in SOA is the detection of suitable Web services for a given task, which commonly is referred to as Web service discovery [Sycara et al., 2003; Preist, 2004]. The aim of SWS research is to provide sophisticated techniques for automating the discovery task, and further to employ this as a heavily used component for dynamically detecting the actual Web services at runtime [Hepp et al., 2005; Bertoli et al., 2007]. With respect to this, two central requirements arise for automated discovery engines: they should

expose (1) a high retrieval accuracy in order to perform the discovery task with an appropriate quality, and (2) a high computational performance in order to serve as a operationally reliable software component in SWS environments.

The first requirement is concerned with the functional quality of automated Web service discovery. In order to ensure that the discovered Web service can actually solve the given client request, the discovery engine should ensure that every discovered Web service is usable (precision) and every usable Web service can be discovered (recall). This can most suitably be achieved by semantic matchmaking on sufficiently rich descriptions of Web services and requests for these, which can in general achieve a higher retrieval accuracy than other techniques. Although there is a wealth of work on semantically enabled Web service discovery (e.g. [Paolucci et al., 2002; Li and Horrocks, 2003; Noia et al., 2003; Benatallah et al., 2005; Keller et al., 2006a]), most existing approaches lack in the achievable retrieval accuracy due to deficiencies in the used functional descriptions as well as a proper support for the client side in SOA applications.

The second requirement addresses the computational performance of automated discovery engines, which becomes in particular relevant for larger search spaces of available Web services that can be expected in real-world applications. In SWS environments, discovery is usually performed as the first processing step which needs to consider all potential candidates. In order to perform this in an adequate time and also to warrant the scalability of the whole system, it is necessary to reduce the average processing time and as well as the computational costs for performing the discovery task. This can be achieved by reducing the relevant search space and minimizing the number of necessary matchmaking operations. While having received far less attention in SWS research than semantic matchmaking techniques, existing approaches address this challenge by clustering Web services in tree-like structures in order to reduce the set of potential candidates by pre-filtering (e.g. [Constantinescu et al., 2005; Verma et al., 2005; Tausch et al., 2006; Abramowicz et al., 2007]). However, these techniques commonly lack in the achievable performance increase as well as in the accuracy of the pre-filtering results.

This thesis aims at contributing to the development of sophisticated Web service discovery technologies with respect to the identified requirements, and at providing scientific progress beyond the state-of-the-art in this field. In contrast to most other works, we take a goal-based approach for Semantic Web services. Therein, clients formulate the objective to achieved in terms of a goal that abstracts from technical details, and the system automatically discovers, composes, and executes suitable Web services for solving this. This allows us to enable problem-oriented Web service usage in SOA systems, and also to overcome the deficiencies in the flexibility that result from hard-wired Web service invocations. We

define a model for describing and processing goals in SWS environments, and develop a two-phased discovery framework for this: the suitable Web services for goal templates as generic and reusable objective descriptions are discovered at design time, and the actual Web services for goal instances that describe concrete client requests are determined at runtime. Web service discovery is performed by semantic matchmaking on the basis of sufficiently rich functional descriptions in order to warrant a high retrieval accuracy. To enhance the computational performance as the time critical operation, a novel technique is developed which adopts the concept of caching to Web service discovery. This captures the relevant knowledge from design time discovery runs, and effectively utilizes this for reducing the search space and minimizing the reasoning effort for runtime Web service discovery. The underlying knowledge structure can be created and maintained automatically, and a better performance increase than existing techniques can be achieved because – in certain cases – the suitable Web services can be discovered without invoking a matchmaker.

To summarize, the overall aim of this work is to elaborate an approach for automated, semantically enabled Web service discovery that satisfies the requirements on a high retrieval accuracy and a high computational performance. For this, we take a goal-based approach and develop a two-phase discovery technique which applies semantic matchmaking on rich functional descriptions and is extended with a caching mechanism for enhancing the computational performance. The aim of the thesis is to design the technique, formally specify the central aspects, and present a prototypical implementation which can be adapted into specific SWS frameworks as well as to other, non-semantically enabled SOA environments. In consequence, the work is centered around the following research questions:

1. How to properly define, model, and utilize goals as formalized client objectives in order to facilitate problem-oriented, dynamic, and automated Web service usage within SWS environments?

2. What is the appropriate distinction between design- and runtime operations in a two-phased discovery framework, and how to formally specify functional descriptions for goals and Web services as well as the necessary semantic matchmaking techniques in order to achieve high precision and recall for Web service discovery?

3. How to define a caching mechanism that correctly captures the relevant knowledge of Web service discovery operations and effectively utilizes this for enhancing the computational performance of subsequent runtime discovery tasks? What is the achievable performance increase in comparison to other optimization techniques, and is the technique beneficially applicable to real-world scenarios?

1.2 Methodology

The aim of this thesis is to address the identified research questions in an appropriate manner, to provide a suitable solution for each one, and to properly position these within related work. In consequence, the work consists of three main parts.

The first part develops a conceptual model for describing and processing goals as formalized client objectives in SWS environments. The aim is to provide a sophisticated basis for realizing goal-driven SOA technologies, and we refine and extend existing approaches for this. The second part is a semantically enabled Web service discovery technique that separates two phases: at design time, Web services for goal templates are discovered, i.e. for generic objective descriptions that are stored in the system. At runtime, a client formulates the concrete objective to be achieved as a goal instance by instantiating a goal template with concrete input values, for which the actually suitable Web services are discovered. In order to ensure a high retrieval accuracy, the discovery techniques developed in this thesis apply semantic matchmaking techniques that work on sufficiently rich functional descriptions of goals and Web services with precise formal semantics.

On this basis, the third part develops a novel technique for optimizing the computational performance of Web service discovery. It automatically creates a graph structure that organizes goal templates in a subsumption hierarchy and captures the minimal knowledge on the usability of the available Web services. This provides an index structure for efficient search of goals and Web services, which is exploited for reducing the search space and minimizing the number of necessary matchmaking operations for runtime discovery. The major gain of this technique is that in certain cases suitable Web services can be discovered without invoking a matchmaker while maintaining a high retrieval accuracy, which appears to be superior to existing optimization techniques. The achievable performance increase is evaluated by a quantitative comparison with less optimized engines, and we verify the applicability of the overall approach in existing, real-world SOA applications.

The following scientific techniques are applied in this work in order to properly elaborate the overall approach and the technical solutions. We use *conceptual analysis* and *examine literature* for the design and scientific positioning of the overall approach. The central aspects of the technical solutions are provided in terms of *formal definitions* along with proofs, and we provide *prototypical implementations* in order to demonstrate the realizability. For the *empirical evaluation*, we apply *statistical analysis* and a *qualitative field study*, and we discuss several *use case scenarios* for illustration and demonstration throughout the thesis. In particular, we use the shipment scenario defined in the Semantic Web Services Challenge – a widely recognized initiative for demonstrating and comparing SWS technologies – as

the use case for the quantitative evaluation, and we show the practical relevance of the developed technology by examining its applicability within one of the largest existing SOA systems that is maintained by the US-based telecommunication provider *Verizon*.

While the individual parts of the thesis are self-contained, the technical solutions developed in this work successively build on top of each other: the two-phase Web service discovery is based upon the distinction of goal templates and goal instances which is defined in the goal model, and the caching mechanism applies the semantic matchmaking techniques. The following outlines the central line of argumentation in order to provide a concise overview of the overall approach that underlies this work. We formulate this in terms of hypotheses that will be addressed and verified in the course of the thesis.

1. The search space, i.e. the number of available Web services in real-world SOA applications will be huge. For instance, the *Verizon* system encompasses about 1.500 Web services that are used by more than 600 client applications. It is commonly expected that the amount of available Web services will grow significantly in the future.

2. The bottleneck for the scalability of SOA environments is Web service discovery, i.e. the detection of suitable Web services out of the available ones. As the first operation for solving a given task, this requires a $1 : n$ search on the complete search space.

3. The automation of Web service discovery can greatly enhance the quality of SOA technology. Techniques based on semantic matchmaking developed in the field of Semantic Web services can achieve a high retrieval accuracy. To increase the computational performance, existing approaches aim at reducing the search space by clustering Web services. This has several deficiencies, in particular with respect to the accuracy of pre-filtering results as well as the achievable reduction of the time and reasoning effort for the discovery task.

4. Goals as formalized descriptions of client objectives provide an appropriate means for lifting the client interaction with a SOA system to the knowledge level. It appears to be expedient to distinguish *goal templates* as generic objective descriptions that are stored in the system and *goal instances* that denote concrete client requests. This allows us to develop efficient Web service discovery techniques that separate design- and runtime operations, and also to ease the goal formulation by clients.

5. In typical SOA applications, there is a significant degree of similarity among client requests when these are described in terms of goals. This seems to be a promising starting point for enhancing the computational performance of automated Web service discovery techniques.

6. It thus is possible to develop a novel optimization technique that adopts the concept of caching to Web service discovery. The idea is to organize goal templates in a subsumption hierarchy with respect to the requested functionalities and capture the relevant knowledge from design time discovery runs. The underlying graph structure can be created automatically, and runtime discovery can be optimized by reducing the search space and minimize the number of necessary matchmaking operations.

7. This technique is able to significantly reduce the computational costs of Web service discovery while exposing a high retrieval accuracy, and thus can overcome the bottleneck for the scalability of SOA environments.

8. The approach is superior to all known approaches for performance optimization of Web service discovery under consideration of the structure and requirements in typical SOA applications. In particular, it enables efficient runtime discovery on the level of goal instances as the expectably most frequent operation in real-world scenarios.

1.3 Outline

After having explained the objectives and the methodology of the work, the following outlines the structure of the thesis.

Chapter 2 introduces into the context of the thesis and identifies the research problem addressed in this work. For this, we review the concept of Service-Oriented Architectures and the existing Web service technologies as well as the idea and the state-of-the-art in Semantic Web services. We then discuss the need and requirements for scalable automated Web service discovery techniques with a high retrieval accuracy in detail, examine existing solutions, and depict their deficiencies. On the basis of this, we motivate and substantiate the approach undertaken in this work.

Chapter 3 presents the goal model for Semantic Web services as the first element of the developed technology. This is a refinement and extension of previous works towards goal-based SWS technologies. We discuss the intended usage of goals in SOA applications, identify the requirements for goal descriptions by examining existing works, and present the conceptual structure and definitions of the refined goal model that integrates several ideas. We finally discuss the suitability of the model and position it within related work.

Chapter 4 presents the semantically enabled Web service discovery technique that builds upon the goal model. This distinguishes design time discovery that is concerned with finding suitable Web services for goal templates as generic objective descriptions that are stored

in the system, and runtime discovery that finds the actual Web services for goal instances which denote specific client requests. We define functional descriptions with precise formal semantics for describing the possible executions of Web services and the possible solutions for goals with respect to the start- and end-states. Upon this, we define semantic matchmaking techniques for precisely determining the usability of a Web service goal templates as well as for goal instances under functional aspects. We present the prototype implementation that uses classical first-order logic (FOL) as the specification language and an automated theorem prover for matchmaking, and we position the approach within related work.

Chapter 5 specifies the caching mechanism for enhancing the computational performance of Web service discovery. As the third element of the developed technology, it works upon the goal model and applies the semantic matchmaking techniques from above. Essentially, it automatically creates a graph structure that organizes goal templates in a subsumption hierarchy and captures relevant knowledge on the usability of available Web services from the results of design time discovery. We explain the design of the technique, formally define the graph structure and specify the algorithms for its automated creation and maintenance, and define the optimized algorithms for Web service discovery that effectively exploit the captured knowledge. We present the prototype implementation, discuss the expectably improvements for optimizing the Web service discovery task and identify complementary technologies, and finally position it within related work.

Chapter 6 presents the evaluation of the work that consists of a quantitative and a qualitative part. The first part is realized as a comparison test between the optimized Web service discovery and other, not or less optimized engines. We use the original scenario and data set from the SWS challenge, and the statistical preparation of the results reveals that the developed technology satisfies the requirements on the high retrieval accuracy and computational performance of automated discovery engines. The second part addresses the relevance of the overall approach for real-world applications. Examining its applicability for the *Verizon* SOA system reveals that (1) the goal-based approach goals can significantly improve the flexibility of the system, (2) that Web service discovery techniques with a high retrieval accuracy are highly desirable, and (3) that the optimization technique can reveal its potential because of the high similarity of the usage requests for Web services in the client applications. Similar observations can be made in other SOA application areas wherein larger numbers of Web services are expected.

Chapter 7 finally summarizes the thesis, discusses the contributions, and outlines possibilities for future research and developments. The appendices provide additional information such as formal proofs and technical specifications, and the accompanying CD-R contains the prototype implementations and the original data of the quantitative evaluation.

Chapter 2

Web Services, SOA, and Semantics

This chapter introduces the research context and identifies the problem addressed in this thesis. For this, we explain the relevant concepts and technologies, discuss the need for scalable Web service discovery techniques with a high retrieval accuracy, and depict the deficiencies of existing technical solutions in order to motivate the relevance of this work.

As the broader research context, Section 2.1 reviews the idea of Web services along with the existing technology stack as well as the emerging concept of Service-Oriented Architectures (SOA). We then explain the idea of Semantic Web services as a prominent approach for realizing more sophisticated SOA technologies, and examine the start-of-the-art in research and development. In Section 2.2, we identify and motivate the research problem addressed in this work. For this, we explain the motivation and expected benefits of the goal-driven approach for SOA technologies, discuss the importance of Web service discovery and the arising requirements for automation, and depict the deficiencies of existing technical solutions. On this basis, we substantiate the approach undertaken in this work, and finally Section 2.3 summarizes the chapter and outlines the remainder of the thesis.

2.1 Research Context

The growth and expansion of the World Wide Web (WWW) in the 1990s triggered several technology developments in order to exploit its potential as a world-wide infrastructure for information exchange and communication. One of these new technologies are Web services that enable the invocation of programs, or, more generally, of computational facilities over the Web [Alonso et al., 2004]. A Web service is accessible via an interface that specifies the messages and further technical information for invocation and consumption. The major

vantage is that the WWW can be used as an infrastructure for intra- and inter-organizational computing while the technology is independent of the platform, language, and protocols used for the technical implementation of the Web services.

This has led to the idea of Service-Oriented Architectures (SOA) as a new paradigm for IT system design that currently receives a lot of attention in industry [Erl, 2005]. The idea is that software systems use Web services as the basic building blocks instead of proprietary solutions. This shall enable a better reuse of existing computational facilities in order to reduce the effort and costs for the development and maintenance of software systems. To provide a generic, domain independent technology that can replace specialized software systems, the overall aim of SOA is to provide technologies for dynamically finding, combining, and executing those Web services that are needed for solving a given request.

The initial Web service technology stack around WSDL, SOAP, and UDDI supports the creation, publication, and consumption of Web services. However, the service descriptions remain on a syntactic level with a strong technical focus, which limits the identification of suitable Web services for a given task to manual inspection. To overcome this, the emerging concept of Semantic Web services (SWS) develops inference-based techniques for automated discovery, composition, and execution of Web services on the basis of rich, semantic descriptions [McIlraith et al., 2001; Fensel et al., 2006]. Most SWS technologies use ontologies as the underlying data model, therewith aiming at treating the interoperability problem on a semantic level as well as at the integration of Web services with the Semantic Web, another prominent technology proposal for the next generation of the WWW.

The realization of suitable SOA technologies, and in particular of SWS-based solutions, is a massive challenge. As the research context of this work, the following explains the central concepts and technologies and reviews the state-of-the-art in research and development.

2.1.1 Web Services

The concept of Web services has been invented in the late 1990s by a mostly industry-driven initiative. The aim was to define a new technology that makes use of the WWW as an infrastructure for computation, and also provides a new means for effectively tackling the intra- and inter-organizational integration of information and services. For this, three contiguous technologies have been specified which are commonly referred to as the basic *Web service technology stack* that has been published by standardization bodies (W3C, respectively OASIS): WSDL as the language for describing the interface of a Web service, SOAP as a messaging protocol for exchanging XML data over the Web, and UDDI as a registry technology for Web services.

The following explains the basics of these technologies. While this appears to be sufficient for our purposes, we refer to the technical specifications as well as to extensive secondary literature for details, e.g. [Alonso et al., 2004; Erl, 2005; Marks and Bell, 2006].[1]

Web Service Description Language (WSDL). This is an XML-based language for describing the interface of a Web service which denotes the heart of Web service technology. Essentially, a WSDL description specifies the supported operations for invoking and consuming the Web service, its physical location, and it supports bindings to several transport protocols and formats for the actual information exchange between the Web service and the requester. The main merit is that WSDL is independent of the platform, language, and protocol used for the implementation of the Web service, and that it supports message exchange over the WWW via SOAP (see below).

The WSDL description of a Web service is an XML document that consists of the following elements as illustrated in Figure 2.1. The *service* element describes the name and the physical location of the Web service, mostly in form of a URI. A Web service can have several physical endpoints. These are called *ports*, for which a *binding* defines the supported transport protocols and formats. While this specifies how to carry out the actual information exchange, the *port type* element specifies the set of operations that are supported by the Web service. An *operation* consists of a set of messages and their direction (i.e. in- or out-going). A *message* describes the data being communicated between the requester and the provider. The message content is described in terms of XML Schemas, for which the used data types are specified in the *type* element of the WSDL description.

[1] The initial specifications have been published in 2000 - 2002; in the meantime, updated versions with minor changes and additions have mostly been recommended as standards:

- Web Services Description Language (WSDL) 1.1, W3C Note, 15 March 2001, online: `http://www.w3.org/TR/wsdl`
- Web Services Description Language (WSDL) Version 2.0, W3C Proposed Recommendation, 23 May 2007, online: `http://www.w3.org/TR/wsdl20-primer`
- Simple Object Access Protocol (SOAP) 1.1, W3C Note, 08 May 2000, online: `http://www.w3.org/TR/2000/NOTE-SOAP-20000508/`
- SOAP Version 1.2, W3C Recommendation (Second Edition) 27 April 2007, online `http://www.w3.org/TR/soap/`
- UDDI Data Structure Reference V1.0, UDDI Published Specification, 28 June 2002, online: `http://uddi.org/pubs/DataStructure-V1.00-Published-20020628.pdf`
- Universal Description, Discovery and Integration v3.0.2 (UDDI), OASIS Standard, 03 February 2005, online: `http://uddi.org/pubs/uddi_v3.htm`

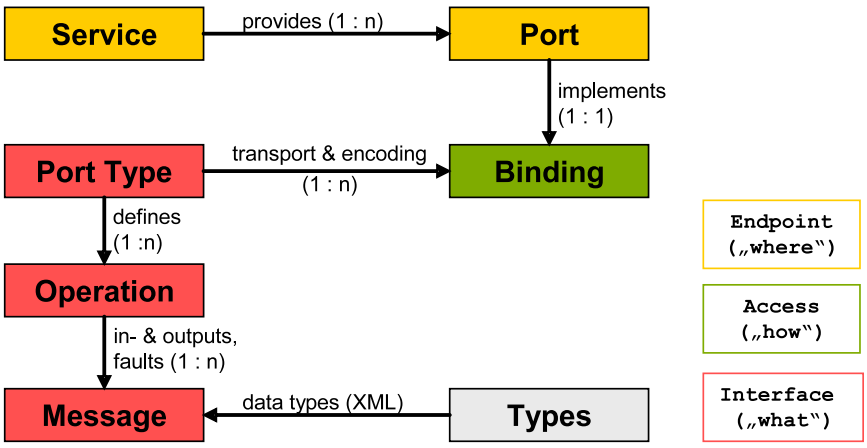


Figure 2.1: Ingredients of WSDL

SOAP. Initially the abbreviation for *Simple Object Access Protocol*, SOAP is a messaging technology for the exchange of XML data over the Web. This has become the standard communication protocol for consuming Web services by the exchange of messages.

As outlined above, every operation in a WSDL description is associated with one or more messages. To consume a Web service, these need to be instantiated with concrete values and then are exchanged between the endpoints via a specific transport protocol. While in the context of Web services SOAP is mostly bound to HTTP in order to facilitate document exchange over the WWW, it can also be bound to other transport protocols. A SOAP message is a XML document which consists of a *header* with technical information, and a *body* that carries the actual content in form of XML data. This is wrapped into an envelope, which then can be bound to a transport protocol for conducting the actual information exchange. Listing 2.1 shows an example for a SOAP message for invoking a Web service for weather forecast. These messages are processed by respective SOAP engines, which denote the heart of execution environments for Web services.

```
<?xml version="1.0" encoding="UTF-8"?>
 <soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <GetWeather xmlns="http://www.webservicex.net">
      <CityName>Innsbruck</CityName>
      <CountryName>Austria</CountryName>
    </GetWeather>
  </soap:Body>
</soap:Envelope>
```

Listing 2.1: An example SOAP message

Universal Description, Discovery and Integration Protocol (UDDI). This is a registry technology intended to support publishing, management, and discovery of Web services. It defines a generic data model for describing Web services with respect to the providing business entity, the technical access information, a natural language description, and a keyword-based classification scheme. In addition, the detailed specification of Web services can be bundled in so-called technical models. The specification comes along with an API in order to support programmatic access to UDDI registries.

The purpose of a UDDI registry is to allow service providers to publish and advertise their Web services, and also to facilitate the search and inspection of suitable Web services by clients. Initially, big vendors such as Microsoft, SAP, and IBM maintained the UDDI Business Registry (UBR) as a single repository for publicly available Web services. However, this effort has been abandoned because the used categorization scheme as well as the UDDI support for publishing and searching Web services turned out to be insufficient. Nowadays, most SOA systems employ registry techniques that are specialized for the specific application scenario. Nevertheless, these proprietary registries follow the principles of UDDI – i.e. describing and organizing Web services in a classification scheme to support clients in the detection of the suitable Web services.

Concluding, the initial Web service technology stack is comprised of three cohesive technologies: (1) WSDL as the standardized description language for Web services, (2) SOAP as the communication protocol for executing Web services, and (3) UDDI as a registry technology for publishing and searching Web services. In addition, several accessory technology standards have been specified for handling usage policies, addressing schemes, security, and other aspects that appear to be relevant for real-world applications [Weerawarana et al., 2005]. A reliable indicator for the thorough adaptation and success of Web services is that essentially all big software vendors committed to this technology.

2.1.2 Service-Oriented Architectures

The invention of Web services and the standardization of the basic technology stack has triggered the concept of Service-Oriented Architectures (SOA) as a new IT system design paradigm [Erl, 2005]. The idea is to use Web services as the basic building blocks of IT systems, and the motivation for this is manifold:

- software fragments from distributed locations that are offered as Web services can be seamlessly combined, which eases the integration and aggregation of services from different providers [Alonso et al., 2004]

- Web services can help to reduce the development and maintenance costs of IT systems by reuse of existing services and by flexible replacement [Marks and Bell, 2006]
- Web services provide a new technology for the integration problem: if two businesses provide their public processes as Web services, then the relevant information can be interchanged while the internal processes remain unchanged [Bussler, 2003].

The initial Web service technology stack as explained above provides a suitable basis for realizing the SOA vision, and the standardization has triggered major research and development efforts in industry as well as in academia. Existing SOA technologies range from freely available tools (e.g. the Methods Web service browser), over open source development kits (e.g. AXIS from Apache), to exhaustive development and management environments from the major software vendors, e.g. the Microsoft's .NET framework, IBM's WebSphere, Oracle's SOA Suite, NetWeaver (SAP), or Crossvision (Software AG). Moreover, the rising interest in Web service and SOA has led to further technology developments such as the integration into business process management (e.g. BPEL4WS, [Andrews et al., 2003]) as well as to service orientation as a new business model [Allen, 2006].

However, the development of sophisticated SOA technologies is an immense challenge. A central problem is the support for the detection of suitable Web services for a concrete client request. This requires an appropriate description that allows clients to determine whether a Web service is actually suitable for the given problem, and SOA systems should support this in an adequate manner. We discuss the deficiencies of the basic Web service technologies for the usability analysis in more detail, which will reveal the motivation for Semantic Web services and in particular for the technology developed in this work.

Figure 2.2 illustrates the procedure of Web service usage by clients on the basis of WSDL, SOAP, and a registry technology like UDDI. The client – which in most cases is the developer of an application wherein Web services shall be used – wants to find a suitable Web service for a certain problem setting. As the first step, the client searches a registry of the available Web services. When a candidate has been found, its actual usability must be determined. This means that the client needs to figure out in what order which messages with what content and under which transport binding must be exchanged with the Web service in order to consume the desired functionality. The relevant information for this is available in the WSDL description of a Web service. However, the client needs to manually analyze the supported operations as well as the required data in order to determine how to invoke the Web service in a way such that it will solve the given task. This problem remains when using tools that automatically generate client stubs, because the generated code merely reflects the WSDL description in a programmatic environment. Once the

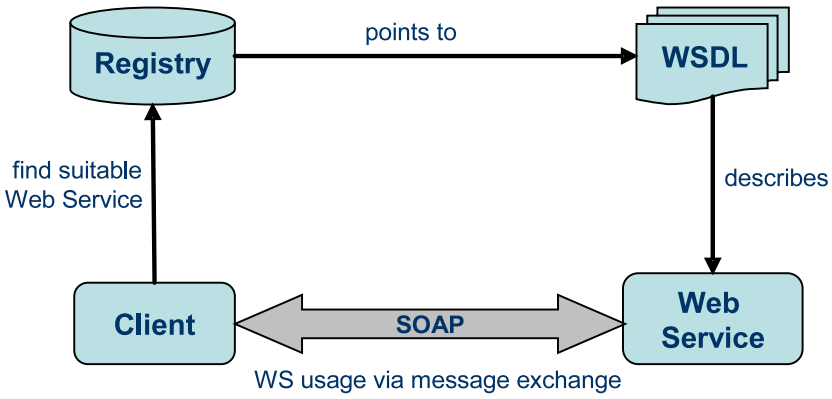


Figure 2.2: Web Service Usage Procedure

usability analysis is completed successfully, the Web service can be invoked and consumed over the specified binding, which usually is SOAP as explained above.

Obviously, the outlined procedure can not be considered to provide sophisticated support for the detection of suitable Web services, because most of the usability analysis tasks are left to manual analysis by the client. Moreover, several problems may arise during the analysis, e.g. that the classification scheme in the repository is too inexpressive so that the candidate search result is imprecise, or that the data of the client and the Web service are incompatible. Thus, more appropriate technologies are needed for supporting Web service detection and the usability analysis, which is at least as important for realizing the SOA vision as the technical infrastructure for the publication and consumption of Web services. One prominent approach that addresses this problem is the emerging concept of Semantic Web services that we will explain in the following.

2.1.3 Semantic Web Services

The aim of Semantic Web services (SWS) is to overcome the deficiencies of the initial Web service technologies, especially for the service detection and usability analysis as discussed above. The approach is to extended Web service descriptions with rich semantic annotations and, upon this, provide inference-based techniques for automating the detection and usage of Web services [McIlraith et al., 2001; Fensel and Bussler, 2002]. Several research and development efforts work on SWS technologies, and there exists a wealth of work on this. We here provide a concise overview that is sufficient in the context of this work, referring to more exhaustive literature for further details (e.g. [Sycara et al., 2003; Cardoso and Sheth, 2006; Stollberg et al., 2006b; Fensel et al., 2006; Studer et al., 2007]).

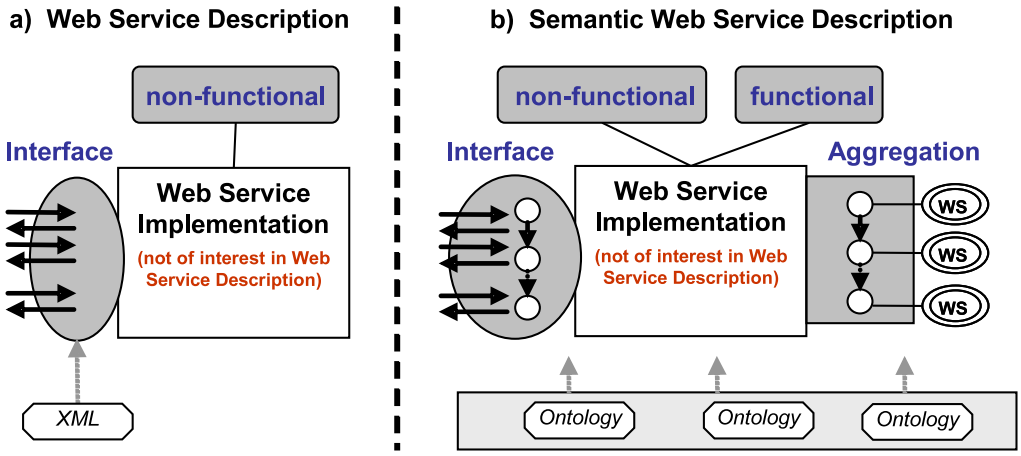


Figure 2.3: From Web Services to Semantic Web Services

Essentially, SWS technologies apply reasoning techniques on formalized descriptions in order to better support the usability analysis of Web services and also to handle the integration problem on a semantic level. The primary tasks that can beneficially be supported by SWS technologies are *discovery* as the detection of suitable Web services for a given task, *composition* as the combination of several Web services to solve a more complex task, and *mediation* as the handling of potentially ocurring heterogeneities that may hamper the interaction between the requester and the provider. For this, the SWS approach extends Web service descriptions as follows (see Figure 2.3):

1. Instead of XML, **ontologies are used as the data model** for describing Web services. These provide formalized knowledge models of a domain that support advanced information processing. Moreover, this pursues the alignment of Web services with the Semantic Web for which ontologies are the base technology (see below).

2. Apart from **non-functional information** such as the owner, usage rights, quality-of-service and financial information, also the **provided functionality** of a Web service is formally described. The primary purpose is to support semantic matchmaking techniques for more precise Web service discovery.

3. The **Web service interface for consumption** is formally described in order to support automated compatibility analysis of the communication behavior supported by the client and the Web service; this corresponds to the WSDL description which merely enlists the supported operations but does not specify in which order these need to be executed.

4. In addition, the **aggregation of Web services** describes how a complex Web service achieves its functionality by combining several other Web services. This aims at automated techniques for analyzing the executability of Web service aggregations in more complex SOA applications.

The following examines the state-of-the-art in SWS research and development in more detail. We commence with the Semantic Web and ontologies, then present and compare the most prominent SWS frameworks, and finally discuss existing SWS technologies for automated discovery, composition, mediation, and execution support for Web services.

Ontologies and the Semantic Web

Ontologies are a modern AI knowledge representation technique. They have been identified as the base technology for the Semantic Web – the grand vision for the further evolution of the WWW [Berners-Lee et al., 2001] – and they are used as the formalized domain knowledge specifications for SWS descriptions. The following explains the definition and the potential of ontologies, and depicts the status of Semantic Web technology developments.

Adopting the denotation from the philosophical study of being and existence, an ontology is defined as a "formal, explicit specification of a shared conceptualization" [Gruber, 1993]. This means that an ontology defines a conceptual model of a domain that ideally represents an agreed consensus among involved parties. The conceptual model is defined in terms of *concepts* that denote the entities in the domain of discourse. These are characterized by *properties*, and *relations* describe associations between concepts whereby subsumption and membership relations define the taxonomic backbone of the ontology. Additional domain knowledge can be specified in terms of *axioms*. Individuals in the domain are represented as *instances* of a concept. The conceptual model is then represented in a formal, machine-processable language upon which reasoning techniques can be applied for advanced information processing. The major merit is that ontologies can bridge the gap between the real world and IT systems [Fensel, 2003], and that heterogeneous data can be integrated on the semantic level by defining mappings between ontologies [Alexiev et al., 2005].

The Semantic Web envisions that Web resources are described in terms of ontologies in order to exploit their potential for advanced and meaning-preserving information processing. Proposed by Tim Berners-Lee – inventor of the WWW and director of the W3C – this is embedded in a larger vision for subsequently augmenting the current WWW with additional languages and technologies that shall be standardized by the W3C. Figure 2.4 shows the so-called *Semantic Web Layer Cake* that illustrates the overall vision: the bottom layers are the already existing WWW technologies (URI, XML, Namespaces) upon which different

ontology languages are defined that are the current focus of standardization work. Op top of this, languages for proof and trust on the Web are targeted as future work.[2]

The idea of the Semantic Web has received high interest in academia and industry, which has led to the formation of a steadily growing, international research community. This has produced a wealth of work that mainly covers:

- formal ontology languages *cf.* [de Bruijn, 2006]) and efficient reasoning techniques (e.g. [Horrocks et al., 2004; Motik et al., 2007]);
- ontology management technologies [Hepp et al., 2007] which cover methodologies and tools for ontology engineering [Goméz-Peréz et al., 2003], scalable ontology repositories (e.g. [Harth and Decker, 2005]), and techniques for ontology versioning and evolution support (e.g. [de Leenheer and Mens, 2006]);
- ontology-based data integration (e.g. [Noy, 2004; Scharffe and de Bruijn, 2005]);
- several applications for Semantic Web technologies [Davis et al., 2006].

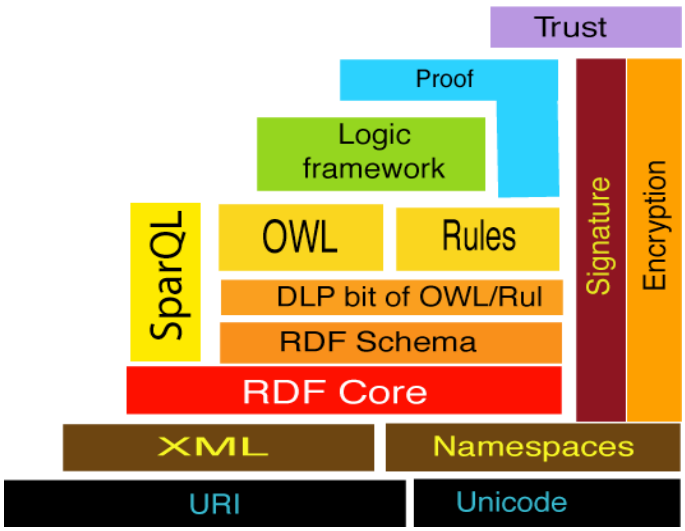


Figure 2.4: The Semantic Web Layer Cake (revised version, 2005)

[2]Figure 2.4 is taken from a keynote talk by Tim Berners-Lee at the WWW 2005 conference, see `www.w3.org/2005/Talks/0511-keynote-tbl/`. At the time of writing, W3C standard recommendations exist for the Resource Description Framework RDF (see `www.w3.org/RDF/`), the Web Ontology Language OWL [McGuinness and van Harmelen, 2004], and the RDF query language SPARQL [Manola and Miller, 2007]; standardization work on a rule language is ongoing, e.g. in the RIF working group (see `http://www.w3.org/2005/rules/wg`).

SWS Frameworks

We now examine existing frameworks that define comprehensive specifications for semantically describing Web services, in general following the SWS approach as outlined above. The aim is to provide an overview of the conceptual frameworks that most of the research in SWS is based upon. As the most relevant ones, we here depict the approaches that have been submitted to or published by standardization bodies.

OWL-S [Martin, 2004]. As the chronologically first approach, OWL-S has been developed in the years 2003 – 2006 by a mostly US-based consortium under the DAML programme (see `www.daml.org`). It defines an upper ontology for semantically annotating Web services that consists of elements as shown in Figure 2.5 (taken from [Martin, 2004]). Every description element is defined on the basis of an domain ontology, and the current standard ontology language OWL is used as the specification language (see above):

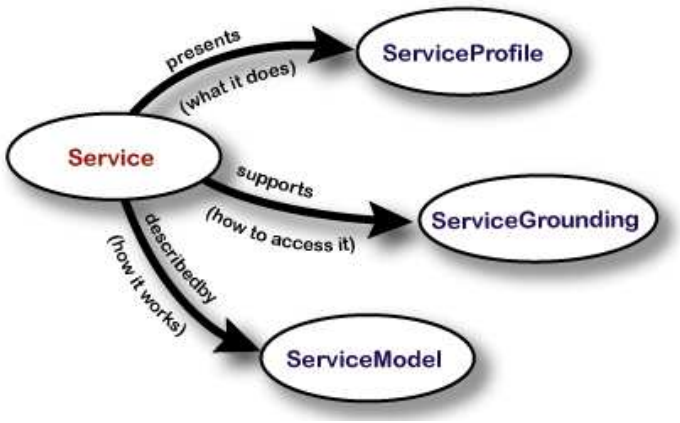


Figure 2.5: Overview OWL-S

1. the *Service Profile* provides information for service advertisement, which contains the name of the Web service, its provider, a natural language description, and a formal functional description that is defined in terms of in- and outputs, preconditions and effects (short: IOPE)

2. the *Service Model* describes how the Web service works. The service is conceived as a process, and the description model defines three types of processes (atomic, simple, and composite processes). These are described by IOPE along with a proprietary process language that defines basic control- and dataflow constructs.

3. the *Service Grounding* gives details of how to access the service, which is realized as a mapping from the abstract descriptions to WSDL.

The intended usage of OWL-S description is as follows. The service profile relate to the information stored in UDDI repositories. While the natural language descriptions are for human consumption, the formal functional description is used for automated Web service discovery by semantic matchmaking (see below). The service model formally describes the external visible behavior of a Web service, i.e. how to invoke and consume the service and what happens when it is executed. This is used to determine whether the communication between a client and the Web service as well as with other aggregated Web services can be carried out successfully. Finally, the service grounding maps the abstract, semantic descriptions to conventional Web service technologies in order to conduct the actual message exchange for execution. Although being criticized on conceptual insufficiencies and especially on the inadequacy of the process description language [Lara et al., 2004], OWL-S has served as the basis for various SWS research and development activities.

WSMO [Lausen et al., 2005]. The Web Service Modeling Ontology WSMO is developed by a European initiative since 2004 (see `www.wsmo.org`). It takes a broader approach than OWL-S, aiming at a comprehensive framework for semantically enabled SOA technologies [Brodie et al., 2005]. For this, it defines four top-level notions as shown in Figure 2.6: *ontologies* that define formalized domain knowledge, *goals* that describe objectives that clients want to achieve by using Web services, semantic description of *Web services*, and *mediators* for resolving potentially occurring heterogeneities.

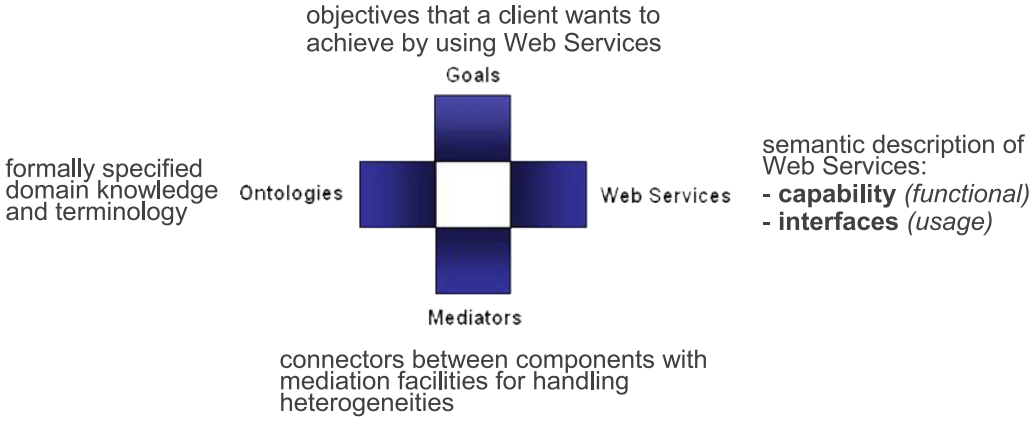


Figure 2.6: WSMO Top Level Notions

In contrast to the other frameworks, WSMO does not only cover the semantic annotation of Web services but propagates a goal-based approach for Semantic Web services along with mediation as an integral part. The idea is that a client formulates requests in terms of a goal, which formally describes the objective to be achieved while abstracting from technical details; the system then automatically detects and executes the suitable Web services in order to solve the goal [Stollberg and Norton, 2007]. The notion of goals provides an explicit element for the client side of SOA applications that enables the lifting of Web service usage by clients to the level of problems that shall be solved. In addition, the integrated mediators support the handling of potentially occurring heterogeneities that can be expected in open and decentralized environments like the Web and may hamper the successful interaction between clients and Web services [Cimpian et al., 2006].

The WSMO framework defines description models and formal specification languages for all four elements. Analogous to Figure 2.3 above, Web services in WSMO are described by *non-functional properties*, a *capability* that specifies the provided functionality in terms of preconditions, assumptions, postconditions, and effects, a *choreography interface* that describes how a client can invoke and consume a Web service, and an *orchestration* that describes how the Web service interacts with other Web services to achieve its functionality. WSMO provides an own specification language called WSML [de Bruijn et al., 2005b], which is a conceptual language for the WSMO elements along with five variants of logical languages that corresponds to the ontology languages developed for the Semantic Web (*cf.* Figure 2.4). Several tools are provided for WSMO, including a suite of reasoners for the different WSML variants and an API for the programmatic management of WSMO elements and definitions. Moreover, there are implementations of execution environments for Semantic Web services, namely WSMX as the WSMO reference implementation (see `www.wsmx.org`) and IRS as a goal-based broker for Semantic Web services [Domingue et al., 2008].

SWSF [Battle et al., 2005]. The Semantic Web Services Framework (SWSF) has been developed by a joint working group of industrial and academic researchers. Essentially, it provides an extension of OWL-S that aims at replacing the initial, insufficient specification model and language for the *Service Model* with an appropriate formal process language. The major contribution of SWSF is a rich behavioral process model based on the Process Specification Language (PSL) [Gruninger and Menzel, 2003]. SWSF provides two axiomizations: (1) FLOWS is based on first-order logic with extensions form situation calculus to model changes of the world; (2) SWSLRules is a logic programming language that serves as both a specification and implementation language and provides support for tasks like discovery, contacting, and policy specification for Semantic Web services.

WSDL-S [Akkiraju et al., 2005]. The WSDL-S approach has been defined in a joint effort of IBM and the University of Georgia. Instead of defining a comprehensive framework for semantically describing Web services, WSDL-S defines extensions to WSDL in order to semantically annotate the XML data types as well as the messages and operations in a WSDL description. For this, a WSDL document is augmented with additional tags that refer to an external domain ontology. In particular, WSDL-S defines three types of annotations independent of the used ontology specification language: (1) WSDL types, i.e. XML data elements, are referenced to concepts of the domain ontology, (2) WSDL operations can be described by preconditions and effects by referencing to respective axioms, and (3) a categorization of Web services can be defined on the basis of the ontology taxonomy.

SAWSDL [Farrell and Lausen, 2007]. While the previously presented approaches have been published as W3C member submissions, *Semantic Annotations for WSDL and XML Schema* (short: SAWSDL) is the only official W3C technology recommendation for Semantic Web services existing at this point in time. It essentially follows the idea of WSDL-S, i.e. the annotation of WSDL documents with additional tags that reference to a domain ontology, independent of the ontology specification language.

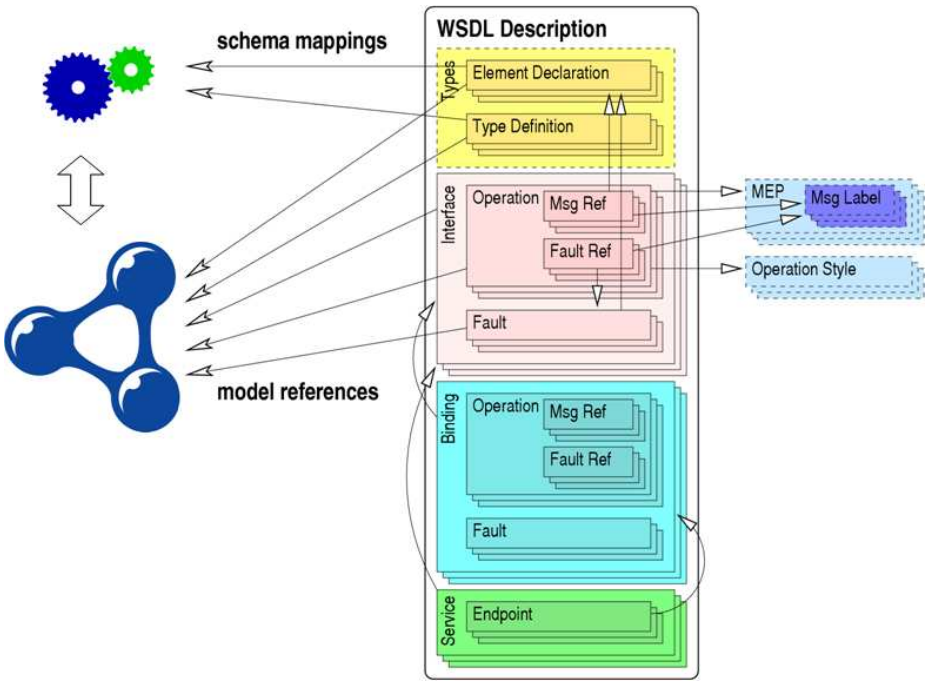


Figure 2.7: SAWSDL Overview

SAWSDL consists of two parts as illustrated in Figure 2.7 (taken from [Kopecky, 2007]): (1) mappings of XML schema definitions to ontology concepts for specifying the correspondence of SOAP message contents to ontology data, and (2) the semantic annotation of WSDL operations. For the latter, SAWSDL defines the annotation by references to only ontology concepts, so that the definition of preconditions and effects in terms of logical expression is not supported. Although this limits the expressivity of the semantic annotations to keywords associated with a domain ontology, several recent works take over this light-weight SWS approach (e.g. [Vitvar et al., 2007a; Martin et al., 2007]).

A comparison of the SWS frameworks reveals the following commonalities and differences. As the chronologically first approach, OWL-S defines a description model for Web services that covers all aspects of the SWS approach as described above – i.e. non-functional aspects, a formal functional description, and formal descriptions of the interfaces for consumption and aggregation of Web services. It uses OWL as the specification language, thus is compliant with the W3C standards for the Semantic Web. SWSF extends this model with a more sophisticated process description language. WSMO is a more exhaustive framework that propagates a goal-driven approach along with integrated mediation facilities. This goes beyond the idea of merely annotating Web services, aiming at an all-embracing framework for semantically enabled SOA technology. WSMO defines an own specification language that covers all ontology languages that are considered for the Semantic Web, and provides reasoners for this along with a set of development tools as well as reference implementations. WSDL-S only partially realizes the SWS approach, therewith can be considered as a light-weight framework. However, it follows the tradition of extending existing technologies standardized by the W3C, and it has served as the conceptual basis for SAWSDL as the only approach for the semantic annotation of Web services that is recommended by a standardization body as of today.

SWS Technologies

After explaining the motivation and overall approach for Semantic Web services, we now turn towards the state-of-the-art in SWS technology research and development. The following focusses on techniques for *discovery* as the detection of suitable Web services for a given task, *composition* as the combination of several Web services to solve a more complex task, *mediation* as the handling of potentially occurring heterogeneities, and *automated execution* of Web services. These are the tasks for which SWS technologies can provide the most surplus value in the service detection and usability analysis phase, and in consequence most existing works address one or more of these topics.

Discovery. This is concerned with the detection of those Web services out of the available ones that are suitable for a given task. This is a central operation in SOA systems for which significant quality increase can be achieved by SWS techniques: on the basis of more precise Web service descriptions, discovery techniques can be developed that expose a higher precision and recall than the syntactic keyword-based search supported by UDDI.

A wealth of work exists on semantically enabled Web service discovery. Most approaches address this by semantic matchmaking of formally described requested and provided functionalities, i.e. OWL-S service profiles or WSMO capabilities as explained above. This allows clients to determine whether a Web service can solve the given request with respect to the preconditions and effects, and is commonly referred to as *functional discovery*. Prominent works for this are [Paolucci et al., 2002; Li and Horrocks, 2003; Kifer et al., 2004; Benatallah et al., 2005; Keller et al., 2006a]. In addition to this, techniques have been developed for handling cases where a match is not given but can be established by relaxing requirements in the request (e.g. [Colucci et al., 2005]), and approaches that integrate other techniques for discovery (e.g. [Klusch et al., 2006; Noia et al., 2003]).

However, also other aspects are considered to be relevant for discovery for which several works present SWS-based techniques. Among these are approaches that consider quality-of-service aspects such as security, robustness, or availability as well as other non-functional aspects (e.g. [Vu et al., 2005; Wang et al., 2006]), ranking techniques that determine a priority list of possible candidates (e.g. [Lu, 2005; Toma et al., 2007]), and techniques that consider the behavioral compatibility as well as input data for determining the suitability of a Web service (e.g. [Stollberg, 2005; Vitvar et al., 2007b]). Also, techniques are proposed for handling cases where a match is not given but can be established by relaxing requirements in the request (e.g. [Colucci et al., 2005; Stollberg et al., 2005a]), and approaches that integrate multiple discovery techniques (e.g. [Klusch et al., 2006; Lara et al., 2006]). We shall discuss the relationship of different discovery techniques in more detail in this work.

Composition. This is concerned with combining several Web services in order to obtain a more complex functionality when this is needed to solve a given client request. The surplus value of Web service composition is that new functionalities can be created that are not provided by the actually existing Web services, which is hardly achievable without any automation support. The aim of composition techniques is to determine possible execution sequences of Web services for solving a given goal. We can distinguish two approaches for automated Web service composition in literature: (1) *functional-level composition* that applies AI Planning techniques to determine suitable compositions on the basis of formal functional descriptions (e.g. [McIlraith and Son, 2002; Wu et al., 2003; Hoffmann et al.,

2007]), and (2) *process-level composition* that takes the behavioral aspects of the Web services into account (e.g. [Berardi et al., 2003; Gerede et al., 2004; Albert et al., 2005]). In fact, both types of composition techniques need to be integrated in order to attain executable compositions of Web services. For this, [Traverso and Pistore, 2004] propose an approach wherein functional composition provides a skeleton of the composition which is then refined by process-level composition. Besides, recent approaches consider Web service discovery and composition as interleaved operations: composition is only needed if a directly usable Web service can not be discovered, and discovery techniques are used to find the candidates during the composition procedure (e.g. [Bertoli et al., 2007]).

Mediation. In the context of Semantic Web services, mediation refers to the handling and resolving potentially occurring heterogeneities which may hamper the interoperability between a requester and a provider. This becomes in particular important within open and distributed environments like the Web where requesters and providers may use different data representation formats, incompatible terminologies, or expose business processes that are not compatible a priori. The main merit of SWS technologies is that such heterogeneities can be handled on the semantic level, i.e. by domain independent techniques that can properly resolve and handle the mismatches [Fensel and Bussler, 2002].

WSMO is the only SWS framework that encompasses mediation as an integral part. It defines several types of mediators along with the necessary techniques for handling different types of heterogeneities, and defines an integrated architecture for the usage of mediators in SWS environments [Cabral and Domingue, 2005; Stollberg et al., 2006a]. The most relevant mediation techniques are (1) *data level mediation* for handling mismatches on terminologies, domain knowledge, and representation formats [Mocan and Cimpian, 2007], and (2) *process level mediation* for establishing a compatible communication behavior between the requester and the provider if this is not given a priori [Cimpian and Mocan, 2005]. The other SWS frameworks do not consider mediation; in fact, they are merely concerned with the semantic description of Web services while remaining orthogonal to all other aspects related to the employment of semantic technology in SOA systems. However, existing techniques for heterogeneity handling can be employed, e.g. ontology-based data integration techniques that have been developed for the Semantic Web (see [Noy, 2004]).

Automated Execution. Once the suitable Web services for solving a given request have been detected, they should be executed automatically in order to minimize the need for human intervention. For this, the semantic descriptions of the Web services need to be mapped to suitable technologies for carrying out the actual information interchange. Commonly, this is achieved by mapping the semantic annotations to a WSDL description,

so that the Web service can be invoked and consumed via SOAP as explained above. This usually also includes an explicit mapping between the XML data types used within SOAP messages and domain ontologies used for the semantic descriptions in order to facilitate the processing of the interchanged data on the semantic level.

This is supported by all the SWS frameworks presented above. OWL-S and SWSF define the mappings in the service grounding element by mapping the domain ontology to an XML Schema and the service model descriptions to WSDL operations. This can be processed by the OWL-S Virtual Machine for automated execution [Paolucci et al., 2003]. The same approach is realized in WSMO: the mappings from the ontology definitions to XML as well as the mapping to WSDL operations is defined within the WSMO choreography interface description, and the WSMX execution component invokes the Web services via WSDL [Kopecký et al., 2006]. Within WSDL-S and SAWSDL, the mappings are defined explicitly by the references to a domain ontology within additional tags in the WSDL document, and this can be processed by respective execution environments.

Concluding, we observe that there exists a wealth of work on Semantic Web services in terms of both overall frameworks and technical solutions for specific tasks. However, these can not yet be considered to be sophisticated as we shall discuss in the following.

2.2 Problem Identification and Approach

After providing an overview of the larger research context, we now turn towards the problem of retrieval accuracy and scalability for automated Web Service discovery engines as the research problem that is addressed in this work.

From the preceding investigations we have learned that Web services support the invocation and consumption of computational facilities over the Internet, and that they are considered as the base technology for SOA as the latest design paradigm for IT systems which receives a lot of attention in academia and industry. The initial Web service technology stack provides a suitable infrastructure for creating and consuming Web services, but it has significant deficiencies in the support for the Web service detection and usability analysis by clients. The emerging concept of Semantic Web services (SWS) is a promising approach to overcome this by developing inference-based techniques for the automated discovery, composition, mediation, and execution of Web services. These work on rich semantic descriptions that are defined on the basis of domain ontologies, and the state-of-the-art examination has shown that a wealth of work on SWS technologies exists in terms of both overall frameworks as well as technical solutions for specific tasks.

However, existing SWS techniques are not yet mature enough to provide sophisticated support for the Web service detection and usability analysis phase. In particular, we identify the following shortcomings of existing solutions:

1. most SWS frameworks and techniques lack in proper support for the client side of SOA technology, i.e. the formulation and handling of client requests for using Web services. A promising approach for this are *goals* that formally describe client objectives while abstracting from technical details for Web service invocation. This can facilitate problem-oriented Web service usage by clients, and support the dynamic detection of Web services in order to enhance the flexibility of SOA systems.

2. Web service discovery is one of the central operations in SOA systems, and the aim of the SWS approach is to automate this. In order to provide a functionally and operationally reliable component in SWS environments, automated discovery engines should expose a high retrieval accuracy and perform the discovery task in an efficient manner. Most existing technical solutions for this are insufficient, in particular with respect to the appropriate integration of both aspects.

3. SOA techniques must be scalable in order to be capable of handling the large amount of Web services that can be expected in real-world applications. This becomes in particular important for SWS environments wherein several expectably complex reasoning operations need to be performed. Under the assumption that usually Web service discovery is performed as the first processing step which needs to consider all potential candidates, it appears to be sufficient to provide scalable discovery techniques in order to warrant the scalability of the whole system.

In order to contribute to the development of more sophisticated SWS technologies, the aim of this work is to elaborate an integrated solution for automated Web service discovery that supports the formulation and processing of client requests on the level of goals, ensures a high retrieval accuracy by semantic matchmaking of sufficiently rich functional descriptions, and is capable of performing the discovery task in an efficient manner also within larger search spaces of available Web services. The following discusses the mentioned deficits of the existing approaches and techniques in more detail, and then outlines the technical solution developed in this thesis. We commence with the motivation for the goal-based approach, then discuss the requirements that arise for the successful deployment of automated discovery engines within SWS environments, and finally explain how the approach taken in this work aims at overcoming the deficiencies of existing solutions.

2.2.1 Motivation for Goal-based SOA Technologies

The existing Web service technologies as well as most works in the field of SWS pay only very little attention to the client side of SOA systems. The following depicts the deficiencies, and outlines how the goal-based approach aims at overcoming these.

The initial Web service technology stack limits the detection of suitable Web services to manual inspection, and merely supports the consumption of Web services by hard-wired invocations (see Section 2.1.2). After a usable Web service has been found by searching a UDDI-like registry and the technical details for the invocation have been determined from the WSDL description, the client can integrate the Web service into the target application via so-called *client stubs* that support the exchange of SOAP messages out of programmatic environments. Apart from the insufficient support for the detection and usability analysis, this hampers the flexibility of the system: whenever a Web service is changed, its invocation in all dependent client applications must be updated. Most SWS techniques merely focus on the semantic descriptions for Web services, in particular those based on OWL-S or WSDL-S (see Section 2.1.3). In order to utilize these for automated Web service detection and usability analysis, clients are expected to formulate queries or conditions on specific description elements. This requires sophisticated knowledge on the description model as well as on the operating mode of the applied SWS techniques.

The idea of the goal-based approach is to overcome these deficiencies by describing client requests in terms of *goals*. Here, a goal is a formal description of the objective that a client wants to achieve by using Web services, which focusses on the problem to be solved while abstracting from technical details. Such a goal is submitted to the SWS system, which then shall automatically detect and execute the necessary Web services for solving the goal. This approach has two central benefits for better supporting the client side of SOA applications as illustrated in Figure 2.8. The first one is that goals provide an abstraction layer for facilitating problem-oriented Web service usage by clients: a client merely describes the objective to be achieved in terms of a goal, while the usability analysis as well as the technical details for the invocation of the necessary Web services is handled by the SWS system [Fensel et al., 2006]. The second one is that goals can serve as a cache from which the actual Web services for a concrete client requested can be detected dynamically at runtime. The figure illustrates this for a business process wherein the activities are defined in terms of goals. At runtime, one of the Web services that are suitable for solving the goal can be selected with respect to the concrete input data defined by the client. This allows us to overcome the problems on the flexibility of Web service usage in dynamic environments which result from hard-wired invocations [Zaremba and Bussler, 2005; Hepp et al., 2005].

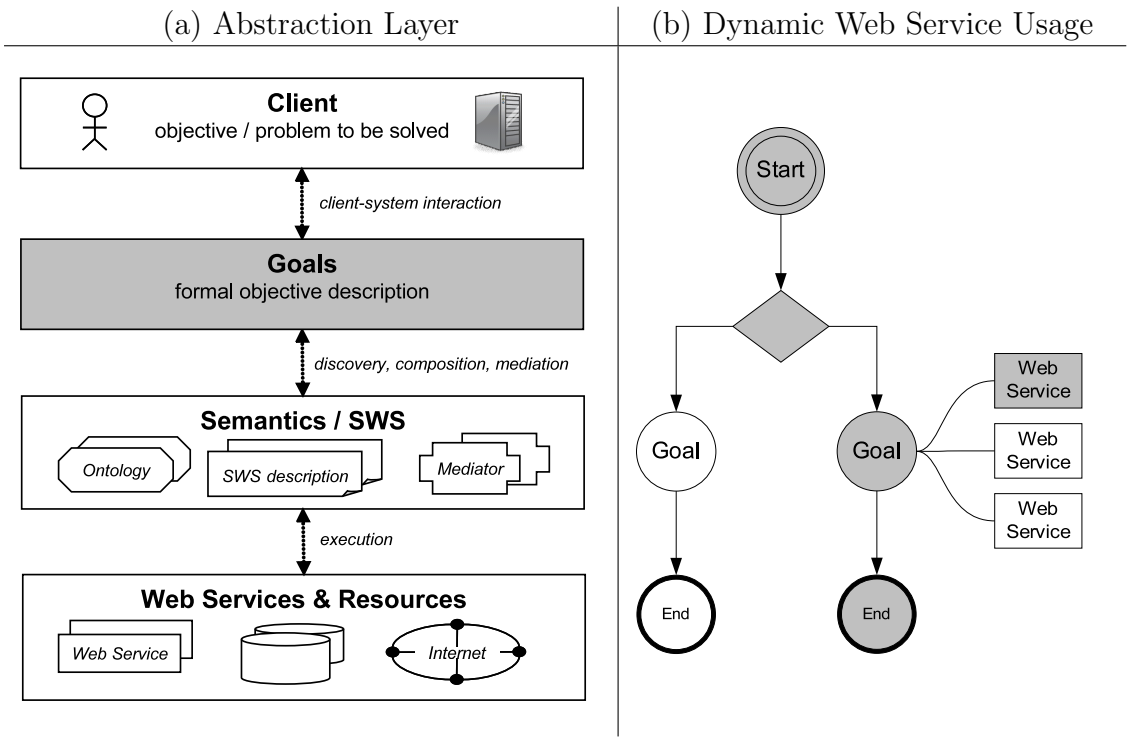


Figure 2.8: Purpose of Goals in SOA

Goal-based techniques with a similar purpose have been developed in AI technologies for automated problem solving, e.g. in BDI agents [Bratman, 1987], cognitive architectures [Newell, 1990], and knowledge engineering (e.g. in UPML [Fensel et al., 2003]). In the field of Semantic Web services, the goal-based approach is propagated by the WSMO framework. It defines goals as one of the top-level elements, which are described by a requested capability and requested interfaces for consuming Web services [Roman et al., 2006].

However, experiences from the development of techniques for the automated discovery, composition, and execution of Web services have shown that the initial model for describing and handling goals as defined in WSMO appears to be not yet sophisticated enough to properly support goal-based Web service usage as outlined above. In particular, it appears to be reasonable to explicitly distinguish *goal templates* as generic and reusable descriptions of client objectives and *goal instances* that describe concrete requests. Also, the approach for the automated invocation and consumption of Web services for solving a goal as well as the support for specifying and processing more complex client objectives appears to be immature. With respect to this, in this thesis we present a refinement of the WSMO goal model in order to better support the goal-based approach for SOA technologies.

2.2.2 Requirements for Automated Web Service Discovery

We now investigate the Web service discovery task in more detail. We commence with the architectural allocation in SWS environments, and then explain the arising requirements for automated discovery engines and depict the shortcomings existing technical solutions.

Figure 2.9 illustrates the procedure for automatically solving goals in a SWS environment by the automated and dynamic detection and execution of Web services. This is an abstraction of the architecture and execution semantics of WSMO reference implementation WSMX [Haller et al., 2005], which complies with early works on SWS system architectures [Preist, 2004] and is currently being specified in detail by the OASIS technical committee for SWS environments (`www.oasis-open.org/committees/semantic-ex/`).

The input is a goal that shall be solved, and procedure specifies the workflow of functional components which perform distinct operations by the SWS techniques explained in Section 2.1.3. The first processing step is Web service discovery. Here, this refers to functional discovery by matchmaking of the formally described requested and provided functionalities. When a suitable Web service has been discovered, the selection & ranking facilities choose one of the candidates or determine a priority list for the further processing with respect to quality-of-service criteria as well as data security requirements and usage rights. Then, the behavioral compatibility is checked in order to ensure that the client can

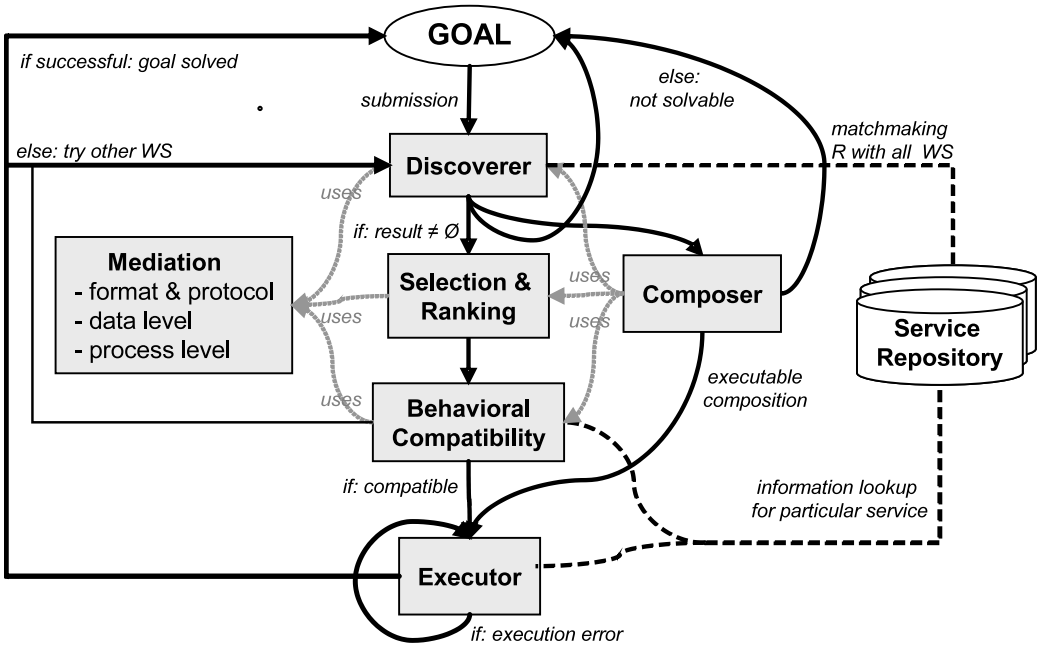


Figure 2.9: Automated Goal Solving in SWS Environments

successfully invoke and consume the chosen Web service. When this is given, the Web service is automatically executed in order to solve the goal. If a single Web service for solving the goal does not exist, then the composition component is invoked in order to create a suitable composition of the available Web services; this utilizes the other components for detecting possible candidates for the composition procedure. In addition, the components can use mediation facilities in order to resolve and handle potentially occurring heterogeneities that hamper the successful interaction between the client and the Web service.

Examining this procedure reveals the importance of automated Web service discovery within SWS environments. It usually is performed as the first processing step which needs to inspect all available Web services, while the successive components merely need to deal with the set of discovered candidates. Moreover, the discovery engine is a heavily used component within the system: it needs to be invoked for each incoming client request in order to facilitate the dynamic Web service usage as outlined above, and it might be invoked several times for solving a single goal – e.g. in the course of automated composition in order to detect the possible candidate Web services. This substantiates the two requirements on automated Web service discovery engines mentioned above: they should expose (1) a high retrieval accuracy in order to perform the discovery task with an appropriate quality, and (2) a high computational performance in order to serve as a operationally reliable software component in SWS environments. The following discusses both aspects in more detail and depicts the deficiencies of existing technical solutions.

On the Retrieval Accuracy

This is related to the quality of the discovery technique for determining the suitability of Web services for a given goal. For this, a high retrieval accuracy appears to be desirable in order to provide a sophisticated filtering facility for potential candidates which are then further inspected in the subsequent processing steps. In terms of the central quality criteria for information retrieval [Baeza-Yates and Ribeiro-Neto, 1999], this means that, under functional aspects, every Web service in the discovery result should be suitable for solving the goal (precision), and that every suitable Web service can be discovered (recall).

As outlined above, this can most adequately be achieved by employing semantic matchmaking techniques for the discovery task. In principle, these can achieve a very high retrieval accuracy: given sufficiently rich functional descriptions that are defined on the basis of an exhaustive domain ontology, one can specify semantic matchmaking techniques for very precisely determining whether a Web service can be used for the given goal or not. However, most existing approaches for semantically enabled discovery lack in exactly this respect.

Some merely work with keyword-based descriptions (e.g. [Oundhakar et al., 2005]), while others only consider in- and outputs for the matchmaking (e.g. [Paolucci et al., 2002]) or simplify this to subsumption reasoning on logically separated elements of the functional descriptions (e.g. [Li and Horrocks, 2003; Noia et al., 2003]). It thus is necessary to define formal functional descriptions for goals and Web services that can precisely describe the requested and provided functionalities, and upon this define semantic matchmaking techniques that warrant a high precision and recall for the discovery task. The discovery techniques developed in this thesis address this challenge, for which existing works can serve as a starting point (e.g. [Kifer et al., 2004; Keller et al., 2006a]).

On the Computational Performance

The second requirement addresses the computational performance of automated discovery engines, which relates to the time and resource efficiency for performing the discovery task. This becomes in particular relevant within larger search spaces of available Web services that can be expected in real-world applications, and also for the employment of discovery engines as heavily used components in SWS environments.

At the time of writing, the SOA system maintained by the US-based telecommunication provider *Verizon* encompasses around 1.500 registered Web services (see Section 6.2.2 for details), and the Web service search engine developed by SeekDa finds around 15.000 Web services that are publicly available on the Internet (see `http://seekda.com/`). The number of both public and access restricted Web services is expected to grow significantly in the next years with the adoption of SOA technology in industry [Marks and Bell, 2006]. SOA technologies need to be scalable in order to adequately operate in such environments, and this becomes especially important for SWS technologies wherein several, expectably expensive reasoning tasks need to be performed. Considering SWS systems as outlined in Figure 2.9 where discovery is performed as the first processing step and is the only operation that needs to consider all available Web services, it appears to be sufficient to provide scalable discovery techniques in order to warrant the scalability of the whole system.

Also related to the computational performance is the efficiency and the stability for performing the discovery task. This becomes especially relevant for advanced SWS technologies that aim at employing a discovery engine as a heavily used component. For example, recent approaches for Web service composition aim at using discovery for detecting the suitable candidates in each step of the composition algorithm [Bertoli et al., 2007]. Similarly, approaches for semantically enabled business process management plan to define specific activities in a process in terms of goals for which the actual Web services shall be

determined at execution time [Hepp et al., 2005]. Considering that compositions or processes can be complex and may consist of several Web services, the absolute overhead and the predictability of the discovery engine becomes critical for realizing such techniques.

Thus, we can judge the computational performance of automated discovery engines by adopting the standard criteria for performance measurement as follows: *efficiency* as the time required for completing a discovery task, *scalability* as the ability to deal with a large search space of available Web services, and *stability* as a low variance of the execution time of several invocations [Ebert et al., 2004]. A discovery engine can be considered to provide an operationally reliable software component in SWS systems if it exposes a sophisticated performance with respect to these criteria. A suitable approach to achieve this is to reduce the relevant search space as well as the necessary reasoning effort for the discovery task. Although reasoning techniques can be optimized for specific tasks (e.g. [Duschka and Genesereth, 1997; Motik et al., 2007]), they in general have significant deficits in efficiency when dealing with large search spaces [Wache et al., 2004; Fensel and van Harmelen, 2007].

Existing approaches address this by applying clustering techniques for Web services. The basic idea is to organize the available Web services in tree-like cluster structures, which is traversed for discovery so that matchmaking is only required for the Web services in a specific cluster [Giunchiglia et al., 2005; Tausch et al., 2006]. Several works address this by extending the classification mechanism that is already supported in UDDI by annotating Web services with concepts of a domain ontology, so that the taxonomy of the ontology constitutes the indexing structure. Upon this, pre-filtering mechanisms for the discovery task are deployed that usually also take non-functional properties into account (e.g. [Srinivasan et al., 2004b; Verma et al., 2005; Lara et al., 2006; Abramowicz et al., 2007]).

However, the central requirement for such pre-filtering mechanisms as an appropriate extension for Web service discovery is to ensure a high retrieval accuracy. This means that the Web services that are grouped in a cluster should be a proper superset of those that a discovery engine would find for a request that corresponds to the cluster. The keyword-based clustering techniques are too imprecise for this. Thus, the per-filtering result may conflict with the discovery result, so that the techniques might become counterproductive. Besides, the referred approaches require the annotation of Web services by the provider as an additional manual effort with the risk of incorrectness. An approach that overcomes these problems is presented in [Constantinescu et al., 2005], which organizes Web services on the basis of their formal functional descriptions. It automatically creates a search tree whose leaf nodes are the Web services while the inner nodes are predicates that define common conditions for all child nodes. However, the approach defines a proprietary model for functional descriptions with significant limitations in expressivity.

In order to overcome these deficiencies, it appears to be necessary to develop an optimization technique that can perform Web service discovery in an efficient manner with a low variance among several invocations and also ensures the scalability in larger search spaces while maintaining a high retrieval accuracy by applying semantic matchmaking techniques that work on sufficiently rich functional descriptions. This work aims at developing a novel technique for this, which we shall outline in the following.

2.2.3 Overview of Approach

After discussing the motivation for the goal-based approach and identifying the requirements for automated Web service discovery, we now outline the approach developed in this work and explain how this aims at overcoming the mentioned deficiencies of existing solutions.

Figure 2.10 provides an overview of the approach that we shall elaborate in detail in the remainder of the thesis. We take the goal-based approach, and explicitly distinguish between *goal templates* as generic and reusable objective descriptions that are kept in the system, and *goal instances* that describe concrete client requests by instantiating a goal template with concrete input values. For this, we define a semantically enabled discovery framework that distinguishes two phases: at design time, the suitable Web services for goal templates are discovered. The result is captured in a specialized knowledge structure, which serves as the heart of the caching mechanism for optimizing the computational performance. At runtime, a client – either a human or a machine – formulates the concrete objective to be achieved in terms of a goal instance. As the time critical and expectably most frequent operation in real-world SOA applications, the discovery of suitable Web services for goal instances at runtime is optimized by exploiting the captured knowledge.

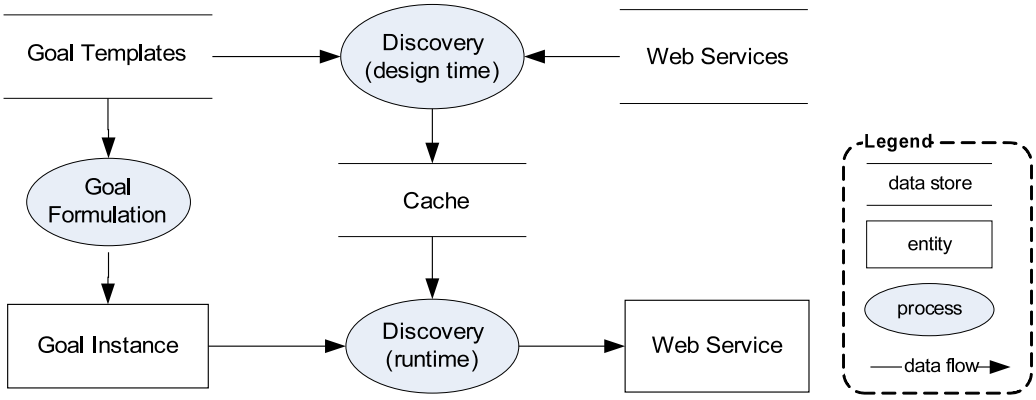


Figure 2.10: Overview of Approach

In order to realize this and to overcome the deficiencies of existing solutions as discussed above, the first part of the thesis develops a conceptual model for describing and processing goals in SWS environments. For this, we refine and extend the initial goal model defined in the WSMO framework. In particular, we define the explicit distinction of goal templates and goal instances, and we revise the approach for enabling the automated invocation and consumption of Web services for solving a goal.

The second part specifies two-phased Web service discovery and defines the necessary semantic matchmaking techniques for this. This allows us to perform potentially expensive operations at design time (which is not time critical), and the results can be used to reduce the reasoning effort at runtime (which is time critical). This follows the idea of *heuristic classification* that has been identified in [Clancey, 1985] as a basic principle for efficient inference-based matchmaking. A given problem is first lifted to a more abstract level that deals with concepts instead of concrete data. Then, suitable candidates for solving the problem are detected by matchmaking on the abstract level, and finally the candidate set is refined with respect to the concrete data and conditions of the given problem. This approach has been proven to facilitate the development of effective search techniques for large-scale and knowledge-intensive application areas [Fensel et al., 2003], and has also been shown to be suitable for developing workable and efficient SWS techniques [Keller et al., 2005]. A similar approach is realized in the IRS system: the suitable Web services for abstract goal descriptions are defined at design time, and at runtime the actual Web services are selected with respect to the concrete inputs defined by a client [Galizia et al., 2007].

In order to ensure a high retrieval accuracy for the Web service discovery task, we define sufficiently rich functional descriptions that precisely describe the requested and provided functionalities on the level of the possible executions of Web services and solutions for goals. These are defined in terms of *preconditions* that specify the possible start-states and *effects* that define the end-states on the basis of domain ontologies. We define the functional descriptions such that they explicitly specify the dependencies between the preconditions and effects, and also consider the computational in- and outputs. Upon this, we define semantic matchmaking techniques for both the design time and runtime discovery task that are able to precisely determine the functional usability of a Web services for solving a goal template as well as for a goal instance. We use classical first-order logic (FOL) as the specification language, and apply an automated theorem prover for the matchmaking. Although FOL is undecidable in the general case, this provides a sufficient expressivity for functional descriptions and also the necessary reasoning facilities. Besides, we expect that most functional descriptions can be expressed within decidable subsets of FOL, so that the limitations of the approach become less significant for real-world applications.

Finally, the third part of the technical solution is a optimization technique for automated Web service discovery. The approach is to organize goal templates in a subsumption hierarchy with respect to the requested functionalities, and capture the relevant knowledge on the usability of the available Web services from design time discovery results. This provides a index structure for the efficient search of goals and Web services, which is generated automatically on the basis of semantic matchmaking and is also properly maintained whenever a goal template or a Web service is added, removed, or modified. The optimized discovery algorithms exploit this knowledge in order to enhance the computational performance by minimizing the relevant search space as well as the number of necessary matchmaking operations. This is a novel approach in the field of Semantic Web services which adopts the concept of caching as a well-established means for performance optimization applied in several areas of computing (e.g. [Astrachan and Stickel, 1992; Handy, 1998; Wessels, 2001]), and also follows a common approach for efficient reasoning techniques by pre-computing relevant knowledge at design time (e.g. [Horrocks et al., 2004; Kiryakov et al., 2005]).

This technique can significantly increase the computational performance of the discovery task, because only the minimal set of potential candidates needs to be inspected and also the necessary reasoning effort is minimized. Moreover, it maintains the retrieval accuracy of our semantically enabled discovery techniques, and therewith overcomes the deficiencies of existing optimization techniques as discussed above. We shall evaluate the achievable performance increase within an exhaustive use case analysis, and also examine the practical relevance of the developed technology within real-world applications.

2.3 Summary and Outlook

In this chapter we have introduced into the field of Web services, SOA, and Semantic Web services, motivated the need for efficient and scalable Web service discovery techniques with a high retrieval accuracy as the research topic of this work, and finally outlined the technical solution for this which shall be elaborated in the course of this thesis.

Web services are a technology for invoking and consuming computational facilities over the Internet. They are considered as the basis for Service-Oriented Architectures (SOA) as a novel design paradigm for IT systems that receives a lot of attention in industry. We have shown that the initial Web service technology stack provides a basic infrastructure for creating and consuming Web services, and we have presented the state-of-the-art in the emerging concept of Semantic Web services (SWS) that aims at developing inference-based techniques for automated discovery, composition, and execution of Web services.

Although there is a wealth of work on SWS frameworks as well as technical solutions for the relevant tasks, existing SWS techniques still have deficiencies for supporting and automating the Web service usage by clients in a sophisticated manner. In particular, most existing solutions only pay little attention to the client side of SOA applications. We have argued that a goal-based approach seems to be suitable for this: a goal formally describes the objective that a client wants to achieve while abstracting from technical details, and SWS techniques can be used to automatically detect and execute the necessary Web services for solving a goal. This allows us to lift the interaction of clients and Web services to the level of problems that can be solved, and also to achieve a better flexibility of SOA systems.

We further have identified Web service discovery as a central operation in SOA, which is concerned with detecting suitable Web services out of the available ones and is usually performed as the first processing step in SWS systems in order to solve a given client request. We have determined two central requirements on automated discovery engines: (1) a high retrieval accuracy in order to perform the discovery task with an appropriate quality, and (2) a high computational performance in order to serve as a operationally reliable software component in SWS systems. The inspection of existing approaches has shown that most technical solutions lack in the achievable quality for either of the requirements, and in particular in the adequate integration of both aspects which appears to necessary in order to provide sophisticated automated Web service discovery components for SWS systems.

With respect to this, we have outlined the approach that is elaborated in this work. This takes a goal-based approach, and defines a two-phased discovery framework that separates design time and runtime operations and applies semantic matchmaking techniques that work on sufficiently rich functional descriptions in order to warrant a high retrieval accuracy. This is extended with a caching mechanism that captures relevant knowledge from design time discovery runs, and effectively utilizes this in order to enhance the computational performance of Web service discovery at runtime.

The remainder of the thesis is concerned with the detailed elaboration of the approach. As the three parts of the technical solution, Chapter 3 specifies the refined goal model for Semantic Web services, Chapter 4 defines the two-phased discovery framework along with the necessary semantic matchmaking techniques, and Chapter 5 specifies the caching mechanism for enhancing the computational performance of automated Web service discovery engines. Chapter 6 evaluates the achievable performance increase and discusses the practical applicability of the developed techniques, and finally Chapter 7 concludes the work.

Chapter 3

A Goal Model for Semantic Web Services

This chapter presents a conceptual model for the specification and usage of goals as formal descriptions of client objectives in the context of Semantic Web services (SWS). A goal describes what a client wants to achieve while abstracting from the technical details related to the invocation and consumption of Web services. The overall aim is to facilitate problem-oriented Web service usage: the client shall merely specify the objective to be achieved in terms of a goal, and the SWS system automatically detects and executes the necessary Web services for solving this. Therewith, goals represent a modeling element for the client side of SOA technology, which is not covered by the initial Web service technology and is also neglected in most SWS approaches.

The aim is to specify a conceptual model for describing goals and their usage in SWS systems in order to better support the idea of goal-driven SOA technology. For this, we refine the goal model defined in the WSMO framework, which is the only SWS approach that identifies goals as a top level element and promotes the idea of goal-driven SWS techniques [Fensel et al., 2006]. The examination of existing works reveals that the following three aspects are desirable for realizing goal-based SWS techniques: (1) the *expressiveness of goal descriptions* should be sufficient for all kinds of objectives that clients may want to achieve by using Web services, (2) the explicit distinction of *goal templates* as generic and reusable objective descriptions and *goal instances* that denote concrete client requests appears to be desirable in order to allow the development of efficient and workable SWS techniques and also to ease the goal formulation by clients, and (3) the support for *automated Web service invocation and consumption* for solving a goal should be enhanced.

We consider these aspects as the requirements for a suitable goal model. The definition of goals in the WSMO framework appears to be too immature in this respect, although it aims at the same usage purpose. We thus refine the specification of the goal model elements while maintaining the general structure as well as the intended usage of goals as defined in WSMO. The main focus of the following elaborations is the conceptual structure of the goal model, i.e. the definition of its elements and the relation to SWS techniques for automated discovery, composition, mediation, and execution of Web services. We use the WSMO specification languages for illustrating the definitions. However, the conceptual model can also be adopted to other specification languages.

The chapter is organized as follows. At first, Section 3.1 explains the origin of the concept of goals and identifies the requirements that arise in SWS systems. Then, Section 3.2 presents the specification of the refined goal model with the main attention on the conceptual structure, the usage purpose, and relationship of its elements. Finally, Section 3.3 discusses the suitability of the goal model and positions it within related work. We use the travel scenario for illustration throughout the chapter, which is often used for demonstration and explanation of SWS techniques [He et al., 2004; Stollberg and Lara, 2004].

3.1 Aim and Requirements Analysis

The following first explains the concept of goals as formal descriptions of client objectives that originates from works on automated problem solving in different AI disciplines. Then, we discuss the specific requirements that arise for describing and using goals in the context of SWS, and, with respect to this, we depict the deficits of the initial WSMO goal model.

3.1.1 Goals – Origin and Purpose

The concept of goals as we understand it here originates from AI research on technologies for automated problem solving. The theoretic foundations root in cognitive science that studies the human mind and behavior as the basis for developing intelligent technologies [Hofstadter, 1979; Wilson and Keil, 1999]. Therein, problem solving is understood as a part of human thinking that is concerned with solving a given task when the procedure for this is not known a priori, and the aim of AI technologies is to automate this in order to enable computers to perform problem solving in a human-like manner [Anderson, 1991].

A primary theory for this has been presented in [Newell and Simon, 1972]. Therein, problem solving is understood as a goal-driven process: a goal describes the desired state of the world, and the task of problem solving is to determine and perform suitable actions

in order to reach the goal state. To automate this, goals are specified in terms of rules and conditions in a logical language. If these hold, then the goal state is reached and the problem is considered to be solved. The applicable operators – which are mostly computational facilities that are able to perform actions – are described by usage conditions and the effects of their execution. Upon this, inference-based techniques automatically determine suitable operators whose execution will change the world from the current state into the goal state. This allows the user to delegate problem solving to a computer system whereby the client-system interaction takes place on the level of goals.

Although being criticized for reducing the model of human mind to a production system for information processing, this model as served as the conceptual basis for further developments of goal-based techniques for automated problem solving. Prominent approaches are the belief-desire-intention (BDI) model for rationale behavior of goal-driven, autonomous agents [Bratman, 1987; Cohen and Levesque, 1990; Rao and Georgeff, 1991], AI planning techniques that are concerned with the automated construction of plans as suitable sequences of operators to solve a given goal [Allen et al., 1990], and the UPML framework that defines a comprehensive framework for describing the reasoning behavior of knowledge-based systems wherein so-called tasks describe the problems to be solved [Fensel et al., 2003]. Also, specific AI technologies adopt the notion of goals, e.g. logic programming languages wherein the user requests are called goals (e.g. in PROLOG, [Bratko, 2000]). We shall discuss these approaches in more detail in Section 3.3.

The main merit of such goal-based techniques is that the client-system interaction can be lifted to the *knowledge level* that is concerned with tasks, actions, effects, and behavior in the world [Newell, 1982]. The connection to the lower *symbol level* that deals with the technical implementation details is handled by the system, and the client (ideally) does not need to care about this. This corresponds to the idea of goal-based SOA technologies for problem-oriented Web service usage as outlined in Section 2.2.1: clients formulate requests in terms of goals, i.e. as objective descriptions on the knowledge level, and the SWS system automatically detects and executes the relevant Web services for solving this.

To realize this, a goal description must carry all information that are necessary to detect the suitable Web services and to invoke them on the symbol level. We shall discuss this in more detail in the following. We here consider Web services to correspond to operators, i.e. to be passive computational facilities that can be used to solve a goal. Web services might provide complex functionalities, but they do not solve problems in an autonomous and proactive manner by themselves, which distinguishes them from intelligent agents [Dickinson and Wooldridge, 2005]. This corresponds to SOA idea of using Web services as basic building blocks in IT systems as discussed in Section 2.1.2.

3.1.2 Requirements on a Goal Model for SWS

We now turn towards the requirements that arise on a goal model for semantic SOA technologies in order to properly support the idea of automated, goal-driven Web service usage as outlined above. The following identifies the requirements by analyzing existing works on goal-based SWS techniques. On this basis, we then examine the goal model defined in the WSMO specification, and we depict its deficiencies in order to motivate the need for the refinements presented in the next section.

Requirements

(1) Expressiveness of Goal Descriptions. This is concerned with the sufficiency of the goal model for describing all kinds of objectives that clients may want to achieve by using Web services. These can range from simple requests to complex problems that require the usage of several Web services. Formal descriptions based on domain ontologies seem to be most suitable for this, also in order to facilitate reasoning within SWS environments.

We find different types of goals in literature. Most are concerned with a desired state of the world (e.g. the goal state in the famous 'Tower of Hanoi' puzzle), with functions between inputs and outputs that shall be performed (e.g. a booking request in the travel scenario), or with notification requests (e.g. on-time reception of stock market information). Goals that abide over a longer time period and require several reconsideration steps are usually assigned to intelligent agents, but we hardly find them as usage requests for Web services. However, we find goals that require a workflow to be sustained for their resolution (e.g. [Albert et al., 2005]), and goals that specify non-functional aspects in addition to the basic objective description (e.g [Galizia et al., 2007; Toma et al., 2007]). Besides, we observe that goals are used for different purposes: (a) to describe the overall aim that a client wants to achieve by using Web services (e.g. [Keller et al., 2005]), and (b) to define the functionality required for performing an activity that is part of a larger process for which the actual Web service shall be detected at execution time (e.g. [Wetzstein et al., 2007]).

(2) Support for Automated Web Service Usage. This relates to the automated execution of the detected Web services in order to automatically solve a goal as envisioned by the SWS approach (see Section 2.1.3). For this, a goal must carry all information that is needed to actually invoke and consume a Web service, in particular: (1) all inputs required by the Web service, (2) a compatible counterpart of the communication behavior supported by the Web service for consuming its functionality, and (3) the actual information exchange can be performed, e.g. via SOAP on the basis of WSDL bindings (see Section 2.1.1).

(3) Distinction of Goal Templates and Goal Instances. This refers to the explicit distinction of generic and reusable goal descriptions (goal templates) and goal descriptions that represent concrete client requests (goal instances). As discussed in Section 2.2.3, this enables the development of efficient SWS techniques that perform expectably expensive operations on goal templates at design time and utilize this to reduce the operational costs for solving goal instances at runtime. Existing SWS environments that support the goal-driven approach already make this distinction, e.g. the Semantic Web Fred system [Stollberg et al., 2005b], the IRS system [Domingue et al., 2008], and also the WSMO reference implementation WSMX [Haller et al., 2005]. However, the separation of design- and runtime operations in SWS environments requires a precise definition of the relationship of goal templates and goal instances. We shall discuss this in more detail throughout this work.

(4) Ease of Goal Formulation by Clients. From a pragmatic perspective, it seems to be reasonable to request as little as necessary from clients for specifying a goal [Domingue et al., 2008]. Moreover, minimizing the manual specification effort by end-users can help to reduce the risk of faulty and imprecise goal definitions [Lara et al., 2006].

The WSMO Goal Model

We complete the requirements analysis with examining the goal description model as defined in the latest final version of the WSMO specification [Roman et al., 2006]. This defines two central description elements for goals. The first one is a *capability* that describes the requested functionality in terms of preconditions, assumptions, postconditions, and effects. This appears to be expressive enough for specifying all kinds of client objectives identified above (*cf.* requirement 1). The second element is a *requested interface*, which is intended to provide the counterpart of a Web service interface description for the automated invocation and consumption (*cf.* requirement 2). While not further defined in the specification, most works use this to specify the communication behavior that is supported by the client for consuming Web services. The descriptions are based on domain ontologies, and mediators can be used to handle data level mismatches and to connect related goals. A WSMO goal can further be described by non-functional properties.

This model does not distinguish between goal templates and goal instances (*cf.* requirement 3), and solutions for supporting the goal formulation by clients is left open to implementations (*cf.* requirement 4). Moreover, the definition of constraints on the resolution process of goals is not supported, and the concept of requested interfaces remains unclear and vague. We thus consider the current WSMO goal model to be not sufficient to support the idea of goal-driven SWS techniques with respect to the identified requirements.

3.2 Conceptual Model and Specification

This section presents the specification of the goal model for Semantic Web services. It extends and refines the existing WSMO goal model, aiming at adequately satisfying the requirements identified above. Figure 3.1 shows the core of the goal model as a UML Class diagram that defines the elements along with their description structure and relationships.

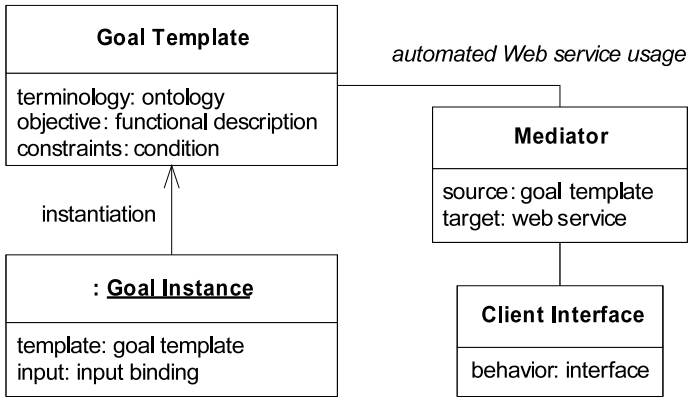


Figure 3.1: Core of the Goal Model

The model explicitly distinguishes *goal templates* as generic and reusable objective descriptions, and *goal instances* that describe concrete client requests. This is an extension to the WSMO goal model in order to enable efficient SWS techniques and also to ease the goal formulation by clients. We shall specify this in more detail in Section 3.2.1. The second central aspect is the approach for automated Web service invocation and consumption. For this, we connect each Web service that is suitable for solving a goal template with a *mediator* that carries a *client interface* for the automated invocation and consumption of the Web service to solve associated goal instances. This is a refinement of the previously insufficient solution in WSMO, and we shall specify this in detail in Section 3.2.2. While the former two aspects are the core elements, we consider the goal model to be extensible. In particular, it supports the specification of composite goals that aggregate several goals in order to describe more complex objectives, and also the definition of non-functional aspects in addition to the requested functionality. We shall discuss this in Section 3.2.3.

The following specifies the goal model in detail with examples from the travel scenario and discusses related works. We use the WSMO specification languages for exemplifying the modeling, in particular the Web Service Modeling Language WSML which supports the specification of both conceptual and logical aspects [de Bruijn et al., 2005b]. However, the conceptual model can be adopted to other specification languages.

3.2.1 Goal Templates and Goal Instances

With respect to requirement 3, the first aspect of the goal model is the explicit distinction goal templates and goal instances. A goal template is a generic and re-usable objective description that is kept in the system; a goal instance describes a concrete client objective that is created by instantiating a goal template with concrete inputs. We consider goal instances as the primary element for end-users to interact with the system, while the creation and management of goal templates mainly resides on the system administration level.

Purpose and Definition

Figure 3.2 illustrates the basic idea for the objective of buying train tickets in Germany. The goal template describes the objective on the schema level. Here, it defines that the origin and destination city must be located in Germany as well as the data type for the travel date. This description is stored in the system. Then, different clients instantiate the goal template with concrete inputs in order to buy tickets for particular trips. The run-time data are not stored permanently in the system, but are only kept as long as the process of solving a goal instance is going on. Below, Listing 3.1 shows the description of the goals in WSML. We can model goal templates as WSMO goals; the listing shows the definition in terms of a WSMO capability with a pre- and a postcondition. Goal instances are not defined in the WSMO framework, and thus not supported in WSML. We define a goal instance as a pair of the corresponding goal template and the set of concrete input values. The listing shows a possible extension to WSML for defining goal instances.

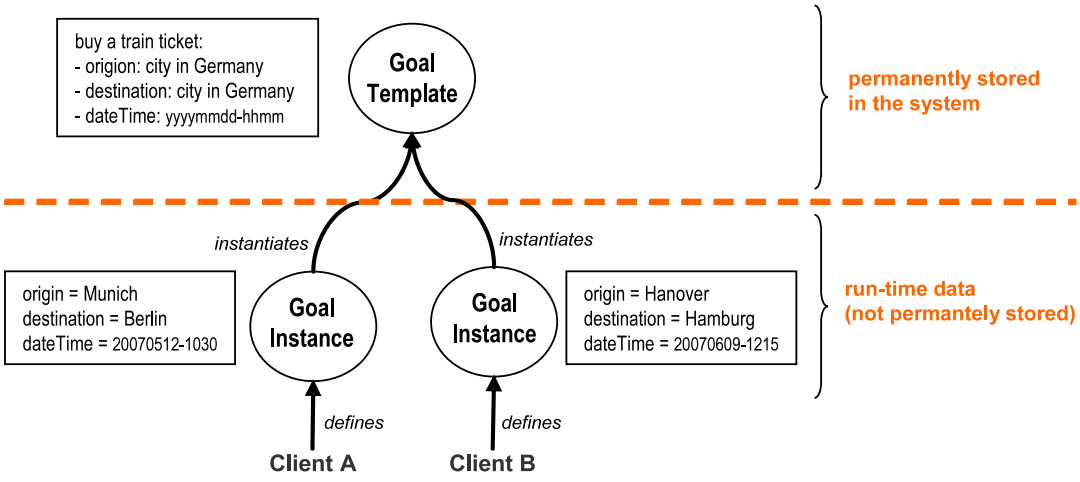


Figure 3.2: Goal Templates and Goal Instances

```
goal trainTicketGermany
 importsOntology _"http://www.wsmo.org/ontologies/trainConnection"
 capability
  sharedVariables {?origin , ?destination , ?dateTime}
  precondition
   definedBy ?origin memberOf city and locatedIn(?origin ,germany) and
             ?destination memberOf city and locatedIn(?destination ,germany) and
             ?dateTime memberOf dateTime.
  postcondition
   definedBy ?ticket [from hasValue ?origin , to hasValue ?destination ,
                      date hasValue ?dateTime] memberOf trainTicket.

goalInstance buyTrainTicketClientA
  correspondingGoalTemplate trainTicketGermany
  inputs ?origin = munich, ?destination = berlin ,
         ?dateTime = 20070512−1030.

goalInstance buyTrainTicketClientB
  correspondingGoalTemplate trainTicketGermany
  inputs ?origin = hanover, ?destination =
         hamburg, ?dateTime = 20070609−1215.
```

Listing 3.1: A Goal Template with Goal Instances in Extended WSML

This conceptual model has two purposes. At first, the detection and usability analysis of Web services can be performed at design time on the level of goal templates, and the captured results of design operations can be used to reduce the necessary reasoning efforts for solving goal instances at runtime as the time critical aspect of SWS technologies (*cf.* requirement 3). Secondly, the model reduces the effort required from clients for the formulation of goal instance to a minimum by merely requesting the provision of inputs for a given goal template. This can be supported by graphical user interfaces in order to reduce the error-proneness of goal definitions (*cf.* requirement 4).

Regarding the description of goals, we consider functional descriptions as the primary description element. Defined in terms of preconditions and effects in a sufficiently expressive formal language, a functional description defines conditions on the initial and the desired final state of the world as the basic description of what shall be achieved. This appears to be expressive enough to describe all kinds of goals that can be expected in the context of Web services (*cf.* requirement 1). In addition, a goal description can define further conditions on suitable Web services, e.g. quality-of-service requirements or usage rights of clients as well as constraints on the workflow that shall be sustained during its resolution (see Section 3.2.3 for details on this). We further consider inputs and outputs as a primary description element of goals, because in the first place Web services are programs that provide computational outputs with respect to the specific inputs that they are invoked with, and the client side element for using Web services should support this. In accordance to the general SWS approach explained in Section 2.1.3, the goal descriptions should be based on ontologies that define the domain terminology and background knowledge.

Goal templates describe the objective on the schema level, meaning that the conditions are defined on the level of concepts in the used domain ontology. A goal instance is defined by a reference to the instantiated goal template, and the set of concrete input values for specifying the particular client objective (see Listing 3.1). We consider the inputs to usually be defined at the time of the goal instance formulation by a client. However, not all necessary inputs must be provided up-front. In particular, inputs whose value is dependent on the interaction with a Web service can only be provided at execution time [Vitvar et al., 2007b]. Nevertheless, independent of when and how the concrete inputs are defined, it is mandatory that the input values are valid for the corresponding goal template, i.e. that they satisfy the conditions and constraints in the goal template description – otherwise, the goal instance is an inconsistent objective description that can not be solved.

Related Work

As mentioned above, existing SWS environments that rely on the WSMO framework already make the distinction of goal templates and goal instances. However, they commonly lack in a precise definition of the relationship between goal templates and goal instances as well as sophisticated techniques for validating goal instantiations:

- The IRS system provides a goal-based broker for Semantic Web services. Goals represent the client objectives, and are described by input and output roles that are defined with respect to domain ontologies; WSMO capabilities are used to define further conditions [Domingue et al., 2008]. Goal instances are created by end-users via a form-based GUI. The instantiation verification is limited to validate the provided input values with respect to the ontology-based conditions defined in the input roles.

- The WSMX system is the reference implementation of WSMO that strictly follows the WSMO specification. It uses WSMO goals as generic objective descriptions in the sense of goal templates. Concrete client objectives are defined by adding an ontology to the goal description that defines the inputs in terms of instances of the used domain ontologies [Zaremba et al., 2006]. However, the system does not perform any explicit instantiation validation.

- The SWF system is an environment for goal-driven, collaborative agents that, among other facilities, use Web services [Stollberg et al., 2005b]. So-called goal schemas are pre-defined in the system, and concrete goal instances are assigned to an agent by either humans or other agents. The inputs in a goal instance are defined as instances of domain ontologies; the instantiation verification is similar to the one in IRS.

We shall discuss this in more detail and present a formal definition of goal templates and goal instances along with instantiation validation in the context Web service discovery and formal functional descriptions in Chapter 4. To summarize, we define the distinction of goal templates and goal instances such that the former are generic and reusable objective descriptions on the schema level that are kept in the system, and the latter denote concrete client objectives for which clients merely need to define the specific inputs. As we shall show in the subsequent chapters of this work, this enables the development of efficient and workable SWS techniques for solving goals.

3.2.2 Automated Web Service Usage

We now turn towards the automated Web service usage that should be supported in goal-based SWS environments. The aim is that the Web services should be executed automatically after having been detected to be suitable for solving a goal. For this, it is necessary that all the inputs needed to invoke and consume the Web service are provided, and that a counterpart of the Web service interface with compatible communication behavior and a technical binding for the actual information interchange is given (*cf.* requirement 2).

In accordance to other approaches – in particular the conceptual model that underlies the IRS system [Domingue et al., 2008] – we consider this as a level of technical detail that clients should not have to deal with. With respect to the overall aim of goal-based SWS approach of lifting the client-system interaction to the level of problem solving, clients should be relieved from dealing with the technical details on the implementation level (see Section 3.1.1). In order to solve a goal, the detected Web services should be executed automatically in a way that no or only minimal involvement of the client is required. Nevertheless, the connections to the implementation level must be defined along with the goal description in order to facilitate the actual invocation of Web services. For this, it is necessary to provide a compatible counterpart for the Web service interfaces which ensures that the result of executing the Web service will solve the goal.

Mediators and Client Interfaces

Figure 3.3 illustrates our conceptual solution for the automated Web service usage for solving goals with minimal intervention by clients. We connect each Web service that is suitable solving a goal template via a mediator that carries a so-called *client interface* which facilitates the invocation and consumption of the Web service for solving the goal. Following the definition of mediators in the WSMO framework [Mocan et al., 2005], a mediator is a logical component that connects a goal template as its source element with a Web service

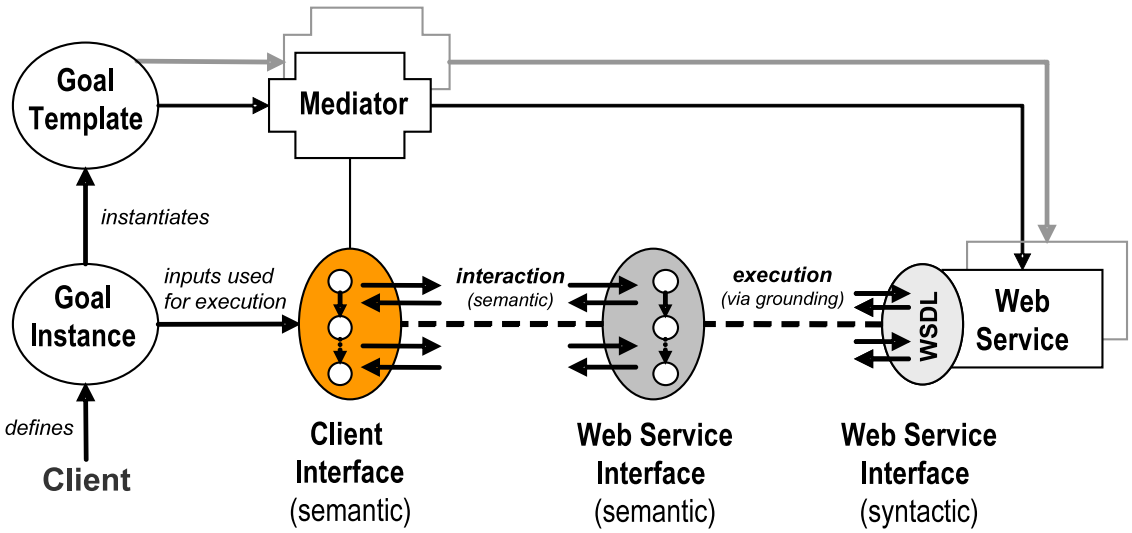


Figure 3.3: Elements for Automated Web Service Usage

as its target element. The client interface defines a semantically described communication behavior for invoking and consuming the target Web services such that the execution result will solve the goal. The interaction is managed on the semantic level between the client interface and the semantically described interface of the Web service, while the technical execution of the Web service is performed via the grounding to WSDL. At runtime, the inputs defined in a goal instance are used to initiate and conduct the interaction with the Web service via the client interface. We shall discuss this below in more detail.

The mediator can further employ mediation facilities for handling heterogeneities on the data and the process level that may occur between the client, the goal description, and the Web service (see Section 2.1.3). Such a mediator is defined for every Web service that is suitable for solving a goal template. Therewith, every Web service that is potentially suitable for solving a goal instance can be executed automatically with the respective inputs, independent of the specific communication behavior supported by the Web service.

The central element for enabling automated Web service usage in this conceptual model is the client interface. This defines a semantically described communication behavior that is compatible to the one of the target Web service, and enables the invocation of this particular Web service in a way that the execution result will solve the goal. The communication behavior defines the order of the information that must be interchanged with the Web service in order to solve the goal; compatibility in this context means that an interaction can take place with respect to the communication behavior defined in the client interface and the one supported by the Web service [Martens, 2003]. The interaction between the

client interface and the Web service can be controlled on the semantic level, while the grounding to WSDL that usually is defined in SWS descriptions of Web services is sufficient to perform the actual information exchange. Thus, the client interface does not need to define a grounding to WSDL or bindings to SOAP messages. The following illustrates the definition and usage of client interfaces within the description model for interaction of clients and Web services defined in the WSMO framework; however, the principle can also be adopted SWS frameworks that use other specification languages.

The WSMO model for describing the interaction behavior of Web services is based on Abstract State Machines [Roman et al., 2007]. A so-called *choreography interface* describes the communication behavior supported by a Web service for consuming its functionality. This is described by a *state signature* which defines the ontology concepts that are used as the content of incoming and outgoing messages, and *guarded transition rules* of the form **if** `condition` **then** `updateFunction` that specify the communication behavior. The functions `add(),delete(),update()` define that information is written, removed, or modified in the common information space wherein the interaction between the client and Web service takes place; so-called `controlStates` allow a partner to control the interaction.

Listing 3.2 shows such a choreography interface description for a German train ticketing Web service offered by the Deutsche Bahn AG (left hand side), and a mediator that connects this Web service with the goal template from Listing 3.1 above along with a client interface (right hand side). For the purpose of illustration, we assume that the Web service supports

```
webService DBticketing                                  |  mediator trainTicketGermany2DBticketing
 interface                                              |   source trainTicketGermany
  choreography                                          |   target DBticketing
   stateSignature                                       |   interface clientInterface
    in                                                  |    stateSignature
     city withGrounding <input-message>                 |     in trainTicket
     dateTime withGrounding <input-message>             |     out {city, dateTime}
    out                                                 |     controlled controlState
     trainTicket withGrounding <output-message>         |    transitionRules
   transitionRules                                      |     if (controlState = init) then
    if (?origin memberOf city and                       |       add(?origin memberOf city and
          locatedIn(?origin,germany) and                |         locatedIn(?origin,germany)),
        ?destination memberOf city and                  |       add(?destination memberOf city and
          locatedIn(?destination,germany)               |         locatedIn(?destination,germany)),
       and ?dateTime memberOf dateTime)                 |       add(?dateTime memberOf dateTime).
    then add(?ticket[from hasValue ?origin,             |     if (?ticket[from hasValue ?origin,
             to hasValue ?destination,                  |         to hasValue ?destination, date hasValue
             date hasValue ?dateTime]                   |         ?dateTime] memberOf trainTicket)
            memberOf trainTicket).                      |      then read(?ticket).
```

Listing 3.2: A Client Interface in the WSMO Choreography Language

a simple request-response behavior. At first, an input message shall provide the necessary inputs (i.e. origin, destination, and travel date), and then an output message provides the train ticket to the requester. The transition rules describe this behavior as explained above. The Web service is suitable for solving the goal template which seeks for train tickets in Germany. Thus, a mediator connects them and defines a client interface for automated invocation and consumption of the Web service. The client interface first writes the requested inputs to the communication space via the `add()` function. If all three inputs are given, the condition of the transition rule in the choreography interface of the Web service is satisfied, and the Web service writes the respective train ticket to the communication space. This result can be obtained via the `read()`-function, and then be presented to the client.

This provides a schema for the successful invocation and consumption of the target Web service in order to solve the goal. At runtime, the concrete inputs defined in a goal instance are used to instantiate the communication behavior defined in the client interface and successively invoke the Web service with concrete data. Given the goal instance `buyTrainTicketClientA` from Listing 3.1 above, the origin city is instantiated with Munich, the destination is Berlin, and the travel date is May 12th, 2007; these data are then used to invoke the `DBticketing` Web service. The interaction between the client interface and the Web service takes place by reading and writing ontology-based information in the communication space. The technical invocation of the Web service is performed via the grounding defined in the Web service choreography interface description, which facilitates the lowering and lifting of the ontology data to XML as well as the necessary connections to WSDL messages [Kopecký et al., 2006]. Thus, the client interface does not need to define a technical grounding because it is only concerned with the behavior for consuming a Web service on the semantic level. The execution engine for client-service interaction on the basis of WSMO choreography descriptions that is provided in the WSMX system already supports this execution model [Haller and Scicluna, 2005; Haselwanter et al., 2006].

So far, we have discussed client interfaces for the automated invocation and consumption of single Web services. If the usage of several Web service is necessary to solve a goal, then Web service composition techniques are applied to determine a suitable execution sequence of the Web services. Mostly, existing techniques for constructing executable compositions for a given goal provide the composition result in form of a new Web service (e.g. [Traverso and Pistore, 2004; Albert et al., 2005; Bertoli et al., 2007]). This means that the constructed aggregation of Web services serves as the implementation of the newly created functionality, and also the communication behavior for invocation and consumption is exposed as a Web service interface. This appears analogously to a single Web service to clients, and for this a client interface for a Web service composition can be constructed as discussed above.

Relation to WSMO and Other Works

The conceptual model of client interfaces that are defined within mediators that connect a goal template with a suitable Web service replaces the notion of requested interfaces in the initial WSMO goal model ([Roman et al., 2006], see Section 3.1.2). As outlined above, the initial solution in WSMO appears to be inappropriate for realizing the idea of goal-driven SWS techniques because it requires the explicit specification of an interface for consuming Web service as a part of the goal description. This contradicts the desired abstraction from technical details. Moreover, defining the requested interface without knowing the Web services to be used is rather an additional restriction on suitable candidates than a means for automated invocation: the compatibility with a Web service interface that is necessary for the successful consumption is left to coincidence, and maybe suitable Web services can not be invoked because required inputs are missing or because presumably occurring mismatches in the communication behaviors can not be resolved. In contrast, the solution presented here supports the automated invocation and consumption of every Web service that is suitable for solving a goal, independent of its specific communication behavior.

Our model maintains the peer-to-peer concept for the interaction of clients and Web services that underlies the WSMO approach: Web services as well as client interfaces describe the supported communication behavior from an individual perspective, and respective algorithms determine whether a successful interaction can take place [Fensel and Bussler, 2002; Stollberg, 2005; Cimpian and Mocan, 2005]. Other approaches that describe the communication behavior between Web services and clients from a global perspective – e.g. the W3C Web Service Choreography Description Language WS-CDL [Kavantzas et al., 2005] – appear to be dispensable in this context. However, there are other application contexts of Web services wherein such global interaction models are desirable, e.g. for the management and maintenance of business cooperations that are realized via Web services.

A similar approach for the automated invocation and consumption of Web services has been developed for the IRS system [Domingue et al., 2005]. Therein, a so-called *choreography* describes how IRS as the broker invokes and communicates with a Web service in order to solve a goal. This is described by SOAP bindings for the input- and output-roles defined in a goal, and uses the WSMO choreography description language for specifying the communication behavior of the IRS to interact with the Web service in terms of guarded transitions. In addition, the management of the interaction as well as the handling execution failures and asynchronous communication is supported on the basis of respective control primitives. This corresponds to the concept of client interfaces introduced above, and the IRS system provides the facilities for executing and controlling the interaction.

Summarizing, we define client interfaces as a compatible counterpart to the semantically described interface of a Web services in order to facilitate the automated invocation and consumption of Web services for solving a goal. Such a client interface is defined for every suitable Web service, and is allocated within a mediator that can employ further mediation facilities in order to resolve possibly occurring heterogeneities. This follows the concept of *client stubs*, a standard element in modern middleware technologies that support the invocation of remote procedures out of a programmatic environment [Birell and Nelson, 1984; Serain and Craiq, 2002]. This is also provided by existing Web service technologies, and there are several tools for automatically generating client stubs from WSDL descriptions (e.g. by the WSDL2Java tool from the Apache Axis project, see `http://ws.apache.org/axis2/`). However, while such client stubs merely support the invocation of specific WSDL operations out of a client application, a client interface as defined here facilitates the automated invocation and consumption of a complete Web service. One can also imagine semi-automated support for the creation of the client interface for a goal and a Web service.

3.2.3 Extensibility

As outlined above, the goal model is extensible. This means that it can be augmented with further constructs and description elements that appear to be desirable in specific application scenarios. In order to remain compatible with the core of the goal model, such extensions should retain the distinction of goal templates and goal instances as well as the concept of client interfaces for automated Web service usage as specified above. The following discusses two examples for such extensions that have been presented in existing works.

Composite Goals

One possible extension of the core goal model is the specification of *composite goals* for describing more complex client objectives as an aggregation of several other goals. This appears to be desirable for describing goals that require a certain workflow to be sustained for their resolution, i.e. a certain order of subgoals for which the actual Web services shall be determined dynamically at runtime. A prominent application area for this are semantically enabled business process management techniques where the distinct process activities are described as goals and the actual Web services are detected at execution time [Weber et al., 2007]. Other examples for goals that consist of multiple aspects and require a certain process to be sustained during their resolution can be found in the area of automated Web service composition (e.g. [Pistore et al., 2004; Albert et al., 2005]).

A composite goal can be described as a collection of subgoals with the control- and data flow among them, i.e. as a decomposition of a more complex objective into smaller parts along with a workflow of how the subgoals shall be solved. For illustration, let us consider the following example setting from the travel scenario. The client objective is to book a flight and a hotel for a holiday trip. In order to get the cheapest offer, the following workflow shall be sustained: at first, a suitable flight shall be found for the earliest departure date and the latest return date. Then, a suitable hotel shall be found where the arrival and departure date fit with the flight dates. If both a flight and a hotel have been found, then first the flight is booked and finally the hotel. Figure 3.4 illustrates this as a composite goal that consists of four sub-goals and defines the necessary control- and data flow.

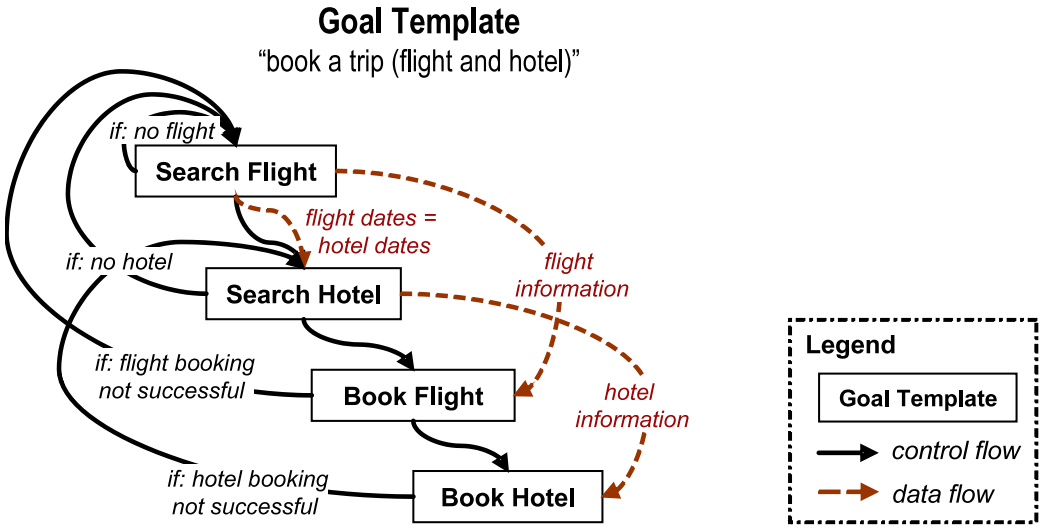


Figure 3.4: Composite Goal Example

This can be described in terms of a WSMO *orchestration* that specifies how a Web service aggregates and interacts with other Web services for achieving its functionality. The specification language for this extends the WSMO choreography language explained above with the `call`-construct that specifies the invocation of other WSMO elements within the update function of a transition rule [Roman and Scicluna, 2007]. This can be used to specify the desired workflow, e.g. by `if (controlState = init) then call flightSearchGoal` in order to solve the flight-search subgoal as the first step for solving the overall goal. We refer to [Stollberg and Norton, 2007] for a more detailed discussion on the modeling of composite goals, including an solution for properly defining the data flow by mediators that specify the necessary variable bindings between the composite goal and the sub-goals.

In our goal model, such composite goals denote a specialization of the basic goals as specified above. The generic structure of the composite goal is defined as a goal template at design time, and at runtime a client creates an instantiation of this by defining concrete input values as explained in above in Section 3.2.1. Then, the actual Web services for the subgoals can be detected dynamically at runtime, and the actual invocation and consumption can be performed via the client interfaces as explained in Section 3.2.2.

Composite goals can be defined explicitly in order to describe a desired workflow that shall be solved by Web services, or they can be created by automated goal decomposition techniques (e.g. [Anton et al., 1994; Giorgini et al., 2003]). Aside from the WSMO orchestration language, also other languages could be used to specify the control- and dataflow, e.g. the composition description language based on UML Activity Diagrams presented in [Albert et al., 2005], the CASHEW language that provides a formal workflow model for Semantic Web services [Norton and Pedrinaci, 2006], or the BPEL4SWS process language that is currently developed in the SUPER project [Haller et al., 2007].

Non-Functional Aspects

Another extension that appears to be relevant for real-world applications is the support for specifying client requirements on non-functional aspects within goal descriptions. While in the above specifications we have mainly focussed on the requested functionality as the basic objective description of *what* shall be achieved, also further aspects might be relevant that are concerned with *how* Web services shall be used to solve a goal. The client requirements on such non-functional aspects can be integrated into the goal model by extending the goal descriptions with respective conditions or policy statements.

We consider non-functional aspects to be concerned with usage conditions on suitable Web services for solving that go beyond the requested functionality as the basic objective description. As the most prominent criteria, this covers quality-of-service aspects (e.g. [Vu et al., 2005; Wang et al., 2006]), negotiation and contracting on the details of a service offer (e.g. [Preist, 2004; Paurobally et al., 2005]), the establishment of trust between interacting entities (e.g. [Olmedilla et al., 2004; Galizia, 2006]), or the handling of service level agreements (SLA) that deal with usage rights and other business-level aspects (e.g. [Ludwig et al., 2003; Oren et al., 2005]). In SWS environments, these criteria are usually evaluated within the selection and ranking component: at first, the functionally suitable candidates are detected by Web service discovery, and then their usability for solving the given goal with respect to non-functional and behavioral aspects is inspected in the subsequent processing steps (see Figure 2.9 in Section 2.2.2).

A common approach for modeling non-functional aspects is to define *policies* that specify conditions along with possible alternatives on how a Web service can be used [Vedamuthu et al., 2007]. In the field of Semantic Web services, this is extended with mostly rule-based conditions that are defined on the basis of a domain ontology [Antoniou et al., 2007]. An approach for modeling this for WSMO goals and Web services is presented in [Toma et al., 2007]. The conditions on a specific non-functional aspect are defined in terms of ontology-based constraints in addition to the capability that describes the requested, respectively the provided functionality. For example, one can specify the cancelation policies for flight tickets that are offered by a ticketing Web service, and analogously define the respective client requirements within the goal description. These policies can then be evaluated in order to provide a ranking of flight offers for a specific client request. A similar approach is presented in [Galizia et al., 2007] where security-related policies are used to select the most adequate candidate out of the functionally suitable Web services.

The goal descriptions in our model can be extended analogously, i.e. with semantically described policies or requirements on non-functional aspects in addition to the requested functionality. The distinction of goal templates and goal instances can be maintained in such extended goal descriptions: the client-side policies can be defined in a generic manner at design time, and then be evaluated at runtime with respect to the concrete input data defined by a client. The automated invocation and consumption of Web services via client interfaces at execution time is orthogonal to this.

3.3 Discussion and Related Work

We conclude the presentation of the goal model with a discussion on its applicability and the relation to previous works on goal-based techniques. For this, the following first summaries the specification of the goal model and discusses its suitability for facilitating the goal-based approach for Semantic Web services. Then, we depict the commonalities and differences to previous works on goal-based techniques that have been developed for automated problem solving in different AI disciplines.

3.3.1 Summary and Applicability

The goal model specified in this work aims at properly facilitating the goal-based approach for SWS technology. It supports the specification of typical client objectives that can be expected in the context of Web services with respect to the problem that shall be solved, and facilitates the automated execution of Web services for solving goals. A central question

is why this appears to be suitable for adequately supporting the goal-based approach. To answer this, the following summarizes the central aspects of the goal model and depicts its benefits in comparison to previously existing approaches.

The overall aim of goal-based SWS technologies is to provide an abstraction layer that allows clients to automatically request and consume the suitable Web services on the level of the problems that can be solved by them while abstracting from the technical details. Based on the examination of existing works, we have identified four central requirements for a sophisticated goal model: the support for expressing all types of goals that can be expected in the context of Web services, the automated execution of Web services to solve a goal, the explicit distinction of goal templates and goal instances in order to develop efficient and workable SWS technologies, and the ease of goal formulation by clients. While WSMO is the only existing SWS framework that promotes a goal-based approach, we have shown that its goal model does not satisfy the requirements in an appropriate manner.

We thus have refined the goal model such that basic objective descriptions for reaching a desired goal state or for performing a function between in- and outputs are defined in terms of preconditions and effects. Clients formulate concrete objectives by instantiating goal templates with concrete inputs, and the Web services for actually solving a given goal instance are invoked and consumed via client interfaces. We have integrated the ideas and conceptual solutions from several works and adopted standard notions and principles for the specification of the goal model: the separation of goal templates as schema-level descriptions handled at design time and goal instances as data-level descriptions that are considered at runtime follows the principle of heuristic classification in order to facilitate the development of efficient and scalable technologies, and the concept of client interfaces adopts the idea of client stubs as a standard element in modern middleware technology.

The core of the goal model adequately satisfies the requirements we have identified for the goal-based approach: it supports the development of efficient SWS techniques that separate design- and runtime operations, the goal formulation for concrete client requests is simplified to the provision of concrete input data, and the concept of client interfaces facilitates the automated usage of Web services with minimal efforts from the client. In addition, the goal model is extensible with further constructs and description elements that appear to be desirable in some application scenarios. As example for this, we have outlined the specification of composite goals for specifying more complex objectives as aggregations of other goals, and the extension of goal description with client-side requirements on non-functional aspects. These extensions support the specification of additional conditions on *how* a goal shall be solved, and can be defined in addition to the requested functionality that specifies *what* shall be achieved as the basic objective description within a goal.

Therewith, the goal model as specified here appears to be suitable for properly supporting the goal-based approach for Semantic Web services, and it overcomes the deficiencies of previously existing solutions. Regarding the usage of goals, we primarily consider them as the element via which clients search and consume Web services, i.e. they replace the hard-wired invocation of Web services that is supported by conventional Web service technologies. The major benefit is that the developers of client applications are relieved from dealing with technical details, and respective SWS techniques can be employed to better support and automate the complete Web service usage process within client applications. However, also end-user applications that utilize Web services via graphical user interfaces can be supported by the goal model.

3.3.2 Relation to Automated Problem Solving in AI

We already discussed specific related works within the explanations of the distinct goal model elements above. Now, we explain the relation to preceding works on goal-based techniques for automated problem solving developed in different AI disciplines.

In accordance to the overall aim of AI research of creating intelligent machines that can solve problems in human-like manner [Turing, 1950], several research efforts have developed techniques for automated problem solving that use goals in a similar way as in the presented goal model for Semantic Web services. Referring to a detailed survey in [Stollberg and Rhomberg, 2006], the most relevant ones in this context are (1) the SOAR system that presents a cognitive architecture on the basis of the theory of human mind and problem solving explained in Section 3.1.1 [Newell, 1990], (2) the belief-desire-intention (BDI) model [Bratman, 1987] that provides the conceptual basis for describing, controlling, and reasoning on the goal-solving behavior of rationale, intelligent agents [Wooldridge and Rao, 1999], (3) AI planning that is concerned with automated construction of plans as a valid sequence of actions for reaching a goal state from an initial state [Allen et al., 1990; Ghallab et al., 2004], and (4) Problem Solving Methods (PSM) that specify generic strategies which can be adapted to several specific problems [Nilsson, 1971; Motta, 1999; Fensel, 2000], and for which the UPML framework provides a comprehensive description model to support logical reasoning and re-use of PSMs in knowledge-based systems [Fensel et al., 2003].

The general purpose of these techniques is the same as in the goal-based SWS approach, i.e. to automatically solve a formally described goal by finding and executing suitable operators or other problem solving facilities. However, there are some differences to our goal model with result from its specialization to Web services. An important one is that in the SOAR system, in classical AI planning (e.g. STRIPS [Fikes and Nilsson, 1971]), and

also in BDI logics [Cohen and Levesque, 1990; Rao and Georgeff, 1991] goals are described by conditions only on the desired final state; the initial state is given by the internal status of the system, respective by the knowledge and beliefs of individual agents. In our model, we describe goals by conditions on both possible start states and desirable goal states. The reason for this is that because of the decoupled and distributed nature of the World Wide Web, we can not know the current state of the world from a global perspective. Describing goals in terms of preconditions and effects ensures that the relevant information on the specific start state and the desired final state of a concrete goal instance is known.

Another difference concerns the types of goals and the resolution techniques for them. The SOAR technology as well as most AI planning techniques are concerned with finding suitable sequence of operators in order to reach the goal state; the applied reasoning techniques resemble those developed for SWS discovery and composition (see Section 2.1.3). In the BDI model, a *desire* denotes the goal that shall be achieved. The agent solves this in an interleaved manner of observation, re-consideration, and interaction with other agents via so-called *intentions* as the intermediate steps. As discussed in Section 3.1, this goal solving technique appears to be dispensable in the context of Web services because they provide passive and non-adaptive computational facilities. Following [Dickinson and Wooldridge, 2005], we thus consider BDI-like goal solving techniques to be suitable for agent-based applications that build on top of Web services and SWS techniques.

A main source of inspiration has been the UPML framework, which also has served as the conceptual predecessor of the WSMO framework [de Bruijn et al., 2005a]. Therein, so-called *tasks* describe the problem that shall be solved in terms of in- and output roles, preconditions, assumptions, and the desired goal state; all description elements are based on ontologies. This corresponds to the notion of functional descriptions as the basic objective descriptions in our goal model (see Section 3.2.1). Furthermore, the description model for complex PSMs by the aggregation of other PSMs has inspired the definition of WSMO orchestration that cab be used to specify composite goal descriptions (see Section 3.2.3).

Chapter 4

Two-Phase Web Service Discovery

This chapter presents an approach for semantically enabled Web service discovery. It supports the goal-based approach for Semantic Web services and facilitates the development of efficient automated discovery techniques by separating two phases: the suitable Web services for goal templates as generic and reusable objective descriptions are discovered at design time, while the actual Web services for goal instances that represent concrete client objectives are discovered at runtime. For this, we specify the necessary semantic matchmaking techniques that work on sufficiently rich functional descriptions in order to warrant a high retrieval accuracy for the discovery task at both design- and runtime.

We briefly recall the overall context from the preceding chapters. Web service discovery is concerned with the detection of suitable Web services for a given request or an application context. This is a central operation in Service-Oriented Architectures (SOA), and research around Semantic Web services (SWS) aims at automating the discovery task. In SWS systems, discovery is usually performed as the first processing step for solving a given goal. It detects suitable Web services out of the available ones with respect to the requested and provided functionalities, and their usability is then further investigated within subsequent processing steps that take other aspects into account (*cf.* Figure 2.9 in Section 2.2.2). A central requirement on automated discovery engines is a high retrieval accuracy, which means that every discovered Web service is suitable (= high precision) and that every suitable Web service can be discovered (= high recall). This appears to be desirable in order to provide a serviceable component for SWS environments, and can most adequately be achieved by employing semantic matchmaking techniques that work on sufficiently rich functional descriptions. The aim of the following elaborations is to specify this for the two-phased discovery approach, and therewith overcome the deficiencies of existing solutions.

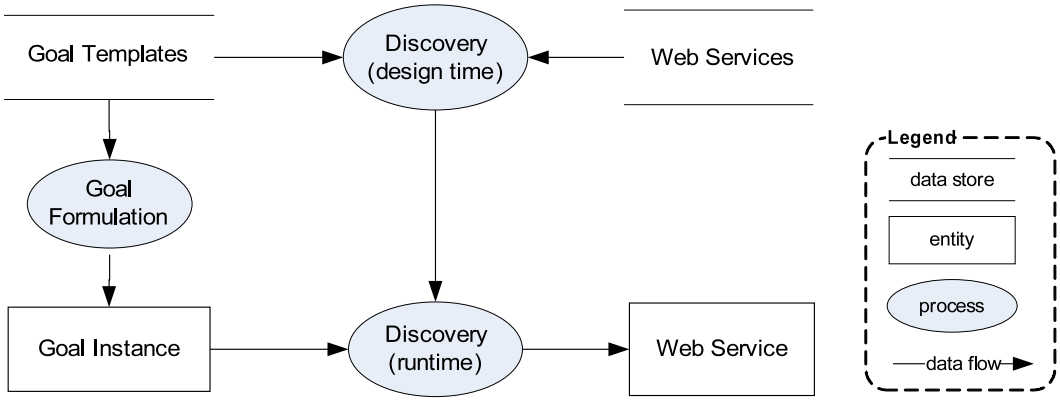


Figure 4.1: Overview of Two-Phase Web Service Discovery

Figure 4.1 provides an overview of the two-phased approach for Web service discovery. Following the goal model presented in Chapter 3, it is based on the explicit distinction of *goal templates* as generic and reusable objective descriptions that are kept in the system, and *goal instances* that represents concrete client objectives by instantiating a goal template with concrete inputs (see Section 3.2.1). This allows us to separate the Web service discovery task into two phases: at design time – that is whenever a goal template or a Web service is added, removed, or changed in the system – the suitable Web services for goal templates are discovered; the actual Web services that are suitable for solving a goal instance are discovered at runtime. The design time discovery results can be used to reduce the necessary matchmaking efforts of runtime discovery, because – due to the instantiation relationship – only those Web services that are suitable for the corresponding goal template are potentially suitable for a goal instance. We therewith can increase the efficiency of runtime Web service discovery as the expectably most frequent operation in SOA applications, and we shall present an extension for the performance optimization in Chapter 5.

In this chapter we formally define the relevant concepts and the necessary semantic matchmaking techniques for this approach. The aim is to ensure a high retrieval accuracy for both design time and runtime discovery, and therewith to overcome the deficiencies of existing approaches as identified in Section 2.2.2. For this, we define functional descriptions with precise formal semantics for accurately describe the overall functionality provided by a Web service and the one requested by a goal. Upon this, we define semantic matchmaking techniques for both design time and runtime discovery for precisely determining the functional usability of Web services, as well as the conditions for the valid instantiation of goal templates during the goal instance formulation process. We use classical first-order logic (FOL) as the specification language, and apply an automated theorem prover for the

matchmaking. Although undecidable in the general case, FOL provides an adequate expressivity while not imposing possibly unnecessary restrictions on the modeling of functional descriptions, and theorem provers provide the necessary reasoning support for the discovery task. Moreover, FOL serves an umbrella for most of the ontology languages developed for the Semantic Web, so that our discovery techniques can be adopted to other specification languages. We shall discuss the limitations of this approach, and also their relevance for the practical applicability of the discovery techniques.

The chapter is organized as follows. Section 4.1 introduces the underlying understanding of goals and Web services and discusses the meaning of a match in the context of Web service discovery. On this basis, Section 4.2 defines the structure and the formal meaning of functional descriptions for goals and Web services, and Section 4.3 specifies the necessary semantic matchmaking techniques for the two-phase Web service discovery. Section 4.4 explains the implementation of the matchmaking techniques within an automated theorem prover for first-order logic, and finally Section 4.5 summarizes the chapter and discusses related work. Throughout this chapter, we use a scenario of finding restaurants in specific cities as well as other examples for illustrating the definitions.

4.1 Foundations

The aim of Web service discovery is to determine the suitability of Web services for solving a goal with respect to the requested and provided functionalities. The common approach for this is to consider the initial usage conditions and the final effects of Web services and match this with the requirements defined in the goal, while abstracting from details on the behavior and technical accessibility. For automating the discovery task, appropriate formal descriptions and semantic matchmaking techniques are required.

Following the standard approach from formal software specification, requested and provided functionalities can most suitably be described in terms of *preconditions* that define the usage conditions and *effects* that describe the results of an execution. Such a functional description is the part of a formal specification that describes *what* is provided by a program, i.e. a "black box" description of the provided functionality that abstracts from how it is achieved. Commonly, preconditions and effects are defined as axioms over a signature that defines the data structure and the relevant background knowledge [Hoare, 1969; Gannon et al., 1994; Fensel, 1995]. The same approach can be used to formally describe the functionality provided by a Web service and the one requested by a goal, and upon this define semantic matchmaking techniques for automated Web service discovery.

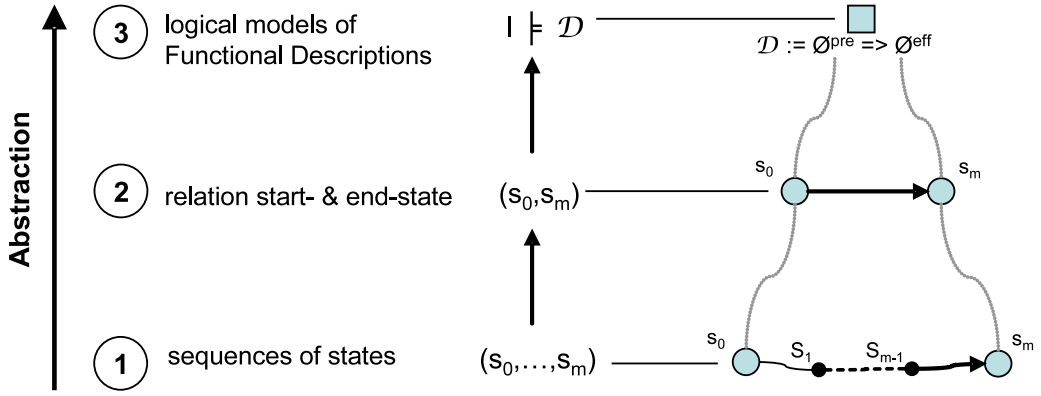


Figure 4.2: Conceptual Abstraction Layers

To ensure a high retrieval accuracy for the discovery task, the functional descriptions of goals and Web services should abstract from unnecessary details but they must properly reflect the actual functionality. Moreover, they should support the employment of reasonably simple reasoning techniques for the necessary matchmaking tasks. For this, we define a conceptual model that distinguishes three levels of abstraction as shown in Figure 4.2.

On the lowest level, we consider the executions of Web services and the solutions of goals as sequences of states $(s_0, \ldots, s_m)$, i.e. from a start-state s_0 via some intermediate states to the end-state s_m. This denotes the actual changes that can be observed in the world when a Web service is executed. On the second level, we take a more abstract view that merely considers the start- and end-states. Here, the abstraction of an observable execution is defined as (s_0, s_m) where s_0 is the start-state and s_m is the end-state, along with the relationship between them. This covers the aspects that appear to be relevant for describing and reasoning on requested and provided functionalities, and the formal relationship between level 3 and level 2 is defined by an abstraction function $(s_0, s_m) := \mathit{abstration}((s_0, \ldots, s_m))$. Finally, on the highest abstraction level we consider the logical models of functional descriptions that formally describe a Web service execution, respectively a solution for a goal. Here, a functional description is a single first-order logic formula that describes the requested and provided functionalities in terms of preconditions and effects, and its formal meaning is defined on the basis of the second abstraction level.

The following elaborates this model in detail. The remainder of this section explains the understanding of Web services and goals in more detail, and discusses the meaning of a match in the context of Web service discovery. Then, we define the structure and formal semantics of functional descriptions in Section 4.2, and on this basis we specify the semantic matchmaking techniques for our two-phased discovery framework in Section 4.3.

4.1.1 Understanding of Web Services and Goals

The following explains the conception of Web services and goals that underlies our approach. We consider Web services on the level of their possible executions that are observable in the world, and formally define the relationship between the lowest level and the second level of the abstraction model outlined above. Then, we explain the meaning of goals that shall be solved by Web services, and define the distinction of goal templates and goal instances.

Web Services

In accordance to the common understanding, we consider a Web service as a computational facility that is invocable over the Internet via an interface (see Section 2.1.1). As an abstraction that is sufficient for our purposes, we define a Web service W as a pair $W = (IF, \iota)$ where IF is the interface of W wherein a finite set of names $(i_1, \ldots, i_n)$ is defined that denotes all inputs that are required for invoking W, and ι is the implementation of W that is executed when W is invoked.

In order to precisely examine the functionality of a Web service W, we consider its actual executions that are observable in the world. For this, we follow the state-based model of the world wherein Web services operate that has been presented in [Keller et al., 2006b]. Therein, a particular execution of W denotes a finite sequence of states $\tau = (s_0, \ldots, s_m)$, i.e. a change of the world from a specific start-state s_0 via some intermediate states to a certain end-state s_m. Such an execution is triggered by the invocation of W with concrete input values in a particular state of the world; we shall refer to this as an input binding β in the following. The resulting execution of W is dependent on the input binding β and the state of the world wherein it has been triggered. It terminates in a specific end-state, after traversing possibly several intermediate states that are dependent on the behavior of W and changes in the world that result from the execution of ι. In consequence, the individual executions of W are unique and have exactly one start-state s_0 and exactly one end-state s_m. Then, we can consider the *overall functionality* provided by W as the set of all its possible executions; we shall denote this by $\{\tau\}_W$ in the following. This can be further differentiated into the distinct sets of possible executions of W for each valid input binding, such that $\{\tau\}_W = \bigcup\{\tau\}_{W(\beta)}$ where $\{\tau\}_{W(\beta)}$ denotes the set of possible executions of W when it is invoked with a particular input binding β.

Figure 4.3 illustrates this understanding of Web services. For illustration, let us consider a Web service for finding the best restaurant in a city. It expects a specific city as the input, and returns the best restaurant in this city as the execution result. Imagine that this Web service is invoked with city A at a state of the world when "Alfredo's Ristorante" is ranked

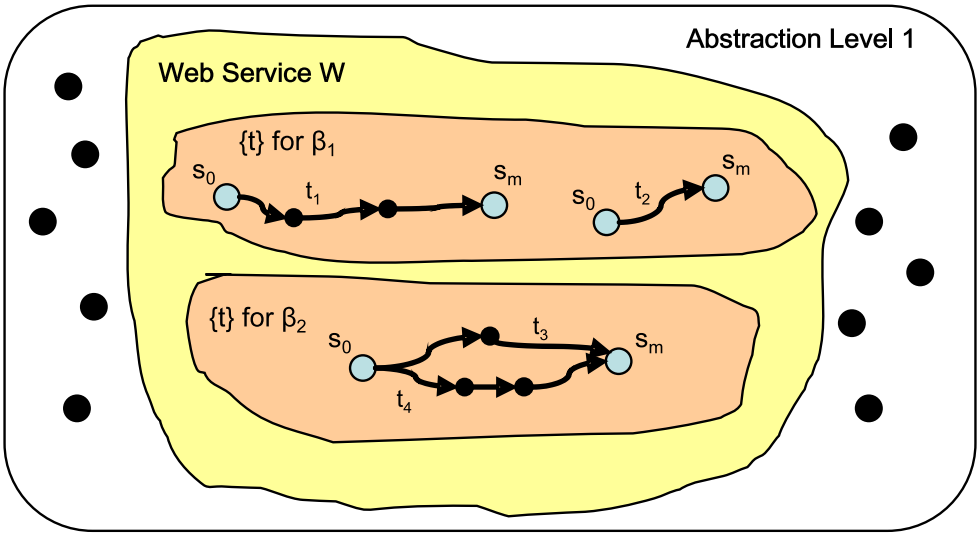


Figure 4.3: Executions of a Web Services

as the best restaurant in city A. This restaurant will then be returned as the execution result. If the Web service is invoked with a different city as the input, naturally a different restaurant will be returned as the execution result. Also, if the Web service is invoked with city A at a different state of the world wherein another restaurant "Aux Michele" is ranked better than "Alfredo's Ristorante", then this will be returned. Analogously, the executions of more complex Web services can be observed as sequences of states whose structure is determined by the actual communication between the client and the Web service.

This conception relates to the lowest level of the abstraction model outlined above. In the context of Web service discovery, we are merely interested in the start- and end-states of the possible Web service executions. For this, we define the tuple $\mathcal{T} = (s_0, s_m)$ as an abstraction that merely considers the start-state s_0 and the end-state s_m of an actual execution $\tau = (s_0, \ldots, s_m)$. Every sequence of states that is observable in the world can be represented by such an abstraction. When there are several executions $\tau_1, \ldots, \tau_n$ of a Web service with the same start- and end-state but with different sequences of intermediate states, these are represented by the same $\mathcal{T} = (s_0, s_m)$ because this level of detail is not differentiated any more. The following defines this formally.

Definition 4.1. *Let $\tau = (s_0, \ldots, s_m)$ be a finite sequence of states that can be observed in the world as an actual execution of a Web service.*

We define $\mathcal{T} = (s_0, s_m)$ as the abstraction of all $\tau = (s_0, s_1, \ldots, s_{m-1}, s_m)$ with the start-state s_0 and the end-state s_m.

This constitutes the second level of our abstraction model. Every execution of W that is observable in the world as a sequence of states $\tau = (s_0, \ldots, s_m)$ can be represented by an abstraction $\mathcal{T} = (s_0, s_m)$ as defined above, and thus we can consider the overall functionality of a Web service W as the set of its abstract executions $\{\mathcal{T}\}_W$.

Definition 4.2. *Let W be a Web Service, and let $\{\tau\}_W$ be the set of all its possible executions that can be observed as a finite sequence of states $\tau = (s_0, \ldots, s_m)$ in the world.*

$\{\mathcal{T}\}_W = \{\mathcal{T} | \mathcal{T} = (s_0, s_m)$ is the abstraction of a $\tau \in \{\tau\}_W\}$ is the smallest set of all abstract executions of W.

This allows us to focus on functional aspects while properly abstracting from details that are irrelevant in the context of Web service discovery. There are two interesting properties that follow from Definition 4.1. The first one is that there is a surjective mapping between the actually observable executions $\tau \in \{\tau\}_W$ and the abstract executions $\mathcal{T} \in \{\mathcal{T}\}_W$: the executions $\tau = (s_0, \ldots, s_m)$ with the same start- and end-state are represented by the same $\mathcal{T} = (s_0, s_m)$, while for all other executions there is a one-to-one correspondence between $\{\tau\}_W$ and $\{\mathcal{T}\}_W$. In consequence, it further holds that every $\mathcal{T} \in \{\mathcal{T}\}_W$ is unique because there can not be two abstract executions with the same start- and end-state.

Goals

We now turn towards the understanding of goals which formally describe the objectives that clients want to achieve by using Web services. As discussed in Chapter 3, a goal describes the desire of a client of getting from the current state of the world into a state wherein the objective is achieved. In the state-based model, we can consider a sequence of states $\tau = (s_0, \ldots, s_m)$ from a start-state s_0 to a desired end-state s_m as a *solution* of a goal. When a goal is solved by a suitable Web service, then the triggered execution is such a solution and it exposes the properties of Web service executions as discussed above.

A central aspect of our approach is the distinction of *goal templates* as generic and reusable objective descriptions that are kept in the system, and *goal instances* that represent specific client objectives by the instantiation of a goal template with concrete inputs. Recalling from the conceptual model presented in Section 3.2.1, a goal template G describes the objective to be achieved on the schema level. The basic objective description is the requested functionality that defines conditions on the initial state and the final desired state of the world. Goal templates can further be described by decompositions into sub-goals as well as by requirements on non-functional aspects (see Section 3.2.3). However, these aspects are not relevant for the Web service discovery task, and thus it appears to be sufficient

to consider the requested functionality on the second level of our abstraction model. Then, in accordance to the above definitions, we can understand the solutions for a goal template G as a set $\{\mathcal{T}\}_G$ where for each abstract solution $\mathcal{T} = (s_0, s_m) \in \{\mathcal{T}\}_G$ the start-state s_0 satisfies the conditions on the initial state and the end-state s_m satisfies the conditions on the desired state of the world. The actual solutions that are observable in the world are the executions of the Web services that are used to solve a goal.

A goal instance $GI(G, \beta)$ describes a concrete client objective. This is defined at runtime by instantiating a goal template G with an input binding β that defines the concrete input values. The input binding β must satisfy the conditions defined in the functional description of the corresponding goal template G; otherwise, the goal instance $GI(G, \beta)$ is an inconsistent goal description that can not be solved. If this is given, then the solutions for $GI(G, \beta)$ are a subset of those for its corresponding goal template G, i.e. it always holds that $\{\mathcal{T}\}_{GI(G,\beta)} \subset \{\mathcal{T}\}_G$. For example, consider a goal template for finding the best restaurant in a German city. A goal instance that defines Berlin (the German capital) as the input binding is a valid instantiation, and its solutions are a subset of the best restaurants in all German cities; a goal instance that defines Innsbruck (located in Austria) as the input binding is not valid because the condition on the inputs is not satisfied. Furthermore, the input binding β defined in a goal instance $GI(G, \beta)$ provides the concrete input values that are used to invoke and consume a suitable Web service for solving the goal instance. This is necessary in order to trigger an execution of the Web service that will provide a solution for the goal instance. The invocation and consumption of the Web service can be conducted via client interfaces as explained in Section 3.2.2.

4.1.2 The Meaning of a Match

After having defined the relevant conceptual entities, we can now turn towards the challenge of Web service discovery. The aim is to find Web services that are suitable for solving a goal with respect to the provided and the requested functionality. As the basis for the following specifications, we define the notion of a *match* as the basic condition under which a Web service is usable to solve a goal under functional aspects.

We consider a match to be given if the Web service can provide an execution that is a solution for the goal. For the functional suitability it is sufficient to consider the abstract executions of Web services and the abstract solutions of goals on the second level of our abstraction model as defined above. Figure 4.4 illustrates this, and below we provide the formal definition of a match for goal templates and for goal instances. These conditions are evaluated by the semantic matchmaking techniques that are defined in Section 4.3.

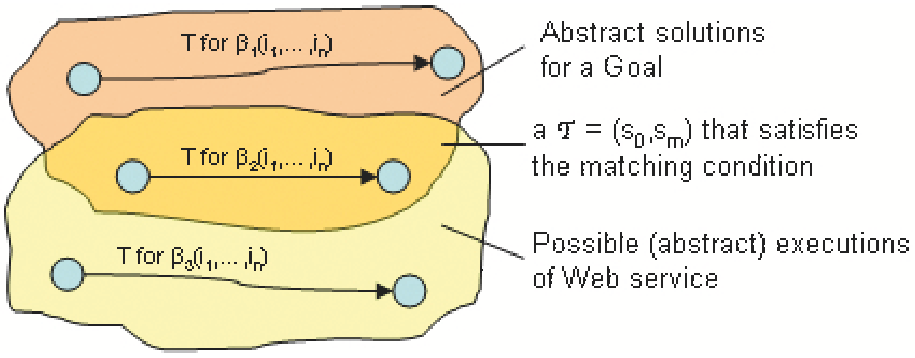


Figure 4.4: The Meaning of a Match

Definition 4.3. *Let W be a Web service, let G be a goal template, and let $GI(G,\beta)$ be a goal instance that instantiates G with an input binding β. Let $\mathcal{T} = (s_0, s_m)$ be the abstraction of sequences of states $\tau = (s_0, \ldots, s_m)$. We define the following sets:*

$\{\mathcal{T}\}_G$	:=	*possible abstract solutions for G*
$\{\mathcal{T}\}_W$	:=	*possible abstract executions of W*
$\{\mathcal{T}\}_{GI(G,\beta)} \subset \{\mathcal{T}\}_G$	:=	*possible abstract solutions for $GI(G,\beta)$ that defines β*
$\{\mathcal{T}\}_{W(\beta)} \subset \{\mathcal{T}\}_W$	:=	*possible abstract executions of W when invoked with β*

We define the basic conditions for the functional suitability of W for solving a goal as:

(i)	$match(G, W)$	:	$\exists \mathcal{T}.\ \mathcal{T} \in (\{\mathcal{T}\}_G \cap \{\mathcal{T}\}_W)$.
(ii)	$match(GI(G,\beta), W)$	:	$\exists \mathcal{T}.\ \mathcal{T} \in (\{\mathcal{T}\}_{GI(G,\beta)} \cap \{\mathcal{T}\}_{W(\beta)})$.

This defines that a Web service W is usable for solving a goal template G under functional aspects if there exists at least one abstract execution of W that is an abstract solution for G (*cf.* clause (i)), and that W is usable for solving a goal instance $GI(G,\beta)$ if there is at least one abstract execution of W that is an abstract solution for $GI(G,\beta)$ when W is invoked with the input binding β that is defined in the goal instance (*cf.* clause (ii)).

Due to the instantiation relationship of a goal instance $GI(G,\beta)$ and its corresponding goal template G it holds that $\{\mathcal{T}\}_{GI(G,\beta)} \subset \{\mathcal{T}\}_G$. Because of this, it also holds that a Web service W that is usable for solving $GI(G,\beta)$ is also usable for the corresponding goal template G: if W can provide an abstract execution $\mathcal{T} = (s_0, s_m)$ such that $\mathcal{T} \in \{\mathcal{T}\}_{GI(G,\beta)}$, then also $\mathcal{T} \in \{\mathcal{T}\}_G$. We can express this as $match(GI(G,\beta), W) \Rightarrow match(G, W)$. As the logical complement, it then also holds that $\neg match(G, W) \Rightarrow \neg match(GI(G,\beta), W)$ [1], i.e. that a Web service that is not usable for a goal template is also not usable for any of its goal

[1] This refers to the following tautology of two terms a, b: $a \Rightarrow b \Leftrightarrow \neg a \vee b \Leftrightarrow \neg b \vee a \Leftrightarrow \neg b \Rightarrow \neg a$.

instances. This constitutes the basis of our two-phase approach for Web service discovery as introduced above: the suitable Web services for goal templates can be determined at design time, and only these need to be inspected to detect the actually suitable Web services for goal instances at runtime (see Figure 4.1). The following defines the formal descriptions for goals and Web services and the necessary semantic matchmaking techniques for this.

4.2 Functional Descriptions

This section formally defines functional descriptions for Web services and goals. In order to provide a sophisticated basis for automated Web service discovery with a high retrieval accuracy, the aim is to define sufficiently rich functional descriptions with accurate formal semantics that allow us to precisely describe the provided and requested functionalities.

For this, the following first identifies the requirements for functional descriptions and examines the state-of-the-art in the field of Semantic Web services. Then, we define the structure and the formal meaning of functional descriptions on the basis of the conceptual model explained above. Finally, we define the representation of a functional description as a single first-order logic formula that allows us to deal with them in terms of classical model-theoretic semantics, which denotes the third level of our abstraction model. We illustrate the definitions within our running example and discuss the limitations of this approach.

4.2.1 Requirements and State of the Art

As outlined above, the concept of functional descriptions is a standard approach in AI technologies for formally describing available computational resources and reasoning on their suitability to solve a given problem. Commonly, these are declarative specifications where *preconditions* specify the usage conditions and *effects* that describe the results of a regular execution. One of the first calculi for this has been presented in [Hoare, 1969] where a so-called Hoare triple $\{P\}\ C\ \{Q\}$ describes a command C as an executable piece of code by a set of preconditions P and a set of effects Q that are defined in predicate logic. The intuitive reading is that whenever P holds before the execution of C, then Q will hold afterwards. This has served as the basis for the definition and usage of formal functional descriptions in several AI techniques, e.g. in AI planning for describing the functionality of available operators [Fikes and Nilsson, 1971] or in formal software specification [Gannon et al., 1994], and several formal specification languages have been developed for this, e.g. algebraic specifications [Bidoit et al., 1991], formal development methods like VDM [Jones, 1990] and Z [Diller, 1994], and in knowledge engineering [Fensel, 1995].

The same approach can be used to formally describe the overall functionality that is provided by a Web service or the one requested by a goal. On the basis of the results from previous works and with respect to the understanding of Web services and goals discussed above, we can identify the following requirements on functional descriptions in order to precisely describe provided and requested functionalities:

1. a sufficiently rich specification language should be used for defining the preconditions and effects, and functional descriptions should be specified on the basis of domain ontologies that define the necessary data structures and the background knowledge in a meaning preserving manner

2. apart from describing the usage conditions (= preconditions) and the possible execution results (= effects), also the computational in- and outputs should be considered as a primary aspect of Web services

3. the relationship between the possible start- and end-states should be defined explicitly, i.e. the formal dependencies between the preconditions and the effects; this is in particular important in order to precisely describe the functionality of Web services whose execution results are dependent on the provided inputs

4. the definition of precise formal semantics for functional descriptions is a pre-requisite for specifying sophisticated semantic matchmaking techniques and for the employment of reasoning techniques for automated Web service discovery.

Analyzing the state-of-the-art in Semantic Web services reveals that most approaches do not satisfy these requirements in a sufficient manner. We here examine the functional description defined in the most prominent frameworks as presented in Section 2.1.3, while we shall discuss other related works below in Section 4.5. In OWL-S [Martin, 2004], the functional description is part of the service profile and is defined in terms of the expected *inputs*, the provided *outputs*, *preconditions* that specify conditions before the Web service can be invoked, and *effects* that hold after a successful execution (short: IOPE). We find the same description model within SWSF [Battle et al., 2005] for describing the functionality that is realized of by process. The descriptions are specified in the OWL ontology language, and domain ontologies define the used terminology and background knowledge. However, the IOPE-elements are defined in a detached manner so that their dependency can not be specified explicitly; also, a formal semantics of functional descriptions is not defined.

In WSMO [Lausen et al., 2005], functional descriptions are called *capabilities* that are defined in terms of preconditions, assumptions, postconditions, and effects. Each element

is specified as a logical statement in the WSML language on the basis of domain ontologies; so-called *shared variables* whose scope is the complete capability are used to define the logical dependencies between the distinct elements. The motivation beyond this model is the distinction of computational aspects from other changes in the world that result from executing Web services [Fensel et al., 2006]. For the former, the *preconditions* constrain the expected inputs and the *postconditions* define the computational results; for the latter, *assumptions* specify further conditions that need to hold in order to invoke the Web service, and the *effects* define the changes in the world that result from a successful execution. The in- and outputs are defined within the interface descriptions of Web services. However, a precise formal semantics of WSMO capabilities has not yet been defined.

While the above frameworks use functional descriptions to describe the overall functionality of a Web service, the WSDL-S approach uses preconditions and effects to formally describe the usage conditions and execution results of individual operations in a WSDL document [Akkiraju et al., 2005]. The conditions are defined by referring to a respective axiom in the domain ontology that is used for annotating the WSDL document; the description of the overall functionality is limited to a keyword-based categorization. The same approach is supported in SAWSDL [Farrell and Lausen, 2007]; however, the annotation of operations is restricted to concepts which limits the expressivity to a keyword-based description. Although these descriptions can be useful to determine the usage conditions and order of the operations for consuming a Web service, we do not consider this approach to be suitable for supporting Web service discovery wherein we are interested in the overall functionalities.

4.2.2 Definition and Semantics

The following defines the functional descriptions that we shall use to precisely describe the overall functionality provided by a Web service as well as the one requested by a goal. The aim is to adequately satisfy the requirements identified above and therewith to overcome the deficiencies of the functional descriptions defined in prominent SWS frameworks.

We commence with defining the structure of functional descriptions. We use first-order logic (FOL, [Smullyan, 1968]) as the specification language, and extend this with additional symbols for specifying the logical dependencies between the description elements. Then, we define the formal meaning of functional descriptions on the second level of our abstraction model that considers the abstract executions of Web services, respectively the abstract solution for goals. Finally, we define the representation of a functional description as a single FOL formula that allows us to consider provided and requested functionalities in terms of logical models and therewith constitutes the third level of our abstraction model.

Elements and Structure

In accordance to the common approach, we define a functional description as a declarative specification that describes the overall functionality provided by a Web service or requested by a goal in terms of *preconditions* that specify conditions on the possible start-states and *effects* that describe the possible end-states. These are defined on the basis of domain ontologies, and, with respect to the requirements identified above, we further define functional descriptions as follows:

- we use classical first-order logic (FOL) to specify preconditions ϕ^{pre} and effects ϕ^{eff}, and define an extended signature $\Sigma^{\star}$ that consists of *static symbols* that are not changed by a Web service execution and *dynamic symbols* that are changed
- we explicitly specify the *inputs* and *outputs* as part of the functional description: the inputs are defined by a set of variables $IN = (?i_1, \ldots, ?i_n)$ which are constrained in the precondition ϕ^{pre}, and the output variables $?o_1, \ldots, ?o_n$ are defined in the effect ϕ^{eff} and explicitly denoted by the predicate `out()`
- we specify the dependency between the start- and the end-states by a set V_{free} of free variables that occur as common variables in both the precondition ϕ^{pre} and the effect ϕ^{eff}; usually, the free variables correspond to the input variables IN, in particular for functionalities whose expected results dependent on the provided inputs.

Definition 4.4. *A Functional Description for a Web service or for a goal is defined as a 5-tuple* $\mathcal{D} = (\Sigma^{\star}, \Omega, IN, \phi^{pre}, \phi^{eff})$ *such that:*

(i) $\Sigma^{\star} = \Sigma_S \cup \Sigma_D \cup \Sigma_D^{pre}$ *is a signature over first-order logic (FOL) that consists of static symbols* Σ_S*, dynamic symbols* Σ_D*, and their pre-variants* Σ_D^{pre} *which are interpreted for an abstract sequence of states* $\mathcal{T} = (s_0, s_m)$ *such that:*
- *for all symbols* $\alpha \in \Sigma_S : \alpha(s_0) = \alpha(s_m)$
- *for all symbols* $\alpha \in \Sigma_D : \alpha(s_0) \neq \alpha(s_m)$
- *for all symbols* $\alpha \in \Sigma_D^{pre} : \alpha_{pre}(s_m) = \alpha(s_0)$

(ii) Ω *defines consistent background knowledge in terms of domain ontologies*

(iii) $IN = (?i_1, \ldots, ?i_n)$ *is the set of input variables; an input binding is a total function* $\beta : (?i_1, \ldots, ?i_n) \rightarrow \mathcal{U}$ *that assigns objects of the universe* $\mathcal{U}$ *to each* $?i \in IN$

(iv) the precondition ϕ^{pre} *is a FOL formula that constrains the possible start-states wherein a set* V_{free} *of free variables occurs with* $IN \subseteq V_{free}$

(v) the effect ϕ^{eff} *is a FOL formula that constrains the possible end-states wherein*
- *the same set* V_{free} *of free variables as in* ϕ^{pre} *occurs with* $IN \subseteq V_{free}$
- *the set of output variables* $?o_1, \ldots, ?o_n$ *are denoted by the predicate* `out()`.

This defines the elements and the structure of functional descriptions in order to precisely describe the overall functionality provided by a Web service as well as the one requested by a goal. While we shall discuss an example below in Section 4.2.3, the following explains the central aspects of the definition in more detail.

The purpose of the extended signature $\Sigma^\star$ defined in clause (i) is to precisely describe the changes on the values of logical symbols that are performed by a Web service execution. For this, we explicitly define dynamic symbols Σ_D whose value is changed by an execution, while the static symbols Σ_S remain unchanged; the meaning of all symbols and the relevant background knowledge is defined in a domain ontology (*cf.* clause (ii)). The pre-variants Σ_D^{pre} of dynamic symbols, denoted by a suffix $_{pre}$, can be used within the effect formulae to explicitly refer to the value of a dynamic symbol in the start-state of an execution. For example, consider a functionality for a bank account withdrawal. The precondition $\phi^{pre} : account(?a) \wedge balance(?a) \geq ?x$ defines the account, its initial balance, and the amount that shall be withdrawn, and the effect $\phi^{eff} : account(?a) \wedge balance(?a) = balance_{pre}(?a) - ?x$ states that the resulting balance is the initial value minus the withdrawn amount. Here, the function symbol $balance()$ is a dynamic symbol. In the effect, we refer to its initial value by the pre-variant $balance_{pre}()$ in order to precisely describe the performed value change. This follows a standard approach for denoting the relationship between the values of logical symbols at different states of the world [Genesereth and Nilsson, 1987].

The clauses (iii) - (v) define our approach for explicitly defining the relationship between the possible start- and end-states. The precondition and the effect define conditions on different states of the world, and thus ϕ^{pre} and ϕ^{eff} each denote a closed FOL formula. To explicate their logical dependency, we define a set $V_{free}(?v_1, \ldots, ?v_m)$ of free variables that occur commonly in both the precondition ϕ^{pre} and the effect ϕ^{eff}. A variable $v \in V_{free}$ is not locally bound to ϕ^{pre} or to ϕ^{eff}, so that under a variable assignment $(v|value)$ the *value* refers to the same object in the universe $\mathcal{U}$ at both the start- and the end-state of a particular execution. We assume that in most cases the free variable that occur in $\mathcal{D}$ are the same as those that define the required inputs, i.e. $V_{free} = IN$. Then, the value assignment for the free variables is defined via an input binding β that is in any case required for defining a goal instance and to actually invoke a Web service. For instance in the above example, the required inputs are the bank account $?a$ and the withdrawal amount $?x$. These two variables denote the data items that that constitute the dependency between the start- and the end-state, and thus they occur as free variables in both the precondition and the effect. In the case that the modeling of a functional description requires free variables that are not required as an input, we define an implicitly universally quantified formula of the form $\forall v_1, \ldots, v_n(\phi^{pre}, \phi^{eff})$ where $v_1, \ldots, v_n \in V_{free}(\phi^{pre}) \cup V_{free}(\phi^{eff}) \setminus IN$.

The final aspect that requires further explanation is the definition of the computational outputs in a functional description. These are defined as as bound variables that are explicitly denoted by the predicate $out(?o_1, \ldots, ?o_n)$ in the effect ϕ^{eff} of a functional description, *cf.* clause (v). These denote the final outputs that are, respectively shall be obtained from a successful Web service execution. A way to precisely specify an output is a statement of the form $\forall ?o.\ out(?o) \Leftrightarrow \phi(?o)$ where the expression $\phi(?o)$ is the part of ϕ^{eff} that constrains the output variable $?o$. The universal quantification along with the logical equivalence states that $?o$ is the output of every possible execution for which $\phi(?o)$ is satisfied. However, the outputs can also be defined by formulae with a different structure.

Formal Semantics

We now define the formal meaning of functional descriptions to enable logic-based reasoning on them. For this, the following defines the conditions under which a functional description $\mathcal{D}$ precisely describes the functionality provided by a Web service W, which we shall denote as $W \models \mathcal{D}$. We here consider the overall functionality of W as the set of its abstract executions $\{\mathcal{T}\}_W$ in accordance to Definition 4.2. The expression $s, \beta \models \phi$ states that the FOL formula ϕ is satisfied at a certain state s under an input binding β that defines a value assignment for each of the input variables defined in $\mathcal{D}$, *cf.* clause (iii) of Definition 4.4.

Definition 4.5. *Let $W = (IF, \iota)$ be a Web service, and let $\{\mathcal{T}\}_W$ be the set of all its possible abstract executions. Let $\mathcal{D} = (\Sigma^\star, \Omega, IN, \phi^{pre}, \phi^{\mathit{eff}})$ be a functional description, and let $\beta : (?i_1, \ldots, ?i_n) \rightarrow \mathcal{U}$ be an input binding for $\mathcal{D}$.*

W provides the functionality described by $\mathcal{D}$, denoted by $W \models \mathcal{D}$, if and only if:

- *(i) there is a bijection $\pi : IN \rightarrow IF$ such that $\mathcal{D}$ defines a corresponding input variable $?i \in IN$ for each input $i \in IF$ that is required by W*
- *(ii) for every input binding β under consideration of the domain ontology Ω and the interpretation of dynamic symbols Σ_D and their pre-variants Σ_D^{pre} it holds for all $\mathcal{T} = (s_0, s_m) \in \{\mathcal{T}\}_W$ that if $s_0, \beta \models \phi^{pre}$ then $s_m, \beta \models \phi^{\mathit{eff}}$.*

This states that a functional description $\mathcal{D}$ precisely describes the functionality of a Web service W if two conditions are satisfied. As the first one, clause (i) requires a one-to-one correspondence of the input variables defined in $\mathcal{D}$ and the inputs required by the Web service. This compatibility is necessary in order to enable the invocation of W via an input binding that is defined on the semantic level. The mapping to the syntactical level is usually defined by the grounding part of a Semantic Web service description (see Section 2.1.3). As the second condition, clause (ii) defines the formal semantics of $\mathcal{D}$ with respect to the

understanding of Web services on the second level of our abstraction model. It states that if under a specific input binding β the start state s_0 of an abstract execution $\mathcal{T} = (s_0, s_m)$ of W satisfies the precondition ϕ^{pre}, then the end-state s_m must satisfy the effect ϕ^{eff}. This must hold for every possible execution of W: if there is a $\mathcal{T} \in \{\mathcal{T}\}_W$ for which this does not hold, then $\mathcal{D}$ does not properly describe the overall functionality provided by W. We refer to this as *implication semantics* which requires that if the precondition is satisfied then also the effect must be satisfied; when the precondition is not satisfied, we can not make any statement about the behavior of W. This corresponds to the formal meaning of functional descriptions defined in earlier works as discussed above in Section 4.2.1.

The structure and formal meaning of the functional descriptions for goals is analogous. For a goal template G, the functional description $\mathcal{D}_G$ formally describes the set of its possible abstract solutions $\{\mathcal{T}\}_G$. In accordance to Definition 4.4, the precondition ϕ^{pre} describes possible start states, and the effect ϕ^{eff} defines conditions on the final desired states wherein the objective is achieved. The set $IN = (?i_1, \ldots, ?i_n)$ defines the required inputs, and the computational outputs that are expected from Web services are explicitly defined in the effect ϕ^{eff}. A goal instance $GI(G, \beta)$ instantiates a goal template G by defining an input binding $\beta : (?i_1, \ldots, ?i_n) \rightarrow \mathcal{U}$ for the input variables IN defined in the functional description $\mathcal{D}_G$. The formal semantics is analog to Definition 4.5: for all solutions of the goal instance $\mathcal{T} = (s_0, s_m) \in \{\mathcal{T}\}_{GI(G,\beta)}$ holds that, under the given input binding β, if $s_0, \beta \models \phi^{pre}$ then $s_m, \beta \models \phi^{\mathit{eff}}$. As discussed above, we require a goal instance $GI(G, \beta)$ to be a valid instantiation of a goal template G; otherwise, $GI(G, \beta)$ is an inconsistent goal description and we can not make any statement about its possible solutions. This is given if both the precondition ϕ^{pre} and the effect ϕ^{eff} defined in $\mathcal{D}_G$ are satisfiable under the input binding defined in $GI(G, \beta)$; we shall denote this by $GI(G, \beta) \models G$ in the following. Then, the solutions $\{\mathcal{T}\}_G$ for a goal template G is the set of the solutions of all possible goal instances $GI(G, \beta)$ where $GI(G, \beta) \models G$ is given.

Representation as a Single Formula

After having defined the structure and the meaning of functional descriptions, we now turn towards the third level of our abstraction model that considers requested and provided functionalities as the logical models of a single FOL formula. For this, the following defines the representation of a functional description as a first-order logic structure that maintains the formal semantics as defined above. The aim is to ease the handling of functional descriptions, and in particular to facilitate the definition of the matchmaking techniques for Web service discovery in terms of classical model-theoretic semantics.

Definition 4.6. *A Functional Description $\mathcal{D}$ of a Web service or a goal is represented by a 4-tuple $\mathcal{D}_{FOL} = (\Sigma^\star, \Omega^\star, IN, \phi^{\mathcal{D}})$ such that:*

(i) $\Sigma^\star = \Sigma_S \cup \Sigma_D \cup \Sigma_D^{pre}$ *is a signature with dynamic symbols and their pre-variants*

(ii) $\Omega^\star = \Omega \cup [\Omega]_{\Sigma_D^{pre}}$ *defines consistent background knowledge in terms of ontologies*

(iii) $IN = (?i_1, \ldots, ?i_n)$ *is the set of input variables*

(iv) $\phi^{\mathcal{D}}$ *is a FOL formula of the form* $[\phi^{pre}]_{\Sigma_D \to \Sigma_D^{pre}} \Rightarrow \phi^{\mathit{eff}}$ *where:*

- ϕ^{pre} *is the precondition with a set* $V_{free} \supseteq IN$ *as the only free variables*
- ϕ^{eff} *is the effect wherein the same set* $V_{free} \supseteq IN$ *of free variables occurs and the output variables* $?o_1, \ldots, ?o_n$ *are denoted by the predicate* `out()`
- $[\phi]_{\Sigma_D \to \Sigma_D^{pre}}$ *is the formula derived from* ϕ *by replacing every dynamic symbol* $\alpha \in \Sigma_D$ *by its corresponding pre-variant* $\alpha_{pre} \in \Sigma_D^{pre}$.

This essentially defines the representation of a functional description as a single FOL formula. For this, we take a functional description $\mathcal{D}$ as defined in Definition 4.4 above, and explicitly define the implication semantics from Definition 4.5 within the FOL formula $\phi^{\mathcal{D}}$ that defines a logical implication between the precondition and the effect formulae, *cf.* clause (iv). To properly handle the explicitly specified value changes of logical symbols within $\phi^{\mathcal{D}}$, the renaming function $[\phi]_{\Sigma_D \to \Sigma_D^{pre}}$ replaces the dynamic symbols that occur in the precondition ϕ^{pre} by their pre-variant. For illustration, let us recall the bank account withdrawal example discussed above. By the renaming function, we obtain $\phi^{\mathcal{D}} = (\mathit{account}(?a) \wedge \mathit{balance}_{pre}(?a) \geq ?x) \Rightarrow (\mathit{account}(?a) \wedge \mathit{balance}(?a) = \mathit{balance}_{pre}(?a) - ?x)$, which is a single FOL formula wherein the initial and the final balance are described by distinct symbols. For this, we need to augment the terminology and background knowledge such that for each dynamic symbol $\alpha \in \Sigma_D$ the meaning of α_{pre} is defined in the extended domain ontology $\Omega^\star$, *cf.* clause (ii). All other elements of $\mathcal{D}_{FOL}$ are the same as in Definition 4.4: the extended signature $\Sigma^\star$ explicitly defines the dynamic symbols for an individual functional description, and the logical dependency of the precondition and the effect is defined by the occurrence of the same free variables in both ϕ^{pre} and ϕ^{eff}, which usually correspond to the required input variables $IN = (?i_1, \ldots, ?i_n)$.

This allows us to deal with functional descriptions on the basis of logical models, i.e. $\Sigma^\star$-interpretations that satisfy $\mathcal{D}_{FOL}$. In particular, there is a bijection between the models of $\mathcal{D}_{FOL}$ and the abstract executions $\{\mathcal{T}\}_W$ of a Web service W that provides the functionality described by $\mathcal{D}$. A $\Sigma^\star$-interpretation $\mathcal{I}$ is a model of $\mathcal{D}_{FOL}$ if $\phi^{\mathcal{D}}$ is satisfied under an input binding β, i.e. if $\mathcal{I}, \beta \models \phi^{\mathcal{D}}$. Because every abstract execution $\mathcal{T} = (s_0, s_m)$ of W is unique in accordance to Definition 4.2, it holds that $W \models \mathcal{D}$ if every model of $\mathcal{D}_{FOL}$ describes exactly one $\mathcal{T} \in \{\mathcal{T}\}_W$. The following defines this formally.

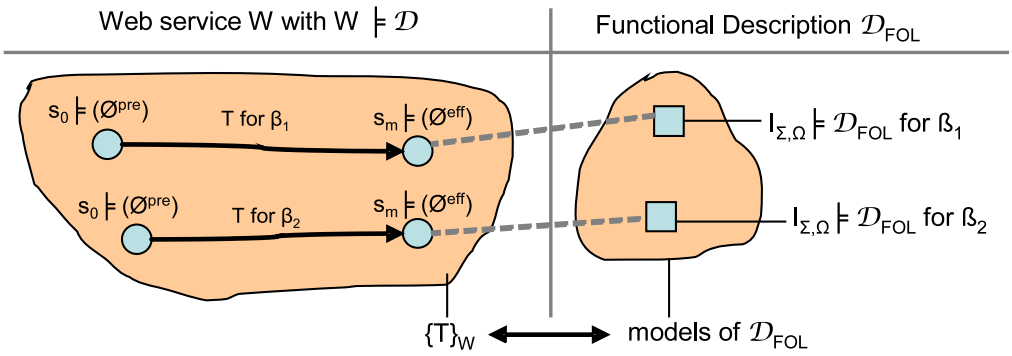


Figure 4.5: The Meaning of a Functional Description

Proposition 4.1. *Let W be a Web service, and let $\{\mathcal{T}\}_W$ be the set of all possible abstract executions of W. Let $\mathcal{D}$ be a functional description that is represented by $\mathcal{D}_{FOL} = (\Sigma^\star, \Omega^\star, IN, \phi^{\mathcal{D}})$, and let $\beta : (?i_1, \ldots, ?i_n) \rightarrow \mathcal{U}$ be an input binding for $\mathcal{D}$.*

W provides the functionality described by $\mathcal{D}$, denoted by $W \models \mathcal{D}$, if and only if:

(i) every $\Sigma^\star$-interpretation $\mathcal{I}$ with $\mathcal{I} \models \Omega^\star$ and $\mathcal{I}, \beta \models \phi^{\mathcal{D}}$ under every input binding β describes a $\mathcal{T} \in \{\mathcal{T}\}_W$, and

(ii) every $\mathcal{T} \in \{\mathcal{T}\}_W$ is described by a $\Sigma^\star$-interpretation $\mathcal{I}$ with $\mathcal{I}, \beta \models \phi^{\mathcal{D}}$ and $\mathcal{I} \models \Omega^\star$ under every input binding β.

Referring to Appendix A.1 for the proof, this states that a Web service W provides the functionality described by $\mathcal{D}$ if each of its possible executions corresponds to exactly one model of $\mathcal{D}_{FOL}$ and vice versa. A state s of the world is constituted by the concrete objects that exist at this point in time. This can be described by an interpretation that assigns concrete objects of the universe $\mathcal{U}$ to the symbols in a logical formula. Analogously, we can define a $\Sigma^\star$-interpretation that describes the objects that exist at both the start- and the end-state of a sequence of states. Following clause (iv) of Definition 4.6, it holds that such a $\Sigma^\star$-interpretation $\mathcal{I}$ satisfies $\phi^{\mathcal{D}}$ under a given input binding β if $\mathcal{I}, \beta \models \phi^{pre}$ and $\mathcal{I}, \beta \models \phi^{eff}$. This describes an abstract sequence of states $\mathcal{T} = (s_0, s_m)$ where $s_0, \beta \models \phi^{pre}$ and $s_m, \beta \models \phi^{eff}$. In accordance to Definition 4.5, this is a possible abstract execution of a Web service W with $W \models \mathcal{D}$. If this holds for every $\mathcal{T} \in \{\mathcal{T}\}_W$, then $\mathcal{D}$ precisely describes the functionality provided by W; if there is a model of $\mathcal{D}_{FOL}$ that does not describe a $\mathcal{T} \in \{\mathcal{T}\}_W$, then $W \models \mathcal{D}$ is not given. Figure 4.5 illustrates this relationship, which holds analogously for functional descriptions that describe the possible solutions of goals.[2]

[2] A stronger definition of $\mathcal{D}_{FOL}$ would be $\phi^{\mathcal{D}}_{+} = [\phi^{pre}]_{\Sigma_D^{pre} \rightarrow \Sigma_D} \wedge \phi^{eff}$ such that only those $\mathcal{T}$ are described for which the precondition is satisfied. However, the implication semantics in Definition 4.6 better reflect the nature of Web services, and it holds that $\phi^{\mathcal{D}}_{+} \models \phi^{\mathcal{D}}$ (see Appendix A.1).

This completes the definition of our 3-leveled abstraction model. On the lowest level we consider the executions of Web services and the solutions of goals as finite sequence of states that are actually observable in the world. On the second level we abstract this so that merely the start- and end-states are considered, and on the highest level the representation of functional descriptions defined above allows us to consider the overall requested and provided functional as the logical models of a single FOL formula. On this basis, we can define the necessary semantic matchmaking for our two-phase discovery framework in terms of classical model-theoretic semantics, which eliminates the need for more complex formalisms that would be required for dealing with dynamic specifications. We therewith follow the approach of the Situation Calculus for dealing with entities in a state-based model of the world in terms of first-order logic as a static knowledge representation language [McCarthy, 1963]. A central difference is that the Situation Calculus uses fluents as non-standard FOL symbols to explicitly defines states, while our functional descriptions merely describe the dependency of start- and end-states on the data level without any notion of states.

4.2.3 Illustrative Example

To illustrate the above definitions, the following exemplifies the specification of functional descriptions for goals and Web services in the best-restaurant-search example. We consider a goal template G for finding the best restaurant in a city that is requested as an input, and a Web service W that provides the best French restaurant in a given city.

Table 4.1 shows the functional descriptions for both G and W. We model $\mathcal{D}_G$ with one *IN*-variable that is constrained to be a *city* in the precondition ϕ^{pre}. The effect ϕ^{eff} defines that a restaurant shall be obtained as the output which is located in the input city and there is no better restaurant in the city. Analogously, $\mathcal{D}_W$ describes the functionality provided by the Web service W. The mere difference occurs in the effect: the output of W is a French restaurant located in the input city such that there does not exist any better French restaurant in the city. For both descriptions, the *best restaurant ontology* defines that restaurants are located in cities and have exactly one distinct type, and it contains axioms to specify that the predicate $better(?r_1, ?r_2)$ denotes a partial order of restaurant rankings; we shall discuss the modeling in more detail in Section 4.4. The upper part of the table shows $\mathcal{D}_G$ and $\mathcal{D}_W$ in accordance to Definition 4.4, and the lower part shows their representation within the FOL structure from Definition 4.6.

As explained above, the dependency between the preconditions and the effects is defined by free variables that occur in both ϕ^{pre} and ϕ^{eff}. Here, the only variable of this type is $?x$, which at the same time is an input variable in both functional descriptions. We also specify

Table 4.1: Examples for Functional Descriptions

Goal "find best restaurant in a city"		**Web Service** "provide best French restaurant in a city"	
Ω:	best restaurant ontology	Ω:	best restaurant ontology
IN:	{?x}	IN:	{?x}
ϕ^{pre}:	$city(?x)$	ϕ^{pre}:	$city(?x)$
ϕ^{eff}:	$\forall ?y.\ out(?y) \Leftrightarrow ($ $restaurant(?y)$ $\wedge in(?y, ?x)$ $\wedge \neg\exists ?z.(restaurant(?z)$ $\wedge\ in(?z, ?x)$ $\wedge\ better(?z, ?y)\ )\ ).$	ϕ^{eff}:	$\forall ?y.\ out(?y) \Leftrightarrow ($ $restaurant(?y)$ $\wedge\ in(?y, ?x)\ \wedge type(?y, french)$ $\wedge \neg\exists ?z.(restaurant(?z)$ $\wedge\ in(?z, ?x)\ \wedge type(?z, french)$ $\wedge\ better(?z, ?y)\ )\ ).$
$\phi^{\mathcal{D}_\mathcal{G}}$:	$city(?x) \Rightarrow ($ $\forall ?y.\ out(?y) \Leftrightarrow ($ $restaurant(?y)$ $\wedge in(?y, ?x)$ $\wedge \neg\exists ?z.(restaurant(?z)$ $\wedge\ in(?z, ?x)$ $\wedge\ better(?z, ?y)\ )\ )\ ).$	$\phi^{\mathcal{D}_W}$:	$city(?x) \Rightarrow ($ $\forall ?y.\ out(?y) \Leftrightarrow ($ $restaurant(?y)$ $\wedge\ in(?y, ?x)\ \wedge type(?y, french)$ $\wedge \neg\exists ?z.(restaurant(?z)$ $\wedge\ in(?z, ?x)\ \wedge type(?z, french)$ $\wedge\ better(?z, ?y)\ )\ )\ ).$

the computational outputs within ϕ^{eff} by a statement of the form $\forall ?o.\ out(?o) \Leftrightarrow \phi(?o)$ as explained above. This example does not contain any dynamic symbols, because neither the goal expects nor the Web services performs value changes of variables, predicates, or function symbols; however, we have illustrated this above within the bank withdrawal example. In accordance to clause (iv) of Definition 4.6, the representation as a single FOL formula $\phi^{\mathcal{D}}$ is defined as a logical implication between ϕ^{pre} and ϕ^{eff}. Both $\phi^{\mathcal{D}}$-formulae shown in the table can be transformed in to the prenex normal form such that all quantifiers are moved to the front. When applying the conventional rules for this on the basis of the de Morgan's laws, then the formal meaning of the formulae will not be changed by such a normalization.

4.2.4 Limitations of the Approach

We conclude the definition of functional description with discussing the limitations of the approach. In particular, we consider the problem of general undecidability that is concerned with the computational complexity, and the Frame Problem that relates to the handling of conditions in the world that are not affected by the execution of a Web service. Both problems arise from the usage of first-order logic (FOL) as the specification language.

As mentioned above, the reason for using FOL is its high expressivity for defining the preconditions and effects of functional descriptions, and also because it serves as an umbrella for several static knowledge representation languages. We therewith follow the approach of first defining the description elements and matchmaking techniques relevant for the discovery task on a general level, while modeling restrictions for functional descriptions to ensure desirable computational properties can be defined later on. We find the same idea within other works for Semantic Web services, in particular in the SWSF framework that uses first-order logic to axiomatize the description model for Web services (FLOWS) and provides a FOL-based specification language (SWSL-FOL) [Battle et al., 2005].

A major drawback of FOL is the general undecidability. Related to the computational complexity, this means that FOL allows the specification problems that can not be decided by any algorithm, i.e. formulae whose validity in a logical theory can not be determined [Börger et al., 1997]. This is downside of the high expressivity of FOL, and mainly results from the support of function symbols and the arbitrary quantification in a formula. A central motivation for the development of other logical languages is to ensure the decidability, which is always a trade-off between the expressivity of a language and its computational complexity. A prominent class of decidable languages are Description Logics (DL), a formal subset of FOL [Baader et al., 2003]. The base description logic $\mathcal{ALC}$ is of polynomial complexity (`PSPACE`-complete, [Horrocks et al., 1998]) but, on the other hand, it has a very limited expressivity. The complexity of more expressive DLs is usually in exponential time, e.g. `ExpTime-complete` for $\mathcal{SHIQ}$ or `NExpTime-complete` for $\mathcal{SHOIN}$ and OWL-DL (see `www.cs.man.ac.uk/~ezolin/dl/` for details on this). While problems of polynomial complexity can usually be solved effectively, this is not given for exponential complexity with respect to the required time and space for the computation [Book, 1974].

However, the general decidability is a rather vague property for judging the suitability of a logical language for a particular application purpose. Instead of merely considering the general solvability of all problems that can possibly be defined by the language, it appears to be more reasonable to deliberate the trade-off between the expressiveness and computational complexity of a language in combination with the relevant reasoning tasks. In our context, this refers to the semantic matchmaking conditions for Web service discovery. We can avoid the undecidability for this by restricting the formulae that can be defined as preconditions and effects of functional descriptions. A promising basis for this is the Bernays–Schönfinkel class, which is a decidable subset of FOL that encompasses formulae without function symbols where all existential quantifiers precede all the universal quantifiers when written in prenex normal form [Bernays and Schönfinkel, 1928]. We shall discuss this in more detail in the context of defining the semantic matchmaking techniques below in Section 4.3.1.

Another general problem of using FOL is to properly handle conditions in the world that are not affected by the execution of a Web service. Referred to as the Frame Problem, the challenge is to properly describe the effects of an execution without the need of explicitly specifying everything that remains unchanged [McCarthy and Hayes, 1969]. Several solutions have been elaborated for defining so-called *frame axioms* that explicitly specify all conditions that are not changed by the execution [Shanahan, 1997]. An approach that appears to be suitable for our framework is presented in [Reiter, 1991]. Therein, so-called *successor state axioms* are defined to state that a condition is true if and only if (1) the execution makes the condition true, or (2) the condition was previously true and the action does not make it false. Following this, we can extend the formal semantics of a functional description such that $W \models \mathcal{D}$ if for all $\mathcal{T} = (s_0, s_m) \in \{\mathcal{T}\}_W$ and for every consistent formula ψ where $\phi^{\mathit{eff}} \not\models \psi$ holds that if $s_0, \beta \models \phi^{pre}, \psi$ then $s_0, \beta \models \mathit{eff}, \psi$.

4.3 Semantic Matchmaking

This section specifies the semantic matchmaking techniques for the two-phase Web service discovery as introduced above (see Figure 4.1). The aim is to provide semantic means that can precisely determine the usability of a Web service with respect to the matching conditions on the goal template and the goal instance level from Definition 4.3 (see Section 4.1.2). For this, we define the matchmaking techniques on the basis of the functional descriptions for goals and Web services as specified above, which allow us to precisely describe requested and provided functionalities on the level of possible executions of Web services and solutions for goals. We commence with semantic matchmaking on the goal template level, and then define the necessary matchmaking techniques for Web service discovery on the goal instance level. Finally, we illustrate the definitions in our running example.

4.3.1 Goal Template Level

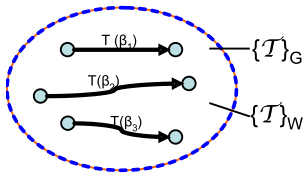
As explained in the introduction of this chapter, Web service discovery on the level of goal templates is performed at design time. The purpose is to detect those Web services out of the available ones that are usable to solve a goal template under functional aspects; this serves as a pre-filter for determining the actual Web services for a goal instance at runtime. As the basic condition under which a Web service W is suitable for solving a goal template G under functional aspects, we have defined a $match(G, W)$ to be given if there is an execution of W that is also a solution for G (*cf.* clause (i) of Definition 4.3). This needs to be evaluated on the basis of the formal functional descriptions of G and W.

For this, we express the functional suitability of Web service W for solving a goal template $\mathcal{G}$ in terms of matching degrees between their functional descriptions $\mathcal{D}_G$ and $\mathcal{D}_W$ with $W \models \mathcal{D}_W$ (*cf.* Definition 4.5). The degrees denote specific relationships between the possible executions of W and possible solutions for $\mathcal{G}$: four degrees – *exact, plugin, subsume, intersect* – denote different situations wherein the basic matching condition is satisfied; the *disjoint* degree denotes that a match is not given. Table 4.2 shows the definitions of the matching degrees on the basis of the formal notions introduced in Section 4.2; we shall discuss this below in more detail. For a better comprehensibility, we also provide a visualization of the matching degrees in terms of Venn diagrams with respect to the abstract executions of W and the abstract solutions for G as defined in Section 4.1.

Table 4.2: Definition of Matching Degrees for Web Service Discovery

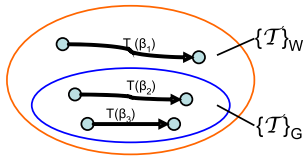
exact$(\mathcal{D}_G, \mathcal{D}_W)$

Definition	$\Omega^\star \models \forall\beta.\ \phi^{\mathcal{D}_G} \Leftrightarrow \phi^{\mathcal{D}_W}$
Meaning	$\mathcal{T} \in \{\mathcal{T}\}_G$ if and only if $\mathcal{T} \in \{\mathcal{T}\}_W$



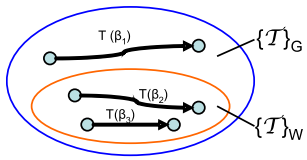
plugin$(\mathcal{D}_G, \mathcal{D}_W)$

Definition	$\Omega^\star \models \forall\beta.\ \phi^{\mathcal{D}_G} \Rightarrow \phi^{\mathcal{D}_W}$
Meaning	if $\mathcal{T} \in \{\mathcal{T}\}_G$ then $\mathcal{T} \in \{\mathcal{T}\}_W$



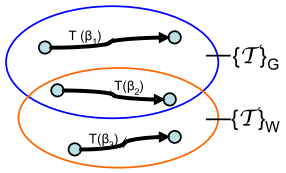
subsume$(\mathcal{D}_G, \mathcal{D}_W)$

Definition	$\Omega^\star \models \forall\beta.\ \phi^{\mathcal{D}_G} \Leftarrow \phi^{\mathcal{D}_W}$
Meaning	if $\mathcal{T} \in \{\mathcal{T}\}_W$ then $\mathcal{T} \in \{\mathcal{T}\}_G$



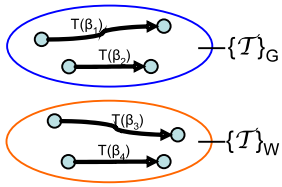
intersect$(\mathcal{D}_G, \mathcal{D}_W)$

Definition	$\exists\beta.\ \bigwedge \Omega^\star \wedge \phi^{\mathcal{D}_G} \wedge \phi^{\mathcal{D}_W}$ is satisfiable
Meaning	$\exists\mathcal{T}.\ \mathcal{T} \in \{\mathcal{T}\}_G \cap \{\mathcal{T}\}_W$



disjoint$(\mathcal{D}_G, \mathcal{D}_W)$

Definition	$\exists\beta.\ \bigwedge \Omega^\star \wedge \phi^{\mathcal{D}_G} \wedge \phi^{\mathcal{D}_W}$ is unsatisfiable
Meaning	$\{\mathcal{T}\}_G \cap \{\mathcal{T}\}_W = \emptyset$



We define the criteria for the matching degrees over the representation of functional descriptions by the FOL structure $\mathcal{D}_{FOL} = (\Sigma^\star, \Omega^\star, IN, \phi^{\mathcal{D}})$ from Definition 4.6. The main merit is that we can define the respective conditions in terms of conventional model-theoretic semantics, and thus can employ reasoning environments for static knowledge representation languages for the technical implementation of a discovery engine. This would be significantly more complicated when defining the matching degrees in a state-based model.

Let us clarify the definition of the matching degrees. The condition for the *exact* degree defines that – with respect to a consistent and homogenous background ontology – under every possible input binding β every model of $\mathcal{D}_{G,FOL}$ as the functional description of the goal template G must also be a model of $\mathcal{D}_{W,FOL}$ that describes the Web service W with $W \models \mathcal{D}_W$, and vice versa. Following Proposition 4.1, this means that the abstract executions of W and the abstract solutions for G must be exactly the same. Analogously, the meaning of the other matching is as follows: the *plugin* degree is given if every solution for G can be provided by W; as the opposite, the *subsume* degree requires that every execution of W is a solution for G. The *intersect* degree requires that there is at least one execution of W that is also a solution of G, and the *disjoint* degree denotes that this is not given.

The matchmaking conditions for the three former degrees are defined as logical entailment relations; the ones for the *intersect* and the *disjoint* degrees are defined as satisfiability checks. We define the conditions via a quantification over input bindings β, which assign concrete values to each of the input variables $IN = (?i_1, \ldots, ?i_n)$ that are defined in a functional description (*cf.* clause (iii) in Definition 4.4). As explained in Section 4.2.2, the input variables are usually a subset of the free variables that define the dependency between the precondition ϕ^{pre} and the effect ϕ^{eff}, and all other free variables are handled by an implicit universal quantification. This ensures that the matchmaking conditions can be evaluated because all free variables are instantiated, respectively bound on the outside.

This definition of the matching degrees on the basis of our functional descriptions for goals and Web services ensures that always the relations between every single $\mathcal{T} \in \{\mathcal{T}\}_W$ and $\mathcal{T} \in \{\mathcal{T}\}_G$ are considered, and not merely subset relations between the possible executions of W and the solutions of G. Moreover, the matchmaking conditions ensure that the signature compatibility between G and W is given, i.e. that $\mathcal{D}_G$ and $\mathcal{D}_W$ define the same or at least semantically equivalent input variables: if this is not given, then there can not be any β under which both $\phi^{\mathcal{D}_G}$ and $\phi^{\mathcal{D}_W}$ are satisfied. Therewith, the semantic matchmaking techniques for Web service discovery on the goal template level as defined here ensure a high retrieval accuracy for the design time discovery task, and also that a Web service W that is functionally be suitable for solving goal template G can be properly invoked in order to solve a goal instance $GI(G, \beta)$ that is a valid instantiation of G.

Similar matching degrees have been defined in several other works on semantically enabled Web service discovery, e.g. [Paolucci et al., 2002; Li and Horrocks, 2003; Benatallah et al., 2005; Keller et al., 2006a]. The central difference is that these works mostly define the matching degrees to denote concept subsumption or logical entailment relations between separate elements of functional descriptions, e.g. between the inputs or the outputs. In contrast, the matching degrees as defined here precisely describe the relationship between the abstract executions of a Web service and the abstract solutions for a goal, which appears to be a much more significant for the purpose of Web service discovery. While we shall discuss related work in more detail in Section 4.5, the following defines possible restrictions on the functional descriptions of goals and Web services which ensure a proper meaning and the general decidability of the semantic matchmaking techniques.

The first restriction is concerned with the suitability of functional descriptions to describe meaningful requested and provided functionalities, which determines the explanatory power of the matchmaking degrees. For this, we require the functional descriptions of Web services and goals to be *consistent*, i.e. that $\mathcal{D}_{FOL}$ is satisfiable which is given if there is at least one $\Sigma^\star$-interpretation $\mathcal{I}$ and an input binding β such that $\mathcal{I}, \beta \models \phi^{\mathcal{D}}$. If this is not given, the matching degrees as defined above might become meaningless because we can not determine any logical model that represents a possible execution of a Web service, respectively a solution for a goal [Keller et al., 2006b]. Under the assumption that all functional descriptions of goals and Web services are consistent, we can always use the highest possible matching degree in order to properly denote the functional suitability of W for solving G on the basis of the following formal relationships between the matching degrees.

Proposition 4.2. *Let $\mathcal{D}_G$ be the consistent functional description of a goal template G, and let $\mathcal{D}_W$ be the consistent functional description of Web service W such that $W \models \mathcal{D}_W$. The following relations hold between the matching degrees of $\mathcal{D}_G$ and $\mathcal{D}_W$:*

(i) $exact(\mathcal{D}_G, \mathcal{D}_W) \Leftrightarrow plugin(\mathcal{D}_G, \mathcal{D}_W) \wedge subsume(\mathcal{D}_G, \mathcal{D}_W)$

(ii) $plugin(\mathcal{D}_G, \mathcal{D}_W) \Rightarrow intersect(\mathcal{D}_G, \mathcal{D}_W)$

(iii) $subsume(\mathcal{D}_G, \mathcal{D}_W) \Rightarrow intersect(\mathcal{D}_G, \mathcal{D}_W)$

(iv) $\neg\, intersect(\mathcal{D}_G, \mathcal{D}_W) \Leftrightarrow disjoint(\mathcal{D}_G, \mathcal{D}_W)$.

The second restriction is concerned with the modeling of functional descriptions in order to warrant the decidability of the semantic matchmaking techniques. As discussed in Section 4.2.4, the problem of undecidability results from using first-order logic (FOL) as the specification language, and we can avoid this by restricting the modeling of functional descriptions such that the conditions for the matchmaking degrees remain in a decidable subset of FOL. We further have identified the Bernays–Schönfinkel class as a suitable FOL

fragment for this, which warrants the decidability if a FOL formula does not contain any function symbols and has a $\exists*\forall*$ quantifier prefix when written in prenex normal form. With respect to this, we can ensure the decidability of the matchmaking conditions by restricting the modeling of functional descriptions and the used domain ontologies as follows.

Proposition 4.3. *Let $\mathcal{D}_G = (\Sigma^\star, \Omega^\star, IN, \phi^{\mathcal{D}_G})$ be the functional description of a goal template G, and let $\mathcal{D}_W = (\Sigma^\star, \Omega^\star, IN, \phi^{\mathcal{D}_W})$ be the functional description of a Web service W with $W \models \mathcal{D}_W$.*

The conditions for all matching degrees are decidable if:

- *(i)* *$\Omega^\star$ as well as $\phi^{pre}, \phi^{eff} \in \mathcal{D}_G, \mathcal{D}_W$ do not contain any function symbols*
- *(ii)* *all formulae $\phi \in \Omega^\star$ have a $\exists * \forall *$ quantifier prefix in prenex normal form*
- *(iii)* *$\phi^{\mathcal{D}_G}$ and $\phi^{\mathcal{D}_W}$ do not have any existential quantifier in prenex normal form.*

Referring to Appendix A.2 for the proof, this ensures the decidability of our semantic matchmaking techniques while maintaining a high expressivity for modeling functional descriptions. Clause (i) requires states that no function symbols occur, i.e. that every symbol with one or more arguments is a predicate that can be evaluated to a truth value and does not define a relationship between individuals, and clauses (ii) and (iii) ensure that the condition for every matching degree is always a FOL formula wherein the existential quantifiers precede the universal quantifiers. If this is given, then the matchmaking conditions are satisfiability problems that remain in the Bernays–Schönfinkel fragment of FOL and thus are decidable. The complexity class for this is `NExpTime-complete` [Lewis, 1980; Papadimitriou, 1994]. Although this means that long processing times might occur for the discovery task, in most real-world scenarios we can work with relatively simple functional descriptions so that the matchmaking can be performed efficiently (see Chapter 6).

4.3.2 Goal Instance Level

We now turn towards the semantic matchmaking techniques for detecting suitable Web services for goal instances at runtime. Recalling from above, a goal instance describes a concrete client objective by instantiating a goal template with specific input values. Goal instances are created at runtime, i.e. whenever a client specifies a concrete objective that shall be solved by using Web services. Our two-phased framework for Web service discovery requires two matchmaking operations for the goal instance level: the first one needs to ensure that a goal instance is a valid instantiation of it corresponding goal template, and the second one is concerned with the detection of functionally suitable Web services for a goal instance (see Figure 4.1). The following defines the formal conditions for both operations.

Goal Instantiation

We commence with the goal instantiation for determining the validity of a goal instance defined by a client. We have defined a goal instance $GI(G, \beta)$ as a pair of the corresponding goal template G and an input binding β that defines concrete values for the input variables defined in $\mathcal{D}_G$ as the functional description of G. As discussed above, we require a goal instance $GI(G, \beta)$ to be a valid instantiation of G, which is given if the input binding β that is defined in $GI(G, \beta)$ satisfies the functional description $\mathcal{D}_G$ of its corresponding goal template at the time of the instantiation. We denote this by $GI(G, \beta) \models G$, and we can only make precise statements about the solutions of $GI(G, \beta)$ if this is given. The following defines the goal instantiation condition formally.

Definition 4.7. *Let G be a goal template, and let $\mathcal{D}_G = (\Sigma^\star, \Omega^\star, IN, \phi^{\mathcal{D}_G})$ be the functional description of G. Let $GI(G, \beta)$ be a goal instance that instantiates G at the time t_0 by defining an input binding $\beta : (?i_1, \ldots, ?i_n) \rightarrow \mathcal{U}$ for $\mathcal{D}_G$.*

$GI(G, \beta)$ is a consistent instantiation of G, denoted by $GI(G, \beta) \models G$, if, at the time t_0 and with respect to $\Omega^\star$, $\phi^{\mathcal{D}_G}$ is satisfiable under the input binding β defined in $GI(G, \beta)$.

Web Service Discovery

We now turn to the discovery of suitable Web services for a goal instance. In our conceptual model, the input binding β defined in a goal instance $GI(G, \beta)$ with $GI(G, \beta) \models G$ is used to invoke a Web service W in order to solve the goal instance: only then an execution of W can denote a solution for $GI(G, \beta)$. With respect to this, we have defined a match on the goal instance level to be given if a solution for a goal instance $GI(G, \beta)$ can be provided by a Web service W when it is invoked with the input binding β defined in the goal instance (*cf.* clause (ii) from Definition 4.3 in Section 4.1.2). The following explains how this can be evaluated on the basis of the formal descriptions.

An input binding $\beta : (?i_1, \ldots, ?i_n) \rightarrow \mathcal{U}$ is a total function that defines a variable assignment over the universe $\mathcal{U}$ for the input variables IN of a functional description $\mathcal{D}$ (*cf.* clause (iii) in Definition 4.4). Given an input binding $\beta = (i_1|value_1, \ldots, i_n|value_n)$, we can instantiate $\mathcal{D}$ by replacing all occurrences of the IN-variables in ϕ^{pre} and ϕ^{eff} by the concrete values defined in β. We shall denote such an instantiated functional description by $[\mathcal{D}]_\beta$ in the following. This does not contain any free variables: the IN-variables are instantiated with the concrete values defined in β, and all other free variables are implicitly universally quantified (see Section 4.2.2). Given a goal instance $GI(G, \beta)$ with $GI(G, \beta) \models G$, we can instantiate $\mathcal{D}_G$ as the functional description of the corresponding goal template G so

that $[\mathcal{D}_G]_\beta$ describes the functionality requested by the goal instance. Analogously, $[\mathcal{D}_W]_\beta$ describes the functionality of a Web service W when it is invoked with β defined in $GI(G, \beta)$. On this basis, we can evaluate the condition for $match(GI(G, \beta), W)$ as follows.

Definition 4.8. *Let $\mathcal{D}_G = (\Sigma^\star, \Omega^\star, IN, \phi^{\mathcal{D}_G})$ be the functional description of a goal template G, and let $GI(G, \beta)$ be a goal instance that defines an input binding $\beta : (?i_1, \ldots, ?i_n) \rightarrow \mathcal{U}$ for $\mathcal{D}_G$ such that $GI(G, \beta) \models G$. Let $\mathcal{D}_W = (\Sigma^\star, \Omega^\star, IN, \phi^{\mathcal{D}_W})$ be the functional description of a Web service W with $W \models \mathcal{D}_W$, and let $[\mathcal{D}]_\beta$ be the β-instantiation of a functional description $\mathcal{D}$ wherein every occurrence of each IN-variable is instantiated with the concrete value assignment defined in the input binding β.*

W is functionally usable to solve $GI(G, \beta)$ if a $\Sigma^\star$-interpretation $\mathcal{I}$ exists such that

$$\mathcal{I} \models \Omega^\star \quad \textit{and} \quad \mathcal{I} \models [\phi^{\mathcal{D}_G}]_\beta \quad \textit{and} \quad \mathcal{I} \models [\phi^{\mathcal{D}_W}]_\beta.$$

This states that a match between a goal instance $GI(G, \beta)$ and a Web service W is given if – with respect to a consistent and homogenous background ontology – there is a common model for $[\mathcal{D}_{G,FOL}]_\beta$ and for $[\mathcal{D}_{W,FOL}]_\beta$, i.e. for the FOL representations of the functional descriptions of the corresponding goal template G and of the Web service W that are both instantiated with the input binding β defined in $GI(G, \beta)$. Formally, this means that the union of the formulae $\Omega^\star \cup \{[\phi^{\mathcal{D}_G}]_\beta, [\phi^{\mathcal{D}_W}]_\beta\}$ must be satisfiable, i.e. that under the input binding β defined in $GI(G, \beta)$ there must exist a $\Sigma^\star$-interpretation $\mathcal{I}$ which satisfies the extended domain knowledge $\Omega^\star$ and is a model for the instantiated FOL representations of the functional descriptions of G and W. In accordance to Proposition 4.1, this describes a $\mathcal{T} = (s_0, s_m)$ that is a solution for $GI(G, \beta)$ and can also be provided by W when invoked with β. We therewith obtain a means for evaluating the basic matching condition on the goal instance level on the basis of the available formal descriptions.

Definition 4.8 also ensures that the input binding β defined in a goal instance provides concrete values for the inputs that are needed to actually invoke the Web service. For this, it must hold that (1) there must be a bijection $\pi_1 : IN_{\mathcal{D}_G} \rightarrow IN_{\mathcal{D}_W}$ such that for every input variable defined in $\mathcal{D}_W$ there is a corresponding input variable in $\mathcal{D}_\mathcal{G}$, and (2) there must be a bijection $\pi_2 : IN_{\mathcal{D}_W} \rightarrow IF$ such that for every input required by the Web service there is a corresponding input variable defined in $\mathcal{D}_W$; this is in any case required for $W \models \mathcal{D}_W$ (*cf.* Definition 4.5 in Section 4.2.2). If this is given, then a Web service W can actually be invoked via a goal instance $GI(G, \beta)$ with $GI(G, \beta) \models G$ because there is a concrete value assignment for every IN-variable of $\mathcal{D}_G$, and these values can subsequently be used to invoke W. If any of the two bijections does not hold, then the matching condition from Definition 4.8 can not be satisfied because there can not be any $\Sigma^\star$-interpretation that is a

model for $[\phi^{\mathcal{D}_G}]_\beta$ and $[\phi^{\mathcal{D}_W}]_\beta$ under the input binding defined in $GI(G, \beta)$. We are aware of that this is not trivial to realize in practice, because it requires a semantic mapping between the input variables of the functional descriptions and the inputs expected by the Web service. Besides, this may require mediation when incompatible ontologies are used in the functional descriptions of the goal and the Web service [Cimpian et al., 2006]. However, in order to invoke a Web service there must be concrete values for all required inputs, and the two bijections define the conditions under which this is given for goal instances.

Another relevant aspect is the handling of incomplete input bindings, i.e. if the β defined in a goal instance $GI(G, \beta)$ misses concrete value assignments for some of the input variables specified for the goal template G. As discussed in Section 3.2.1, this might occur when concrete values for some input variables are not known at the time of the goal instance formulation. For this situation, we can maintain the applicability of the matchmaking techniques specified above by specifying unnamed constants as placeholders for the missing input values. For example, consider a functional description $\mathcal{D}_G$ wherein the input variable $?i_j \in IN$ is constrained to be of type `person` in the precondition, and let us assume that this is the only IN-variable for which an input binding β does not define a value assignment. We then define a constant p with $person(p)$, and add the value assignment $(?i_j|p)$ to the input binding β. Therewith, we can extend the previously incomplete input binding so that it eventually defines concrete values all input variables. The actual value for an initially missing variable assignments can then be determined during the goal resolution process.

Definition 4.8 above defines the basic semantic matchmaking condition for Web service discovery on the goal instance level. However, within our two-phase discovery framework we assume that the results of design time discovery runs on the level of goal templates are known at runtime. This means that we usually know the matching degree under which a Web service W is usable for solving a goal template G as defined above in Table 4.2, and we can use this knowledge to simply the necessary matchmaking effort for determining the suitability of W for solving a goal instance $GI(G, \beta)$ that instantiates G as follows.

Theorem 4.1. *Let G be a goal template that is described by a functional description $\mathcal{D}_G$. Let W be a Web service, and let $\mathcal{D}_W$ be a functional description such that $W \models \mathcal{D}_W$. Let $GI(G, \beta)$ be a goal instance such that $GI(G, \beta) \models G$.*

W is usable for solving $GI(G, \beta)$ if and only if:

(i)	$\texttt{exact}(\mathcal{D}_G, \mathcal{D}_W)$	*or*	
(ii)	$\texttt{plugin}(\mathcal{D}_G, \mathcal{D}_W)$	*or*	
(iii)	$\texttt{subsume}(\mathcal{D}_G, \mathcal{D}_W)$	*and*	$\bigwedge \Omega^\star \wedge [\phi^{\mathcal{D}_W}]_\beta$ *is satisfiable, or*
(iv)	$\texttt{intersect}(\mathcal{D}_G, \mathcal{D}_W)$	*and*	$\bigwedge \Omega^\star \wedge [\phi^{\mathcal{D}_G}]_\beta \wedge [\phi^{\mathcal{D}_W}]_\beta$ *is satisfiable.*

Referring to Appendix A.3 for the formal proof, this defines the functional suitability of Web service W for solving a goal instance $GI(G, \beta)$ under consideration of the matching degree between W and the corresponding goal template G. Under both the *exact* and the *plugin* degree W can be used for solving any goal instance $GI(G, \beta) \models G$ because – due to the goal instantiation condition and the definition of the matching degrees – it holds that $\{\mathcal{T}\}_{GI(G,\beta)} \subset \{\mathcal{T}\}_G \subseteq \{\mathcal{T}\}_W$ and $\mathcal{T} \in \{\mathcal{T}\}_{GI(G,\beta)} \Leftrightarrow \mathcal{T} \in \{\mathcal{T}\}_{W(\beta)}$. Under the *subsume* degree it holds that $\{\mathcal{T}\}_G \supseteq \{\mathcal{T}\}_W$, i.e. every execution of W can solve G but there can be solutions of G that cannot be provided by W. Hence, W is only usable for solving $GI(G, \beta)$ if the input binding β defined in $GI(G, \beta)$ can be used to invoke W. This is given if there is a $\Sigma^\star$-interpretation that is a model for $[\phi^{\mathcal{D}_W}]_\beta$ and the conjunction of the axioms in $\Omega^\star$. Under *intersect* as the weakest degree where $match(G, W)$ is given, the complete matchmaking condition for the goal instance level must hold because there can be solutions for G that can not be provided by W and vice versa. The *disjoint* degree denotes that W is not usable for solving G, and thus is also not usable for any of its instantiations.

We therewith obtain a semantic matchmaking technique for runtime Web service discovery on the goal instance level that requires minimal matchmaking effort. This appears to be desirable in real-world applications where the computational efficiency of a discovery engine becomes a critical success factor. The integrated matchmaking conditions also guarantee the signature compatibility on the expected and the provided inputs, because we use the same matchmaking conditions for the goal template and the goal instance level as defined above. When the matchmaking on the goal template level is decidable in accordance to Proposition 4.3, then also the necessary matchmaking on the goal instance level is decidable because the goal instantiation condition as well as the additional matchmaking conditions for Web service discovery are satisfiability problems that remain in the Bernays–Schönfinkel fragment of FOL and work on the instantiated functional descriptions. Although the general complexity for this is still `NExpTime`, this can usually be evaluated effectively, e.g. by partial instantiation techniques that reduce the problem to a finite sequence of satisfiability problems in propositional logic [Gallo and Rago, 1994].

4.3.3 Illustrative Example

In order to demonstrate the above specifications and to show the retrieval accuracy that is achievable with the semantic matchmaking techniques, the following discusses examples for Web service discovery within the best-restaurant-search scenario. We commence with an example for discovery under the *intersect* matching degree between a goal template and a Web service, and then discuss the matchmaking techniques under other degrees.

Discovery under the Intersect Degree

The following discusses the matchmaking techniques for the goal of finding the best restaurant in a city and a Web service that provides the best French restaurant in a city. This is an example for the *intersect* degree and hence requires the full range of the additional matchmaking on the goal instance level. We shall explain the technical implementation of the semantic matchmaking techniques for this example below in Section 4.4.

We have already defined the functional descriptions for the goal template G and for the Web service W above in Table 4.1 (see Section 4.2.3). Therein, the functional description $\mathcal{D}_G$ of the goal template defines one input variable $?x$ which is constrained to be a *city* in the precondition ϕ^{pre}, and the effect ϕ^{eff} defines that the best restaurant in that city shall be obtained as the output. Analogously, the functional description $\mathcal{D}_W$ of the Web service defines a city to be provided as an input, and the output is the best French restaurant in the city. The *best restaurant ontology* provides the terminology and background knowledge for $\mathcal{D}_G$ and $\mathcal{D}_W$. It defines that restaurants are located in cities and have exactly one distinct type, and the predicate $better(?r_1, ?r_2)$ which states that a restaurant $?r_1$ is ranked to be better than $?r_2$ along with axioms that specify the transitivity of the rankings.

For illustrating the semantic matchmaking techniques, it is sufficient to consider a city A wherein the best restaurant is French and a city B wherein the best restaurant is not French. We then define two input bindings $\beta_1 = (?x|A)$ and $\beta_2 = (?x|B)$, and examine the resulting executions of W as well as the solutions for G and its goal instances under each input binding. Table 4.3 below provides a concise overview of the relevant information for our discussion. The first part shows the specifications for the three best restaurants in city A and in city B as background ontologies $\Omega_1, \Omega_2 \subseteq \Omega$. The second part shows the instantiations of the functional description $\mathcal{D}_G$ under both input bindings, i.e. $[\mathcal{D}_G]_{\beta_1}$ and $[\mathcal{D}_G]_{\beta_2}$ which denote the functional descriptions of goal instances (*cf.* Definition 4.8). Analogously, the third part shows $[\mathcal{D}_W]_{\beta_1}$ and $[\mathcal{D}_W]_{\beta_2}$ as the functional descriptions of W when instantiated with the input bindings. Finally, the fourth part identifies $\Sigma^\star$-interpretations $\mathcal{I}$ which serve as witnesses for demonstrating the matchmaking techniques.

We commence with discussing the matching degree between the goal template G and the Web service W. We observe that under the input binding β_1 there is a $\Sigma^\star$-interpretation $\mathcal{I}_1$ which is consistent with Ω and satisfies both $[\phi^{\mathcal{D}_G}]_{\beta_1}$ as the instantiated functional description of G and $[\phi^{\mathcal{D}_W}]_{\beta_1}$ as the instantiated functional description of W. As explained above, this represents an abstract execution of W that is also a solution for G. Thus, $match(G, W)$ is given, and also the condition for the *intersect* degree is satisfied (*cf.* Table 4.2 in Section 4.3.1). Moreover, we observe from the right-hand column that under

Table 4.3: Relevant Information for Semantic Matchmaking Illustration

City A: $\Omega_1 \subseteq \Omega$	**City B:** $\Omega_2 \subseteq \Omega$
$\Omega_1 = \{city(A)$ $restaurant(r1A)$ $in(r1A, A), type(r1A, french)$ $restaurant(r2A)$ $in(r2A, A), type(r2A, italian)$ $restaurant(r3A)$ $in(r3A, A), type(r3A, french)$ $better(r1A, r2A)$ $better(r2A, r3A)\}$	$\Omega_2 = \{city(B)$ $restaurant(r1B)$ $in(r1B, B), type(r1B, italian)$ $restaurant(r2B)$ $in(r2B, B), type(r2B, french)$ $restaurant(r3B)$ $in(r3B, B), type(r3B, french)$ $better(r1B, r2B)$ $better(r2B, r3B)\}$
$[\phi^{\mathcal{D}_G}]_{\beta_1}$ **with** $\beta_1 = \{?x\|A\}$	$[\phi^{\mathcal{D}_G}]_{\beta_2}$ **with** $\beta_2 = \{?x\|B\}$
$city(A) \Rightarrow ($ $\forall ?y.(out(?y) \Leftrightarrow ($ $restaurant(?y) \wedge in(?y, A)$ $\wedge \neg\exists ?z.(restaurant(?z)$ $\wedge\ in(?z, A)$ $\wedge\ better(?z, ?y)\ )\ )\ )$	$city(B) \Rightarrow ($ $\forall ?y.(out(?y) \Leftrightarrow ($ $restaurant(?y) \wedge in(?y, B)$ $\wedge \neg\exists ?z.(restaurant(?z)$ $\wedge\ in(?z, B)$ $\wedge\ better(?z, ?y)\ )\ )\ )$
$[\phi^{\mathcal{D}_W}]_{\beta_1}$ **with** $\beta_1 = \{?x\|A\}$	$[\phi^{\mathcal{D}_W}]_{\beta_2}$ **with** $\beta_2 = \{?x\|B\}$
$city(A) \Rightarrow ($ $\forall ?y.(out(?y) \Leftrightarrow ($ $restaurant(?y)$ $\wedge\ in(?y, A) \wedge type(?y, french)$ $\wedge \neg\exists ?z.(restaurant(?z)$ $\wedge\ in(?z, A) \wedge type(?z, french)$ $\wedge\ better(?z, ?y)\ )\ )\ )$	$city(B) \Rightarrow ($ $\forall ?y.(out(?y) \Leftrightarrow ($ $restaurant(?y)$ $\wedge\ in(?y, B) \wedge type(?y, french)$ $\wedge \neg\exists ?z.(restaurant(?z)$ $\wedge\ in(?z, B) \wedge type(?z, french)$ $\wedge\ better(?z, ?y)\ )\ )\ )$
$\mathcal{I}_1$ **with** $\mathcal{I}_1 \models \Omega \cup \{[\phi^{\mathcal{D}_G}]_{\beta_1}, [\phi^{\mathcal{D}_W}]_{\beta_1}\}$	$\mathcal{I}_2$ **with** $\mathcal{I}_2 \models \Omega \cup \{[\phi^{\mathcal{D}_G}]_{\beta_2}, [\phi^{\mathcal{D}_W}]_{\beta_2}\}$
$\Omega_1 \cup \Omega_2 \cup \{out(r1A),$ $better(r1A, r3A), better(r1B, r3B)\}$	No such I_2 can exist!

the input binding β_2 there can not exist such a common model for $[\phi^{\mathcal{D}_G}]_{\beta_2}$ and $[\phi^{\mathcal{D}_W}]_{\beta_2}$. The reason is that the goal requests to find the *best* restaurant while the Web service only provides the *best French* restaurant in the input city. Thus, only those $\Sigma^\star$-interpretations can be a model of $[\phi^{\mathcal{D}_G}]_{\beta_2}$ for which the output object is the best restaurant in city B, i.e. if $?y = r1B$. However, the $\Sigma^\star$-interpretations that can be models of $[\phi^{\mathcal{D}_W}]_{\beta_2}$ must define $?y = r2B$ because $r2B$ is the best French restaurant in city B; under all other $\Sigma^\star$-interpretations $[\phi^{\mathcal{D}_W}]_{\beta_2}$ is not satisfiable. This means that there are input bindings under which the possible executions of W and the solutions for G are different. Because of this, neither the condition for the *subsume* nor the one for the *plugin* matching degree is satisfied, and thus also not the one for the *exact* degree. Hence, the matching degree is

$intersect(\mathcal{D}_G, \mathcal{D}_W)$. The matchmaking conditions for this example are decidable in accordance to Proposition 4.3: both functional descriptions as well as the background ontology Ω do no contain function symbols, and we can transform both $\phi^{\mathcal{D}_G}$ and $\phi^{\mathcal{D}_W}$ to a prenex normal form $\forall x, y, z.\ in(x) \wedge \phi(x) \rightarrow (out(y) \leftrightarrow \phi(y) \wedge\ (\phi(z))$ that only has universal quantifier. This is possible in this special case because $\forall y.\ out(y) \leftrightarrow \neg\exists z.\ \phi(z) \Leftrightarrow ((\forall y.\ out(y) \rightarrow \neg\exists z.\ \phi(z)) \wedge (\forall y.\ out(y) \leftarrow \neg\exists u.\ \phi(u))\ ) \Leftrightarrow \forall y.\ out(y) \leftrightarrow \neg\exists z.\ \phi(z)$.

We now turn towards discovery on the goal instance level. Because of the *intersect* matching degree between the goal template G and the Web service W, clause (iv) of Theorem 4.1 must hold for W to be usable for solving a goal instance $GI(G, \beta)$ that instantiates G. This requires that the complete matchmaking condition from Definition 4.8 holds, i.e. that under consideration of the domain ontology Ω there is a $\Sigma^\star$-interpretation $\mathcal{I}$ which is a common model for $[\phi^{\mathcal{D}_G}]_\beta$ and $[\phi^{\mathcal{D}_W}]_\beta$. Let us consider a goal instance $GI_1 = (G, \beta_1)$ that instantiates G with $\beta_1 = (?x|A)$, and another goal instance $GI_2 = (G, \beta_2)$ that instantiates G with $\beta_2 = (?x|B)$. Both goal instances are valid instantiations of the goal template G, because under both input bindings the precondition and effect of $\mathcal{D}_G$ are satisfiable. For GI_1, the matchmaking condition for the goal instance level is satisfied. The witnessing $\Sigma^\star$-interpretation is $\mathcal{I}_1$, which is a model of both $[\phi^{\mathcal{D}_G}]_{\beta_1}$ and $[\phi^{\mathcal{D}_W}]_{\beta_2}$ because by coincidence $r1A$ as the best restaurant is also the best French restaurant in City A. Thus, the Web service W is functionally suitable to solve the goal instance GI_1. For GI_2, this is not given because – as discussed above – a common model for $[\phi^{\mathcal{D}_G}]_{\beta_2}$ and $[\phi^{\mathcal{D}_W}]_{\beta_2}$ can not exist. Also here the additional matchmaking is decidable because they work on the instantiated functional descriptions which are in the Bernays–Schönfinkel fragment of FOL.

Discovery under the Other Matching Degrees

To complete the illustration of the Web service discovery techniques, we briefly discuss examples for the other matching degrees on the goal template level.

Let W_2 be a Web service that provides a search functionality for the best restaurants in Austrian cities. Its functional description is identical to the one of the goal template G above, with the mere difference that the precondition is defined as $\phi^{pre} := city(?x) \wedge in(?x, austria)$. The matching degree between G and W_2 is $subsume(\mathcal{D}_G, \mathcal{D}_{W_2})$, because Austrian cities are a subset of all the cities in the world. Let us consider a goal instance $GI_3 = (G, \beta_3)$ that instantiates G with $\beta_3 = (?x|innsbruck)$. This is a valid instantiation of G and also W_2 can be successfully invoked with β_3, so that W_2 is usable for GI_3. However, for a goal instance $GI_4 = (G, \beta_4)$ that properly instantiates G with $\beta_4 = (?x|berlin)$, i.e. the German capital, we can not make any statement about the usability of W_2 because β_4

can not be used to properly invoke the Web service. This is reflected in the matchmaking condition, which requires that $[\phi^{\mathcal{D}_{W_2}}]_{\beta_4}$ must be satisfiable (*cf.* clause (iii) of Theorem 4.1). However, because β_4 does not satisfy the precondition of W_2 we can not identify a particular $\Sigma^\star$-interpretation that is a model of $[\phi^{\mathcal{D}_{W_2}}]_{\beta_4}$ and thus would represent a solution for GI_4. Because of this, we consider W_2 to be not usable for solving the goal instance GI_4.

For the other degrees, let us consider another goal template G_2 for finding the best restaurant in a city in Tyrol, which is the state of Austria that the city of Innsbruck is located in. Let $\mathcal{D}_{G_2}$ define the precondition $\phi^{pre} := \mathit{city}(?x) \wedge \mathit{in}(?x, \mathit{tyrol})$, and let the background ontology Ω specify that Tyrol is located in Austria. Then, the matching degree between G_2 and the Web service W_2 from above is $\mathit{plugin}(\mathcal{D}_{G_2}, \mathcal{D}_{W_2})$, because every city that is located in Tyrol is also located in Austria. Any goal instance $GI(G_2, \beta)$ of G_2 must define a Tyrolian city in the input binding β – otherwise the goal instantiation condition $GI(G_2, \beta) \models_{\mathcal{A}} G_2$ is not satisfied. If this is given, then also $[\phi^{\mathcal{D}_{W_2}}]_\beta$ is satisfiable under every possible input binding defined in a goal instance $GI(G_2, \beta)$ with $GI(G_2, \beta) \models_{\mathcal{A}} G_2$, so that W_2 is usable for any goal instance of G_2. The same holds for a Web service W_3 that provides a search facility for the best restaurant in a Tyrolian city. Its functional description would be identical to the one of G_2, so that $\mathit{exact}(\mathcal{D}_{G_2}, \mathcal{D}_{W_3})$. In consequence, the possible solutions of W_3 are exactly the same as the possible solutions for G_2, and under every input binding defined in a goal instance $GI(G_2, \beta)$ with $GI(G_2, \beta) \models_{\mathcal{A}} G_2$ the invocation of W_3 will trigger an execution that is a solution of the respective goal instance.

To conclude, the examples show that the matchmaking techniques specified above are suitable for precisely determining the functional usability of Web services for both goal templates and goal instances. We thus consider the aim of specifying semantic matchmaking techniques with a high retrieval accuracy for our two-phase Web service discovery to be achieved. The remainder of this chapter explains the technical implementation of the matchmaking techniques, and positions the approach within related work.

4.4 Implementation in Vampire

This section explains the technical realization of the semantic matchmaking techniques for Web service discovery. We use VAMPIRE as the reasoning engine for the matchmaking, which is a resolution-based automated theorem prover for classical first-order logic with equality [Riazanov and Voronkov, 2002]. This allows us to implement the matchmaking techniques exactly as specified above, i.e. to determine the suitability of Web services for goal templates as well as for goal instances on the basis of our functional descriptions.

Automated theorem proving (ATP) is a subfield of AI research on automated reasoning that is concerned with proving mathematical theories by computer programs. This has been subject to research since more than three decades, and several standard techniques for automated reasoning have been developed in the context of ATP [Loveland, 1978; Gallier, 1986]. ATP is concerned with solving problems of the following kind: given a proof obligation formula ψ and a logical theory that is represented by a finite set of first-order logic axioms $\phi_1, \ldots, \phi_n$, show that ψ is logically implied by the theory. We apply this for Web service discovery such that the functional descriptions of goals and Web service as well as the background ontologies are modeled as the theory in first-order logic, and the distinct conditions for semantic matchmaking are defined as proof obligations.

VAMPIRE is a state-of-the-art automated theorem prover developed at the University of Manchester. It realizes a refutational strategy which performs proof by showing that the set $\phi_1, \ldots, \phi_n, \neg\psi$ is unsatisfiable, i.e. that an interpretation which is a model for the theory and the proof obligation does not exist. It applies standard resolution extended with advanced techniques for handling equality and optimizing the computational efficiency; we refer to [Riazanov, 2003] for a comprehensive presentation. We choose VAMPIRE for implementing the semantic matchmaking techniques, which is one of the most mature and efficient automated theorem provers. It has dominated the international ATP competitions in the last years (in particular the CADE ATP System Competition CASC, see `www.cs.miami.edu/~tptp/CASC/`), and is also positively evaluated in other benchmark tests, e.g. [Nieuwenhuis et al., 2003; Denney et al., 2006]; furthermore, VAMPIRE has been successfully applied for instance-level reasoning with ontologies [Tsarkov et al., 2004].

The following explains the modeling of domain ontologies and of functional descriptions for goals and Web services, and defines the realization of the matchmaking conditions in terms of proof obligations. We use the best-restaurant-search scenario discussed above for illustration; in particular, we provide the complete modeling as well as the matchmaking results for the example discussed in Section 4.3.3.

4.4.1 Modeling in TPTP

The specification language for VAMPIRE is TPTP, a first-order syntax that has been developed to provide a common language for the ATP community [Sutcliffe and Suttner, 1998]. In principle, the mathematical theory as well as proof obligations are specified in terms of FOL formulae. The formulae that constitute the theory are annotated as an *axiom*, and the ones that denote proof obligations are annotated with *conjecture*. The TPTP syntax and semantics for logical expressions is analog to classical FOL. Referring to `www.tptp.org`

for details, the most relevant logical symbols in TPTP for our purposes are the following: (1) lower-case names denote *constants*, and upper-case names denote *variables*; (2) the logical connectives between terms t, t_1, t_2 are represented as follows: t_1 & t_2 is a conjunction, $t_1 \mid t_2$ is a disjunction, $t_1 \Rightarrow t_2$ is an implication, $t_1 \Leftrightarrow t_2$ defines an equivalence, and $\sim t$ is a negation; (3) a universally quantified formula $\forall x : \phi$ is specified as ! $[X] : (\phi)$, and an existentially quantified formula $\exists y : \phi$ is specified as ? $[Y] : (\phi)$.

Ontologies

We commence with the modeling of ontologies in TPTP, which define the terminology and background knowledge used in functional descriptions (*cf.* Definition 4.4). In general, ontologies are defined in terms of *concepts* that denote the entities in the domain which are characterized by *attributes*; *relations* describe the relationships between concepts whereby the subclass- class membership relations define the taxonomic backbone of the ontology. Additional domain knowledge can be specified in terms of *axioms*, and the individuals in the domain are represented as *instances* [de Bruijn and Fensel, 2005].

We represent ontologies as an FOL theory as follows. Concepts are defined by a unique name `concept`, and their associated attributes are defined as binary predicates of the form `attribute(concept,type)` where `attribute` is a unique name and `type` defines the attribute value type by referring to another concept. Relations are defined analog as n-ary predicates, and axioms are defined in terms of FOL formulae. Instances are defined by unary predicates of the form `concept(x)` where `concept` denotes the class membership and `x` is a constant which defines the unique name of the instance; the predicate `attribute(x,y)` defines `y` as the concrete value of an attribute for the instance `x`. We define subclass relations by predicates of the form `isA(concept`$_1$`,concept`$_2$`)` such that `concept`$_1$ is a sub-concept of `concept`$_2$. In order to properly handle the attributes within subclass relations we further define that (1) a sub-concept inherits the attributes of its super-concept so that for each attribute holds $\forall\ C_1, C_2 :\ isA(C_1, C_2) \wedge attribute(C_2, type) \Rightarrow attribute(C_1, type)$, and (2) every instance of the sub-concept is also an instance of the super-concept, i.e. $\forall\ Instance :\ isA(concept_1, concept_2) \wedge concept_1(Instance) \Rightarrow concept_2(Instance)$.

In order to enable reasoning on ontology-based descriptions, we represent concepts by so-called *generic instances*. For a concept with two associated attributes `attr`$_1$`(concept,type`$_1$`)` and `attr`$_2$`(concept,type`$_2$`)`, the generic instance is defined as a formula of the form $\forall\, T_1, T_2 :\ \exists\ C :\ concept(C) \wedge attr_1(C, T_1) \wedge attr_2(C, T_2)$. Essentially, this defines that an unnamed instance of the concept along with unnamed values for each of the associated attributes exists in the knowledge base. Such a generic instance is specified for every concept defined

in an ontology. Given the set of generic instances for all concepts, a theorem prover can find interpretations that satisfy a proof obligation in case that an actual instance for which this is given is not defined in the ontology. This is in particular needed for determining an *intersect* matching degree whose condition is satisfied if there is at least one common model for the functional description of a goal and the one of a Web service.

To clarify the above explanations, Listing 4.1 illustrates the modeling of the *best restaurant ontology* that defines the terminology and background knowledge for our running example. We here show three input-formulae; the complete ontology modeling is provided below in Section 4.4.3. The first formula defines *cityA* as an instance of the concept *city*. The second formula defines the generic instance for representing the concept *restaurant* with an attribute *in* that defines the city wherein the restaurant is located, and another attribute *type* that defines the type of a restaurant (e.g. French or Italian). The third formula defines the axiom that specifies the transitivity of the relation $better(?r_1, ?r_2)$ for describing the ranking of restaurants. This states that if a restaurant $R1$ is ranked better that $R2$ and $R2$ is ranked better that $R3$, then also $R1$ is ranked better that $R3$. Note that in the TPTP representation all these formulae are annotated to be "axioms", which means that they constitute the theory upon which proof obligations shall be shown.

```
input_formula(instanceCityA,axiom,( city(cityA) )).
input_formula( restaurantGenericInstance ,axiom,(
 ! [City,Type] : ( ? [Restaurant] : (
    restaurant(Restaurant)
    & in(Restaurant,City)
    & type(Restaurant,Type) ) ) )).
input_formula( transitivityBetterRelation ,axiom,(
 ! [R1,R2,R3] : (
      restaurant(R1) & restaurant(R2) & restaurant(R3) & better(R1,R2) & better(R2,R3)
      => better(R1,R3) ) )).
```

Listing 4.1: Ontology Representation in TPTP

Functional Descriptions

We now turn towards the specification of functional descriptions for goals and Web services in TPTP. For this, we use the representation $\mathcal{D}_{FOL} = (\Sigma^\star, \Omega^\star, IN, \phi^{\mathcal{D}})$ where the precondition and the effect are defined in a single formula $\phi^{\mathcal{D}} := [\phi^{pre}]_{\Sigma_D \rightarrow \Sigma_D^{pre}} \Rightarrow \phi^{eff}$ (*cf.* Definition 4.6 in Section 4.2.2). To specify this in TPTP, we need to define two minor extensions to the above definition of $\mathcal{D}_{FOL}$: (1) a universal quantification of the input variables $IN = (i_1, \ldots, i_n)$ because VAMPIRE does not support free variables, and (2) the explicit definition of predicates $\texttt{goal}(i_1, \ldots, i_n, o_1, \ldots, o_n) \Leftrightarrow \phi^{\mathcal{D}_G}$, respectively $\texttt{ws}(i_1, \ldots, i_n, o_1, \ldots, o_n) \Leftrightarrow \phi^{\mathcal{D}_W}$

Table 4.4: Example for a Functional Description in TPTP

$\mathcal{D}_{FOL}$ **of a Goal Template** "find best restaurant in a city"	**TPTP Representation** of $\phi^{\mathcal{D}_G}$
Ω: best restaurant ontology IN: {?x} $\phi^{\mathcal{D}_G}$: $city(?x) \Rightarrow ($ $\forall ?y.\ out(?y) \Leftrightarrow ($ $restaurant(?y) \wedge in(?y, ?x)$ $\wedge \neg\exists ?z.(restaurant(?z)$ $\wedge\ in(?z, ?x) \wedge\ better(?z, ?y)\ )\ )\ ).$	**input_formula**(goaltemplate, **axiom**,(! [X,Y] : (goal(X,Y) <=> (city (X) => (out(Y) <=> (restaurant (Y) & in(Y,X) & ~ ? [Z] : (restaurant (Z) & in(Z,X) & better(Z,Y)))))))).

which we can refer to within the proof obligations in order to specify the matchmaking conditions (see below). As the information that usually are of interest, we define the argument of these predicates to be the input variables IN defined in $\mathcal{D}$ and the output variables which are denoted by `out(`$?o_1, \ldots, ?o_n$`)` in the effect. For illustration, Table 4.4 shows the TPTP representation of the goal template for finding the best restaurant in a city that is provided as input which we have explained above in Section 4.2.3.

4.4.2 Matchmaking as Proof Obligations

On this basis, we can now define the proof obligation for the semantic matchmaking techniques. As mentioned above, a proof obligation in TPTP is defined as a FOL formula ψ that is annotated with "conjecture", and the meaning is that ψ shall be proved to hold for the theory which is defined by the formulae annotated with "axiom".

Table 4.5 shows the definition of the proof obligations for the matching degrees defined for Web service discovery on the goal template level in Table 4.2 (see Section 4.3.1). For the *exact* degree it is required that the functional description of the goal template is semantically equivalent to the one of the Web service for all pairs of input and output variables. We define the proof obligation formula on the basis of the predicates `goal(`$i_1, \ldots, i_n, o_1, \ldots, o_n$`)` $\Leftrightarrow \phi^{\mathcal{D}_G}$ and `ws(`$i_1, \ldots, i_n, o_1, \ldots, o_n$`)` $\Leftrightarrow \phi^{\mathcal{D}_W}$ as explained above; we here denote the set of input variables by `IN`, and the set of output variables by `OUT`. The proof obligations for the other matching degrees are defined analogously: the *plugin* and the *subsume* degree require a logical implication between the goal and the Web service description, and the *intersect* degree requires that at least one input-output pair exists under which there is common model for $\phi^{\mathcal{D}_G}$ and $\phi^{\mathcal{D}_W}$. We do not need to define a proof obligation for the *disjoint* degree because it is given if none of the other matching degrees holds (*cf.* Proposition 4.2).

Table 4.5: Proof Obligations for Matching Degrees in TPTP

Matching Degree	Proof Obligation
$\text{exact}(\mathcal{D}_G, \mathcal{D}_W)$	**input_formula**(exact, **conjecture**,(! [IN,OUT] : (goal(IN,OUT) <=> ws(IN,OUT)))).
$\text{plugin}(\mathcal{D}_G, \mathcal{D}_W)$	**input_formula**(plugin , **conjecture**,(! [IN,OUT] : (goal(IN,OUT) => ws(IN,OUT)))).
$\text{subsume}(\mathcal{D}_G, \mathcal{D}_W)$	**input_formula**(subsume, **conjecture**,(! [IN,OUT] : (ws(IN,OUT) => goal(IN,OUT)))).
$\text{intersect}(\mathcal{D}_G, \mathcal{D}_W)$	**input_formula**(intersect , **conjecture**,(? [IN,OUT] : (goal(IN,OUT) & ws(IN,OUT)))).

The proof obligations for the necessary matchmaking operations on the goal instance level are defined analogously. To evaluate $GI(G, \beta) \models G$, i.e. whether a goal instance $GI(G, \beta)$ is a consistent instantiation of its corresponding goal template G, we need to check whether the FOL representation of the functional description $\mathcal{D}_G$ is satisfiable under the input binding β defined in the goal instance (*cf.* Definition 4.7). The proof obligation for this is defined as `?[OUT] : (goal(`β`,OUT))`, which is provable when there exists a model $\phi^{\mathcal{D}_G}$ under the input binding defined in $GI(G, \beta)$.

For Web service discovery on the goal instance level we merely need two additional proof obligations, namely if a Web service W is usable for the corresponding goal template G under the *subsume* or the *intersect* degree (*cf.* Theorem 4.1). These are also defined as satisfiability checks: if $\mathit{subsume}(\mathcal{D}_G, \mathcal{D}_W)$, then the proof obligation `?[OUT] : (ws(`β`,OUT))` checks whether $[\phi^{\mathcal{D}_W}]_\beta$ is satisfiable under the input binding defined in the goal instance $GI(G, \beta)$ with $GI(G, \beta) \models G$. For the additional matchmaking required under the *intersect* degree, the proof obligation `?[OUT] : (goal(`β`,OUT) & ws(`β`,OUT))` checks whether a common model for $[\phi^{\mathcal{D}_G}]_\beta$ and $[\phi^{\mathcal{D}_W}]_\beta$ exists under the input binding defined in $GI(G, \beta)$.

4.4.3 Illustrative Example

To illustrate the above specifications, the following provides the complete modeling for the best-restaurant-search example as discussed in Section 4.3.3 along with the results of the matchmaking techniques obtained from VAMPIRE.

At first, Listing 4.2 shows the *best restaurant ontology* in TPTP. It defines: (1) the generic instances for the concepts *city* and *restaurant*, (2) the better-relation as a transitive, partial order on the ranking of restaurants, and (3) the knowledge base for City A wherein the best restaurant is French and City B wherein the best Restaurant in not French.

```
% generic instance for concept CITY
input_formula( cityGenericInstance ,axiom,(
  ! [Location] : ( ? [City] : ( city(City) & locatedIn(City,Location) ) ) )).
% generic instance for concept RESTAURANT
input_formula( restaurantGenericInstance ,axiom,(
  ! [City,Type] : ( ? [Restaurant] : (
        restaurant(Restaurant)
        & in(Restaurant, City)
        & type(Restaurant,Type) ) ) )).
% better-relation is a partial order
input_formula( betterRelationPartialOrder , axiom,(
 ! [R1,R2] : ( restaurant(R1) & restaurant(R2) & better(R1,R2) => ~better(R2,R1) ) )).
% transitivity of better-relation
input_formula( transitivityBetterRelation , axiom,(
 ! [R1,R2,R3] : (
    restaurant(R1) & restaurant(R2) & restaurant(R3) & better(R1,R2) & better(R2,R3) => better(R1,R3) ) )).
% KNOWLEDGE BASE: instances for 3 best restaurants in city A and in city B
input_formula(cityA, axiom,(city(cityA) )).
input_formula(r1A,axiom,(restaurant(r1A) & in(r1A,cityA) & type(r1A,french) )).
input_formula(r2A, axiom,(restaurant(r2A) & in(r2A,cityA) & type(r2A, italian ) )).
input_formula(r3A, axiom,(restaurant(r3A) & in(r3A,cityA) & type(r3A,french) )).
input_formula(betterCityA1, axiom,(better(r1A,r2A) )).
input_formula(betterCityA2, axiom,(better(r2A,r3A) )).
input_formula(cityB, axiom,(city(cityB) )).
input_formula(r1B, axiom,(restaurant(r1B) & in(r1B,cityB) & type(r1B, italian ) )).
input_formula(r2B, axiom,(restaurant(r2B) & in(r2B,cityB) & type(r2B,french) )).
input_formula(r3B, axiom,(restaurant(r3B) & in(r3B,cityB) & type(r3B,french) )).
input_formula(betterCityB1, axiom,(better(r1B,r2B) )).
input_formula(betterCityB2, axiom,(better(r2B,r3B) )).
```

Listing 4.2: Best Restaurant Ontology

Listing 4.3 shows the definition of the functional descriptions for the goal template of finding the best restaurant in a city that is provided as input, and the Web service with returns the best French restaurant in a given city as defined in Table 4.1 (see Section 4.3.3).

```
% GOAL: find best restaurant in a city
input_formula(goaltemplate, axiom,(
 ! [X,Y] : (
 goal(X,Y) <=> (
  city(X) => (
   out(Y) <=> (
     restaurant(Y) & in(Y,X)
     & ~ ? [Z] : (
        restaurant(Z) & in(Z,X)
        & better(Z,Y) )
) ) ) ) )).
```

```
% WEB SERVICE: give best French restaurant in a city
input_formula(webService, axiom,(
   ! [X,Y] : (
    ws(X,Y) <=> (
       city(X) => (
         out(Y) <=> (
           restaurant(Y) & in(Y,X) & type(Y,french)
           & ~ ? [Z] : (
             restaurant(Z) & in(Z,X) & type(Z,french)
             & better(Z,Y) )
 ) ) ) ) )).
```

Listing 4.3: Functional Descriptions

Finally, Listing 4.4 shows the proof obligations for the necessary matchmaking operations for Web service discovery on both the goal template and the goal instance level. Here, the `include`-statement inserts the domain ontology and the functional descriptions as defined in the above listings as the background theory upon with the proof obligations are evaluated. We first show the proof obligations for the matchmaking degrees as defined above in Table 4.5 along with the results obtained from their evaluation in VAMPIRE: PROVED means that the proof obligation has been proved to hold, and UNPROVABLE means denotes that this is not given. Although these proof obligations are decidable as shown in Section 4.3.3, VAMPIRE requires a remarkably long time for the evaluation. We can enhance this by defining additional axioms that explicate relevant aspects of the domain knowledge. Here, we can significantly improve the proof efficiency by explicitly defining the best and the best French restaurant in City B. At last, we show the goal instantiation test results and the results for Web Service discovery on the goal instance level.

```
include('bestRestaurantOntology.ax').
include('functionaldescriptions.ax').

% discovery for goal template level
% exact(G,W): UNPROVABLE
input_formula(po, conjecture,( ! [I,O] : ( goal(I,O) <=> ws(I,O) ) )).
% plugin(G,W): UNPROVABLE
input_formula(po, conjecture,( ! [I,O] : ( goal(I,O) => ws(I,O) ) )).
% subsume(G,W): UNPROVABLE
input_formula(po, conjecture,( ! [I,O] : ( ws(I,O) => goal(I,O) ) )).
% intersect(G,W): PROVED
input_formula(po, conjecture,( ? [I,O] : ( goal(I,O) & ws(I,O) ) )).
% additional axioms to enhance the proof for intersect(G,W)
input_formula(bestB, axiom,(~ ? [R] : ( restaurant(R) & in(R,cityB) & better(R,r1B) ) )).
input_formula(bestFrenchinCityB, axiom,(~ ? [R] : (
    restaurant(R) & in(R,cityB) & type(R,french) & better(R,r2B) ) )).

% goal instantiation check
% for GI(G,β1) with β1 = (?x|cityA): PROVED
input_formula(po, conjecture,( ? [O] : ( goal(cityA,O) ) )).
% for GI(G,β2) with β2 = (?x|cityB): PROVED
input_formula(po, conjecture,( ? [O] : ( goal(cityB,O) ) )).

% discovery on the goal instance level under intersect(G,W)
% for GI(G,β1): PROVED
input_formula(po, conjecture,( ? [O] : ( goal(cityA,O) & ws(cityA,O) ) )).
% for GI(G,β2): UNPROVABLE
input_formula(po, conjecture,( ? [O] : ( goal(cityB,O) & ws(cityB,O) ) )).
```

Listing 4.4: Proof Obligations and Results

To conclude, we have defined the modeling of ontologies and functional descriptions of goals and Web services in terms of FOL theories, as well as the definition of the matchmaking conditions relevant for Web service discovery in terms of proof obligations which can be evaluated by automated theorem provers. This allows us to realize the semantically enabled discovery techniques exactly as specified in the preceding sections, and therewith to provide a technical infrastructure for automated discovery in our two-phase framework which ensures a high retrieval accuracy for the discovery task at both design and runtime.

While we here have illustrated the modeling in TPTP for the best-restaurant-search example, the general modeling structure can be adopted to other scenarios. We however need to allude to the limitations of this approach for realizing a general purpose matchmaker. At first, standard TPTP does not support arithmetic operations on natural numbers, which means that we can not express conditions like $price < 100$. There are extensions for TPTP that define generic theories for dealing with natural numbers, e.g. [Schulz and Sutcliffe, 2005; Prevosto and Waldmann, 2006]. However, we usually expect functional descriptions to deal with concepts rather than with concrete numbers so that conditions of the mentioned kind are expressed in terms of price categories.

Secondly, there might occur inadequately long processing times for the discovery task. This results from the high computational complexity as the downside of the high expressiveness and retrieval accuracy. In our approach, the complexity for the matchmaking is `NExpTime-complete` if the proof obligations are decidable in accordance to Proposition 4.3. However, OWL-DL has the same complexity, so that the same efficiency problems might occur when using this as the specification language instead of FOL. As illustrated in the above example, one can increase the processing efficiency of an automated theorem prover by adding further axioms that explicate relevant aspects of the domain ontology. Although not specifying new knowledge, such additional axioms bridge the gap between the generic domain knowledge defined in the background ontology and the specific aspects which constitute the relevant relationship of the functional descriptions of goals and Web services. There are semi-automated techniques for identifying such additional axioms (e.g. [Fensel and Schönegge, 1998; de Nivelle and Piskac, 2005]), which we can consider as complementary techniques for realizing semantic matchmaking with automated theorem provers.

Finally, it is to remark that within our test scenarios VAMPIRE exposes a significantly better performance than other ATP engines that work on TPTP. For this, we have compared the time required for evaluating a proof obligation with other automated theorem provers (namely Otter, SPASS, and Waldmeister). While VAMPIRE in average requires between 0.1 and 1 second, the other engines require significantly more time or do not terminate at all for the same proof obligations.

4.5 Summary and Related Work

This chapter has presented the formal specification of the semantic matchmaking techniques for a two-phase Web service discovery approach on the basis of rich functional descriptions for goals and Web services. The following first summarizes the central aspects of our approach, and then positions it within related works for the different aspects.

Summary

Our Web service discovery framework follows the goal-based approach for Semantic Web services. A goal formally describes the objective that a client wants to achieve when using a Web service while abstracting from technical details. The purpose of Web service discovery is to determine those Web services out of the available ones that are suitable for solving a goal. This serves as a first filtering step in SWS environments for which we consider functional aspects as the primary criterion; the usability of the discovered candidates for solving a given goal is then further inspected in subsequent processing steps which take other aspects into account. In accordance to the goal model elaborated in Chapter 3, our Web service discovery approach distinguishes two phases: at design time, suitable Web services for goal templates as generic and reusable objective descriptions are discovered, and the actual Web services for goal instances which denote concrete client objectives by instantiating a goal template with concrete inputs are discovered at runtime.

A central requirement for automated Web service discovery engines is a high retrieval accuracy, meaning that under functional aspects every discovered Web service is suitable (= high precision) and every suitable Web service can be discovered (= high recall). This can most adequately be achieved by semantic matchmaking of sufficiently rich functional descriptions. For this, we have defined formal functional descriptions that can precisely describe the overall functionality provided by a Web service with respect to the start- and end-states of its possible executions, and, analogously, can precisely specify the basic objective description for a goal by formally defining the start- and end-states of its possible solutions. Following the standard approach from formal software specification, our functional descriptions are defined in terms of preconditions and effects on the basis of domain ontologies. We explicitly specify the dependency of the preconditions and effects by free variables that occur in both conditions, and also explicitly specify the computational in- and outputs. We use classical first-order logic (FOL) as the specification language which – although undecidable in general – provides a sufficiently high expressivity and does not impose possibly unnecessary modeling restrictions. Moreover, this facilitates the adaption of our model to the specific ontology languages developed for the Semantic Web.

We have defined a representation of functional descriptions as a single first-order logic formula that allows us to consider the possible executions of Web services and the solutions of goals as logical models in terms of classical model-theoretic semantics. On this basis, we have defined the necessary matchmaking techniques for the two-phased Web service discovery. For the design time discovery task, we have defined four matching degrees (*exact*,*plugin*,*subsume*, and *intersect*) which differentiate the situations where a Web service is functionally usable to solve a goal template; the *disjoint* degree states that this is not given. The matchmaking conditions are defined as proof obligations, and we have shown that marginal restrictions on the modeling of functional descriptions are sufficient to ensure the decidability. This supports the employment of standard FOL reasoning techniques, and warrants a high retrieval accuracy for the discovery task because the used functional descriptions precisely describe the requested and provided functionalities.

At runtime, a goal instance is created by defining a concrete variable assignment for the inputs required by the corresponding goal template. We require a goal instance to be a valid instantiation so that the functional description of its corresponding goal template is satisfiable under the defined input values. For a Web service to be functionally suitable for solving a goal instance, it must be invocable with the defined inputs and the resulting execution must be a solution for the goal instance. In our two-phase discovery framework, we can use the design time discovery results in order to minimize the necessary matchmaking efforts at runtime. For this, we have shown that those Web services that are suitable for a goal template are potential candidates for its goal instances. Furthermore, if a Web service is usable for a goal template under the *exact* or the *plugin* degree, then it is also usable for all its goal instances; under the *subsume* and the *intersect* degree, additional matchmaking is necessary at runtime. We have exemplified the modeling of functional descriptions for goals and Web services within an illustrative example and demonstrated the retrieval accuracy of the discovery techniques therein, and we have presented the implementation of the semantic matchmaking techniques within an automated theorem prover.

Related Work

Due to its relevance for SWS technology, automated Web service discovery has been subject to many research efforts that have provided significant contributions and insights. To properly position our approach therein, the following discusses related works with respect to (1) the conceptual model of our two-phase discovery approach and its architectural allocation in SWS environments, (2) works on functional Web service discovery by semantic matchmaking, and (3) other techniques that can be considered to be complementary.

Conceptual Model. The first aspect relates to the overall conceptual model of our two-phase Web service discovery framework. As summarized above, we separate design time and runtime operations, and provide semantic matchmaking techniques for Web service discovery under functional aspects for both phases.

In principle, this correlates with the purpose of Web service discovery and its architectural allocation within SWS environments that support the complete process of automated Web service detection and execution in order to solve a client request (see Section 2.2.2). Therein, Web service discovery is considered as the first processing step which identifies possible candidate Web services whose usability is then investigated in subsequent processing steps, and the compatibility on the level of the provided and the requested functionality is considered the most expedient indicator for this [Preist, 2004]. We find the same conceptual model in other application areas wherein requests and offers need to be reconciled, e.g. within electronic marketplaces (e.g. [Schmid and Lindemann, 1998]), in multi-agent systems (e.g. [Martin et al., 1999]), or for service detection in mobile environments (e.g. [Avancha et al., 2002]). Other conceptual models differentiate several filters for Web service detection whereby each filter treats a different aspect or a different level of abstraction (e.g. [Sycara et al., 2002; Lara et al., 2006]). Although this does not necessarily contradict our approach, it appears to be mandatory that the result of a preceding filter does not contradict with the results of its successors. If this is not given, there might be too much redundancy or – even more undesirable for real-world applications – actually usable Web services might not be detectable due to an insufficient retrieval accuracy.

In contrast to most other works, we take a goal-driven approach as proposed by the WSMO framework [Fensel et al., 2006]. Goals are an explicit modeling element for the client side of SOA technology, which is commonly neglected in most other SWS technologies. In consequence, clients are expected to formulate discovery requests as queries on formal Web service descriptions, and then to consume Web services via hard-wired invocations. This is overcome by our approach by formally describing the objective to be achieved in terms of a goal, and the automated invocation of Web services with the concrete inputs that are defined in goal instances (see Chapter 3). Moreover, our two-phase discovery approach co-aligns with the heuristic classification methodology [Clancey, 1985] which has been identified to be suitable for developing workable and efficient Web service discovery techniques [Keller et al., 2005]: a goal instance represents the concrete problem to be solved, and the valid instantiation of its corresponding goal template defines the *abstraction* to the level of generic objective descriptions; the design time discovery on the goal template realizes the most relevant aspects of the *matchmaking*, and the runtime discovery denotes the *refinement* of the findings from the abstract to the concrete level of goal instances.

Web Service Discovery by Semantic Matchmaking. We now turn towards works on semantically enabled Web service discovery by the matchmaking of formal functional descriptions, which relate to the semantic matchmaking techniques specified in this chapter. The general approach relies on matchmaking techniques developed in the area of formal software specification and reuse (e.g. [Zaremski and Wing, 1997; Meyer, 2000]) as well as in multi-agent systems (e.g. [Kuokka and Harada, 1996; Sycara et al., 1999]). Two aspects seem to be relevant in order to position our approach within existing works: the definition of a match and the matchmaking conditions for this, and the suitability of the used functional descriptions in order to warrant a high retrieval accuracy.

In accordance to [Noia et al., 2003], a suitable basis for discussing the meaning of a match in the context of Web service discovery is the distinction of (1) a *guaranteed match* where the execution of the Web service will always solve the given request or goal, (2) a *possible match* where the Web service might be usable, and (3) *non-match* where the Web service is not suitable. Commonly, the first category is defined as the *plugin* match – which however is defined differently in different works. For example, [Zaremski and Wing, 1997] defines the *plugin* match as $(Q_{pre} \Rightarrow S_{pre}) \wedge (S_{\mathit{eff}} \Rightarrow Q_{\mathit{eff}})$, i.e. the service S is considered to be suitable for the request Q if it can be properly invoked and the execution result of S is a specialization of the result expected by Q. Other works define the *plugin* match exactly the opposite way, in particular such that the effect of S must be a generalization of the one expected by Q in order to guarantee that every execution of S will solve Q (e.g. [Paolucci et al., 2002; Keller et al., 2006a]). Our *plugin* match follows the latter conception, because this defines the situation where the suitability of a Web service is guaranteed for solving every goal instance that properly instantiates the respective goal template.

Regarding the formal functional descriptions, [Paolucci et al., 2002] defines semantic matchmaking for in- and outputs in OWL-S, and [Li and Horrocks, 2003] extends this to requested and advertised service profiles that are described by DL formulae. Both define the matching degrees that we use to denote the usability of Web services for goal templates with the ordering $\mathit{disjoint} > \mathit{intersect} > \mathit{subsume} > \mathit{plugin} > \mathit{exact}$. However, a match is defined on the basis of subsumption relations subsumption among the individual concepts in the functional descriptions. This results from the insufficiency of OWL-S wherein the IOPE-elements are separated logical statements whose dependency can not be explicated (see Section 4.2.1). In consequence, the matchmaking algorithms can merely detect ontological relationships between corresponding description elements, but determine whether the execution of a Web services will satisfy a client request. Naturally, keyword-based matchmakers that merely consider keywords without a suitable description model have the same deficits in the achievable retrieval accuracy (e.g. [Oundhakar et al., 2005]).

An approach that aims at overcoming the limitations of keyword- and concept-based matchmaking is presented in [Kifer et al., 2004]. Therein, a match is considered to be given if the concrete inputs provided by the client satisfy the precondition of a Web service description and then, for the given inputs, the effect of the Web service logically entails the goal description. In principle, this corresponds to our definition of a match on the goal instance level. As a follow-up work, [Lara, 2006] presents a two-phased Web service discovery approach: the first phase detects potential candidates by matchmaking of the effect descriptions of the goal and the Web service, and in the second phase the inputs that the client needs to provide to invoke one of the candidates are determined. However, this does not distinguish goal templates and goal instances and thus does not separate design- and runtime operations. Our matchmaking criteria appear to be more precise because we already consider the start- and the end-state constraints for Web service discovery on the goal template level. Also, the creation of goal instances by merely defining input bindings minimizes the necessary effort for goal formulation by clients.

Another approach that explicitly defines the dependency of the distinct elements of functional descriptions is presented in [Hull et al., 2006]. Therein, a functional description is defined as $\langle \vec{x} : \vec{X};\ \vec{y} : \vec{Y};\ \phi(\vec{x}, \vec{y}) \rangle$ where $\vec{x}$ denotes the input variables with the type definition, $\vec{y}$ are the out variables, and the formula $\phi(\vec{x}, \vec{y})$ describes the functionality as a relationship of the in- and outputs. This corresponds to our approach of defining free variables that occur in both the precondition and the effect. However, we only consider the input variables to usually correspond to these free variables while the outputs are only constrained within the effect description. Besides, this is restricted to stateless Web services and hence only covers a subset of the functionalities supported by our approach.

Works that have served as starting points for the definition of functional descriptions and the semantic matchmaking techniques are [Keller and Lausen, 2006] which discusses the formal meaning of WSMO capabilities on the basis of the Abstract State Space model, and [Keller et al., 2006a] which defines the matchmaking degrees between goals and Web services in terms of generic set-theoretic criteria but without specifying the necessary functional descriptions. We have refined this approach and provided a comprehensive formalization. We further have extended the discovery framework to also cover the level of goal instances. We use classical first-order logic (FOL) as the specification language for functional descriptions. However, the structure and formal meaning of functional descriptions as well as the definition of the matchmaking conditions are applicable to all ontology languages with model theoretic semantics, in particular to Descriptions Logics via the translation defined in [Borgida, 1996] in order to be compatible to OWL as the currently recommended standard ontology language for the Semantic Web.

Complementing Techniques. We complete the discussion with related works that address other aspects which are also relevant for automated Web service discovery.

The first group of works is concerned with the handling situations where the matchmaker determines a *disjoint* match but only a subset of the conditions defined in a goal conflict with the Web service description. This has been identified as an interesting situation in [Noia et al., 2003], with the aim of providing techniques for revising goals that are not solvable. An example for this is that a goal requests a flight ticket from Innsbruck to Vienna for € 100, but there is no offer which satisfies this. To make the goal solvable, one can weaken the conditions by defining a higher price limit or defining a different means of transportation. [Colucci et al., 2005] presents a technique for identifying the conflicting conditions by determining the logical difference between the formal descriptions. Similar techniques have been developed for establishing the usability of a problem-solving method for a concrete task if this is not given a priori [Fensel and Straatman, 1998], and [Stollberg et al., 2005a] adopts this for WSMO capabilities. As a different approach, [Klusch et al., 2006] presents a hybrid matchmaker for OWL-S which applies heuristic retrieval techniques in addition to logical matchmaking in order to identify "nearest-neighbor matches". These techniques can be considered as possible extensions to the basic semantic matchmaking in order to determine suitable candidate Web services when a proper match is not given.

Another group of works addresses Web service discovery with respect to non-functional and quality-of-service criteria, e.g. [Vu et al., 2005; Lu, 2005; Wang et al., 2006]). We consider such techniques to be applied within the subsequent processing steps after functional discovery: the discovery result is an unordered set of potential candidates which then is refined and ordered by respective selection and ranking techniques (*cf.* Section 2.2.2). We further consider ontology-based data mediation (e.g. [Noy, 2004; Mocan and Cimpian, 2007]) as complementary techniques, which can be applied to ensure that the goal and Web service descriptions use homogeneous background ontologies.

Finally, we consider techniques for the creation and validation of goal and Web service descriptions to complement the techniques specified in this work. This covers techniques for validating the correctness and consistency of the formal functional descriptions as discussed in [Keller et al., 2006b], as well as advanced techniques for supporting the identification of concrete input values to complement the basic conditions for the goal instance formulation as defined in our approach (e.g. [Born et al., 2007; Vitvar et al., 2007b]).

Chapter 5

Semantic Discovery Caching

This chapter presents a technique for enhancing the efficiency and scalability of automated Web service discovery. This appears to be essential for the applicability of discovery engines in real-world SOA scenarios, and becomes in particular important for their employment as a heavily used component in Semantic Web service environments. We address this challenge by extending the two-phase Web service discovery approach presented in Chapter 4 with a mechanism that captures relevant knowledge of design time discovery results and effectively uses this for enhancing the computational performance of runtime discovery operations. This adopts the concept of caching to the context of Web service discovery, and thus we refer to the technique as *Semantic Discovery Caching* (short: SDC).

As discussed in Section 2.2.2, the need for efficient and scalable Web service discovery techniques arises because SOA technologies must be able to deal with the large and steadily growing number of available Web services that can be expected in real-world settings. This becomes in particular relevant for Semantic Web service (SWS) environments wherein Web service discovery is a central and expectably often performed operation: it detects the suitable candidates out of the available Web services, and is usually performed as the first processing step for solving a given goal that needs to be performed for each new request. In order to ensure the operational reliability of the Web service discovery component in such architectures, it is necessary to reduce both the average time for individual discovery tasks as well as the time variance among several invocations. To provide a sophisticated extension to semantically enabled Web service discovery, the optimization technique should maintain the achievable retrieval accuracy and not require additional annotations of Web services or of goals. This challenge has received only little attention in the SWS community so far, in particular in comparison to semantic matchmaking techniques for Web service discovery.

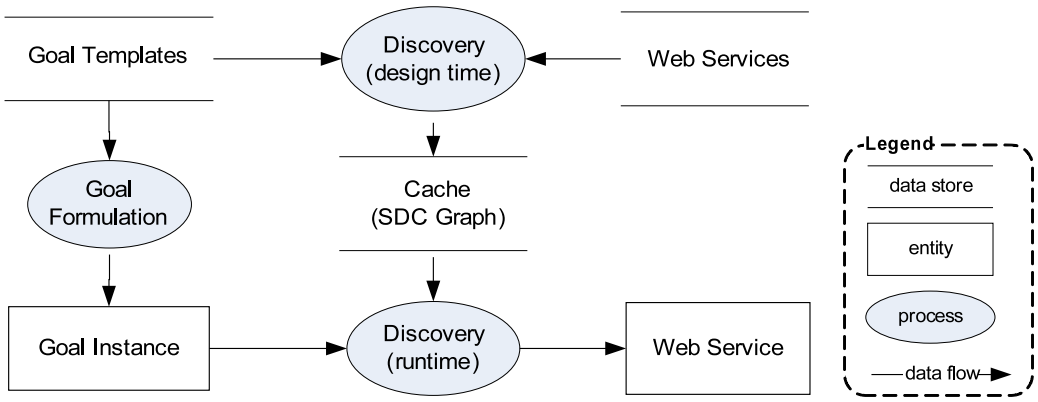


Figure 5.1: Overview of SDC-Optimized Web Service Discovery

Figure 5.1 provides an overview of our approach for the performance optimization. This extends the two-phased Web service discovery model from Chapter 4 with a caching mechanism for enhancing the efficiency of runtime discovery. We consider three central entities: *Web services* that have a formal description, *goal templates* as generic objective descriptions that are stored in the system, and *goal instances* that describe concrete requests by instantiating a goal template with concrete inputs. At design time, the Web services for goal templates are discovered by matchmaking of their formal functional descriptions. The result is stored in the *SDC graph*, a special knowledge structure which organizes goal templates in a subsumption hierarchy and captures the minimal knowledge on the suitability of the available Web services. This provides an unambiguous index for the efficient search of goals and Web services that can be generated automatically on the basis of semantic matchmaking. At runtime, a concrete client request is formulated as a goal instance for which the actual Web services need to be discovered. We consider this as the expectably most frequent operation in SOA applications because – in our model – goal instances denote the primary element for clients to detect and consume Web services. We thus optimize the Web service discovery at runtime by exploiting the SDC graph in order to reduce the search space and minimize the number of necessary matchmaking operations.

The aim of this chapter is to define the necessary concepts, data structures, and the basic operations of the SDC technique. In particular, we formally define the SDC graph and specify the algorithms for its creation and maintenance as well as its usage for optimizing the runtime Web service discovery. We here define the SDC technique on the basis of the functional descriptions for goals and Web services and the semantic matchmaking techniques specified in Chapter 4 that warrant a high retrieval accuracy for both design- and runtime discovery. However, the general approach can be adapted to frameworks that follow the

goal-based approach for Semantic Web services discussed in Chapter 3 but apply other specification languages. We use the shipment scenario from the SWS Challenge as the running example throughout this chapter, a widely recognized initiative for demonstration and comparison of semantically enabled Web service discovery techniques ([Petrie et al., 2008], see: `www.sws-challenge.org`). The scenario defines several Web services for package shipment between different destination countries on the basis of existing real-world services, along with several examples of client requests. We shall use the original data set of this scenario for the quantitative evaluation of the SDC technique in Chapter 6.

This chapter is structured as follows. At first, Section 5.1 explains the aim and approach of the SDC technique in more detail and recalls the foundations from the previous chapters. Then, Section 5.2 formally defines the SDC graph and discusses its relevant properties, and Section 5.3 specifies the algorithms for its automated creation and maintenance. Section 5.4 explains how the SDC graph is used to optimize the Web service discovery at runtime, and Section 5.5 presents the prototype implementation of the SDC-enabled Web service discovery engine. Finally, Section 5.6 concludes the chapter and discusses related work.

5.1 Motivation and Overview

The aim of the SDC technique is to enhance the computational performance of automated Web service discovery, in particular for runtime discovery which is concerned with the detection of suitable Web services for goal instances and thus denotes the time critical operation in our approach. The following discusses the need for scalable discovery techniques, recalls the underlying conceptual model and the Web service discovery techniques from the preceding chapters, and provides an informal overview of the SDC technique that we shall specify in detail in the subsequent sections.

5.1.1 The Need for Scalable Web Service Discovery

The need for efficient and scalable technologies arises in application areas with a larger and continuously changing number of available resources, and commonly performance optimization techniques are employed in order to ensure the operational reliability of software systems that operate in such environments [Crawford et al., 2000]. This naturally becomes important in SOA applications where numerous Web services shall be used as the basic building blocks of an IT system. For this, the fast and reliable execution of Web services by information exchange over the Internet as well as the management of larger number of available Web services are considered as critical success factors [Cohen, 2006].

In the context of semantically enabled SOA technologies, we consider Web service discovery as the first processing step in SWS environments for solving goals by the automated detection and execution of Web services. It determines the candidates out of the available Web services under functional aspects; their actual usability is then further investigated in subsequent processing steps that take other aspects into consideration (see Section 2.2.2). Hence, discovery denotes the bottleneck for the scalability of such environments because it requires a $1 : n$ search on all available Web services while the subsequent processing steps only need to deal with the discovered candidates. Thus, under consideration of larger search spaces that can be expected in real-world applications, the computational performance of automated discovery engines is a critical success factor for the scalability of the whole system. This becomes in particular important for advanced SWS techniques that envision to employ automated discovery engines as heavily used components, e.g. in recent approaches for dynamic Web service composition where the candidates shall be detected at each iteration step of the composition algorithm [Bertoli et al., 2007], or for semantically enabled business process management where the actual Web services for specific process activities shall be determined at runtime [Hepp et al., 2005; Wetzstein et al., 2007].

We take a goal-based approach for Semantic Web services. A goal formally describes the objective that a client wants to solve by using Web services, which provides an abstraction layer for enabling problem-oriented Web service usage by clients (see Chapter 3). We explicitly distinguish *goal templates* as generic and reusable objective descriptions that are kept in the system, and *goal instances* that describe concrete client objectives by instantiating a goal template with concrete inputs. On this basis, we have defined a Web service discovery framework that separates two phases: at design time, the suitable Web services for goal templates are discovered by matchmaking of formal functional descriptions, and at runtime the actual Web services for goal instances are discovered with respect to the concrete input values. We have defined semantic matchmaking techniques for both design- and runtime discovery which expose a high retrieval accuracy (see Chapter 4).

In this two-phase approach, the runtime discovery on the level of goal instances is the time critical operation. We consider goal instances as the primary element for clients to demand Web services and automatically consume them. Each new client request is formulated in terms of a goal instance for which the actual Web services need to be discovered and automatically executed. This facilitates dynamic Web service usage in order to overcome the limitations of hard-wired invocations out of a client application (see Section 3.2.1). Thus, the runtime discovery should be performed in an efficient manner in order to provide an operationally reliable filter for the subsequent processing steps, also for larger search spaces of available Web services. The Web service discovery on the level of goal templates appears

to be not time critical because it can be performed at design time, i.e. whenever a goal template or a Web service is added, removed, or changed in the system. Besides, we expect that in real-world settings the runtime discovery is a much more frequent operation than the design time discovery: the former is required for every single client request, while the latter is only needed when the existing goal templates or the available Web services change. In accordance to the standard quality measurements for software performance [Ebert et al., 2004], we consider the following characteristics for judging the computational performance of an automated discovery engine: *efficiency* as the time required for finding a suitable Web service, *scalability* as the ability to deal with larger search spaces of available Web services, and *stability* as a low variance of the execution time among several invocations.

5.1.2 The SDC Approach

The approach for enhancing the computational performance of Web service discovery that underlies the SDC technique is to reduce the search space and minimize the necessary matchmaking operations for individual discovery operations. We choose this optimization strategy with respect to the known performance deficiencies of reasoning techniques: the fewer matchmaking operations are needed for obtaining a valid discovery result, the faster the discovery process can be completed (see Section 2.2.2).

To realize this, we exploit the relationships between goal templates, goal instances, and Web services that result from the formal functional descriptions. The central element for this is the SDC graph that organizes goal templates in a subsumption hierarchy and keeps the relevant knowledge on the suitability of the available Web services for each goal template by capturing the results of design time discovery runs. This provides an efficient search index for goals and Web services, extending the property of goal templates as a cache for the candidate services that are functionally suitable for solving goal instances.

The SDC graph is defined such that there is only one unambiguous graph structure that properly represents the relevant relationships for a given set of goals and Web services. The basic idea is to organize goal templates with respect to their semantic similarity. Two goal templates G_i and G_j are considered to be similar if they have at least one common solution: then, mostly the same Web services are usable for them. We express this in terms of *similarity degrees* as the matching degree between the functional descriptions $\mathcal{D}_{G_i}$ and $\mathcal{D}_{G_j}$, which are defined analog to the matching degrees for denoting the functional usability of a Web service for a goal template (see Section 4.3.1). To illustrate this, Table 5.1 shows the functional descriptions of two goal templates from the shipment scenario: G_1 for shipping packages within Europe, and G_2 for shipping packages in Germany with a maximal weight

Table 5.1: Example for Semantically Similar Goal Templates

Goal Template G_1 "ship a package of any weight in Europe"		**Goal Template** G_2 "ship a package in Germany, max 50 kg"	
Ω:	location & shipment ontology	Ω:	location & shipment ontology
IN:	{?s,?r,?p,?w}	IN:	{?s,?r,?p,?w}
ϕ^{pre}:	$address(?s) \wedge in(?s, europe)$	ϕ^{pre}:	$address(?s) \wedge in(?s, germany)$
	$\wedge\, address(?r) \wedge in(?r, europe)$		$\wedge\, address(?r) \wedge in(?r, germany)$
	$\wedge\, package(?p) \;\wedge weight(?p, ?w)$		$\wedge\, package(?p) \;\wedge weight(?p, ?w)$
	$\wedge\, maxWeight(?w, heavy)$.		$\wedge\, maxWeight(?w, 50)$.
ϕ^{eff}:	$\forall ?o, ?price.\ out(?o) \Leftrightarrow ($	ϕ^{eff}:	$\forall ?o, ?price.\ out(?o) \Leftrightarrow ($
	$shipmentOrder(?o, ?p)$		$shipmentOrder(?o, ?p)$
	$\wedge\, sender(?p, ?s) \;\wedge receiver(?p, ?r)$		$\wedge\, sender(?p, ?s) \;\wedge receiver(?p, ?r)$
	$\wedge\, costs(?o, ?price)\ )$.		$\wedge\, costs(?o, ?price)\ )$.

of 50 kg. As defined in Section 4.2, a functional description $\mathcal{D} = (\Sigma^\star, \Omega, IN, \phi^{pre}, \phi^{eff})$ formally describes the possible solutions for a goal in terms of a precondition ϕ^{pre} that constrains the possible start-states and an effect ϕ^{eff} that describes the final desired state and explicitly defines the expected computational outputs in the predicate $out(?o_1, \ldots, ?o_n)$. The dependency between the start- and the end-state is defined via free variables that occur in both ϕ^{pre} and ϕ^{eff}, which usually are the same as the required inputs $IN = (?i_1, \ldots, ?i_n)$. The relevant terminology and background knowledge is defined in a domain ontology Ω.

The similarity degree of the goal templates in Table 5.1 is $subsume(G_1, G_2)$ because every shipment order whose sender and receiver are located in Germany is also a shipment order whose the sender and receiver are located in Europe, but not vice versa. In consequence, only those Web services that are usable to solve G_1 can possibly be usable for G_2, because a Web service that does not provide package shipment within Europe can also not ship packages within Germany. We use this relation as the constituting principle for the SDC graph: it organizes the existing goal templates with respect to the requested functionalities such that the only occurring similarity degree is $subsume(G_i, G_j)$. This enables efficient search, because under this similarity degree the solutions for the child G_j are a subset of those for the parent G_i, and thus the Web services that are usable for G_j are a subset of those usable for G_i. We explicate the subsumption relation between goal templates by directed arcs (G_i, G_j), which define the so-called *goal graph* as the basic index structure of the SDC graph. We can properly allocate every goal template in the goal graph under every possible similarity degree. To ensure the proper subsumption hierarchy, we introduce so-called *intersection goal templates* as additional nodes in the goal graph that describe

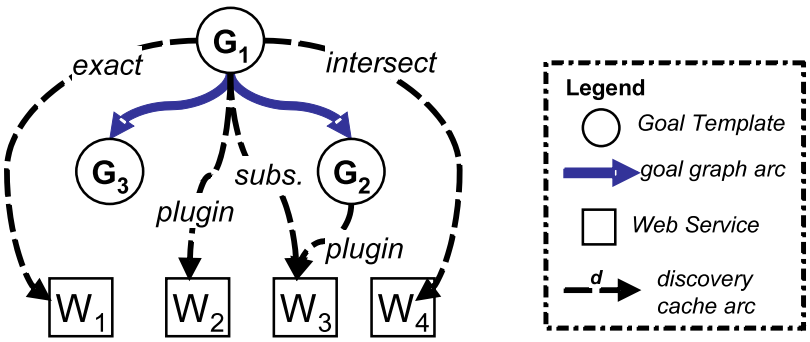


Figure 5.2: Example of a SDC Graph

the common solutions of two goal templates when their similarity degree is *intersect*. In addition to this, the SDC graph explicates the functional usability of the available Web services for each goal template by capturing the result of design time discovery runs. This is defined by directed arcs (G, W) between a goal template G and a Web service W, which are annotated with the matching degree $d(G, W)$ as the relevant knowledge for optimizing the discovery operations. We refer to this as the *discovery cache*.

Figure 5.2 illustrates the structure of the SDC graph for our running example. We here consider three goal templates: G_1 for package shipment in Europe, G_2 for Germany, and G_3 for Switzerland. Their similarity degrees are $subsume(G_1, G_2)$ and $subsume(G_1, G_3)$ as discussed above; this is explicated in the goal graph. It also holds that $disjoint(G_2, G_3)$ because the solutions for G_2 and for G_3 are disjoint subsets of those for G_1. Let the following Web services be among the available ones: W_1 for package shipment in Europe, W_2 in the whole world, W_3 in the European Union, and W_4 in the Commonwealth. The usability degree of every Web service for each goal template is explicated in the discovery cache. The figure shows the actual design time discovery results that are obtained from matchmaking of the functional descriptions (see Section 4.3.1). To keep the SDC graph minimal, we omit redundant discovery cache arcs whose usability degree can be directly inferred as we shall explain below. We observe that a SDC graph is a directed acyclic graph whose inner nodes are the goal templates and whose leaf nodes are the usable Web services.

The basis for exploiting the SDC graph are inference rules of the form $d(G_i, G_j) \wedge d(G_i, W) \Rightarrow d(G_j, W)$, i.e. between the similarity degrees of goal templates and the usability degrees of Web services which result from the formal model. These provide the logical foundation for effectively reducing the search space for Web service discovery operations, and also for maintaining the SDC graph. Regarding the omittance of redundant arcs, in the above example it holds that $subsume(G_1, G_3) \wedge plugin(G_1, W_2) \Rightarrow plugin(G_3, W_2)$ because

$\{\mathcal{T}\}_{W_2} \subset \{\mathcal{T}\}_{G_1} \subset \{\mathcal{T}\}_{G_3}$. In this case we omit the discovery cache arc (G_3, W_2) because the usability degree can be directly inferred from (G_1, W_2). Analogously, we can also omit the arc (G_3, W_1) because its usability degree can be directly inferred by $subsume(G_1, G_3) \wedge exact(G_1, W_1) \Rightarrow plugin(G_3, W_1)$; the same holds for the arcs (G_2, W_1) and (G_2, W_2). The omittance of such redundant arcs allows us to keep the discovery cache minimal while we still know the particular usability degree of every Web service for each goal template.

To optimize the Web service discovery for goal instances at runtime, we can use the SDC graph as follows. Consider a goal instance GI for shipping a package from Bern to Zürich, i.e. within Switzerland. Let G_1 be defined as the initial corresponding goal template. As the first step, we can revise the corresponding goal template to be G_3 in order to obtain a more precise goal description. This results in a reduction of the search space of possible candidates from the 4 Web services that are usable for G_1 to the 2 Web services that are usable for G_3; we recall that only those Web services can be usable for a goal instance which are usable for its corresponding goal template (*cf.* Definition 4.3 in Section 4.1.2). Moreover, we know that both W_1 and W_2 are usable for G_3 under the *plugin* degree as explained above. Under this degree, the Web service is also usable for every goal instance that properly instantiates the goal template without the need of additional matchmaking (*cf.* Theorem 4.1 in Section 4.3.2). We thus will detect W_1 and W_2 as the only Web services that are usable to solve GI under functional aspects. This shows that the runtime discovery task can be performed with a minimal number of necessary matchmaking operations.

This is a novel approach for enhancing the computational performance of automated Web service discovery by adopting the concept of caching as a well-established optimization technique. In comparison to existing approaches that mostly apply clustering techniques, the main benefits are that (1) it maintains the high retrieval accuracy that is achievable by our semantic matchmaking techniques, (2) it can, in certain cases, perform Web service discovery at runtime without invoking a matchmaker, and (3) the SDC graph as the underlying data structure can be generated automatically on the basis of semantic matchmaking without the need of additional annotations.

While we shall discuss related works in detail in Section 5.6, the following sections provide the detailed specification of the SDC technique. In particular, we define the necessary techniques and algorithms to ensure that the SDC graph properly captures the relevant knowledge and that its structure and properties are maintained at all times, and we define the optimized algorithms for Web service discovery at runtime. While we here focus on the theoretic foundations and the technical realization, we shall show that the SDC technique can achieve significant improvements for runtime Web service discovery in terms of efficiency, scalability, and stability, on the basis of a detailed evaluation in Chapter 6.

5.2 Concepts and Definitions

This section defines the central concepts and data structures of the SDC technique. We commence with defining the similarity measurement of goals, and the inference rules between similar goal templates and their usable Web services. Then, we formally define the SDC graph and explain how it can be created and maintained automatically. Finally, we discuss the computational complexity for managing a SDC graph as well as its properties as a search index for goal templates and Web services.

5.2.1 Goal Similarity and Inference Rules

The following defines the measurement notion for the semantic similarity of goals as the basic notion of the SDC technique. On this basis, we also define all relevant inference rules between similar goal templates and the usability of Web service for them, which provides the logical foundation for defining and exploiting SDC graphs.

Goal Similarity

The constituting principle of the SDC graph is to organize goal templates in a subsumption hierarchy with respect to the requested functionalities. This defines the goal graph as the skeletal indexing structure of the SDC graph. As the basis for this, we need to define a formal measurement for the semantic similarity of goal templates.

As outlined above, we consider two goal templates G_1 and G_2 to be similar if they have at least one common solution because then mostly the same Web services are usable for them. To explain and formally substantiate this, we briefly recall the understanding of goals and the meaning of their descriptions from Section 4.1. A goal expresses the desire to get from the current state of the world into a state wherein the objective is achieved, and the basic objective description of a goal template G formally describes this in terms of conditions on the possible start-states and conditions on the final desired state. We consider $\mathcal{T} = (s_0, \ldots, s_m)$ as the abstraction of an execution of a Web service that is observable in the world as a sequence of states from a start-state s_0 to an end-state s_m (*cf.* Definition 4.2 in Section 4.1). This is a solution for G if s_0 satisfies the conditions on the start-state and s_m satisfies those on the final desired state; we denote the set of all possible abstract solutions for G by $\{\mathcal{T}\}_G$. With respect to this, we define two goal templates G_1 and G_2 to be semantically similar if there is a sequence of states $\mathcal{T} = (s_0, \ldots, s_m)$ which is a solution for G_1 and for G_2. A Web service that can provide such a $\mathcal{T}$ is then usable for both goal templates. The following defines this notion of goal similarity formally.

Definition 5.1. *Let G_1 be goal template with $\{\mathcal{T}\}_{G_1}$ as the set of its possible abstract solutions, and let G_2 be goal template with $\{\mathcal{T}\}_{G_2}$ as the set of its possible abstract solutions. G_1 and G_2 are semantically similar if and only if*

$$\exists \mathcal{T}.\ \mathcal{T} \in (\{\mathcal{T}\}_{G_1} \cap \{\mathcal{T}\}_{G_2}).$$

This is the basic condition for the similarity of goal templates, which is defined analog to the meaning of a match between a goal template and a Web service (*cf.* Definition 4.3 in Section 4.1.2). We choose this goal similarity measurement with respect to the primary focus of our Web service discovery approach on the compatibility of the requested and provided functionalities. One could also consider other aspects for this, e.g. that goals are described on the basis of the same domain ontology or that they are defined in the same application area. However, the goal similarity as defined here appears to be appropriate for the context of the SDC technique because it allows us to organize goal templates in a way such that we can effectively determine the functionally suitable Web services for them.

The semantic matchmaking techniques in our discovery framework are defined over functional descriptions. These formally describe the possible executions of Web services as well as the possible solutions of goals with respect to the start- and end-states, which provides an abstract but sufficiently rich description of the overall provided and requested functionalities (see Section 4.2). Because the SDC technique shall provide an optimization for this, it is sufficient to evaluate the similarity of goal templates on the basis of their functional descriptions. In order to precisely differentiate the grade of similarity between goal templates, we define *similarity degrees* which denote the matching degree between their functional descriptions – analog to the matching degrees between a goal template and a Web service (see Section 4.3.1). The four similarity degrees *exact, plugin, subsume* and *intersect* distinguish different situations wherein the basic similarity condition from Definition 5.1 is satisfied, and the *disjoint* degree denotes that this is not given.

Table 5.2 provides the definitions of the goal similarity degrees in a concise manner. Their meaning is as follows: $exact(G_1, G_2)$ denotes that the solutions for G_1 and G_2 are exactly the same, $plugin(G_1, G_2)$ denotes that every solution of G_1 is also a solution for G_2 and $subsume(G_1, G_2)$ denotes the opposite relationship; $intersect(G_1, G_2)$ states that there is at least one common solution, and $disjoint(G_1, G_2)$ denotes that a common solution for G_1 and G_2 does not exist. In order to support reasoning in terms of model-theoretic semantics, we define the conditions for the similarity degrees over the representation of functional descriptions by the first-order logic structure $\mathcal{D}_{FOL} = (\Sigma^\star, \Omega^\star, IN, \phi^{\mathcal{D}})$ from Definition 4.6. Here, the FOL formula $\phi^{\mathcal{D}} := [\phi^{pre}]_{\Sigma_D \to \Sigma_D^{pre}} \Rightarrow \phi^{eff}$ defines a logical implication

Table 5.2: Definition and Meaning of Goal Similarity Degrees

Denotation	Definition	Meaning
$\mathbf{exact}(\mathcal{D}_{G_1}, \mathcal{D}_{G_2})$	$\Omega^\star \models \forall\beta.\ \phi^{\mathcal{D}_{G_1}} \Leftrightarrow \phi^{\mathcal{D}_{G_2}}$	$\mathcal{T} \in \{\mathcal{T}\}_{G_1}$ if and only if $\mathcal{T} \in \{\mathcal{T}\}_{G_2}$
$\mathbf{plugin}(\mathcal{D}_{G_1}, \mathcal{D}_{G_2})$	$\Omega^\star \models \forall\beta.\ \phi^{\mathcal{D}_{G_1}} \Rightarrow \phi^{\mathcal{D}_{G_2}}$	if $\mathcal{T} \in \{\mathcal{T}\}_{G_1}$ then $\mathcal{T} \in \{\mathcal{T}\}_{G_2}$
$\mathbf{subsume}(\mathcal{D}_{G_1}, \mathcal{D}_{G_2})$	$\Omega^\star \models \forall\beta.\ \phi^{\mathcal{D}_{G_1}} \Leftarrow \phi^{\mathcal{D}_{G_2}}$	if $\mathcal{T} \in \{\mathcal{T}\}_{G_2}$ then $\mathcal{T} \in \{\mathcal{T}\}_{G_1}$
$\mathbf{intersect}(\mathcal{D}_{G_1}, \mathcal{D}_{G_2})$	$\exists\beta.\ \bigwedge \Omega^\star \wedge \phi^{\mathcal{D}_G} \wedge \phi^{\mathcal{D}_W}$ is satisfiable	$\exists\mathcal{T}.\ \mathcal{T} \in \{\mathcal{T}\}_{G_1} \cap \{\mathcal{T}\}_{G_2}$
$\mathbf{disjoint}(\mathcal{D}_{G_1}, \mathcal{D}_{G_2})$	$\exists\beta.\ \bigwedge \Omega^\star \wedge \phi^{\mathcal{D}_G} \wedge \phi^{\mathcal{D}_W}$ is unsatisfiable	$\{\mathcal{T}\}_{G_1} \cap \{\mathcal{T}\}_{G_2} = \emptyset$

between the precondition ϕ^{pre} and the effect ϕ^{eff}, and the renaming function $[\phi]_{\Sigma_D \to \Sigma_D^{pre}}$ along with the extended background ontology $\Omega^\star$ ensures the proper handling of dynamic symbols (see Section 4.2.2). The matchmaking conditions are defined over input bindings $\beta : (?i_1, \ldots, ?i_n) \to \mathcal{U}$, i.e. concrete value assignments for each input variable $?i \in \mathit{IN}$. This ensures that two similar goal templates define the same or at least semantically equivalent input variables. We further assume that the functional descriptions of goals are consistent so that at least one solution exists, and thus the following formal relations holds between the similarity degrees: (1) *exact* $\Leftrightarrow$ *plugin* $\wedge$ *subsume*, (2) *plugin* $\Rightarrow$ *intersect*, (3) *subsume* $\Rightarrow$ *intersect*, and (4) $\neg$ *intersect* $\Leftrightarrow$ *disjoint* (*cf.* Proposition 4.2 in Section 4.3.1).

Inference Rules

We now turn towards the inference rules between similar goal templates and their usable Web services. As outlined above, we can define rules of the form $d(G_i, G_j) \wedge d(G_i, W) \Rightarrow d(G_j, W)$ in order to infer knowledge about the usability degree of a Web service W for a goal template G_j when the similarity degree between G_j and another goal template G_i as well as the usability degree of W for G_i is known. These rules result from the formal definitions of the matching degrees, and provide the logical basis for the SDC technique.

Theorem 5.1 below defines all inference rules for every possible similarity degree between two goal templates. Referring to Appendix B.1 for the formal proof, this essentially states that (1) under $\mathit{exact}(G_i, G_j)$ all Web services that are usable for G_i are also usable G_j under the same usability degree, (2) under $\mathit{plugin}(G_i, G_j)$ all Web services usable for G_i are also usable for G_j but not vice versa, (3) under $\mathit{subsume}(G_i, G_j)$ the set of usable Web services for G_j is a subset of those usable for G_i, (4) under $\mathit{intersect}(G_i, G_j)$ there can be Web services that are usable for both G_i and G_j, and (5) under $\mathit{disjoint}(G_i, G_j)$ we can mostly not make any statement between the usable Web services for G_i and for G_j.

Theorem 5.1. *Let G_1 and G_2 be goal templates, and let $d(G_1, G_2)$ denote their similarity degree. Let $d(G, W)$ denote the usability degree of a Web service W for a goal template G. Given $d(G_1, G_2)$ and $d(G_1, W)$, all the following holds for $d(G_2, W)$:*

(1) $exact(G_1, G_2) \wedge$
- *(1.1)* $exact(G_1, W) \Rightarrow exact(G_2, W)$.
- *(1.2)* $plugin(G_1, W) \Rightarrow plugin(G_2, W)$.
- *(1.3)* $subsume(G_1, W) \Rightarrow subsume(G_2, W)$.
- *(1.4)* $intersect(G_1, W) \Rightarrow intersect(G_1, W)$.
- *(1.5)* $disjoint(G_1, W) \Rightarrow disjoint(G_2, W)$.

(2) $plugin(G_1, G_2) \wedge$
- *(2.1)* $exact(G_1, W) \Rightarrow subsume(G_2, W)$.
- *(2.2)* $plugin(G_1, W) \Rightarrow exact(G_2, W) \vee plugin(G_2, W) \vee subsume(G_2, W) \vee intersect(G_2, W)$.
- *(2.3)* $subsume(G_1, W) \Rightarrow subsume(G_2, W)$.
- *(2.4)* $intersect(G_1, W) \Rightarrow subsume(G_2, W) \vee intersect(G_2, W)$.
- *(2.5)* $disjoint(G_1, W) \Rightarrow d(G_2, W) = any\ degree.$

(3) $subsume(G_1, G_2) \wedge$
- *(3.1)* $exact(G_1, W) \Rightarrow plugin(G_2, W)$.
- *(3.2)* $plugin(G_1, W) \Rightarrow plugin(G_2, W)$.
- *(3.3)* $subsume(G_1, W) \Rightarrow d(G_2, W) = any\ degree.$
- *(3.4)* $intersect(G_1, W) \Rightarrow plugin(G_2, W) \vee intersect(G_2, W) \vee disjoint(G_2, W)$.
- *(3.5)* $disjoint(G_1, W) \Rightarrow disjoint(G_2, W)$.

(4) $intersect(G_1, G_2) \wedge$
- *(4.1)* $exact(G_1, W) \Rightarrow intersect(G_2, W)$.
- *(4.2)* $plugin(G_1, W) \Rightarrow plugin(G_2, W) \vee intersect(G_2, W)$.
- *(4.3)* $subsume(G_1, W) \Rightarrow subsume(G_2, W) \vee intersect(G_2, W) \vee disjoint(G_2, W)$.
- *(4.4)* $intersect(G_1, W) \Rightarrow d(G_2, W) = any\ degree.$
- *(4.5)* $disjoint(G_1, W) \Rightarrow d(G_2, W) = any\ degree.$

(5) $disjoint(G_1, G_2) \wedge$
- *(1)* $exact(G_1, W) \Rightarrow disjoint(G_2, W)$.
- *(2)* $plugin(G_1, W) \Rightarrow d(G_2, W) = any\ degree.$
- *(3)* $subsume(G_1, W) \Rightarrow disjoint(G_2, W)$.
- *(4)* $intersect(G_1, W) \Rightarrow d(G_2, W) = any\ degree.$
- *(5)* $disjoint(G_1, W) \Rightarrow d(G_2, W) = any\ degree.$

5.2.2 The SDC Graph

The following defines the SDC graph, which serves as the data structure for optimizing Web service discovery operations and thus denotes the heart of the SDC technique. We commence with the overall definition, and then explain the necessary constructs and refinements for obtaining the desirable structure from a given set of goal templates and Web services.

The SDC graph organizes the existing goal templates in a subsumption hierarchy and captures the relevant knowledge on the functional suitable of the available Web services from design time discovery results. As outlined above, we distinguish two layers of a SDC graph: the upper layer is the *goal graph* which organizes the goal templates such that *subsume* is the only occurring similarity degree, and the lower layer is the *discovery cache* which explicates the usability degree of every available Web service for each goal template. All edges in the SDC graph are defined as *binary directed arcs*, i.e. each arc has exactly one source and one target element. In the goal graph, (G_i, G_j) is a directed arc from the goal template G_i to the goal template G_j such that $subsume(G_i, G_j)$. In order to properly handle all possible situations we introduce so-called *intersection goal templates* as virtual goal descriptions in the goal graph; we shall explain this below in more detail. The discovery cache consists of directed arcs (G, W) between a goal template G and a Web service W which are annotated with the usability degree $d(G, W)$. The following defines this formally.

Definition 5.2. *Let $\mathcal{G}$ be a set of goal templates, and let $\mathcal{W}$ be a set of Web services. Let $d(G_i, G_j)$ denote the similarity degree of goal templates $G_i, G_j \in \mathcal{G}$, and let $d(G, W)$ denote the usability degree of a Web service $W \in \mathcal{W}$ for a goal template $G \in \mathcal{G}$.*

The SDC graph for $\mathcal{G}, \mathcal{W}$ is a directed acyclic graph $(V_\mathcal{G} \cup V_\mathcal{W}, E_{sim} \cup E_{use})$ such that:

(i) $V_\mathcal{G} := \mathcal{G} \cup \mathcal{G}^I$ *is the set of inner vertices where*
- $\mathcal{G} = \{G_1, \ldots, G_n\}$ *are the goal templates and*
- $\mathcal{G}^I := \{G^I(G_i, G_j) \mid G_i, G_j \in \mathcal{G}, d(G_i, G_j) = \mathsf{intersect}, \phi^{\mathcal{D}_{G^I}} = \phi^{\mathcal{D}_{G_i}} \wedge \phi^{\mathcal{D}_{G_j}}\}$ *is the set of intersection goal templates for $\mathcal{G}$*

(ii) $V_\mathcal{W} := \{W_1, \ldots, W_m\}$ *is the set of leaf vertices representing Web services*

(iii) $E_{sim} := \{(G_i, G_j) \mid G_i, G_j \in V_\mathcal{G}\}$ *is the set of directed arcs where*
- $d(G_i, G_j) = \mathsf{subsume}$ *and*
- *not exists $G \in V_\mathcal{G}$ such that $d(G_i, G) = \mathsf{subsume}$, $d(G, G_j) = \mathsf{subsume}$*

(iv) $E_{use} := \{(G, W) \mid G \in V_\mathcal{G}, W \in V_\mathcal{W}\}$ *is set of directed arcs where*
- $d(G, W) \in \{\mathsf{exact}, \mathsf{subsume}, \mathsf{intersect}\}$ *or*
- $d(G, W) = \mathsf{plugin}$ *and not exists $G_i \in V_\mathcal{G}$ such that $d(G_i, G) = \mathsf{subsume}$ and $d(G_i, W) \in \{\mathsf{exact}, \mathsf{plugin}\}$.*

This defines a SDC graph as outlined above. The goal graph defines the skeletal structure by organizing the existing goal templates as well as their intersection goal templates in a proper subsumption hierarchy so that the only occurring goal similarity degree is *subsume* and redundant arcs do not exist (*cf.* clauses (i) and (iii)). The discovery cache defines the minimal set of arcs that is necessary to determine the precise usability degree of each available Web service for every existing goal template (*cf.* clauses (ii) and (iv)).

For any set of goal templates and Web services, there is only one possible SDC graph that properly describes the relevant relationships because its actual structure is determined by the formally described requested and provided functionalities. This graph defines an indexing structure where the more general goal templates are allocated at the top. The lower levels contain the more specialized goal templates, for which successively fewer Web services are usable. This can be effectively used for optimizing Web service discovery operations. In particular, for the runtime discovery task we can first determine the most appropriate goal template for the given goal instance by traversing the SDC graph, and then detect the suitable Web services for solving the goal instance with minimal time and reasoning effort. The following explains the techniques and the constructs for the creation and maintenance of SDC graphs. While we here focus on the theoretic foundations, we shall specify the algorithms for the automated creation and maintenance of SDC graphs as well as for optimizing Web service discovery operations in the subsequent sections in detail.

Automated Creation and Maintenance

A central feature of the SDC technique is that the SDC graph as the underlying knowledge structure can be created and maintained automatically. To ensure that the SDC graph exposes the structure defined above at all times, it must (1) correctly represent the relevant relations between the goal templates and the Web services in all possible situations, and (2) properly reflect the changes that can occur in dynamically evolving environments.

This can be ensured by algorithms that automatically create a correct SDC graph for a given set of goal templates and Web services, and also update this correctly whenever a goal template or a Web service is added, removed, or changed. The basis for this are two principles for correctly handling the possibly occurring situations: the first one is concerned with establishing the goal graph such that every goal template is allocated properly without any redundancy, and the second one is concerned with defining the discovery cache such that redundant arcs are omitted in order to enhance its manageability. The following explains both principles and defines their theoretic basis, upon which we shall specify the algorithms for the creation and maintenance of SDC graphs in detail below in Section 5.3.

Establishing the Goal Graph. The goal graph is the skeletal structure of the SDC graph. To enable efficient search of goal templates and Web services, we have defined the goal graph to represent a subsumption hierarchy of the existing goal templates with respect to the semantic similarity of the requested functionalities.

In particular, the goal graph is defined such that (1) the only occurring similarity degree is $subsume(G_i, G_j)$ because then the solutions for the child G_j are a subset of those for the parent G_i and thus the usable Web services for G_j are a subset of those usable for G_i, and (2) the subsumption hierarchy is defined by the minimal number of necessary arcs (*cf.* clause (iii) in Definition 5.2). To properly handle all possible situations that can occur with respect to the semantic similarity of goal templates, we create the goal graph for two goal templates G_i, G_j under the other similarity degrees defined in Table 5.2 as follows:

- if $exact(G_i, G_j)$, then only one of the goal templates is kept in the SDC graph while the other one is considered to be redundant in the context of Web service discovery: the requested functionalities are semantically equivalent, and the same Web services are usable for both under the same usability degree (*cf.* rules 1.1 – 1.5 of Theorem 5.1)

- if $plugin(G_i, G_j)$, then we store the arc (G_j, G_i) in the SDC graph so that G_i becomes a child of G_j; this is possible because $plugin(G_i, G_j) \Leftrightarrow subsume(G_j, G_i)$ given that the functional descriptions $\phi^{\mathcal{D}_{G_i}}$ and $\phi^{\mathcal{D}_{G_j}}$ are consistent (*cf.* Proposition 4.2)

- if $intersect(G_i, G_j)$, then we define an intersection goal template $G^I(G_i, G_j)$ that describes the common solutions of G_i and G_j and becomes an additional child node of both goal templates in the goal graph; we shall explain this below in more detail

- if $disjoint(G_i, G_j)$, then both goal templates are kept as disconnected nodes in the goal graph; in consequence, the SDC graph can have separated subgraphs where each consists of semantically similar goal templates along with their usable Web services.

The SDC graph is created by the subsequent insertion of goal templates. In each insertion operation we add the necessary goal graph arcs and remove the redundant ones. For example, consider two goal templates G_1 and G_3 with $subsume(G_1, G_3)$ to be given, and a new goal template G_2 with $subsume(G_1, G_2)$ and $subsume(G_2, G_3)$ is added. Here, we add the new goal graph arcs (G_1, G_2) and (G_2, G_3), and remove the previously existing arc (G_1, G_3). This is done analogously for inserting a goal template under the other similarity degrees and also when a goal template is removed or modified. Therewith, every goal template will be allocated at the only possible position in the SDC graph, and the goal graph always defines a proper and non-redundant subsumption hierarchy.

The similarity degree $intersect(G_i, G_j)$ denotes that G_i and G_j have at least one common solution but there are also exclusive solutions for each. This is problematic because the direction of goal graph arcs for representing this is ambiguous, and – much more critical – the connection of multiple goal templates under the *intersect* similarity degree might cause cycles which disable the search properties of a SDC graph. To avoid this, we introduce an *intersection goal template* $G^I(G_i, G_j)$ that is defined as the logical conjunction of the functional descriptions of G_i and G_j (*cf.* clause (i) in Definition 5.2). The solutions for $G^I(G_i, G_j)$ are exactly those that are common for G_i and G_j, and thus it holds that $subsume(G_i, G^I(G_i, G_j))$ and $subsume(G_j, G^I(G_i, G_j))$. In consequence, the intersection goal template becomes a common child of G_i and G_j in the goal graph. We extend the subsumption hierarchy by defining two new arcs $(G_i, G^I(G_i, G_j))$ and $(G_j, G^I(G_i, G_j))$, which precisely reflects the semantic relationship of the requested functionalities. This is applied for every occurrence of an *intersect* similarity degree, so that eventually all similar goal templates are organized in a subsumption hierarchy. The existence of intersection goal templates determines the SDC graph to be a *directed acyclic graph* (short: DAG), because an intersection goal template by definition has two parent nodes and the goal graph can not contain any cycles [Bang-Jønsen and Gutin, 2000]. We shall formally define intersection goal templates and discuss their use within a SDC graph in more detail below.

Moreover, the SDC graph can contain subgraphs which are disconnected from each other. This results from the handling of goal templates with the similarity degree $disjoint(G_i, G_j)$ where we keep both G_i and G_j in the SDC graph, because each one is a potentially relevant objective description. Each of such disconnected goal templates might be associated with a set of similar goal templates, and so there can be several separate subgraphs as illustrated in Figure 5.3. In an application context, we can understand each subgraph to cover a specific problem domain, e.g. one for the shipment scenario and another one for flight ticketing. This does not contradict with the definition of the SDC graph, because a DAG does not need to be completely connected and also can have several root nodes.

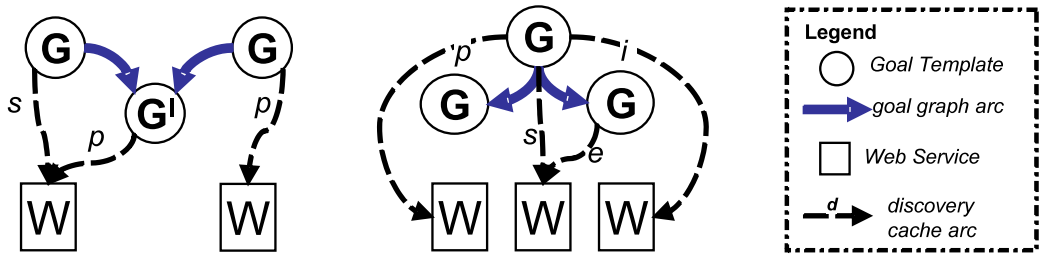


Figure 5.3: Disconnected Sub-Graphs in the SDC graph

Creation of the Discovery Cache. We now explain the discovery cache in more detail. This captures the knowledge on the usability of the available Web services, and is defined by the minimal set of arcs that are necessary to determine the usability degree of each Web service for every existing goal template (*cf.* clause (iv) in Definition 5.2).

For the optimization of Web service discovery operations, we must know the precise usability degree of every Web service W for each goal template G. This is explicated in the discovery cache by directed arcs of the form (G, W) which are annotated with the matching degree between G and W. However, in order to avoid redundancy, we omit discovery cache arcs for which the usability degree can be inferred from the SDC graph. Following from the inference rules 3.1 and 3.2 in Theorem 5.1, it holds that if $\mathit{subsume}(G_i, G_j)$, then the usability degree of a Web service W for the child G_j is always *plugin* if W is usable for the parent G_i under the degrees *exact* or *plugin* because the relationship of the possible executions of W and the solutions for G_i, G_j is $\{\mathcal{T}\}_W \supseteq \{\mathcal{T}\}_{G_i} \supset \{\mathcal{T}\}_{G_j}$. Thus, the arc (G_j, W) is not defined in the SDC graph, and so the discovery cache is kept minimal.

This relationship also holds for subsequent occurrences of the *subsume* similarity, which constitute the goal graph as explained above. If there are three goal templates such that $\mathit{subsume}(G_1, G_2)$ and $\mathit{subsume}(G_2, G_3)$ and a Web service W with $\mathit{plugin}(G_1, W)$, then the only possible usability degree of W for G_2 as well as for G_3 is also *plugin* and thus the arcs (G_2, W) and (G_3, W) are both omitted. In consequence, we can omit every discovery cache arc that connects a child of a goal template G in the goal graph with a Web service W whose usability degree for G is either *exact* or *plugin*. Figure 5.4 illustrates this for the example scenario discussed above in Section 5.1.2. The minimality of the discovery cache is not mandatory with respect to the formal properties of the SDC graph, but greatly eases its manageability in dynamically changing environments.

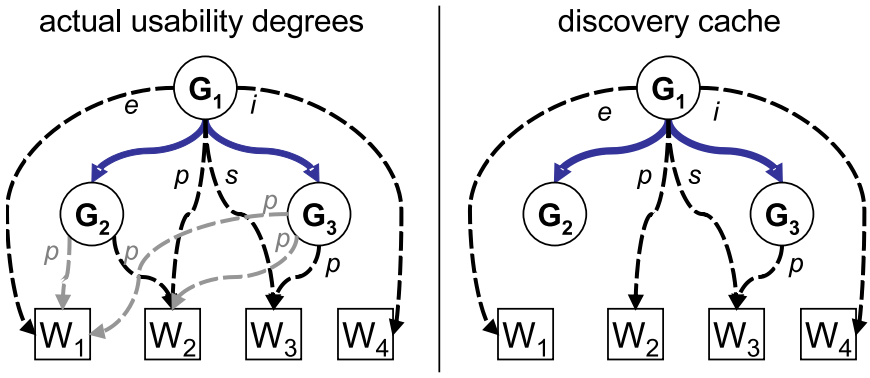


Figure 5.4: Minimality of Discovery Cache

Intersection Goal Templates

The following explains the concept of intersection goal templates in more detail. As outlined above, an intersection goal template describes the solutions which are common for two goal templates under the *intersect* similarity degree, and it is added as a child node of both original goal templates in the goal graph. This is necessary in order to organize all non-disjoint goal templates in a proper subsumption hierarchy, and therewith warrant the desirable search properties of the SDC graph. We commence with the formal definition of intersection goal templates, and then explain how these allow us to avoid cycles as well as other undesirable situations in the SDC graph.

Definition 5.3. *Let G_i and G_j be goal templates, and let their semantic similarity degree be intersect(G_i, G_j). Let $\mathcal{D}_{G_i} = (\Sigma^\star, \Omega^\star, IN_{G_i}, \phi^{\mathcal{D}_{G_i}})$ be the functional description of G_i, and let $\mathcal{D}_{G_j} = (\Sigma^\star, \Omega^\star, IN_{G_j}, \phi^{\mathcal{D}_{G_j}})$ be the functional description of G_j.*
We define $G^I(G_i, G_j)$ as the intersection goal template of G_i and G_j such that

$$\mathcal{D}_{G^I(G_i,G_j)} = (\Sigma^\star, \Omega^\star, IN_{G_i} \cup IN_{G_j}, \phi^{\mathcal{D}_{G_i}} \wedge \phi^{\mathcal{D}_{G_j}}).$$

This defines an intersection goal template $G^I(G_i, G_j)$ as the logical conjunction of the functional descriptions of the original goal templates G_i and G_j. Defined over the FOL structure for representing functional descriptions where $\phi^{\mathcal{D}} := [\phi^{pre}]_{\Sigma_D \rightarrow \Sigma_D^{pre}} \Rightarrow \phi^{\mathit{eff}}$, the $\Sigma^\star$-interpretations that are a model of $\mathcal{D}_{G^I(G_i,G_j)}$ are exactly the common models for $\mathcal{D}_{G_i}$ and $\mathcal{D}_{G_j}$. In accordance to Proposition 4.1 in Section 4.2.2, this means that the abstract solutions for $G^I(G_i, G_j)$ that are described by the logical models of $\mathcal{D}_{G^I(G_i,G_j)}$ are exactly those solutions that are common for G_i and G_j, i.e. $\{\mathcal{T}\}_{G^I(G_i,G_j)} = \{\mathcal{T}\}_{G_i} \cap \{\mathcal{T}\}_{G_j}$.

In fact, the abstract solutions for $G^I(G_i, G_j)$ are the solution of G_i and G_j that constitute the similarity degree $\mathit{intersect}(G_i, G_j)$. It thus holds that $\mathit{subsume}(G_i, G^I(G_i, G_j))$ and $\mathit{subsume}(G_j, G^I(G_i, G_j))$. In consequence, the Web services that are usable to solve $G^I(G_i, G_j)$ are exactly those that are usable for both G_i and G_j. Recalling from Definition 4.3, a match between a goal template G and a Web service W is given if $\exists \mathcal{T}.\ \mathcal{T} \in (\{\mathcal{T}\}_G \cap \{\mathcal{T}\}_W)$. Thus, it holds that $\mathit{match}(G^I(G_i, G_j), W) \Leftrightarrow \mathit{match}(G_i, W) \wedge \mathit{match}(G_j, W)$ (see Section 4.1.2). Note that $G^I(G_i, G_j)$ considers the union of the input variables defined for G_i and for G_j: these must be semantically equivalent because otherwise the condition for $\mathit{intersect}(G_i, G_j)$ can not be satisfied. Intersection goal templates are only meaningful under the *intersect* similarity degree. Under $\mathit{subsume}(G_i, G_j)$ it holds that $\{\mathcal{T}\}_{G_i} \supset \{\mathcal{T}\}_{G_j}$ and in consequence $\{\mathcal{T}\}_{G^I(G_i,G_j)} = \{\mathcal{T}\}_{G_j}$. The same holds analogously for the *plugin* and the *exact* similarity degrees, and if $\mathit{disjoint}(G_i, G_j)$ then $\{\mathcal{T}\}_{G^I(G_i,G_j)} = \emptyset$.

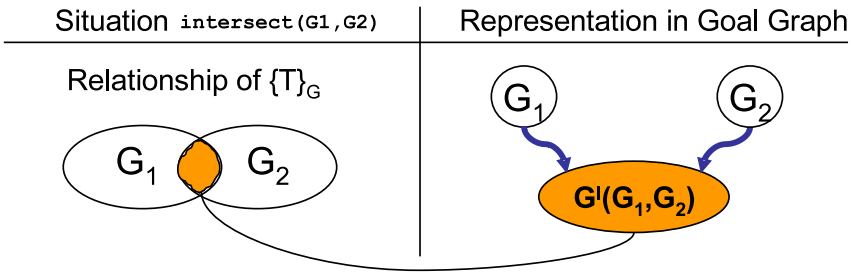


Figure 5.5: Example for an Intersection Goal Template

On this basis, we represent the situation where $intersect(G_1, G_2)$ in the goal graph as illustrated in Figure 5.5: we add the intersection goal template $G^I(G_1, G_2)$ along with two arcs $(G_1, G^I(G_1, G_2))$ and $(G_2, G^I(G_1, G_2))$ which denote the subsumption hierarchy. In our running example, consider a goal template G_1 for shipment in Germany for packages of any weight, and a goal template G_2 for shipment in Europe for packages with a maximal weight of 50 kg. Their similarity degree is *intersect* because shipping a package of 62.5 kg from Munich to Berlin is a solution for G_1 but not for G_2. The intersection goal template $G^I(G_1, G_2)$ describes the objective of shipping packages in Germany with a maximal weight of 50 kg. We then can treat $G^I(G_1, G_2)$ as a conventional goal template for search and discovery in the SDC graph. Its functional description does not have to be materialized because we can reason on it via the functional descriptions of the original goal templates.

Avoidance of Cycles. The major merit of intersection goal templates as defined above is that they allow us to properly handle problematic situations that might occur among semantically similar goal templates, in particular to avoid potential cycles in the SDC graph. The following explains this on the basis of several situations, and shows how they can be properly handled by adding intersection goal templates to the goal graph. We here content ourselves with informal discussions that appear to be sufficient for this purpose.

We consider situations where the similarity of goal templates does not denote a proper subsumption relationship to be undesirable, because this can not be represented straightforward in the SDC graph without hampering its desirable search properties. This can only occur under the $intersect(G_i, G_j)$ similarity degree which denotes that G_i and G_j have at least one common solution but there are also exclusive solutions for each; under all other similarity degrees we can unambiguously represent the subsumption relationship of the requested functionalities in the SDC graph. Figure 5.6 illustrates such situations, and shows their representation in the goal graph on the basis of intersection goal templates.

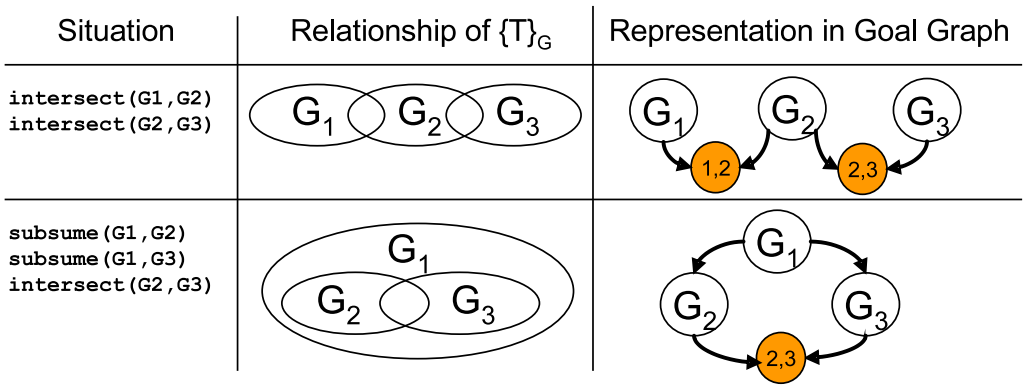


Figure 5.6: Intersection Goal Templates in the Goal Graph

The first situation considers three goal templates G_1, G_2, G_3 whose similarity degrees are $intersect(G_1, G_2)$ and $intersect(G_2, G_3)$. Here, it is not possible to unambiguously define directed arcs in the goal graph. For representing $intersect(G_1, G_2)$, we could define the arc (G_1, G_2) but also the arc (G_2, G_1). The same problem occurs for representing $intersect(G_2, G_3)$. To avoid the risk of unambiguity, we represent the relationships between the three goal templates by the intersection goal templates $G^I(G_1, G_2)$ and $G^I(G_2, G_3)$. In the second situation shown in the figure, we analogously represent the relation between two non-disjunct children G_2, G_3 of a goal template G_1 by defining their intersection goal template $G^I(G_2, G_3)$. This allows us to better reduce the search space for runtime Web service discovery. A goal instance that satisfies the goal instantiation condition for all three goal templates is also a consistent instantiation of the intersection goal template $G^I(G_2, G_3)$. In consequence, we only need to investigate the usable Web services for $G^I(G_2, G_3)$, which by definition are those that are commonly usable for G_2 and G_3.

We now turn towards the avoidance of cycles in the goal graph. In contrast to the situations discussed above, a cycle effectively disables the search properties because a search algorithm can run into infinite loops. Figure 5.7 below illustrates three examples for situations that would cause cycles in the goal graph: (1) if three or more goal templates are concatenated via *intersect* similarity degrees and there is no common solution for them, (2) if three goal templates ore more are concatenated via *intersect* similarity degrees and there is at least common solution for them, and (3) if there are three goal templates with one *subsume*- and two *intersect* similarity degrees. We can avoid each of these as well as all other potential cycles in the goal graph by adding the respective intersection goal templates with a similar effect as discussed above. The figure shows the patterns for avoiding the cycles by intersection goal templates; although this does not cover all possible situations,

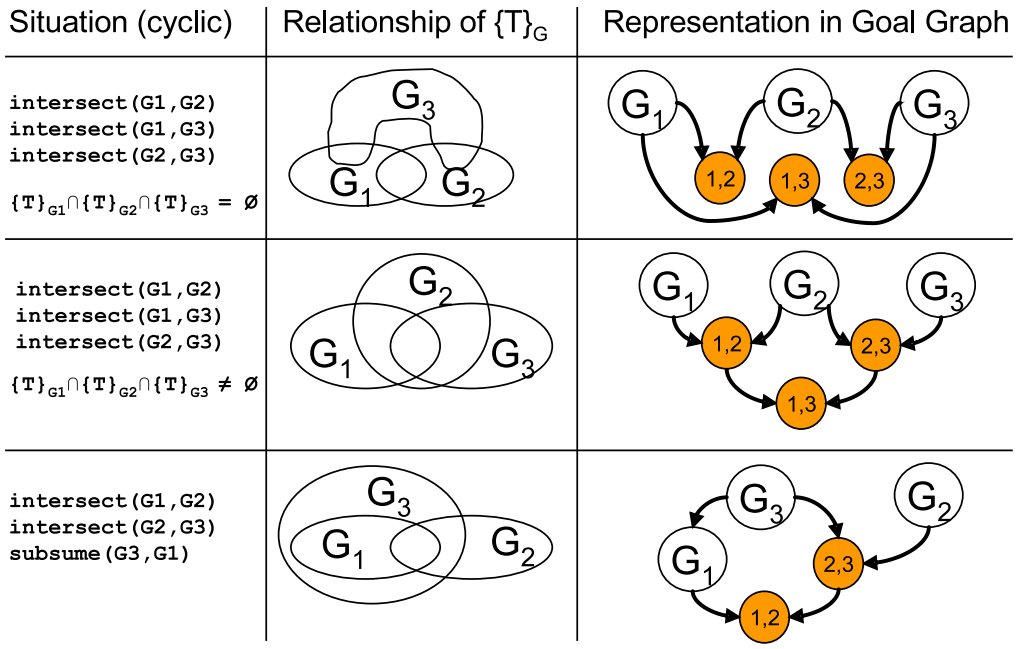


Figure 5.7: Avoidance of Cycles in the Goal Graph

the resulting patterns for resolving other potential cycles are analog. In situations (2) and (3), the intersection goal template that describes the common solutions of all involved goal templates becomes a child node at the lowest level of the goal graph. To obtain this goal graph structure, we need to iteratively check for new occurrences of the *intersect* similarity degree and resolve them accordingly. We explain this for an illustrative example.

Illustrative Example. The following exemplifies the avoidance of a cycle on the basis of intersection goal templates and the iterative refinement of the goal graph. This illustrates the basic principles of the algorithms for the creation and maintenance of the SDC graphs that we shall specify in detail below in Section 5.3.

In order to show that the SDC technique is also applicable for other scenarios, we here consider an example from the best-restaurant-search scenario that we have discussed in detail in Chapter 4. Let us consider three goal templates: G_1 for finding the best restaurant in an Austrian city, G_2 for finding the best French restaurant in any city of the world, and G_3 for finding the best restaurant in a European city. Their similarity degrees are $intersect(G_1, G_2)$, $intersect(G_2, G_3)$, and $subsume(G_3, G_1)$. This situation corresponds to the third type of cycles in Figure 5.7. To reach the representation pattern, we iteratively resolve the occurrences of *intersect* similarity degrees in a top down-top manner.

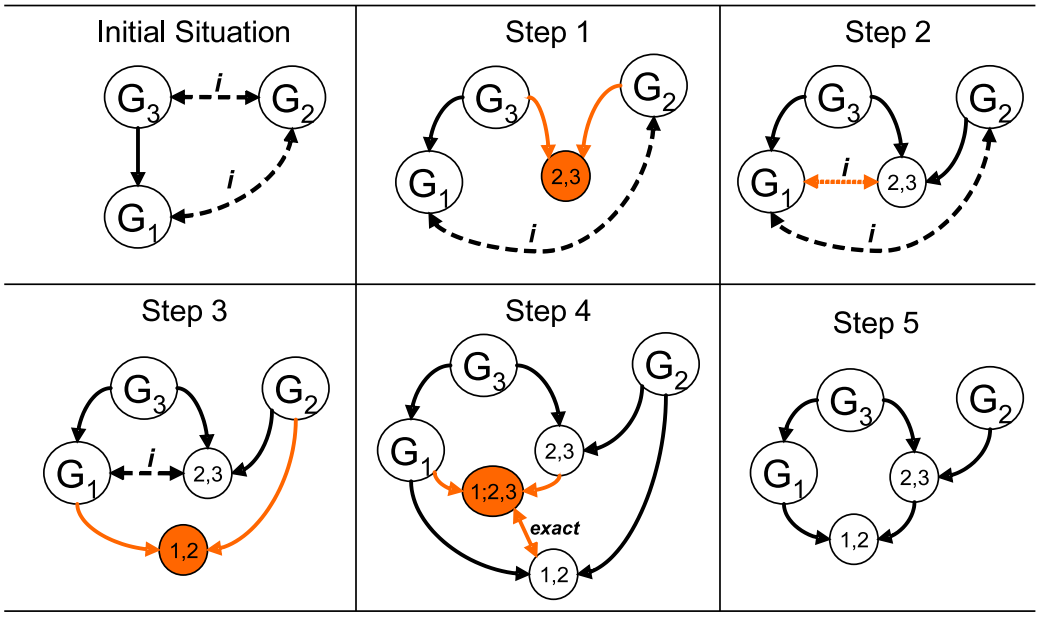


Figure 5.8: Example for Avoiding Cycles in the Goal Graph

Figure 5.8 shows the relevant steps. We commence with $intersect(G_2, G_3)$ as the top-most occurrence of the *intersect* similarity degree. The resulting intersection goal template $G^I(G_2, G_3)$ describes the objective of finding the best French restaurant in a European city, and becomes a child node of G_2 and of G_3 (*cf.* Step 1). This results in a new occurrence of the *intersect* similarity degree because now $intersect(G_1, G^I(G_2, G_3))$, i.e. G_1 and the new node $G^I(G_2, G_3)$ are non-disjoint child nodes of G_3 (*cf.* Step 2). Before handling this, we first resolve $intersect(G_1, G_2)$ as the top-most occurrence at this stage. The resulting intersection goal template $G^I(G_1, G_2)$ describes the objective of finding the best French restaurant in an Austrian city, and becomes a child node of both G_1 and G_2 (*cf.* Step 3).

We now address the new occurrence, i.e. $intersect(G_1, G^I(G_2, G_3))$. The resulting intersection goal template would describe the objective of finding the best French restaurant in an Austrian city. We observe that this is semantically equivalent with the already existing intersection goal template $G^I(G_1, G_2)$: both describe the solutions which are common to all three original goal templates, and there similarity degree is *exact* (*cf.* Step 4). Under this similarity degree, we only keep one goal template while the other one is redundant for the purpose of the SDC technique (see Section 5.2.2). Here, we keep the already existing intersection goal template $G^I(G_1, G_2)$ while the newly created, third intersection goal template is not kept (*cf.* Step 5). We therewith finally obtain the representation pattern for the cyclic situation in the goal graph as outlined above in Figure 5.7.

5.2.3 Properties of the SDC Graph

After having defined the SDC graph along with the constructs and principles for obtaining this for a given set of goal templates and Web services, we now discuss its relevant properties as a search index for optimizing Web service discovery. In particular, we show that the SDC graph holds all relevant knowledge that is relevant for optimizing discovery operations in a concise manner, and we analyze the computational complexity for creating a SDC graph as well as its search properties for optimizing runtime discovery operations.

Inferential Completeness and Minimality

The first aspect is concerned with the correctness and conciseness of the SDC graph, i.e. that it keeps all relevant knowledge in an unambiguous and non-redundant manner. Recalling from Definition 5.2, the goal graph organizes all existing goal templates in a proper subsumption hierarchy such that the only occurring similarity degree is *subsume* and redundant arcs do not exist, and the discovery cache captures the minimal knowledge on the functional suitability of every available Web service for each existing goal template.

This structure ensures that the SDC graph satisfies two properties that appear to be relevant for its application purpose. The first one is that from the SDC graph when can infer (a) the precise semantic similarity between every pair of goal templates, and (b) the precise usability degree of each Web service for every existing goal template. This is a pre-requisite for optimizing Web service discovery operations: the knowledge about the semantic similarity of goal templates in necessary to effectively reduce the search space, and the knowledge of the precise usability degrees of Web service is required for optimizing the discovery operations, in particular at runtime. We thus refer to this as the *inferential completeness* of the SDC graph for the context of Web service discovery. The second property is that the SDC graph only keeps the minimal necessary knowledge: every arc in the SDC graph is necessary to establish the inferential completeness, and every additional arc that describes a correct relationship between the goal template and Web services is redundant because it can be inferred from the existing arcs. We call this the *inferential minimality* of the SDC graph, which greatly eases its maintainability.

The following defines this formally. We here consider $d(G_i, G_j)$ and $d(G, W)$ as atoms that can be either `true` or `false` and denote the similarity degree of goal templates, respectively the usability degree of a Web service for a goal template where the possible matching degree d is either *exact,plugin,subsume,intersect*, or *disjoint*. A_{SDC} is the set of such atoms that is explicitly defined in the SDC graph, and $cl^{\star}(A_{SDC})$ is its deductive closure that encompasses all atoms can be deduced by the inference rules $\mathcal{IR}$ defined in Theorem 5.1.

Theorem 5.2. *Let $\mathcal{G}$ be a set of goal templates, let $\mathcal{W}$ be a set of Web services, and let $SDC = (V_{\mathcal{G}} \cup V_{\mathcal{W}}, E_{sim} \cup E_{use})$ be the SDC graph over $\mathcal{G}, \mathcal{W}$. Let $A_{SDC} = \{d(x,y) | d \in \{\mathsf{exact,plugin,subsume,intersect}\}, x \in \mathcal{G}, y \in (\mathcal{G}, \mathcal{W}), (x,y) \in (E_{sim}, E_{use})\}$ be all the atoms defined in SDC, let $\mathcal{IR}$ be the set of all inference rules that hold between $d(x,y)$, and let $cl^{\star}(A_{SDC}) = \{d(x,y) | A_{SDC} \wedge \mathcal{IR} \models d(x,y)\}$ be the deductive closure of A_{SDC} over $\mathcal{IR}$.*

The SDC graph is inferentially complete and minimal such that:

(i) $d(x,y) \in cl^{\star}(A_{SDC})$ *if and only if* $d(x,y)$ *is* true *for* $\mathcal{G} \times \mathcal{W}$*, and*

(ii) for all $d(x,y) \in A_{SDC} : cl^{\star}(A_{SDC} \setminus d(x,y)) \subset cl^{\star}(A_{SDC})$.

Referring to Appendix B.2 for the proof, this states that the SDC graph defines all relevant knowledge on the semantic similarity of goal templates and the usability of the available Web services in a concise and redundance-free manner. Clause (i) states that the inferential completeness is given because the SDC graph arcs together with the knowledge that can be inferred by the rules defined in Theorem 5.1 correctly describe all relevant relations between the goal templates and the Web services, and clause (ii) states that this completeness would be disabled when a single arc is removed from the SDC graph.

Complexity and Search Properties

We now turn towards the computational complexity of the SDC graph. We commence with analyzing the complexity for the automated creation and maintenance of the SDC graph, which covers the operations carried out at design time in our two-phased discovery framework. Then, we discuss the general properties of the SDC graph as a search index for optimizing the Web service discovery task at runtime.

SDC Graph Management. The SDC graph is created and maintained at design time, i.e. whenever a goal template or a Web service is added, removed, or modified. As explained above, the SDC graph is created by the subsequent insertion of goal templates. This denotes the central operation for which we need to insert the new goal template into the goal graph and perform the necessary design time discovery runs to create the discovery cache. We thus focus on the insertion of single goal templates for the complexity analysis; all other management operations on SDC graphs require significantly less computational effort.

A new goal template G_{new} is inserted into the SDC graph by first allocating it at the only possible position in the goal graph, and then update the discovery cache so it properly captures the knowledge on the usability of the available Web services for G_{new} as well as for all previously existing goal templates. The base operation for this is semantic matchmaking on the goal template level, for determining both the semantic similarity of goal templates as

well as the usability degree of a Web service. If this is decidable, then the complexity of a single matchmaking operation is `NExpTime-complete` (*cf.* Proposition 4.3 in Section 4.3.1). We can distinguish two situations with respect to the number of necessary matchmaking operations: (1) when G_{new} becomes a new root node of the SDC graph we have to inspect the usability of all available Web services, while (2) in all other situations we only need to inspect the Web services which are usable for the previously existing goal templates.

For the first situation, we need to first ensure that there does not exist any root node G_{root} in the SDC graph where $subsume(G_{root}, G_{new})$, because otherwise G_{new} would become a child node of G_{root}. Then, we need to inspect the usability of all existing Web services for G_{new}, because under the *disjoint* similarity degree we can not infer relevant knowledge on this from the SDC graph (*cf.* rules 5.1 - 5.5 in Theorem 5.1). In all other situations, we first need to allocate G_{new} at the appropriate position within the subsumption hierarchy of the already existing goal templates. For this, we need to first find the existing root node where $subsume(G_{root}, G_{new})$ which indicates the root of the SDC subgraph wherein G_{new} shall be inserted. Then, we need to stepwise traverse the goal graph by checking the semantic similarity with the already existing goal templates until reaching the appropriate position for inserting G_{new}. Note that by definition there can only be one of the possibly separated subgraphs where $subsume(G_{root}, G_{new})$, and there also is only one possible position for every goal template in the SDC graph. Finally, we need to determine the usability of only those Web services that are usable under the *subsume* or the *intersect* degree for the goal templates G_p that now are parents of G_{new} (*cf.* clauses 3.1 - 3.5 in Theorem 5.1).

The following defines this formally. We use the following standard notions from graph theory: the diameter of the goal graph $diam(SDC_{GG})$ is the maximal distance between a root and a leaf node, and the branching factor $b(SDC_{GG})$ denotes the maximal number of children of a goal template in the goal graph [Diestel, 2005].

Proposition 5.1. *Let $SDC = (V_{\mathcal{G}} \cup V_{\mathcal{W}}, E_{sim} \cup E_{use})$ be the SDC graph for a set $\mathcal{G}$ of goal templates and a set $\mathcal{W}$ of Web services. Let $SDC_{GG} = (V_{\mathcal{G}}, E_{sim})$ be the goal graph with the diameter $diam(SDC_{GG})$ and the branching factor $b(SDC_{GG})$. Let $\mathcal{G}_{root} \subseteq V_{\mathcal{G}}$ be the set of root nodes of SDC, and let $\mathcal{W}_G \subseteq \mathcal{W}$ be the usable Web services for a goal template G.*

The computational costs for inserting a new goal template G_{new} to SDC are

- *(i)* $O(|\mathcal{G}_{root}| + |\mathcal{W}|)$ *when G_{new} becomes a new root node of SDC, otherwise*
- *(ii)* $O(|\mathcal{G}_{root}| + (diam(SDC_{GG}) * b(SDC_{GG})) + |\mathcal{W}_{G_p}|)$ *with* $(G_p, G_{new}) \in E_{sim}$ *and* $d(G_p, W) \in \{subsume, intersect\}$

where the complexity of a single matchmaking operation is `NExpTime-complete` if the proof obligation remains in the Bernays–Schönfinkel fragment, undecidable otherwise.

The complexity for the creation of SDC graphs is relatively high. This corresponds to our two-phased approach wherein expectably expensive operation are performed at design time, i.e. orthogonal to the time critical discovery of suitable Web services for a goal instance at runtime. Besides, we expect than in most application scenarios there are only a few root nodes of the SDC graph so that the expensive goal template insertion operation is rather rare, and also that both $diam(SDC_{GG}), b(SDC_{GG}) \ll |\mathcal{G}|$ and usually $|\mathcal{W}_G| \ll |\mathcal{W}|$ so that the insertion operation for new child nodes is significantly less expensive.

Search Properties. We now discuss the properties of the SDC graph as a search index for goal templates and Web services. This mainly relates to its primary usage purpose for optimizing the discovery of suitable Web services for a goal instance at runtime.

As outlined above, the approach for enhancing the computational performance of the runtime discovery task is to minimize the necessary matchmaking operations. For a given goal instance $GI(G, \beta)$, we first reduce the relevant search space by determining the most appropriate goal template G' that is a (possibly indirect) child of the originally defined corresponding goal template G, and of which the goal instance is a valid instantiation. For this, we subsequently traverse the SDC graph by checking the goal instantiation condition, and then inspect the suitability of those Web services that are usable for G' on the basis of the integrated matchmaking conditions defined in Theorem 4.1 (see Section 4.3.2).

The following defines the computational costs for this formally, while we shall specify the algorithms for the optimized runtime discovery below in Section 5.4. Note that by definition there can only be one most appropriate goal template G' for a goal instance, and there always is at least one path $G \rightarrow \ldots \rightarrow G'$ in the SDC graph where $GI(G, \beta)$ is a valid instantiation of every goal template. The base operation for this search are the matchmaking techniques on the goal instance level as defined in Section 4.3.2.

Proposition 5.2. *Let $SDC = (V_{\mathcal{G}} \cup V_{\mathcal{W}}, E_{sim} \cup E_{use})$ be the SDC graph for a set $\mathcal{G}$ of goal templates and a set $\mathcal{W}$ of Web services. Let $SDC_{GG} = (V_{\mathcal{G}}, E_{sim})$ be the goal graph with the diameter $diam(SDC_{GG})$ and the branching factor $b(SDC_{GG})$. Let $GI(G, \beta) \models G$ be a goal instance where $G, G' \in V_{\mathcal{G}}$ with $subsume(G, G')$ and $GI(G, \beta) \models G'$, and let $\mathcal{W}_G \subseteq \mathcal{W}$ be the usable Web services for a goal template G.*

The costs for finding a suitable Web service W for a goal instance $GI(G, \beta)$ in SDC are

$$O(\ (diam(SDC_{GG}) * b(SDC_{GG})) + |\mathcal{W}_{G'}| \text{ where } d(G', W) \in \{subsume, intersect\}\ )$$

and the complexity of the necessary matchmaking operations is `NExpTime` *if the functional descriptions $[\mathcal{D}_G]_\beta, [\mathcal{D}_W]_\beta$ instantiated with the β defined in $GI(G, \beta)$ do not contain function symbols and have a $\exists^\star\forall^\star$ quantifier prefix when written in prenex normal form.*

5.3 Management and Maintenance

This section specifies the techniques for managing and maintaining SDC graphs, which covers the automated creation as well as the handling of changes in evolving environments. In order to warrant the operational correctness of the SDC graph at all times, we need to ensure that its structure and properties are maintained in all possible situations when a goal template or a Web services is added, removed, or modified.

The following defines the necessary algorithms for this, which follow the principles explained above. We commence with the creation of the SDC graph. This is realized by the subsequent insertion of new goal templates such that after each insertion the SDC exposes the structure and properties as defined above. We explain this central operation in detail, and illustrate the algorithm for a comprehensive example from the shipment scenario. We then define the algorithms for managing changes on the existing goal templates and the available Web services, which provide the basic evolution support for the SDC technique. We here focus on the design and the functional correctness of the algorithms, while we shall explain the technical realization below in Section 5.5.

The creation and maintenance algorithms provide the basic management facilities for the SDC graph. We amend this with discussing extensions and complementing techniques, in particular the automated generation of additional goal templates and the integration with other SWS techniques. All management and maintenance operations are performed at design time, i.e. orthogonal to the discovery of suitable Web services for goal instances at runtime, and thus are considered to be not time critical in our two-phased framework.

5.3.1 SDC Graph Creation

As outlined above, we realize the automated creation of the SDC graph via the subsequent insertion of goal templates. Each new goal template is first inserted into the goal graph, and then the necessary arcs are added to the discovery cache. We assume that in real-world scenarios the goal templates are specified in a stepwise manner so that the SDC graph grows successively. However, we can also generate the SDC graph for a given set of goal templates and Web services by subsequently inserting the individual goal templates.

Figure 5.9 provides an overview of the algorithm for this in terms of a flow chart, which follows the principles and constructs for creating SDC graphs as explained above in Section 5.2.2. The first part is concerned inserting a new goal template G_{new} at the proper position in the goal graph. We commence with inspecting the semantic similarity of G_{new} and already existing goal templates which are root nodes in the SDC graph. If there is a

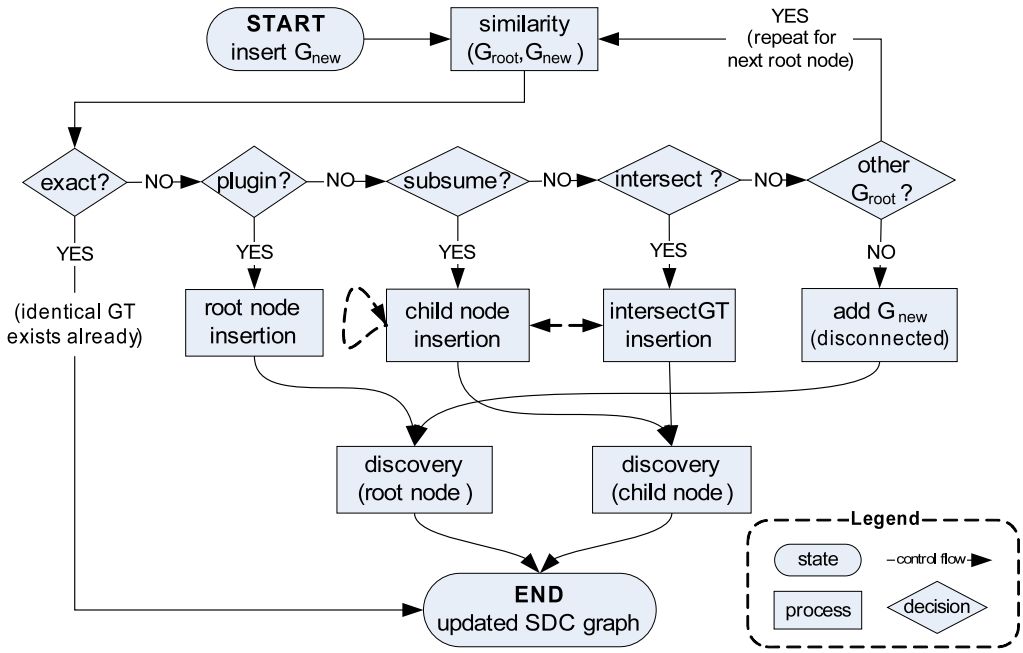


Figure 5.9: Overview of SDC Graph Creation Algorithm

root node G_{root} for which the similarity degree is not *disjoint*, then G_{new} will be allocated in the subgraph of G_{root}. There can only be one potential target subgraph because by definition the goal templates in disconnected subgraphs are not semantically similar. We then insert G_{new} under the possible similarity degrees as follows:

- if $exact(G_{new}, G_{root})$ then G_{new} is not added to the SDC graph because a semantically equivalent goal template already exists
- if $plugin(G_{new}, G_{root})$ then G_{new} becomes a new root node of the subgraph of G_{root}; the goal graph update is handled by the *rootNodeInsertion* sub-procedure
- if $subsume(G_{new}, G_{root})$ then G_{new} becomes a new child node in the subgraph of G_{root}; this is handled by the iterative sub-procedure *childNodeInsertion*
- if $intersect(G_{new}, G_{root})$ then we represent this by the respective intersection goal template in the goal graph (*cf.* Definition 5.3); all occurrences of the *intersect* similarity degree are handled by the sub-procedure *intersectGTInsertion*
- if a semantically similar G_{root} does not exists, then G_{new} is inserted as a disconnected node so that it constitutes a new separated subgraph.

The second part of the algorithm is concerned with the creation of the discovery cache, i.e. with adding the arcs which are necessary to deduce the precise usability degree of every available Web service for G_{new} by capturing design time discovery results. Every newly inserted goal template is either a root or a child node in the goal graph: a disconnected goal template can be consider to be a root node without children, and intersection goal templates are child nodes by definition. We thus merely need to distinguish two sub-procedures for the discovery cache creation: *rootNodeDiscovery* detects the suitable Web services and updates the discovery cache when G_{new} is a root node, and *childNodeDiscovery* handles the case when G_{new} has been inserted as a child node. For this, we can make use of the inference rules defined in Theorem 5.1 in order to reduce the necessary matchmaking effort.

After each run of the algorithm, the resulting SDC graph must expose the structure and properties from Definition 5.2, i.e. (1) G_{new} is allocated at the correct position and there are no redundant arcs in the goal graph, and (2) the extended discovery cache defines the minimal set of arcs that is necessary to determine the precise usability degree of every available Web service for G_{new} and for all other goal templates. This is ensured by the sub-procedures which we shall specify in detail in the following.

Listing 5.1 shows the overall control procedure of the goal template insertion algorithm in Java-style pseudo code. Initially, the SDC graph is empty and G_{new} becomes the first goal template; the subsequent insertions are performed as explained above. We use the following elements: `sdcgraph` is the complete SDC graph, and `goalgraph` is its goal graph. The function *position(G)* denotes the position of a goal template G in the goal graph which can have two values: `root` if G does not have any incoming arcs, and `child` otherwise. Besides, `disjointFlag` is a local boolean constant for the internal status management.

```
insert (Gn){
  if ( goalgraph = empty ) then { goalgraph = + Gn; discoveryCacheCreation(Gn); }
  else { forall ( G2 and position(G2) = root ) {
          if ( exact(Gn,G2) ) then { disjointFlag = false, return sdcgraph};
          else if ( plugin(Gn,G2) ) then { rootNodeInsertion(Gn,G2); discoveryCacheCreation(Gn);
               disjointFlag = false}
          else if ( subsume(Gn,G2) ) then { childNodeInsertion{Gn,G2}; discoveryCacheCreation(Gn);
               disjointFlag = false}
          else if ( intersect (Gn,G2) ) then { goalgraph =+ Gn; intersectGTInsertion{Gn,G2};
              discoveryCacheCreation(Gn); disjointFlag = false}
          else { disjointFlag = true; } }
       if ( disjointFlag = true) then { goalgraph = + Gn; discoveryCacheCreation(Gn); }
  }
  return sdcgraph;
}
```

Listing 5.1: Algorithm for Goal Template Insertion

Sub-Procedures for Goal Graph Creation

We now define the sub-procedures for inserting G_{new} into the goal graph. To warrant the required structure and properties of the resulting goal graph, these procedures need to ensure that (1) in all possible situation G_{new} is properly inserted at the only possible position in the goal graph, (2) all occurrences of the *intersect* similarity degree are properly resolved, and (3) all redundant goal graph arcs that may result from the insertion are removed.

The following specifies the three sub-procedures defined in Listing 5.1, and shows that they satisfy the requirements. These operations require semantic matchmaking in order to determine the similarity degree of goal templates as defined in Table 5.2 (see Section 5.2.1). This can technically be realized analogously to the matchmaking techniques for Web service discovery as explained in Section 4.4. We only perform matching operations when necessary in order to minimize the computational costs of the SDC graph creation algorithm.

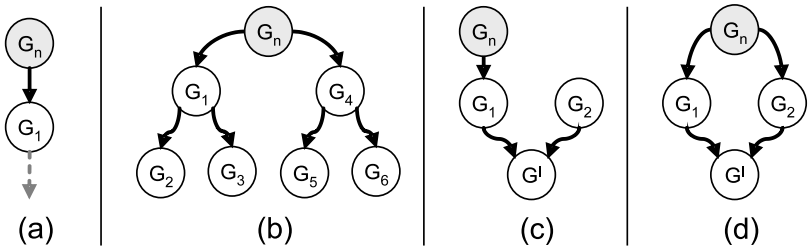
Insertion of a New Root Node. We commence with the sub-procedure *rootNodeInsertion*, which inserts G_{new} as a new root node of a previously existing subgraph. This is invoked from the main control procedure when the similarity degree of G_{new} and an existing root node G_{root} is *plugin*. In this case, essentially G_{new} becomes a new root node of the respective subgraph with G_{root} as a direct child. Listing 5.2 shows the algorithm for this: we add G_{new} to the goal graph, and a new arc (G_{new}, G_{root}) that defines the subsumption relationship. Although this algorithm is relatively simple, it can properly handle all possibly occurring situations as we shall discuss below.

```
rootNodeInsertion{Gn,Gr} {
 goalgraph = + Gn + (Gn,Gr);
 return goalGraph;
}
```

Listing 5.2: Root Node Insertion Algorithm

Table 5.3 shows four possible situations for the insertion of G_{new} if the similarity degree is $plugin(G_{new}, G_{root})$. In case (a), G_{new} becomes the new root node of a subgraph that had only one root node beforehand. Here, the above algorithm obviously generates the correct structure of the goal graph; this also covers the case when G_{new} becomes the parent of a previously disconnected goal template. In case (b), G_{new} becomes the common root node of two previously disconnected subgraphs. This situation is also covered by the above algorithm because it is invoked for each existing root node whose similarity degree with G_{new} is *plugin* (see Listing 5.1). The other two cases show the insertion of G_{new} as a new root node of a subgraph that previously had two or more root nodes. This can

Table 5.3: Situations for New Root Node Insertion



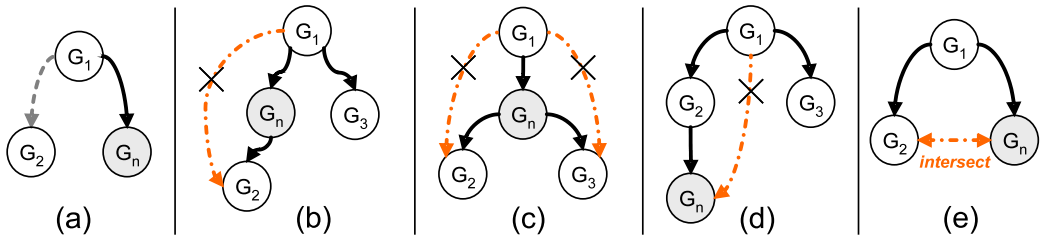
only occur if the similarity degree of the previous root nodes G_1, G_2 is *intersect*, which is represented in the goal graph by their intersection goal template $G^I(G_1, G_2)$. In case (c), G_{new} becomes the new parent of either G_1 or of G_2. In the figure, the similarity degrees are $plugin(G_{new}, G_1)$ and $intersect(G_{new}, G_2)$. The resolution of the latter situation does not result in an additional intersection goal template because the common solutions of G_1, G_2, and G_{new} are already described by $G^I(G_1, G_2)$. In case (d), G_{new} becomes the common root node of the subgraph. This is supported by the algorithm analogously to case (b).

The lower levels of existing subgraphs are not changed if G_{new} becomes a new root node. Thus, we do not need to investigate them. Moreover, every separated subgraph with more than one root node must contain at least one intersection goal template. In consequence, all its root nodes are non-disjoint, and cases (c) and (d) are the only possible situations for inserting G_{new} as a new root node of such a subgraph. Thus, Table 5.3 covers all possible situations for inserting a new root node, and the algorithm specified above in Listing 5.2 handles all of them properly.

Insertion of a New Child Node. We now turn towards the insertion of G_{new} as a new child node into the goal graph. The sub-procedure *childNodeInsertion* is invoked if there exists a root node where $subsume(G_{root}, G_{new})$, so that it will be inserted as child node in respective subgraph. Listing 5.3 below specifies the algorithm for this, and Table 5.4 illustrates the relevant situations that must be handled.

The algorithm commences with adding G_{new} and the arc (G_{new}, G_p) to the goal graph. Here, G_p is the root node for which the similarity degree $subsume(G_{new}, G_p)$ has triggered the invocation from the main procedure in Listing 5.1. This is sufficient to handle case (a) as the basic situation where G_{new} becomes a new child node of a goal graph whose subsumption hierarchy only has two levels, and it also covers the situation when G_{new} becomes a child node of a previously disconnected goal template. In more complex goal graphs, we then have to inspect the relationship of G_{new} and the already existing direct child nodes G_c of

Table 5.4: Situations for Insertion of a New Child Node



G_p. If the similarity degree is $exact(G_{new}, G_c)$, then G_{new} is not kept in the goal graph because it is redundant; we thus remove the previously added elements. If the similarity degree is *plugin*, then G_{new} becomes an intermediate node between G_p and G_c. We here need to remove the arc (G_p, G_c) because it would be redundant in the goal graph. This relates to the cases (b) and (c), which are successively handled by iterations of the *forall*-loop. Case (d) is the situation where G_{new} becomes a child node of G_c, which occurs if $subsume(G_{new}, G_c)$. Here, we can inspect the next lower level of the subsumption hierarchy in a depth-first manner by iteratively invoking the *childNodeInsertion* algorithm. In each iteration, we remove the previously added arc (G_{new}, G_p). The last possible situation is case (e) where $intersect(G_{new}, G_c)$. This is handled by the *intersectGTInsertion* procedure which we shall define below. This algorithm ensures that eventually G_{new} is inserted at the only possible position without any redundant arcs in the goal graph, and also that all direct children of every goal template are disjoint.

```
childNodeInsertion (Gn,Gp) {
  goalgraph = + Gn + (Gp,Gn);
   forall ( Gc and (Gp,Gc) in goalgraph ) {
     if ( exact(Gc,Gn) ) then {
       goalgraph =− Gn − (Gp,Gn);
       return goalgraph; }
     else if ( plugin(Gc,Gn) ) then {
       goalgraph = + (Gn,Gc);
       goalgraph = − (Gp,Gc); }
     else if ( subsume(Gc,Gn) ) then {
       goalgraph = − (Gp,Gn);
       childNodeInsertion{G , Gc}; }
     else if ( intersect (Gc,Gn) ) then {
       intersectGTInsertion{Gn , Gc}; }
  }
  return goalGraph;
}
```

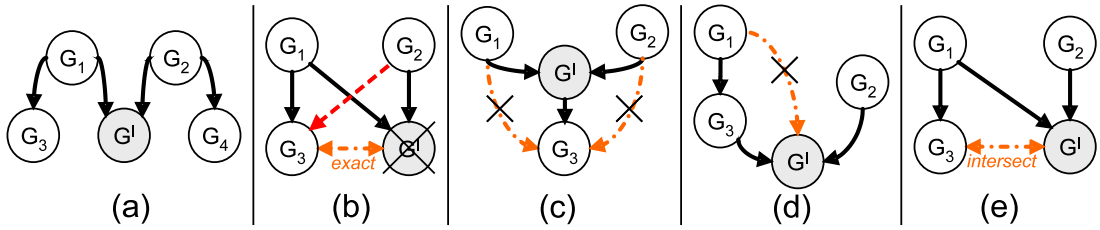
Listing 5.3: Child Node Insertion Algorithm

Handling the *Intersect* Similarity Degree. The last sub-procedure for creating the goal graph is concerned with handling the occurrences of the *intersect* similarity degree. We represent these situations by intersection goal templates in order to avoid the ambiguity as well as potential cycles in the goal graph. Given two goal templates G_1, G_2 such that $intersect(G_1, G_2)$, the intersection goal template $G^I(G_1, G_2)$ describes exactly the common solutions and thus becomes a child node of both G_1 and G_2 (*cf.* Definition 5.3).

Listing 5.4 below shows the algorithm for adding intersection goal templates into the goal graph, which is invoked by the previous sub-procedures whenever an *intersect* similarity degree occurs. Similar to the other algorithms, after adding $G^I(G_1, G_2)$ we must inspect and maybe further resolve the relationship between the intersection goal template and the other direct children of G_1 and G_2. Table 5.5 illustrates the relevant situations, and we explain how these are properly handled by the algorithm.

The algorithm commences with adding the intersection goal template to the goal graph as explained in Section 5.2.2. This allows us to handle case (a) as the situation where $G^I(G_1, G_2)$ becomes a child of G_1 and G_2 and is disjoint from the already existing children. The other cases distinguish situations where this is not given. These can occur if two goal templates already exist (e.g. G_1, G_3 with $subsume(G_1, G_3)$), and then another one is added where an *intersect* similarity degree occurs (e.g. G_2 with $intersect(G_1, G_2)$). If then the similarity degree between G_3 and $G^I(G_1, G_2)$ is *exact*, we do not keep the intersection goal template because it is redundant (case (b)). However, we need to add the arc $(G2, G3)$ because then G_3 describes the common solutions for G_1 and G_2. If $plugin(G_3, G^I(G_1, G_2))$, then G_3 becomes a child of the intersection goal template (case (c)). We here need to remove the previously existing incoming arcs of G_3. In case (d) where $subsume(G_3, G^I(G_1, G_2))$, the intersection goal template becomes a child of G_3. For this, we invoke the *childNodeInsertion* algorithm in order to properly handle the relationship to other existing children of G_3. Before that, we must remove the previously added arc $(G_1, G^I(G_1, G_2))$ because it would be redundant in the resulting goal graph. For case (e) where $intersect(G_3, G^I(G_1, G_2))$, we again invoke the *intersectGTInsertion* procedure.

Table 5.5: Situations for Insertion of an Intersection Goal Template



```
intersectGTInsertion (G1,G2) {
  goalgraph = + iGT;
  goalgraph = + (G1,iGT) + (G2,iGT);
  forall ( G3 != iGT and ( (G1,G3) or (G2,G3) ) in goalgraph ) {
    if ( exact(G3,iGT) ) then {
        goalgraph = − iGT − (G1,iGT) − (G2,iGT);
        if ( ! (G1,G3) in goalgraph ) then goalgraph = + (G1,G3);
        if ( ! (G2,G3) in goalgraph ) then goalgraph = + (G2,G3); }
    else if ( plugin(G3,iGT) ) then {
        goalgraph = + (G3,iGT);
        if ( (G1,G3) in goalgraph ) then goalgraph = − (G1,G3);
        if ( (G2,G3) in goalgraph ) then goalgraph = − (G2,G3); }
    else if ( subsume(G3,iGT) ) then {
        if ( (G1,G3) in goalgraph ) then goalgraph = − (G1,iGT);
        if ( (G2,G3) in goalgraph ) then goalgraph = − (G2,iGT);
        childNodeInsertion (iGT,G3); }
    else if ( intersect (G3,iGT) ) then iArcResolution (G3,iGT);
  }
  discoveryCacheCreation(iGT);
  return sdcgraph;
}
```

Listing 5.4: Intersection Goal Template Insertion Algorithm

This can handle all possible occurrences of the *intersect* similarity degree so that resulting goal graph organizes the existing goal templates in a proper subsumption hierarchy without redundant arcs. Also, potential cycles as well as all other undesirable situations are avoided as explained in Section 5.2.2. As the last step, we need to create the discovery cache for the newly inserted intersection goal template, which is not covered by the overall procedure because intersection goal templates are additional elements in the goal graph.

Sub-Procedures for Discovery Cache Creation

We now turn towards the second part of the SDC graph creation algorithm, which is concerned with creating the discovery cache. For this, we perform design time Web service discovery by semantic matchmaking of the functional descriptions of goal templates and Web services as defined in Section 4.3.1, and then capture the relevant knowledge in the SDC graph by the minimal set of discovery cache arcs which are necessary to deduce the precise usability degree of each available Web service for each goal template that exists in the goal graph. The following defines the necessary algorithms for this.

For optimizing discovery operations, it is necessary to know the precise usability degree of a Web service W for a goal template G. Listing 5.5 defines the basic algorithm for this. Initially, the usability degree is set to *disjoint*, which denotes that W is not usable

for G. Then, we check the conditions for the *plugin* and for the *subsume* degree as defined in Table 4.2, which can technically be realized as described in Section 4.4. The boolean constants `pluginFlag` and `subsumeFlag` keep the results, and on this basis we can decide whether the actual usability degree is *exact* because $plugin(G, W) \wedge subsume(G, W) \Leftrightarrow exact(G, W)$, *cf.* Proposition 4.2. If this is not given, we finally check the condition for the *intersect* usability degree and update the resulting degree accordingly.

```
matchmaking(G,W){
  pluginFlag = false;
  subsumeFlag = false;
  degree = disjoint;
  if ( plugin(G,W) ) then { pluginFlag = true; degree = plugin; }
  if ( subsume(G,W) ) then { subsumeFlag = true; degree = subsume; }
  if ( pluginFlag and subsumeFlag ) then degree = exact;
  else if ( intersect(G,W) ) then degree = intersect;
  return degree;
}
```

Listing 5.5: Matchmaking Algorithm

We now specify the algorithm for creating the discovery cache for a newly inserted goal template. This is invoked by the overall insertion algorithm after inserting G_{new} into the goal graph (see Listing 5.1), and also for a newly added intersection goal template (see Listing 5.4). According to Definition 5.2, the `discoverycache` consists of directed arcs between a goal template and a Web service that is annotated with the respective usability degree. To ensure the inferential completeness and minimality of the resulting SDC graph in accordance to Theorem 5.2, we need to extend and refine the previously existing discovery cache such that it contains the minimal set of arcs which are necessary to deduce the precise usability degree of every Web service for the new as well as for all other goal templates that exist in the goal graph. Listing 5.6 shows the control algorithm for this, which consists of three sub-procedures that we shall specify below in detail: *childNodeDiscovery* is used if G_{new} has been inserted as a child node in the goal graph and also when a new intersection goal template has been added, and *rootNodeDiscovery* is used otherwise; the third sub-procedure re-establishes the minimality of the extended discovery cache.

```
discoveryCacheCreation(G) {
  if ( position(G) = child ) then childNodeDiscovery(G);
  if ( position(G) = root ) then rootNodeDiscovery(G);
  minimizeDiscoveryCache(G);
  return discoverycache;
}
```

Listing 5.6: Discovery Cache Creation Algorithm

```
childNodeDiscovery(G){
 forall ( Gp and (Gp,G) in goalgraph ) {
   forall ( W and subsume(Gp,W) ) {
      degree = matchmaking(G,W);
      if (! degree = disjoint ) then discoverycache = + (G,W); }
   forall ( W and intersect(Gp,W) ) {
      if ( plugin(G,W) ) then degree = plugin;
      if ( intersect (G,W) ) then degree = intersect ;
      if (! degree = disjoint ) then discoverycache = + (G,W); } }
 return discoverycache ;
}
```

Listing 5.7: Child Node Discovery Algorithm

Listing 5.7 defines the algorithm for extending the discovery cache when a new child node or an intersection goal template has been inserted into the goal graph. We here merely need to inspect those Web services that are usable for G_p as a direct parent of G under the degrees *subsume* or *intersect*, because with to the inference rules from Theorem 5.1 it holds that (1) every Web service that is not usable for G_p is also not usable for G (*cf.* rule 3.5), and (2) if $exact(G_p, W)$ or $plugin(G_p, W)$ then we can directly infer $plugin(G, W)$ from rules 3.1 and 3.2; moreover, in these cases the arc (G, W) is in anyway omitted in the SDC graph (*cf.* clause (iv) in Definition 5.2). Furthermore, also the necessary discovery operations are based on the inference rules: if $subsume(G_p, W)$, then any usability degree of W for G is possible (*cf.* rule 3.3), and if $intersect(G_p, W)$ then the possible usability degrees are *plugin, intersect* or *disjoint* (*cf.* rule 3.4). We determine the actual degree by matchmaking, and add the discovery cache arc (G, W) only if W is actually usable for G.

Listing 5.8 below defines the algorithm for extending the discovery cache when G has been inserted as a new root node in the goal graph. We here can make use of the captured knowledge on usable Web services for the direct children of G, which is facilitated by the inference rules under the *plugin* similarity degree (*cf.* rules 2.1 - 2.5 in Theorem 5.1). However, there can also be Web services which are usable for G but not for any of its children; we thus need to investigate all other available Web services as well. If G is a new disconnected node in the goal graph, we need to inspect the usability of each available Web service. Both algorithms for extending the discovery cache may create redundant arcs – in particular the *rootNodeInsertion* algorithm but also if a new child node has been inserted at an intermediate level in the subsumption hierarchy. We recall that the arc (G_c, W) is redundant if there is a G which is a parent of G_c and the usability degree of W for G is *exact* or *plugin* (*cf.* clause (iv) in Definition 5.2). Listing 5.9 shows the algorithm for removing all redundant arcs and therewith maintain the minimality of the discovery cache.

```
rootNodeDiscovery(G){
    forall ( G2 and (G,G2) in goalgraph ) {
        forall ( W and exact(G2,W) ) { degree = subsume; discoverycache = + (G,W); }
        forall ( W and plugin(G2,W) ) {
            degree = intersect ;
            if ( plugin(G,W) ) then { pluginFlag = true; degree = plugin; }
            if ( subsume(G,W) ) then { subsumeFlag = true; degree = subsume; }
            if ( pluginFlag and subsumeFlag ) then degree = exact;
            discoverycache = + (G,W); }
        forall ( W and subsume(G2,W) ) { degree = subsume; discoverycache = + (G,W); }
        forall ( W and intersect(G2,W) ) {
            if ( subsume(G,W) ) then degree = subsume; else degree = intersect;
            discoverycache = + (G,W); }
    }
    forall (W and ! (W in sdcgraph) ) {
        degree = matchmaking(G,W);
        if (! degree = disjoint ) then discoverycache = + (G,W);
    }
    return discoverycache;
}
```

Listing 5.8: Root Node Discovery Algorithm

```
minimizeDiscoveryCache(G) {
    forall ( Gc and (G,Gc) in goalgraph ) {
        forall ( W and (G,W) in discoverycache ) {
            if ( degree(G,W) = exact or degree(G,W) = plugin ) then {
                discoverycache = − (Gc,W); } } }
    return discoverycache;
}
```

Listing 5.9: Algorithm for Minimizing the Discovery Cache

5.3.2 Illustrative Example

The insertion of individual goal templates is the central operation for creating SDC graphs. In order to demonstrate the algorithms defined above, the following illustrates the creation of the SDC graph for a non-trivial example from the shipment scenario defined in the Semantic Web service Challenge (see `www.sws-challenge.org`).

We here follow the original scenario description which is concerned with shipping packages from the USA. We consider three goal templates: `gtUS2world` for shipping packages of any weight from the USA to anywhere in the world, `gtUS2NA` for package shipment from the USA to North America, and `gtNA2NAlight` for shipping light packages within North America. We further consider the following three Web services from the original scenario

description: `wsMuller` which offers shipment from the USA to almost anywhere in the world for packages with a maximal weight of 50 kg, `wsRunner` for shipping packages of any weight from the USA to all continents apart from North America and Africa, and `wsWeasel` which offers shipment within the USA for packages of any weight.

Figure 5.10 provides a concise overview of the relevant information for our discussion. The upper part shows the semantic similarity of the goal templates as well as the usability degrees of the Web services which have been determined by actual matchmaking. The lower part shows the steps for creating the SDC graph which we shall explain below.

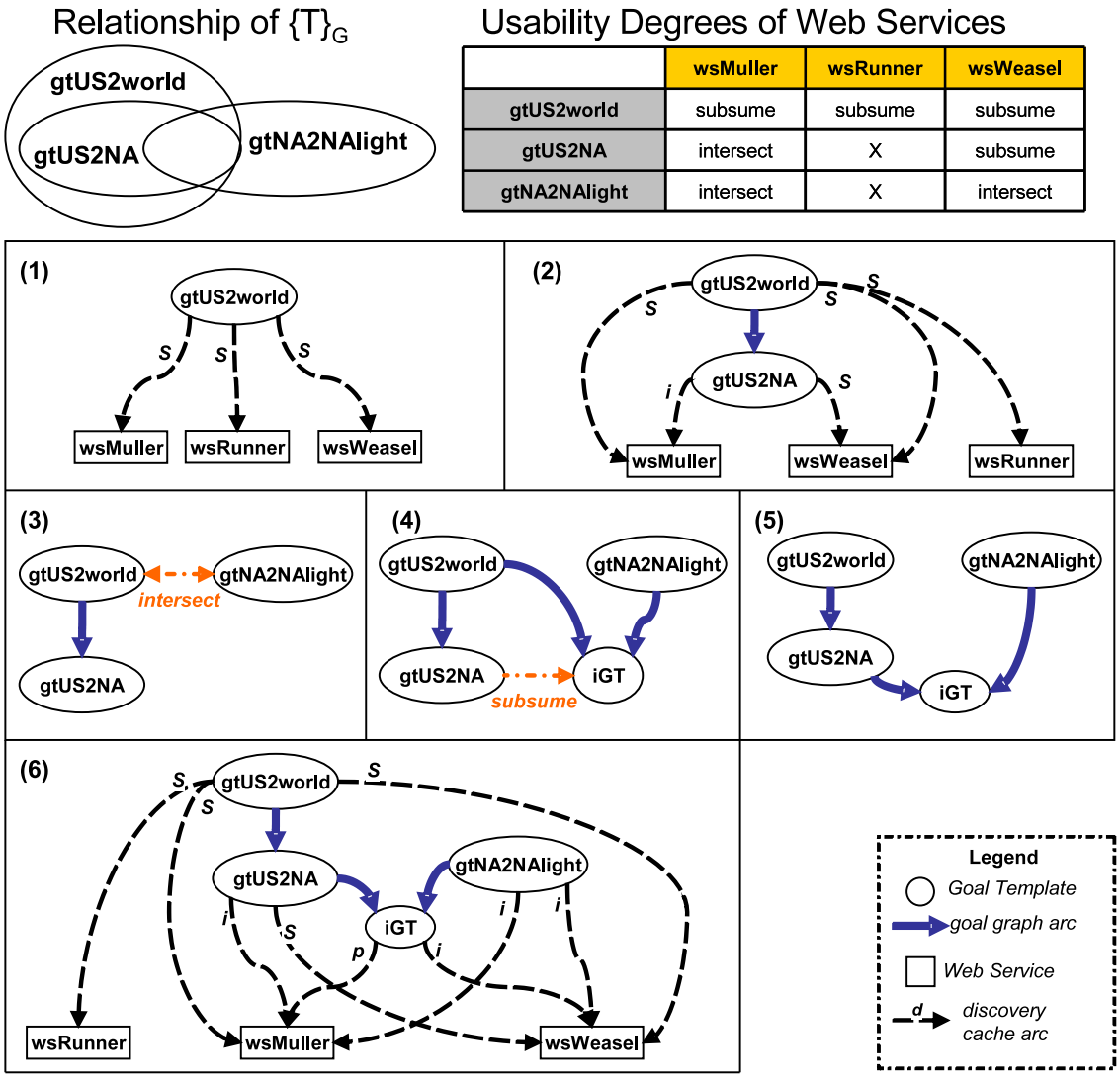


Figure 5.10: Illustrative Example for SDC Graph Creation

We commence the SDC graph creation with inserting the goal template `gtUS2world`. This becomes the first node in goal graph (see Listing 5.1), and then the discovery cache is created by the *rootNodeDiscovery* algorithm from Listing 5.8; the resulting SDC graph is shown as step (1) in the figure. We then continue with inserting `gtUS2NA` into the SDC graph. Because *subsume*(`gtUS2world`, `gtUS2NA`), the main procedure will invoke the *childNodeInsertion* algorithm from Listing 5.3 which will insert `gtUS2NA` as a child node of `gtUS2world`. Then, the *childNodeDiscovery* algorithm from Listing 5.7 is invoked to update the discovery cache, which results in the SDC graph shown as step (2).

We then add `gtNA2NAlight` to the SDC graph, see step (3). The current root node is `gtUS2world`, and because of *intersect*(`gtUS2world`, `gtNA2NAlight`) the main procedure will invoke the *intersectGTInsertion* algorithm from Listing 5.4. This adds the intersection goal template `iGT`, which describes the objective of shipping light packages from the USA to North America. Then, the algorithm will detect that *subsume*(`gtUS2NA`, `iGT`): both are concerned with shipment from the USA to North America, but `gtUS2NA` allows the package to be of any weight while `iGT` is restricted to light packages (see step (4)). This is an example for case (d) in Table 5.5, which is handled by the algorithm such that `iGT` becomes a child of `gtUS2NA`; the resulting goal graph is shown as step (5). Finally, the insertion operation for `gtNA2NAlight` is completed by extending the discovery cache. Because of the nested algorithm, first the discovery cache for `iGT` will be created via the *childNodeDiscovery* algorithm, and then the one for `gtNA2NAlight` is created via the *rootNodeDiscovery* algorithm. In this example, the discovery cache creation algorithms do not create redundant arcs. The finally resulting SDC graph is shown as step (6) in the figure.

5.3.3 Evolution Support

We now turn towards the maintenance of the SDC graph. This becomes necessary with respect to the dynamic nature of SOA applications wherein usually the available Web services as well as goal descriptions are changing over time. In order to provide a serviceable index structure for optimizing Web service discovery operations in such environments, we need to ensure that the SDC graph exposes its structure and properties at all times. For this, the following specifies the algorithms for maintaining the SDC graph when changes on the existing goal templates or the available Web services occur. We consider this as the basic evolution support for the SDC technique which, in combination with the SDC graph creation by the subsequent insertion of goal templates, provides a set of algorithms for retaining the desirable structure of the SDC graph whenever a goal template or a Web service is added, removed, or modified.

Change Management of Existing Goal Templates

We commence with the maintenance of the SDC graph on the level of goal templates. We consider the subsequent addition of new goal templates as the default operation for extending the SDC graph, which is supported by the SDC graph creation algorithm defined above in Section 5.3.1. However, there might occur changes on the already existing goal templates during the life-time of an application, e.g. when goal templates are outdated or if their descriptions are updated in order to reflect changes in the application context. The following defines the necessary algorithms for maintaining the structure of the SDC graph when a goal template is removed or modified. We consider these to be performed in the course of manual maintenance activities on the existing goal templates, which can be expected to only occur rarely in real-world applications. They should be carried out carefully because the existing goal templates constitute the goal graph as the skeletal structure of SDC graph, and thus changes on them can significantly degrade its quality as an efficient search index for the existing goal templates and the available Web services.

Goal Template Removal. The first algorithm is concerned with maintaining the SDC graph when an existing goal template is removed. In order to warrant the operational reliability of the SDC graph, this must ensure that the SDC graph will still be inferentially complete and minimal after removing the goal template (*cf.* Theorem 5.2). For this, we must adjust the goal graph such that it organizes the remaining goal templates in a proper subsumption hierarchy, and we also need to refine the discovery cache such that it defines the minimal set of arcs which are necessary to deduce the precise usability degree of every available Web service for each remaining goal template.

Listing 5.10 defines the algorithm for removing a goal template G from the SDC graph. When G has been a root or a disconnected node in the goal graph, we remove G and all its outgoing arcs so that no reduncancy remains in the goal graph nor in the discovey cache. We also need to materialize the previously omitted discovery cache arcs for each G_c which is direct child of G. If a Web service W is usable for G under the *exact* or the *plugin* degree, then the arc (G_c, W) has been omitted in the SDC graph because $plugin(G_c, W)$ can be directly inferred (*cf.* Definition 5.2): we now need to explicate it in order to keep the relevant knowledge in the SDC graph. When G has been a child node, we adjust the goal graph such that every direct child node of G becomes a direct child of the former parents of G, and we refine the discovery cache analogously to the first situation. We also need to adjust possibly existing intersection goal templates of which G has been a direct parent. This is performed at first in the algorithm as we shall explain below in more detail.

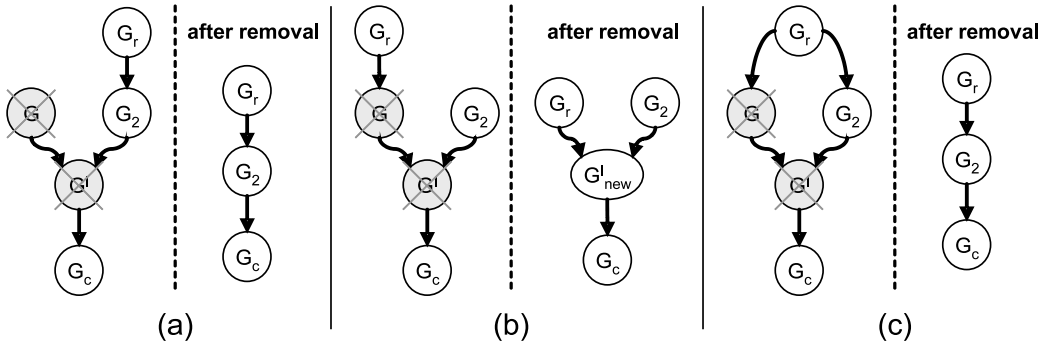
```
remove(G) {
 forall ( iGT and (G,iGT) in goalgraph ) { removeIntersectGT(G,iGT); }
 if ( position(G) = root ) then {
   forall ( G2 and (G,G2) in goalgraph ) {
      goalgraph = − (G,G2);
       forall ( W and (G,W) in discoverycache ) {
          addOmittedDiscoveryCache(G,G2,W);
          discoverycache = − (G,W); } }
   goalgraph = − G; }
 else if ( position(G) = child ) then {
   forall ( Gp and (Gp,G) in goalgraph ) {
       forall ( G2 and (G,G2) in goalgraph ) {
          goalgraph = − (G,G2);
          goalgraph = + (Gp,G2);
           forall (W and (G,W) in discoverycache) {
              addOmittedDiscoveryCache(G,G2,W);
              discoverycache = − (G,W); } } }
   goalgraph = − G; }
 return sdcraph;
}
addOmittedDiscoveryCache(G1,G2,W) {
 if ( degree(G1,W) = (exact,plugin) ) then {
   degree(G2,W) = plugin;
   discoverycache = + (G2,W); }
  return discoverycache; }
```

Listing 5.10: Goal Template Removal Algorithm

The above algorithm handles the removal of original goal templates, i.e. which have an explicitly specified functional description. An intersection goal template is a logical construct in the SDC graph whose functional description is not explicitly specified. To perform matchmaking on an intersection goal template, we need to know the original goal templates for which it has been created. Thus, when removing an original goal template G from the SDC graph we also need to adjust its dependent intersection goal templates.

Listing 5.11 shows the algorithm for this, and Table 5.6 illustrates the adjustment of the goal graph for the relevant situations. In case (a), G has been a root node, and the dependent intersection goal template has G_2 as its other direct parent. Here, we remove $G^I(G, G_2)$ and its incoming goal graph arcs, and the possibly existing child nodes of $G^I(G, G_2)$ become direct child nodes of G_2. If in this case G_2 has a parent G_r it must hold that $intersect(G, G_r)$ – otherwise the goal graph would have a different structure. Here G_2 can be an intersection goal template itself: this is handled by the *forall*-loop in Listing 5.10 for iteratively adjusting all intersection goal templates whereof G is a direct parent. The other cases differentiate two situations when G has been a child node in the goal graph. We always need to remove

Table 5.6: Situations for Removal of Intersection Goal Templates



$G^I(G, G_2)$ because one of its original parents will no longer exist. In case (b), there is a G_p which is a parent of G but not of G_2. Here, it must hold that $intersect(G_p, G_2)$; thus, if not existing before, we need to add a new intersection goal template and the previous children of $G^I(G, G_2)$ become child nodes of this. In case (c), G_p is a common parent of both G and G_2 as the parents of the relevant intersection goal template. Here, the resulting goal graph has the same structure as in case (a). We hence can define one algorithm for adjusting the SDC graph which covers all cases, and also refines the discovery cache by materializing previously omitted arcs. This is invoked from the overall algorithm in Listing 5.10 before removing the original goal template.

```
removeIntersectGTofChild(G,iGT) {
 if ( Gr and (Gr,G) in goalgraph and ! ( (Gr,G2) in goal graph ) ) then {
   iGTnew = intersectionGoalTemplate(Gr,G2);
   goalgraph = + iGTnew + (Gr,iGTnew) + (G2,iGTnew);
   discoveryCacheCreation(iGTnew);
    forall (Gc and (iGT,Gc) in goal graph) {
       goalgraph = + (iGTnew,Gc);
        forall (W and (iGT,W) in discoverycache) {
           addOmittedDiscoveryCache(iGT,Gc,W);
           discoverycache = - (iGT,W); } }
   goalgraph = - iGT - (G,iGT) - (G2,iGT);
 } else {
    forall (Gc and (iGT,Gc) in goal graph) {
       goalgraph = + (G2,Gc);
        forall (W and (iGT,W) in discoverycache) {
           addOmittedDiscoveryCache(iGT,Gc,W);
           discoverycache = - (iGT,W); } }
     goalgraph = - iGT - (iGT,G) - (iGT,G2); }
 return sdcgraph; }
```

Listing 5.11: Adjustment of Intersection Goal Templates

Goal Template Update. The second algorithm handles the update of a goal template, i.e. if its description is changed. We define the necessary adjustment of the SDC graph as an update operation which replaces the old version G_o of the goal template by its new version G_n. The basic way to handle this is to first remove G_o and then re-insert G_n.

Listing 5.12 shows the algorithm for this. We therein distinguish some situations where we can reduce the necessary computational effort. The first one is if the similarity degree between the new and the old version is *exact*, which can occur if the changes in the goal template description to not affect the meaning of the functional description. We here can simply replace G_o by G_n while the rest of the SDC graph remains the same. Other situations where we can simplify the update management of the SDC graph are (1) when G_o has been a disconnected node and G_n denotes a semantic specialization, (2) when G_o has been the single root node of an SDC subgraph and G_n denotes a semantic generalization, and (3) when G_o has been a child node without any outgoing goal graph arcs and G_n denotes a semantic specialization. In all other situations we perform the default operation.

```
update(Go,Gn) {
   if ( exact(Go,Gn) ) then {
      goalgraph = + Gn;
       forall ( (Go,G) in goalgraph ) { goalgraph = + (Gn,G) − (Go,G); }
       forall ( (G,Go) in goalgraph ) { goalgraph = + (G,Gn) − (G,Go); }
       forall ( (Go,W) in discoverycache ) { discoverycache = + (Gn,W) − (Go,W); }
      goalgraph = − Go; }
   else if ( disconnected(Go) and subsume(Go,Gn) ) then {
      goalgraph = + Gn + (Go,Gn);
      childNodeDiscovery(Gn);
      goalgraph = − Go − (Go,Gn); }
   else if ( singleRoot(Go) and plugin(Go,Gn ) then {
      goalgraph = + Gn + (Gn,Go);
      rootNodeDiscovery(Gn);
       forall ( W and (Go,W) in discoverycache ) { discoverycache = − (Go,W); }
       forall ( Gc and (Go,Gc) in goal graph) { goalgraph = + (Gn,Gc) − (Go,Gc); }
      goalgraph = − Go − (Gn,Go); }
   else if ( lowestChild(Go) and subsume(Go,Gn) then {
      goalgraph = + Gn + (Go,Gn);
      childNodeDiscovery(Gn);
       forall ( W and (Go,W) in discoverycache ) { discoverycache = − (Go,W); }
       forall ( Gp and (Gp,Go) in goalgraph ) { goalgraph = + (Gp,Gn) − (Gp,Go); }
      goalgraph = − Go − (Go,Gn); }
   else {
      remove(Go);
      insert (Gn); }
  return sdcgraph;
}
```

Listing 5.12: Goal Template Update Algorithm

Change Management of Available Web Services

The second compulsory aspect for maintaining the SDC graph is the handling of changes on the available Web services. This refers to the addition of new Web services to the system, as well as the removal or modification of existing ones. Such changes can be expected to repeatedly occur in real-world scenarios, and they should be properly reflected in the SDC graph in order to warrant its operational reliability. The following defines the necessary algorithms for this, which are simpler than handling changes of goal templates because we merely need to adjust the discovery cache. These algorithms can be automatically triggered by changes in the Web service registry that is associated with the overall SWS system.

Web Service Insertion. The first algorithm is concerned with adding a new Web service W_{new} into the SDC graph. Essentially, we need to add the arcs to the discovery cache which are necessary to deduce the usability degree of W_{new} for every existing goal template. Listing 5.13 shows the algorithm for this. We start with inspecting the usability of W_{new} for all root nodes in the goal graph, and then successively proceed with the child nodes in a depth first-manner via the iterative sub-procedure *childNodeWSInsertion* that takes the inference rules from Theorem 5.1 into account. This keeps the necessary matchmaking effort minimal, and also ensures the minimality of the updated discovery cache.

```
insert (W) {
  forall (G and position(G) = root) {
    matchmaking(G,W);
    if (! d = disjoint ) then {
      discoverycache = + d(G,W);
      childNodeWSInsertion(G,W); } }
 return sdcgraph;
}
childNodeWSInsertion(G,W) {
  if ( degree(G,W) = (exact,plugin) ) then return sdcgraph;
  else { forall ( G2 and (G,G2) in goalGraph ) {
            if ( degree(G,W) = subsume ) then {
                matchmaking(G2,W);
                if (! degree = disjoint ) then discoverycache = + (G2,W); }
            if ( degree(G,W) = intersect ) then {
                if ( plugin(G2,W) ) then { degree = plugin; discoverycache = + (G2,W); }
                if ( intersect (G2,W) ) then { degree = intersect ; discoverycache = + (G2,W); }
            }
            childNodeWSInsertion(G2,W); }
     return sdcgraph; }
  }
```

Listing 5.13: Web Service Insertion Algorithm

Web Service Removal. When a Web service is no longer available, we need to remove it from the SDC graph. The algorithm for this as shown in Listing 5.14 is straightforward: we delete all discovery cache arcs for the removed Web serivce. While this does not affect the formal properties of the SDC graph, a result of this operation can be that for some goal templates no a usable Web service exists any longer – i.e. if the removed Web service has been the only one. To ensure that every existing goal template can be solved, one could remove the respective goal templates from the SDC graph. However, we assume that usually there is more than one usable Web service for a goal template and thus do not further handle this situation.

```
remove(W) {
   forall  ( (G,W) in discoverycache ) { discoverycahe = – (G,W); }
   return sdcgraph; }
```

Listing 5.14: Web Service Deletion Algorithm

Web Service Update. The third algorithm handles modifications on the description of a Web service W which already exists in the SDC graph. Similar to updates of goal templates, we define the necessary adjustment of the discovery cache as a replacement of the old version W_o of the Web serivce by the new version W_n. Listing 5.15 shows the algorithm for this, which merely performs the default handling of first removing W_o and then inserting W_n. We here resign from handling situations where the necessary adjustment could be achieved with less computational effort, because the computational costs of the above algorithms for the removal and insertion of Web services are acceptably low.

```
update(Wo,Wn) {
    remove(Wo);
    insert (Wn);
    return sdcgraph; }
```

Listing 5.15: Algorithms for Deletion and Update of a Web Service Description

To conclude, the set of algorithms for creating and maintaining the SDC graph ensure that it exposes the desirable structure and properties whenever a goal template or a Web service is added, removed, or modified. Therewith, the SDC graph provides a correct index structure of the existing goal templates and the available Web services with respect to their actual functional descriptions at all times, which is necessary to warrant the operation reliability of the SDC technique in dynamic environments where goal templates and Web services steadily change and evolve.

5.3.4 Complementing Techniques

After specifying the necessary management techniques for the SDC graph, we now discuss possible extensions for increasing the effectiveness for its intended usage. In particular, we present an approach for automatically generating additional goal templates as a central element of our approach, and we outline possibilities for the integration of the SDC graph with other SWS techniques. We consider these as optional extensions which are not necessary to warrant the operational reliability of the SDC technique, and thus content ourselves with explaining the basic ideas and the expectable benefits.

Automated Goal Template Generation

Goal templates are a central element in our approach. They are needed to support the formulation of concrete client objectives in terms of goal instances. Furthermore, they are the constituting element of the SDC graph: the more goal templates exist, the more fine-grained becomes the goal graph, and the more effective can be the search space reduction for optimizing Web service discovery operations. With respect to this, techniques for automatically creating goal templates appear to be desirable in order to enhance the support for potential client requests as well as the operational effectiveness of the SDC technique.

One possibility for this is to generate additional goal templates from a given one on the basis of the underlying domain ontologies. We explain this for our running example from the shipment scenario. Let the goal template G_1 for shipping packages of any weight in Europe be given. Its functional description is based on the `location & shipment ontology`. Table 5.7 provides the relevant information for the following discussion.

Table 5.7: Basis for Automated Goal Template Generation

Goal Template G_1 "ship a package of any weight in Europe"		**Domain Ontology** Ω excerpt of location & shipment ontology
Ω:	location & shipment ontology	$\forall ?x.\ continent(?x) \Rightarrow in(?x, world)$.
IN:	{?s,?r,?p,?w}	$continent(europe)$.
ϕ^{pre}:	$address(?s) \wedge in(?s, europe)$	$country(germany)$.
	$\wedge\ address(?r) \wedge in(?r, europe)$	$in(germany, europe)$.
	$\wedge\ package(?p)\ \wedge weight(?p, ?w)$	$continent(northAmerica)$.
	$\wedge\ maxWeight(?w, heavy)$.	$country(usa)$.
ϕ^{eff}:	$\forall ?o, ?price.\ out(?o) \Leftrightarrow$ (	$in(usa, northAmerica)$.
	$shipmentOrder(?o, ?p)$	$weightClass(heavy)$.
	$\wedge\ sender(?p, ?s)\ \wedge receiver(?p, ?r)$	$weightClass(light)$.
	$\wedge\ costs(?o, ?price)$).	$\forall ?x.\ weight(?x, light) \Rightarrow weight(?x, heavy)$.

The precondition of G_1 is defined by three logical atoms that refer to instances defined in the domain ontology: the sender and the receiver must be located in `europe`, and the weight class `heavy` defines that the package can have any weight. From this, we can generate new goal templates by changing these atoms with respect to the taxonomic structure and the instances defined in the ontology. For example, we can define G_2 for shipping a heavy package from anywhere in the world to Europe by changing the atom for the sender location from `europe` to `world`; this is facilitated by the axiom $\forall ?x.\ continent(?x) \Rightarrow in(?x, world)$ in the ontology. Then, the similarity degree is $plugin(G_1, G_2)$, so that G_2 will become a parent of G_1 in the SDC graph. We can analogously generate a goal template G_3 for package shipment in Germany by changing the atoms for the sender and receiver location from `europe` to `germany`. Then, G_3 will become a child of G_1 in the SDC graph because it denotes a semantic specialization, i.e. the similarity degree is $subsume(G_1, G_3)$.

This allows us to generate further goal templates from a given one by exploiting background knowledge that is defined in the underlying domain ontology. Eventually, we can generate all goal templates that can be expressed on the basis of the ontology, and therewith provide a sophisticated basis for the formulation of specific goal instances in the problem domain. Moreover, all the generated goal templates will be semantically similar: the logical structure of their functional descriptions is identical, and they only differ within the constraining atoms that refer to semantically related concepts of instances in the underlying domain ontology. We thus will obtain a fine-grained subsumption hierarchy in the SDC graph, which improves its effectiveness for optimizing Web service discovery. The generation of additional goal templates can be automated. Although the algorithms for is will dependent on the specific goal template descriptions and the underlying domain ontology, the outlined technique appears to be generally applicable for all kinds of goal templates.

Another challenge is the definition of the goal descriptions that are relevant for an application scenario, i.e. to identify the client objectives that should be described by goal templates. A supportive technique for this is presented in [Lambert et al., 2007]: the domain experts – i.e. the prospective clients – formulate desirable goals in natural language descriptions, which then is transformed into formal goal descriptions by trained knowledge engineers via several iterations and on the basis of Wikis for decentralized collaboration.

Integration with Other SWS Techniques

We here have defined the SDC graph with respect to the functional compatibility of goal templates and Web services, which appears to be sufficient for the purpose of optimizing the Web service discovery task. This could be extended with further knowledge that is relevant

for other SWS techniques, e.g. quality-of-service or other non-functional aspects that are considered within the selection and ranking components of SWS systems (see Section 2.2.2). The relevant information could be kept in the SDC graph in addition to the discovery cache arcs that explicate the functional suitability of a Web service for solving a goal template, and could then be used to support the selection of the actual Web services at runtime.

Apart from extending the SDC graph with additional knowledge, it is also possible to use it within other SWS techniques that appear to be relevant for solving goals by the automated detection and execution of Web services. For example, we can use the SDC graph for visualizing the search space of available Web services. This allows clients to browse and investigate the Web services on the level of goals that can be solved by them while abstracting from technical details. For this, we can display the SDC graph in a graphical user interface and provide respective browsing facilities; we shall present a prototypical solution for this below in Section 5.5. Another possibility is use the knowledge kept in the SDC graph to generate the necessary infrastructure for the automated invocation and consumption of Web services as explained in Section 3.2.2. For this, we can generate the mediators between every goal template and its usable Web services. The relevant knowledge for this is kept in the discovery cache of the SDC graph. Then, the respective client interfaces need to be defined in order to support the automated execution of the Web services.

5.4 Optimized Web Service Discovery

After having specified the necessary management techniques for SDC graphs, we now turn towards the optimization of Web service discovery as the primary aim of the SDC technique. We in particular focus on runtime discovery, which is concerned with detecting the actual Web services for a given goal instance and denotes the time critical operation in our framework. The following briefly recalls the relevant aspects.

In our approach, goal instances are the central element for clients to request and consume Web services. Every concrete objective that a client wants to achieve is described in terms of a goal instance, and for each goal instance the suitable Web services need to be detected and executed dynamically at runtime. Thus, the runtime discovery task is the time critical operation in our approach. It further is the expectably most frequent operation in real-world scenarios, and it should be performed in an efficient and reliable manner in order to adequately accomplish the first processing step in SWS environments. With respect to this, the aim is to provide an efficient and reliable runtime discovery component and, therewith, overcome the bottleneck for the scalability of SWS systems (see Section 2.2.3).

A goal instance $GI(G, \beta)$ is defined as a tuple of G as the corresponding goal template, and the input binding β that defines a value assignment for the input variables specified in the functional description of G (see Section 4.3.2). We require goal a instance $GI(G, \beta)$ to be a consistent instantiation of its corresponding goal template G. The goal instantiation condition $GI(G, \beta) \models G$ is given if the functional description of G is satisfiable when instantiated with the input binding β, i.e. if both the precondition and the effect are satisfiable for the defined input values (*cf.* Definition 4.7). This is a pre-requite for automated Web service discovery: only if $GI(G, \beta) \models G$ we can precisely identify the logical models which represent the solutions for $GI(G, \beta)$, and on this basis determine the suitable Web services by semantic matchmaking. We can reduce the necessary matchmaking effort for this by considering the design time discovery results: if a Web service is usable for G under the *exact* or the *plugin* degree then it is also usable for $GI(G, \beta)$, while under the degrees *subsume* and *intersect* additional matchmaking is needed; all other Web service are not usable to solve $GI(G, \beta)$ (*cf.* Theorem 4.1).

The approach for enhancing the computational performance of runtime discovery operations is to exploit the knowledge kept in the SDC graph in order to effectively reduce the search space and minimize the number or necessary matchmaking operations. In particular, we optimize the discovery of suitable Web services for a given goal instance $GI(G, \beta)$ as follows. At first, we reduce the relevant search space by determining G' as the most specialized goal template whereof $GI(G, \beta)$ is a valid instantiation. This G' is allocated as a (possibly indirect) child of the originally defined corresponding goal template G in the SDC graph. By definition, there can only be one most appropriate goal template G' for every goal instance, and we can detect this by subsequently traversing the SDC graph on the basis of the goal instantiation condition. Because G' is the most specialized goal template, its set of usable Web services is minimal, and these are the only potential candidates for solving the given goal instance. We then can determine their actual suitability on the basis of the integrated matchmaking techniques for the goal instance level summarized above.

The primary purpose of this optimization strategy is to minimize the number of necessary matchmaking operations for completing a discovery task. We therewith aim at enhancing the computational performance in two ways: (1) a significant decrease of the average time can be achieved by reducing the usually expensive matchmaking operations to a minimum, and (2) the efficiency of Web service discovery in larger search spaces can be increased because only a subset of the available Web services needs to be inspected. The following specifies the optimized algorithms for runtime Web service discovery, illustrates them within our running example, and finally discusses the achievable performance increase that can be expected in real-world application scenarios.

5.4.1 Runtime Discovery Algorithms

The following specifies the algorithms for the optimized runtime discovery in detail. As the two flavors of Web service discovery that appear to be relevant in SWS systems, we define one algorithm for detecting a single Web service and another one for finding all usable Web services for a given goal instance.

Discovery of a Single Web Service

Listing 5.16 defines the algorithm for the discovery of one Web service that is usable for solving a given goal instance under functional aspects. Although in general there can be several usable Web services for a goal instance, for certain application purposes the detection of one of these appears to be sufficient, e.g. if the SWS system performs the discovery task in an interleaved manner with the subsequent processing steps.

```
discoverSingleWS(GI(G,b)) {
    if ( instantiationCheck(GI(G,b)) = false ) then return "invalid goal instantiation";
    if ( lookup(G) = W ) then return W;
    while ( (G,Gc) in goalgraph and instantiationCheck(GI(Gc,b)) ) do {
        G = Gc;
        forall ( (G,W) in discoverycache ) {
            if ( degree(G,W) = (exact,plugin) ) then return result = W; } }
    if ( checkOtherWS(G,b) = W ) then return W;
    else { return "no Web service found"; }
}
```

Listing 5.16: SDC-enabled Runtime Discovery Algorithm – Single Web Service

The input of the algorithm is a goal instance $GI(G, \beta)$ for which a suitable Web services shall be discovered. As the first step, the function *instantiationCheck(*$GI(G, \beta)$*)* checks whether the goal instantiation condition $GI(G, \beta) \models G$ is satisfied; this is a pre-requisite for Web service discovery by semantic matchmaking as explained above. Then, the algorithm consists of three methods which require increasing matchmaking efforts for the discovery task, and thus are invoked successively if the preceding one has not been successful.

At first, the *lookup*-method searches for a Web service W that is usable for the corresponding goal template G under the *exact* or the *plugin* degree: we know that this W is also usable for solving $GI(G, \beta)$ without the need of matchmaking at runtime (*cf.* Theorem 4.1). Listing 5.17 below shows the algorithm for finding such a W in the SDC graph. We first inspect the existing discovery cache arcs for the corresponding goal template G. As soon as an arc that is annotated with the *exact* or *plugin* degree is detected, the respective Web service is returned at the discovery result for $GI(G, \beta)$. If this is not successful, we search

```
lookup(G) {
    result = empty;
    forall ( (G,W) in discoverycache ) {
        if ( degree(G,W) = (exact,plugin) ) then return result = W;
        else { forall ( Gp and (Gp,G) in goalgraph ) { lookup(Gp); } } }
    if ( result = empty ) return result ;
}
checkOtherWS(G,b) {
    result = empty;
    forall ( (G,W) in discoverycache and degree(G,W) = subsume ) {
        if ( satisfiable (W,b) ) then return result = W; }
    forall ( (G,W) in discoverycache and degree(G,W) = intersect ) {
        if ( satisfiable (G,W,b) ) then return result = W; }
    if ( result = empty ) then return result ;
}
```

Listing 5.17: Sub-Procedures for Single Web Service Discovery

for a W that is usable for G but for which the discovery cache arc is omitted in the SDC graph. If there is a goal template G_p which is a direct parent of G and there is a W whose usability degree for G_p is either *exact* or *plugin*, then the arc (G, W) is omitted in the SDC graph (*cf.* clause (iv) in Definition 5.2). We can directly infer that $plugin(G, W)$, and under this usability degree we know that W is usable for $GI(G, \beta)$. The algorithm detects such Web services by an an inverse depth-first search in the SDC graph, which is realized by the iterative invocation of the *lookup* sub-procedure. This is the most efficient method for runtime Web service discovery because it does not require any matchmaking at all.

If the discovery by lookup is not successful, we continue with the *refinement*-method. This reduces the relevant search space by replacing the initially defined corresponding goal template G with a goal template G_c which is a child node of G in the SDC graph and for which $GI(G, \beta) \models G_c$ is satisfied. We then need to consider fewer candidates for the runtime discovery because the usable Web services for G_c are a subset of those usable for G. The structure of the SDC graph facilitates the effective search for such goal templates. For $\mathcal{G}_{child(G)} = (G_1, \ldots, G_n)$ as the set of direct children of G in the goal graph it holds that each pair $(G_i, G_j) \in \mathcal{G}_{child(G)}$ is *disjoint* if there is no intersection goal template for them (*cf.* clause (iii) in Definition 5.2). Hence, there can only be *one* possible G_c for the refinement: for $G_1, G_2 \in \mathcal{G}_{child(G)}$ and $disjoint(G_1, G_2)$ it holds that if $GI(G, \beta) \models G_1$ then $GI(G, \beta) \not\models G_2$ because if $GI(G, \beta) \models G$ then $\{\mathcal{T}\}_{GI(G,\beta)} \subset \{\mathcal{T}\}_G$ (*cf.* Definition 4.7). If there is an intersection goal template $G^I(G_1, G_2)$ for $G_1, G_2 \in \mathcal{G}_{child(G)}$, then $GI(G, \beta) \models G^I(G_1, G_2)$ only if $GI(G, \beta) \models G_1$ and $GI(G, \beta) \models G_2$; this can be determined by traversing either the path $G \rightarrow G_1 \rightarrow G^I(G_1, G_2)$ or the path $G \rightarrow G_2 \rightarrow G^I(G_1, G_2)$.

We thus can effectively search the SDC graph in a depth-first manner in order to find G' as the lowest goal template in the goal graph for which $GI(G, \beta) \models G'$ is satisfied. This is the most suitable goal template for efficient runtime discovery because the set of usable Web services for G' is minimal in comparison to the initially defined corresponding goal template G and all other ones on every path $G \rightarrow \ldots \rightarrow G'$ in the SDC graph. The algorithm performs this refinement in an iterative manner. To further optimize the detection of a single suitable Web service, in each refinement step we invoke the *lookup*-method explained above in order to find a Web service that is usable for the current G_c under the *exact* or *plugin* degree.

If neither the *lookup*- nor the *refinement*-method has been successful in finding a suitable Web service for $GI(G, \beta)$, we finally inspect the Web services which are usable for the (possible refined) corresponding goal template under the *subsume* or the *intersect* degree. This requires additional matchmaking for each Web service; Listing 5.17 shows the algorithm which performs the necessary satisfiability tests as in defined in Theorem 4.1. A Web service for which this holds is immediately returned as the discovery result. If this is not given for any of the candidates, then a Web service for solving $GI(G, \beta)$ does not exist.

This algorithm effectively realizes the optimization aim performing runtime discovery with the minimal number of necessary matchmaking operations. At first, we try to find a suitable Web service by the *lookup*-method which does not require any matchmaking; if not successful, the *refinement*-method is used to reduce the search space. As the last option, matchmaking is performed for only the minimal number of potential candidates.

Discovery of All Usable Web Services

The second algorithm detects all usable Web services for solving a given goal instance under functional aspects. This can be applied in SWS environments that perform discovery and the subsequent operations for the usability analysis in a stepwise manner.

Listing 5.18 shows the algorithm for this, which essentially applies the same methods for enhancing the computational performance for runtime discovery tasks as explained above. We first check the goal instantiation condition $GI(G, \beta) \models G$ in order to ensure the validity of the provided goal instance. To minimize the relevant search space, we then refine the goal instance by replacing the initially defined corresponding goal template G by G' as the lowest goal template in the goal graph with $GI(G, \beta) \models G'$ as explained above. Then, we determine the discovery result by adding all Web services which are usable for the (possibly refined) corresponding goal template G under the *exact* or the *plugin* degree by applying the *lookup*-method. Finally, we add those Web services that usable for G under the *subsume* or the *intersect* degree and for which the additional matching conditions are satisfied.

```
discoverAllWS(GI(G,b)) {
    if ( instantiationCheck(GI(G,b)) = false ) then return "invalid goal instantiation";
    result = empty;
    while ( (G,Gc) in goalgraph and instantiationCheck(GI(Gc,b)) ) do { GI(G,b) = GI(Gc,b); }
    result = + lookupAllWS(G);
    forall ( (G,W) in discoverycache and degree(G,W) = subsume ) {
        if ( satisfiable (W,b) ) then result = + W; }
    forall ( (G,W) in discoverycache and degree(G,W) = intersect ) {
        if ( satisfiable (G,W,b) ) then result = + W; }
    if ( result = empty ) then return "no Web service found";
    else return result ;
}
lookupAllWS(G) {
    result = empty;
    forall ( (G,W) in discoverycache ) {
        if ( degree(G,W) = (exact,plugin) ) then result = + W;
        forall ( Gp and (Gp,G) in goalgraph ) { lookupAllWS(Gp); } }
    return result ;
}
```

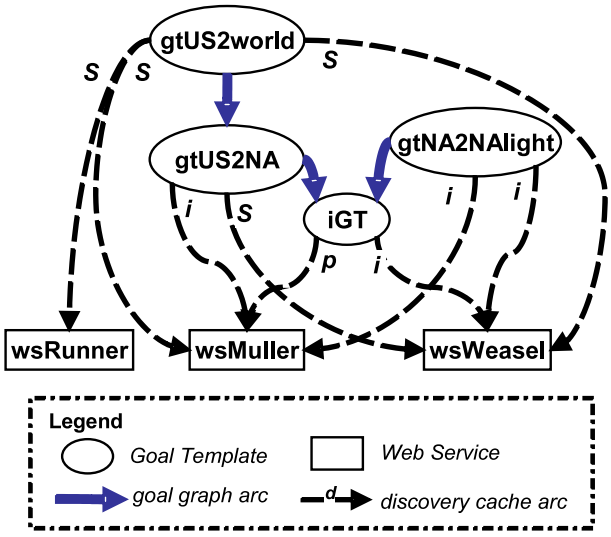
Listing 5.18: SDC-enabled Runtime Discovery Algorithm – All Web Services

This algorithm also realizes our optimization aim: it first cuts down the relevant search space to a minimum, and matchmaking is only performed for the minimal set of candidate Web services. As discussed in Section 2.2.2, it seems to be reasonable to perform functional Web service discovery as the first processing step in integrated SWS environments. Afterwards, the actual usability of the discovered Web services can be further inspected with respect to non-functional and behavioral aspects. Under this assumption, the optimized runtime discovery algorithms as defined here are sufficient to warrant the efficiency and the scalability of the whole system: the bottleneck is the number of the available Web services, which is only relevant for discovery as the first processing step.

5.4.2 Illustrative Example

The optimization of the runtime discovery operations is the primary purpose of the SDC technique, in particular with respect to the overall aim of this work of developing efficient and scalable Web service discovery techniques. In order to illustrate the algorithms defined above, the following discusses the discovery of Web services for a goal instance within the shipment use case. For this, we recall the scenario setting for which we have exemplified the creation of the SDC graph above in Section 5.3.2. We here focus on the operating principles of the runtime discovery algorithms, while we shall discuss the performance increase that can be achieved with the SDC technique in more detail below.

Figure 5.11 provides the relevant information for our discussion. Following the original scenario description as defined in the SWS Challenge, we consider three goal templates and three Web services which are concerned with package shipment from the USA. The goal templates are (1) `gtUS2world` for shipping packages of any weight from the USA to anywhere in the world, (2) `gtUS2NA` for package shipment from the USA to North America, and (3) `gtNA2NAlight` for shipping light packages within North America. The Web services are (1) `wsMuller` which offers shipment from the USA to almost anywhere in the world for packages with a maximal weight of 50 kg, (2) `wsRunner` for shipping packages of any weight from the USA to all continents apart from North America and Africa, and (3) `wsWeasel` which offers shipment within the USA for packages of any weight. As discussed in Section 5.3.2, the SDC graph for this scenario has, in addition to the original goal templates, an intersection goal template `iGT` which describes the objective of shipping light packages from the USA to North America and is allocated as a common child of `gtUS2NA` and `gtNA2NAlight`.



Goal Template	Sender	Receiver	Weight
gtUS2world	USA	world	heavy
gtUS2NA	USA	NA	heavy
gtNA2NAlight	NA	NA	light

Web Service	Sender	Receiver	Weight
Muller	USA	AF,EU,NA,AS	w50lq
Runner	USA	EU,AS,SA,OC	heavy
Weasel	USA	USA	heavy

Usability	wsMuller	wsRunner	wsWeasel
gtUS2world	subsume	subsume	subsume
gtUS2NA	intersect	X	intersect
gtNA2NAlight	intersect	X	subsume
iGT	plugin	X	intersect

Figure 5.11: Illustrative Example for Optimized Runtime Discovery

For illustrating the runtime discovery algorithms, let us consider the client objective of shipping a package of 5.16 kg from San Francisco to New York City. Let this be described as a goal instance $GI(\texttt{gtUS2world}, \beta)$ with `gtUS2world` as the corresponding goal template and the input binding $\beta = (?s|sanFrancisco, ?r|newYorkCity, ?w|5.16\ kg)$. Note that the chosen goal template is suitable for describing the objective but it is not the most appropriate one among the existing goal templates. We assume this to be a typical situations in real-world applications: clients choose an existing goal template or define a new one

which is sufficient to express the concrete objective to be solved, while there might exist more appropriate goal templates which the client is not aware of. Besides, the input binding merely covers the inputs related to functional aspects of the objective description. There might be further requirements on non-functional aspects, e.g. to find the cheapest offer or to only use Web services that support a certain payment method. However, we consider such constraints to be evaluated by selection and ranking mechanisms which are performed after Web service discovery, and thus do not consider them here any further.

We first discuss the discovery of a single Web service for $GI(\texttt{gtUS2world}, \beta)$ by the algorithm defined in Listing 5.16. We commence with checking the goal instantiation condition: $GI(\texttt{gtUS2world}, \beta) \models \texttt{gtUS2world}$ is given because San Francisco is located in the USA, New York City is located in the world, and the package weight of 5.16 kg is included in the weight class `heavy`. Then, we try to find a suitable Web service via the *lookup*-method. This is not successful because there are only Web services whose usability degree for `gtUS2world` is *subsume*. Hence, we continue with the *refinement*-method. In the first iteration, this will define `gtUS2NA` as the new corresponding goal template, because this is a direct child of `gtUS2world` in the SDC graph, and the goal instantiation condition $GI(\texttt{gtUS2world}, \beta) \models \texttt{gtUS2NA}$ is satisfied. We therewith have reduced the relevant search space to *wsMuller* and *wsWeasel* as the only Web services which are usable for `gtUS2NA`. We then invoke the *lookup*-method for `gtUS2NA` in order to find a suitable Web service for the goal instance. This is also not successful because neither of the Web service is usable for `gtUS2NA` under the *exact* or the *plugin* degree. Thus, we continue with the second iteration of the *refinement*-method. This defines the intersection goal template `iGT` as the new corresponding goal template, because it is a direct child of `gtUS2NA` and also $GI(\texttt{gtUS2world}, \beta) \models \texttt{iGT}$ is satisfied. When we then invoke the *lookup*-method for `iGT`, we will detect *wsMuller* to be usable for solving the goal instance because it is usable for `iGT` under the *plugin* degree.

The algorithm from Listing 5.18 for discovering all usable Web services for the goal instance works analog. At first, the goal instantiation for `gtUS2world` will be evaluated positively. Then, we refine the corresponding goal template to be the intersection goal template `iGT`, which is the most appropriate one for the goal instance. We then only need to investigate the two Web services which are usable for `iGT` as the minimal relevant search space. We will detect *wsMuller* to be usable for $GI(\texttt{gtUS2world}, \beta)$ via the *lookup*-method as explained above, and we will also detect *wsWeasel* to be usable because it offers shipment within the USA for packages of any weight. The Web service *wsRunner* is not usable for `iGT` because it does not support shipment to the USA. In consequence, it is also not usable for our goal instance.

5.4.3 Expectable Performance Improvements

We complete the specification of the SDC-enabled runtime Web service discovery with discussing the performance increase that can be achieved with this optimization. This is dependent on the specific application scenario, in particular on the structure of the SDC graph that can be created for the existing goal templates and the available Web services.

We already defined the general computational costs for the optimized runtime discovery as $O(\ (diam(SDC_{GG}) * b(SDC_{GG})) + |\mathcal{W}_{G'}|$ where $d(G', W) \in \{subsume, intersect\}$): the former part refers to the number of goal instantiation checks for the *refinement*-method, and the latter denotes the additional matchmaking that is necessary under the *subsume* and *intersect* degree (*cf.* Proposition 5.2 in Section 5.2.3). This means that in the worst case we need to inspect all existing goal templates and perform additional matchmaking for all available Web services, so that no performance increase is gained. However, we assume that in typical application scenarios it is possible to construct the SDC graph such that its structure facilitates efficient runtime Web service discovery. In particular, we can expect significant performance increases when the following is given:

- the goal templates can be organized so that $diam(SDC_{GG}) \ll |\mathcal{G}|$ and $b(SDC_{GG}) \ll |\mathcal{G}|$, because then $O(diam(SDC_{GG}) * b(SDC_{GG}))$ should be a significant saving over $O(|\mathcal{G}|)$; the ideal case is when the goal graph is a tree where every non-leaf node has at least two sons: then $diam(SDC_{GG}) \leq log_2(|G|)$ so that the savings are exponential
- the most frequently occurring usability degree of Web services for the goal templates at the lower levels of the goal graph is *plugin*: these can be detected by the *lookup*-method, and then the *refinement*-method saves a significant amount of unnecessary matchmaking operations because $O(diam(SDC_{GG}) * b(SDC_{GG})) \ll O(|\mathcal{W}_G|)$
- the necessary matchmaking operations are satisfiability tests of a low complexity that can be performed in an efficient manner.

We shall discuss this in more detail in the course of evaluating the developed techniques below in Chapter 6. In particular, we will quantify the actual performance increase on the basis of an exhaustive use case analysis (see Section 6.1). This shall show that the SDC technique can achieve significant performance improvements for the runtime discovery task and thus can be considered as a sophisticated optimization technique. Nevertheless, there are also application scenarios where no or only marginal enhancements are achievable. We shall discuss examples for this, and provide a model for estimating the expectable performance enhancements in concrete application scenarios (see Section 6.2).

5.5 Prototype Implementation

This section presents the prototype implementation of the SDC technique, which includes the management techniques for SDC graphs and the optimized runtime Web service discovery as specified in the preceding elaborations. The aim is to show that the above specifications can be implemented as an extension of existing SWS technologies, and also to provide the necessary infrastructure for demonstrating and evaluating the SDC technique. The prototype is implemented as a discovery component of the WSMX system, which is the reference implementation of the WSMO framework. The following explains the technical design and architecture of the SDC prototype; information on the availability of the software along with further technical details are provided in Appendix B.3. We also present a graphical visualization of the SDC graph which allows clients to browse and inspect the available Web services on the level of goals which can be solved by them.

5.5.1 Design and Architecture

Although the SDC technique is specified independently of a particular Semantic Web service framework, it appears to be reasonable to realize the prototype implementation within such a framework in order to integrate it with existing SWS technologies. We thus validate our approach within a WSMO-based implementation, because – in contrast to the other prominent SWS frameworks – this defines goals as a top-level element and supports the goal-driven approach that underlies the SDC technique [Fensel et al., 2006]. Moreover, WSMO provides a reference implementation of along with further tooling support that can be used as the basic infrastructure for our prototype implementation.

The Web Service Execution Environment WSMX (`www.wsmx.org`) provides a goal-based execution environment for Semantic Web services as the reference implementation of the WSMO framework [Haller et al., 2005; Haselwanter et al., 2006]. Essentially, it realizes the procedure for solving goals as formally described client objectives by the automated discovery, composition, and execution of Web services as discussed in Section 2.2.2. For this, the WSMX system provides the programmatic interfaces and defines execution semantics for the necessary software components [Zaremba et al., 2005]. It uses the Web Service Modeling Language WSML [de Bruijn et al., 2005b] as the specification language, and supports the programmatic handling of WSMO elements via WSMO4J as a Java API (`wsmo4j.sourceforge.net`). The interface for Web service discovery component requires a goal as input and the functionally suitable Web services for this as the output; we thus realize the SDC prototype as an implementation of a WSMX discovery component.

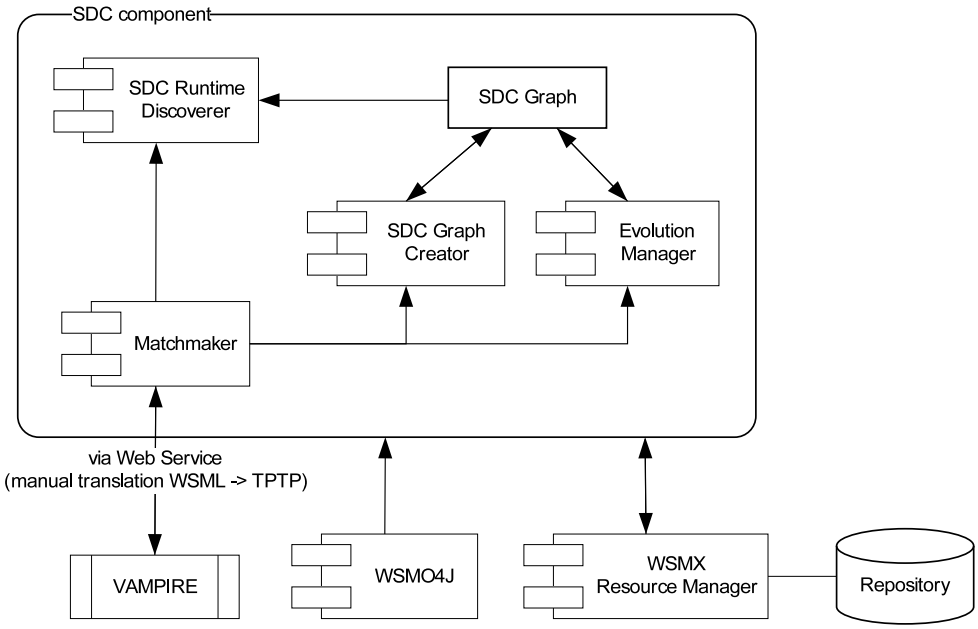


Figure 5.12: SDC Prototype Architecture

Figure 5.12 shows the technical architecture of the SDC prototype. The `SDC Runtime Discoverer` implements the optimized runtime discovery algorithms specified in Section 5.4. This provides the Web service discovery facilities required by the WSMX architecture for goal instances, i.e. for concrete client objectives. Although the distinction of goal templates and goal instances is not defined WSMO and hence not explicitly supported in WSMX, existing solutions for specific use cases realize the Web service discovery component in a similar manner (e.g. [Zaremba et al., 2006]). The other components of the SDC prototype are concerned with the management of the SDC graph: the `SDC Graph Creator` implements the algorithm for creating the SDC graph as specified in Section 5.3.1, and the `Evolution Manager` implements the maintenance algorithms defined in Section 5.3.3. The SDC graph is kept in form of a WSML knowledge base as we shall explain below.

The `Matchmaker` provides the technical implementation of all matchmaking operations that are needed for the SDC technique. This covers the matchmaking of functional descriptions for Web service discovery on the goal template level as well as for determining the semantic similarity degree of goal templates, and the matchmaking operations required for runtime discovery on the goal instance level. We use the automated theorem prover VAMPIRE that performs the matchmaking as explained in Section 4.4. Technically, VAMPIRE is invoked via a Web service with a currently manual translation from WSML to

TPTP (see below). We handle goals, Web services, ontologies, as well as the SDC graph as WSMO entities. For this, we use the WSMO4J API for the programmatic handling and the WSMX Resource Manager for storage and retrieval of the resources. While referring to Appendix B.3 for the detailed technical documentation, the following explains the usage of WSML as the specification language for the prototype and defines the ontology schema for the internal management of SDC graphs as a WSML knowledge base.

WSML as Specification Language

In the preceding elaborations, we have used classical first-order logic (FOL) as the specification language for describing goals and Web services as well as for the underlying domain ontologies. The WSMX system uses WSML as the specification language for all WSMO elements. Furthermore, the functional descriptions of goals and Web services are expected to be defined as WSMO capabilities, which slightly differ from the functional descriptions defined in our framework. However, we can handle this by translating the WSML definitions to the syntax and structure that is required for matchmaking in the SDC prototype.

WSML has been defined as the specification language for the WSMO framework. It consists of a conceptual part for specifying WSMO elements and their description models, and five logical variants which cover the different ontology languages which are considered for the Semantic Web [de Bruijn et al., 2005b]: (1) *WSML Core* relates to Description Logic Programs, (2) *WSML DL* as a Description Logic that is compatible with OWL-DL, (3) *WSML Flight* which is based on F-Logic, (4) *WSML Rule* as fully qualified rule language, and (5) *WSML Full* as first-order logic with auto-epistemic extensions which defines a logical umbrella for all variants. Although not defined as an explicit variant, WSML also defines a first-order logic language with classical model-theoretic semantics. This is referred to as *WSML FOL*, which has the same semantics as classical FOL and is defined as *WSML Full* without the symbols `naf` as negotiation by failure, `!-` for defining constraints, `ofType` for constraining attribute values, and without cardinality constraints [de Bruijn and Heymans, 2006]. We use this as the specification language for the SDC prototype. However, it is not possible to use any other WSML variant when we want to maintain the structure and formal semantics of functional descriptions as defined in Section 4.2 because (1) *WSML Core* does not support variables that are needed for specifying functional descriptions, (2) *WSML DL* is not expressive enough because it does not support nominals that would be needed to define input bindings for our functional descriptions, and (3) *WSML Flight* and *WSML Rule* have minimal model semantics (in contrast to the classical model-theoretic semantics of FOL), and the available reasoners do not provide the necessary reasoning facilities.

Table 5.8: Example for Translation from WSML FOL to TPTP

WSMO Capability in WSML FOL	**Functional Description $\mathcal{D}_{FOL}$** TPTP Representation
goal gtRoot **importsOntology** {_"http://... shipment.wsml#", _"http://... location.wsml#"} **capability** gtRootCapability **sharedVariables** {?SendLoc,?RecLoc,?Package,?Weight} **precondition definedBy** SendLoc[locatedIn **hasValue** loc#world] **memberOf** loc#Location **and** ?RecLoc[locatedIn **hasValue** loc#world] **memberOf** loc#Location **and** ?Package[weight **hasValue** ?Weight] **memberOf** sho#Package **and** ?Weight[includedIn **hasValue** sho#heavy]. **postcondition definedBy** **forall** (?O). ?O [from **hasValue** ?SendLoc, to **hasValue** ?RecLoc, item **hasValue** ?Package , price **hasValue** thePrice] **memberOf** sho#shipmentOrder.	**include**('shipment.ax'). **include**('location.ax'). **input_formula**(gtRoot,**axiom**,(! [SendLoc,RecLoc,Package,Weight,O] : (gt(SendLoc,RecLoc,Package,Weight,O) <=> (% PRECONDITION (locatedIn(SendLoc, world) & locatedIn(RecLoc, world) & package(Package) & weight(Package,Weight) & includedIn(Weight,heavy)) => % EFFECT (shipmentOrder(O) & from(SendLoc) & to(RecLoc) & item(O,Package) & price(O,price)))))).

We need to translate the WSMO capabilities of goals and Web services to the TPTP representation of functional descriptions required for matchmaking with VAMPIRE. Table 5.8 illustrates this for the goal template `gtRoot` which describes the objective of shipping a package of any weight from anywhere to anywhere in the world. The left column shows the WSMO capability description in *WSML FOL*. We define the correspondence to a functional description $\mathcal{D} = (\Sigma^\star, \Omega, IN, \phi^{pre}, \phi^{eff})$ from Definition 4.4 as follows: the **importsOntology** keyword defines the used domain ontologies Ω, the **sharedVariables** correspond to the input variables IN, the **precondition** defines ϕ^{pre} as the conditions on possible start-states, and the **postcondition** relates to ϕ^{eff} which defines the desired end-state and the expected computational outputs. The logical symbols and connectives defined in *WSML FOL* have exactly the same meaning as the classical FOL, so that logical expressions can be translated by syntactic transformation. In order to obtain the TPTP representation of a functional description as a single FOL formula in accordance to Definition 4.6, we define a TPTP input-formula of the form $\forall IN,\ gt(IN) \Leftrightarrow ([\phi^{pre}]_{\Sigma_D \rightarrow \Sigma_D^{pre}} \Rightarrow \phi^{eff})$. The result is shown in the right column of the above table, which serves as the input for performing matchmaking via proof obligations with VAMPIRE as explained in Section 4.4.

This translation is straightforward without loss of information. In the SDC prototype, this is implemented by mapping the WSML descriptions to pre-defined TPTP descriptions. While this defines the rules for the translation, an automated translation from *WSML FOL* to TPTP that is planned in WSMX can be used in the future.

SDC Graph Ontology

As mentioned above, the prototype implementation provides and maintains the SDC graph in form of a knowledge base. For this, we specify an ontology whose schema defines the constructs of the SDC graph while the actual elements of a SDC graph for a particular application are represented as instances of this ontology. This follows the common approach for handling meaningful data in WSMX, which supports the meaning-preserving exchange of information and the reasoning upon the knowledge captured in SDC graphs.

Listing 5.19 shows the ontology schema of the SDC Graph. We define this in *WSML Core* as the least expressive variant of WSML. Essentially, the ontology defines concepts for the elements of SDC graphs, namely: goal templates, intersection goal templates, goal graph arcs, and discovery cache arcs (*cf.* Definition 5.2). The instances of these concepts form the knowledge base which is kept in the system. The goal and Web services descriptions are stored separately, and are referenced in the SDC graph ontology via their unique identifiers. We also define concepts for goal instances and for the matching degrees defined in our approach in order to avoid confusion with similar terminology definitions in other works.

```
wsmlVariant _"http://www.wsmo.org/wsml/wsml-syntax/wsml-core"
namespace {_"http://members.deri.at/~michaels/ontologies/SDContology.wsml#",
          wsml _"http://www.wsmo.org/wsml/wsml-syntax#" }
ontology _"http://members.deri.at/~michaels/ontologies/SDContology.wsml"
 concept goalTemplate
    description impliesType wsml#goal
 concept intersectionGoalTemplate subConceptOf goalTemplate
    parent1 impliesType goalTemplate
    parent2 impliesType goalTemplate
 concept goalInstance
    correspondingGoalTemplate impliesType goalTemplate
    inputbinding impliesType {wsml#instance,wsml#datatype}
 concept goalGraphArc
    sourceGT impliesType goalTemplate
    targetGT impliesType goalTemplate
 concept discoveryCacheArc
    sourceGT impliesType goalTemplate
    targetWS impliesType wsml#webService
     usability impliesType matchingDegree
 concept matchingDegree
 instance exact memberOf matchingDegree
 instance plugin memberOf matchingDegree
 instance subsume memberOf matchingDegree
 instance intersect memberOf matchingDegree
 instance disjoint memberOf matchingDegree
```

Listing 5.19: Ontology Schema for the SDC Graph in WSML Core

From the perspective of conceptual modeling, it would be more appropriate to represent the SDC graph arcs as tenary relations of the form $arc(source, target, degree)$. However, such constructs can not be represented in OWL. Thus, we decided to define the SDC graph ontology in terms of concepts and instances because this can be represented in any ontology language, even as RDF triples. This allows users to easily transfer the captured knowledge to other applications, which is supported by the translation facilities provided by existing WSMO tools, e.g. the OWL2WSML tool in WSMO Studio [Dimitrov et al., 2007].

5.5.2 A Goal-based Web Service Browser

The SDC graph provides an index structure of the available Web services with respect to the goals that can be solved by them. Apart from its primary usage as the search index for optimizing Web service discovery, this knowledge structure can also be used within other SWS techniques for supporting the Web service usage process. As one of the most beneficial techniques, the following presents a prototypical realization of a goal-based Web service browser. Based on the graphical visualization of the SDC graph, this allows clients to browse and inspect Web services on the level of goals that can be solved by them.

Search and browsing facilities for Web services are an important feature for supporting application developers in the detection and inspection of potential candidate services for a specific problem. In conventional Web service technologies, browsing facilities for Web service registries are intended to support the discovery task by manual search and inspection. This is already supported by UDDI on the basis of keyword-based classification schemes [Clement et al., 2004]. In the context of Semantic Web services, the detection and detailed usability analysis of Web services for a given request is automated by respective semantic techniques. However, browsing facilities are also desirable in SWS environments in order to better support the planning and design of client applications.

These should provide an overview of the available Web services on an abstract level, while the technical details are left to the respective SWS techniques. Most existing tools follow the UDDI approach, i.e. they work on keyword-based categorizations of Web service registries and the browsing support resides on a technical level [Bachlechner et al., 2006]. To overcome these deficiencies, we can use the SDC graph in order to provide browsing facilities on the level of requested and provided functionalities, abstracting from the technical details. A main merit for this is that the SDC graph organizes the existing goal templates and the available Web services in a tree-like subsumption hierarchy. This eases the understandability for humans, so that clients can better comprehend the available resources and their relationships on an abstract level which neglects the technical details.

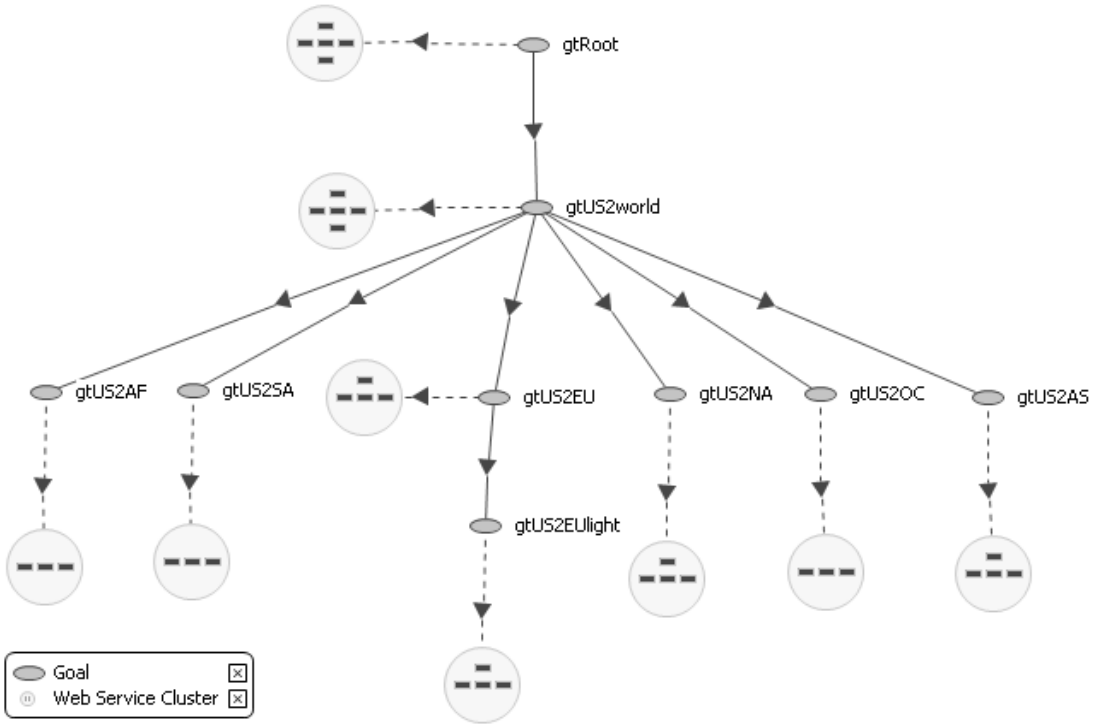


Figure 5.13: SDC Graph Visualization in WSMT

The goal-based Web service browser is implemented as a new plugin for the Web Service Modelling Toolkit WSMT, which is the Integrated Development Environment for the SWS technologies developed around the WSMO framework [Kerrigan et al., 2007]. Figure 5.13 provides a screenshot for the shipment scenario. To provide a comprehensive overview of the available resources, we use the subsumption hierarchy defined in the goal graph as the basic browsing structure. The set of suitable Web services for each goal template is displayed in form of a cluster, which is derived from the discovery cache of the SDC graph.

This SDC graph visualization is initially presented to the user that allows the inspection of the available resources on the level of requested and provided functionalities. For example, from the cluster associated with the goal template `gtRoot` we can easily see that there are five Web services for package shipment, whereof only three are usable for the goal template `gtUS2AF` to ship a package from the USA to Africa. The user can then subsequently navigate to more detailed perspectives, down to the level of the actual element descriptions. This is supported by the already existing browsing and editing facilities provided by WSMT; we refer to [Stollberg and Kerrigan, 2007] for further details. Eventually, this can be extended towards a comprehensive graphical user interface for goal-based SWS environments.

5.6 Summary and Related Work

In this chapter we have specified the Semantic Discovery Caching technique (short: SDC) for enhancing the computational performance of automated Web service discovery engines, which we have identified as the bottleneck for the efficiency and scalability of SWS environments. The following summarizes the central aspects of the SDC technique, positions it within related work, and discusses its applicability as well as complementing techniques.

The primary aim of the SDC technique is to enhance the computational performance of runtime Web service discovery operations. In our two-phase discovery framework, this is concerned with detecting the Web services which are suitable for a given goal instance under functional aspects. A goal instance describes a particular client objective by instantiating a goal template with concrete inputs. Goal templates are generic and reusable objective descriptions which are kept in the system, and the suitable Web services for them are determined at design time. Thus, runtime discovery denotes the time critical and the most frequently required operation in our approach: it is invoked for each client request, and it should be performed in an efficient and reliable manner in order to adequately accomplish the discovery task as the first processing step in SWS environments.

The approach for providing an efficient, scalable, and stable runtime discovery component is to capture the relevant knowledge of design time discovery results, and then effectively use this to minimize the average time for completing runtime discovery tasks. The basis for this is the SDC graph as a unique indexing structure for the efficient search of goal templates and Web services. Its skeletal structure is the goal graph, which organizes the existing goal templates in a subsumption hierarchy with respect to their semantic similarity. Two goal templates G_1, G_2 are considered to be similar if they have at least one common solution, because then mostly the same Web services are usable for them. We define the goal graph such that the only occurring similarity degree is $subsume(G_1, G_2)$, because then the Web services that are usable for the child G_2 are a subset of those usable for parent G_1. The outer layer of the SDC graph is the discovery cache, which captures the minimal knowledge on the functional usability of every Web service for each existing goal template. We have defined inference-rules between similar goal templates and their usable Web services as the logical basis of the SDC technique, shown that there only exists one inferentially correct and minimal SDC graph for a given set of goals and Web services, and specified the algorithms for the automated creation and maintenance of SDC graphs.

On this basis, we have defined the optimized runtime Web service discovery algorithms. In principle, these exploit the knowledge captured in the SDC graph to reduce the search space and minimize the number of matchmaking operations that are necessary to detect

the suitable Web services for a goal instance. We have shown that this enables efficient runtime Web service discovery also within larger search spaces, because only the minimal set of relevant candidates needs to be inspected. This optimization strategy has been chosen with respect to the known performance deficiencies of reasoning techniques. While in this chapter we have specified the the data structures, techniques and algorithms for the SDC technique and presented the prototype implementation, the achievable performance increase is subject to the overall evaluation of this work (see Chapter 6).

Only few existing works address the challenge of enhancing the computational performance of semantically enabled Web service discovery, although this is commonly considered to be important for ensuring the success of SWS technology [Preist, 2004; Cardoso and Sheth, 2006; Fensel and van Harmelen, 2007]. In particular, we are not aware of any other approach that addresses the problem in a similar way as the SDC technique. Existing works mostly apply clustering techniques for Web services in order to reduce the search space. A major difference to SDC is that these works do not take a goal-based approach for Semantic Web services. The following discusses related works for optimizing Web service discovery in more detail, and also investigates the relationship of the SDC technique to caching techniques for performance optimization in other areas.

Web Service Categorization. The majority of related works aims at reducing the search space for Web service discovery by organizing the available Web services in keyword-based categorization schemes. This essentially follows the registry categorization already supported in UDDI [Clement et al., 2004]: each Web service is annotated with one or more keywords which constitute a mostly hierarchical categorization scheme. In the field of SWS, this idea is extended by using ontologies as the underlying categorization scheme. Prominent works are [Srinivasan et al., 2004a] wherein OWL-S descriptions are integrated into a UDDI repository, and [Verma et al., 2005] which uses domain ontologies for annotating and categorizing Web services in accordance to the WSDL-S approach. Other works that follow the same idea are [Tausch et al., 2006; Lara et al., 2006; Abramowicz et al., 2007].

The basic idea is the same as in SDC: the set of relevant candidates is reduced to the Web services in the relevant category as a pre-filtering step before the actual matchmaking. However, the major deficiency of this approach is the imprecision of the keyword-based annotations in comparison to the rich functional descriptions used within the SDC technique. Imagine that a Web service of our running example is annotated with the keywords "package shipment, USA, North America"; this does not state whether "USA" refers to the sender or the receiver, or maybe that US-American regulations hold for the shipment

service. Moreover, this imprecision may cause contradictions with the actual discovery results. Besides, most of the mentioned works require the necessary annotations to be defined by the Web service providers so that the clustering becomes error-prone. In consequence, keyword-based clustering techniques appear to be inadequate for the purpose of optimizing semantically enabled discovery techniques with a high retrieval accuracy. In fact, they appear to be dispensable because the keyword-based annotations merely denote an addendum to the functional descriptions which in anyway are required for the semantic matchmaking.

Indexing of Formal Capability Descriptions. A more sophisticated approach for enhancing the efficiency of Web service discovery in larger search spaces has been presented in a series of papers around [Constantinescu et al., 2005]. This organizes the available Web services in a search tree with respect to their formal functional descriptions. These are defined by so-called *interval constraints* $S = \{\{IN(r,t)\}, \{OUT(r,t)\}, \{PRE(c_x)\}, \{EFF(c_y)\}\}$ – i.e. the sets of inputs and outputs which are described by their role and type, and preconditions and effects which refer to concepts in a background ontology. The search tree is defined to be balanced. Its indexing nodes group a set of Web services with respect to the upper bound $\overline{S}$ as the union of their interval constraints and the lower bound $\underline{S}$ as their intersection, so that $\underline{S} \models S \models \overline{S}$. The actual Web services denote the leaf nodes, defined by pointers from the indexing nodes. The user can pose discovery requests via a special query language. These are evaluated by a best-first traversal of the tree, i.e. first an appropriate indexing node is determined and then its lead nodes are inspected in detail.

Although this approach overcomes the problems of imprecision and error-proneness of keyword-based clustering techniques discussed above, it has several disadvantages in comparison to the SDC technique. At first, the interval constraints merely define sets of predicates and concepts for describing Web services, which is significantly less expressive than our functional descriptions (see Chapter 4). Secondly, the indexing technique is less precise than the SDC graph: while here the indexing nodes merely denote the logical union and intersection of interval constraints – i.e. between sets of predicates and concepts – the SDC graph properly reflects the semantic relationships of goals and Web services on the basis of precise functional descriptions. Thirdly, for each new incoming request the search tree needs to be traversed in a top-down manner until a suitable Web service is detected, and for each step a matchmaker needs to be invoked. In contrast, the SDC technique enables discovery-by-lookup, i.e. the detection of suitable Web services for a given client request without invoking a matchmaker. Finally, it remains unclear how the search tree is created for a given set of Web services, while the automated creation and maintenance of the SDC graph is an integral part of the SDC technique.

Caching Techniques. Caching is a well-established means for performance optimization applied in several areas of computing, e.g. for memory management [Handy, 1998] or for traffic management on the Web [Wessels, 2001]. The basic idea is to capture the results of recent user requests in a cache and then answer new requests from this intermediate storage, which usually is faster than performing the actual computation. An extended form are semantic caching techniques [Keller and Basu, 1996] which keep user requests as logical expressions: if a new query Q' that is semantically similar to a previous query Q, then the answer set can be derived from the cache. Respective studies show that caching techniques can achieve the highest efficiency increase if there are many similar requests [Godfrey and Gryz, 1997; Chidlovskii et al., 1999]. Caching techniques have also been successfully applied for enhancing the computational performance of reasoning techniques (e.g. [Astrachan and Stickel, 1992; Clayton et al., 2002; Nieuwenhuis et al., 2003]).

The SDC technique adopts the principles of caching to the context of Web service discovery. The SDC graph can be considered as the cache structure, whereby the goal templates denote the abstract, semantic descriptions of client requests. The achievable efficiency increase is proportionally dependent on the number of similar goals and Web services in an application. An important difference to other caching techniques is that the SDC graph is stored in a persistent memory, and only the relevant goal and Web service descriptions are loaded into the working memory at runtime. Thus, its size is not critical for the operational reliability so that a periodical cache clearing is dispensable.

To conclude, the SDC technique is a novel and promising approach to enhance the computational performance of automated Web service discovery. Such techniques appear to be necessary in order to warrant the applicability of SWS techniques in real-world scenarios. This can not be achieved by optimizing general purpose reasoning techniques (e.g. [Horrocks et al., 2004; Kiryakov et al., 2005]), because Web service discovery requires several, possibly expensive matchmaking operations on potentially complex functional descriptions.

While we here have specified the data structures and algorithms on the basis of rich functional descriptions which warrant a high retrieval performance, the general approach can be adapted to other descriptions models and languages. We further have mainly focussed on the optimization of runtime Web service discovery. However, also other beneficial techniques can be built upon the SDC graph – e.g. suitable graphical user interfaces for goal-driven SWS techniques. Moreover, the SDC technique can be further extended, e.g. with advanced techniques for managing and reasoning on goals (e.g. [Giorgini et al., 2003]).

Chapter 6

Evaluation

This chapter presents the evaluation of the Web service discovery techniques developed in this work. In order to properly assess the achievable improvements for automated discovery as a central component in SWS environments, the evaluation consists of two parts.

The first one is a *quantitative evaluation* which investigates the retrieval accuracy of the semantically enabled discovery techniques specified in Chapter 4, and in particular determines the enhancements for the computational performance of automated Web service discovery that is achievable with the SDC technique defined in Chapter 5. For this, in Section 6.1 we examine the behavior of our discovery techniques in larger search spaces of available Web services and compare it with other, not or less optimized techniques in terms of time efficiency measurements. This will show that performance optimization is necessary in order to provide efficient and scalable Web service discovery techniques, and that the SDC technique can achieve significant improvements while maintaining the high retrieval accuracy of our semantic matchmaking techniques.

The second part is a *qualitative evaluation* which examines the practical relevance of the developed technology within real-world SOA applications. The aim is to ascertain whether the goal-based approach for Semantic Web services presented in Chapter 3 can be beneficially applied to existing SOA systems, in particular in a way such that the optimized Web service discovery can reveal its potential. For this, in Section 6.2 we define general criteria for the practical applicability of our technology, and then discuss the possible increase in the quality and flexibility of one of the largest actually used SOA systems that is maintained by the US-based telecommunication provider Verizon. We further discuss the employment of our technology within other SOA application areas, in particular in prominent use case scenarios that have been considered in research efforts around Semantic Web services.

6.1 Performance Analysis

We commence with the quantitative evaluation. With respect to the overall aim of this work, the aim is to evaluate the retrieval accuracy of our Web service discovery techniques and in particular to quantify the performance increase that is achievable with the SDC technique. The following explains the methodology for the performance analysis, presents the use case scenario and modeling, and discusses the results of the evaluation tests.

6.1.1 Methodology

The overall aim of this work is to develop an efficient and scalable Web service discovery technique with a high retrieval accuracy. With respect to this, the aim of the quantitative evaluation is to assess the improvements that can be achieved with the techniques specified in the preceding elaborations in terms of quantitative measurements.

For this, we specify evaluation tests that allow us to properly evaluate both the design time and the runtime operations of our two-phased discovery framework with respect to the retrieval accuracy and the computational performance. The first test addresses the creation and management of SDC graphs as the design time operations in our framework. Here, we are interested in the operational correctness of the automatically created SDC graphs as well as the required time for performing update operations. The second test is concerned with runtime Web service discovery, i.e. the detection of functionally suitable Web services for concrete goal instances which denotes the time critical operation in our framework. In order to evaluate the enhancements in computational performance, we compare our optimized runtime discovery techniques with other, not or less optimized engines. We specify the tests such that they cover all relevant aspects, and in particular allow us to examine the behavior of the discovery techniques in larger search spaces of available Web services.

We use the shipment scenario from the Semantic Web service challenge as the use case for the performance analysis, which we already have discussed as the running example in Chapter 5. This is a widely recognized initiative for the demonstration and comparison of semantically enabled Web service discovery techniques based on real-world services (see `http://www.sws-challenge.org`). We closely follow the original scenario description, which defines five Web services for package shipment from the USA to different destinations along with several examples of concrete client requests for Web service discovery. We shall explain the representation of this use case in our framework as well as the resource modeling below in Section 6.1.2. The following specifies the evaluation tests in detail; we shall present and discuss the results below in Section 6.1.3.

Evaluation Test for SDC Graph Creation and Management

The first test is concerned with the design time operations of our two-phase Web service discovery framework. This covers the discovery of suitable Web services for goal templates, and in particular the creation and maintenance of SDC graphs. We consider these operations to be not time critical because they are performed orthogonal to the discovery of suitable Web services for a goal instance at runtime.

As discussed in the detailed specification in Section 5.3, the algorithms for the automated creation and maintenance of SDC graphs can correctly handle all possible situations. However, the following two aspects appear to be important for assessing the usability of the SDC technique in real-world applications: (1) the computational complexity of creating and maintaining the SDC graph for a given and evolving set of goals and Web services, in particular regarding the required processing times, and (2) the correctness of the SDC graph at all times. The latter is a pre-requisite for the proper operation of the SDC technique: the results of the optimized runtime discovery will only be correct if the SDC graph properly captures the results of the design time discovery runs, and also the update operations for handling changes on the resources require a previously correct SDC graph.

In consequence, the evaluation test examines the automated management of the SDC graph for the shipment scenario with respect to these criteria. For this, we define a set of 10 goal templates for package shipment from the USA, and we consider a set of 50 available Web services that consists of the 5 Web services from the original scenario description and 45 other Web services which do not offer package shipment services and thus are not usable for any of the goal templates. This appears to be a sufficiently large search space for the evaluation. Then, we examine the automated SDC graph management with respect to:

1. the required time and the operational correctness for the automated creation of the SDC graph by the subsequent insertion of the goal templates in a top-down manner as well as in a different order

2. the required times for performing change management operations on the existing goal templates and on the available Web services as well as the operational correctness of the updated SDC graph.

This test covers all relevant aspects for evaluating the functional correctness as well as the performance of the design time operations in our framework. The design time Web service discovery is an integral part of the SDC graph creation algorithm, and the use case scenario covers most of the situations that occur in dynamically evolving applications and require updates of the SDC graph, in particular the insertion of goal templates at different

positions, the handling of intersection goal templates, design time Web service discovery for root nodes and child nodes in the SDC graph, and essential change management operations. We perform the test with the SDC prototype implementation presented in Section 5.5.

Evaluation Test for Runtime Web Service Discovery

The second test is concerned with the detection of functionally suitable Web services for a given goal instance as the time critical runtime operations in our framework.

Recalling from the preceding elaborations, a goal instance formally describes a particular client objective by instantiating a goal template with concrete inputs. In order to gain a better flexibility, we expect every usage request for Web services to be described in terms of a goal instance for which the usable Web services are detected and executed dynamically. The first processing step in SWS environments for automated goal solving is functional Web service discovery which determines the suitable candidates out of the available Web services; their actual usability is then further inspected within subsequent processing steps. This is required for each new goal instance, whereby the discovery engine needs to take all available Web services into account while the subsequent mechanisms merely need to work with the discovered candidates.

Because of this, the discovery of functionally suitable Web services at runtime is the time critical operation in our framework. It should be performed in an efficient and reliable manner in order to adequately accomplish the first processing step and therewith warrant the operational suitability of SWS environments for effectively solving concrete client requests. Our optimized runtime discovery techniques address this challenge by exploiting the knowledge captured in the SDC graph in order to minimize the relevant search space as well as the number of necessary matchmaking operations for completing a runtime discovery task (see Section 5.4). In consequence, the main evaluation interest is to quantify the increase in the computational performance for runtime discovery that is achievable with the SDC technique, and also the retrieval accuracy of the applied matchmaking techniques.

To evaluate this in terms of quantitative measurements, we examine the behavior of our runtime discovery engine in increasingly larger search spaces and compare it with other engines. As the test data, we define 10 goal instances for package shipment from the USA which instantiate the goal templates defined in the first test (see above). To simulate realistic scenarios, we define the set of available Web services to always contain the five Web services from the original shipment scenario description; all others do not offer package shipment and thus are not usable for any of the goal instances. We further perform two comparison tests in order to properly assess the achievable enhancements. The first one compares the

SDC-enabled runtime discoverer with a naive engine that does not use any optimization techniques, and the second one compares the full SDC-enabled engine with an engine that merely utilizes the captured design time discovery results without further exploiting the SDC graph. We adopt the following standard criteria for judging the computational performance of the discovery engines: *efficiency* as the time required for completing a discovery task, *scalability* as the ability to deal with a large search space of available Web services, and *stability* as a low variance of the execution time of several invocations [Ebert et al., 2004]. The following explains the design of the comparison tests in more detail.

SDC Runtime Discoverer vs. Naive Engine. The first comparison test examines the behavior of the SDC-enabled runtime discoverer against a naive engine that does not use any optimization techniques. The aim is to show the necessity for the optimization of runtime Web service discovery, and to determine the increase rate that is achievable with the SDC technique. We compare the behavior for both the discovery of a single Web service and the discovery of all functionally suitable Web services for a given goal instance.

We use the runtime discovery component from the SDC prototype for the comparison test, which implements the optimized discovery algorithms specified in Section 5.4. The comparison engine simulates a naive Web service discovery for goal instances without any optimization. It investigates the available Web services in a randomized order, and checks the basic matching condition for the functional suitability of each Web service for the given goal instance (*cf.* Definition 4.8 in Section 4.3.2). This is implemented as an extension of the SDC prototype so that both engines expose the same retrieval accuracy because they use the same descriptions of goals and Web services, the same matchmaking techniques, and the same technical infrastructure. We choose this engine for the comparison test in order to evaluate the effectiveness of our optimized discovery algorithms. A quantitative comparison with existing Web service discovery techniques appears to be dispensable because a comparable goal-based discovery engine does not exist and most other approaches lack in the achievable retrieval accuracy due to less expressive resource descriptions and less precise matchmaking techniques. We refer to Section 4.5 for the detailed discussion on this.

The comparison test is performed by a sufficiently large number of repetitive test runs in order to examine and compare the durable behavior of the discovery engines. For the discovery of a single Web service, we consider sets of 10 up to 2000 available Web services which always contain the 5 Web services from the original shipment scenario description while all others are not usable. This simulates the size and structure of search spaces that can be expected in real-world scenarios. We then perform 50 repetitive test runs for each of our 10 test goal instances under every set of available Web services. Finally, we prepare

the test results in terms of statistical standard notions such as the arithmetic mean, the median, and the standard deviation in order to analyze and compare the behavior of the discovery engines with respect to the computational efficiency. The test set-up for comparing the discovery of all functionally suitable Web services is analog. Because of the smaller variance that can be expected from the naive engine for this discovery task, it is sufficient to consider search spaces of 10 up to 500 Web services and perform 25 repetitive test runs. Regarding the retrieval accuracy, both engines always detect the same Web services to be usable because they use the same matchmaking techniques as explained above.

SDC$_{full}$ vs. SDC$_{light}$. The second test compares the SDC-enabled runtime Web service discovery with a discovery engine for goal instances that merely uses the knowledge on suitable Web services for goal templates without further exploiting the SDC graph. The motivation beyond this is that a significant part of the achievable efficiency increase for runtime discovery results from the pre-filtering of the potential candidates on the basis of the corresponding goal template that is defined for the goal instance. With respect to this, aim is to evaluate the computational performance of our optimized runtime discovery in comparison to a simpler optimization technique that merely performs pre-filtering.

In our two-phase discovery framework, we only need to consider those Web services as potential candidates for a given goal instance $GI(G, \beta)$ which are suitable for the corresponding goal template G (*cf.* Theorem 4.1 in Section 4.3.2). In addition to this pre-filtering, a central method of our optimized runtime Web service discovery algorithms is the *refinement* of the provided goal instance. This replaces the initially defined corresponding goal template G by the most appropriate goal template G' as the lowest existing child node of G in the SDC graph for which the goal instantiation condition $GI(G, \beta) \models G'$ is satisfied; the set of suitable Web services for G' is minimal in comparison to all other existing goal templates, so that the relevant search space is reduced to a minimum (see Section 5.4).

In order to determine the performance enhancements that is achievable with this exhaustive exploitation of the SDC graph, we define the comparison engine SDC$_{light}$ to perform runtime Web service discovery the same way as our optimized algorithms but without the *refinement*-method. We implement this as an extension of the SDC prototype so that both engines use the same technical infrastructure and expose the same retrieval accuracy. We extend the original shipment scenario with additional Web services for this comparison test, because the *refinement*-method unfolds noticeably for larger numbers of potential candidates. As for the above comparison test, a quantitative comparison with other optimization techniques for Web service discovery appears to be dispensable because the differences to the SDC technique reside on the qualitative level as already discussed in Section 5.6.

6.1.2 Use Case Scenario and Modeling

We now explain the definition and resource modeling for the shipment scenario in our framework. The use cases in the SWS challenge are described in natural language without imposing a specific conceptual model for Semantic Web services [Petrie et al., 2008]. Thus, a central part of the demonstration and testing of our Web service discovery techniques is to map the generic description of the shipment scenario into our conceptual model.

The original scenario description defines five Web services for package shipment from the USA to different destinations under specific conditions, and several examples of concrete client requests for Web service discovery. In our model, these relate to goal instances for which Web services shall be discovered at runtime. Elements that correspond to goal templates are not defined in the original scenario description. The following explains the definition and modeling of the goals and the Web services in our framework, which very closely follows the original scenario description. We also present the resulting SDC graph and define the goal instances used for the evaluation tests.

Resource Modeling. Our two-phased discovery framework is based on the distinction of *goal templates* as generic and reusable objective descriptions and *goal instances* that represent particular client requests by instantiating a goal template with concrete inputs. The part of goal and Web service descriptions which is relevant for Web service discovery are *functional descriptions* that formally describe the requested and provided functionalities while abstracting from the behavioral as well as other technical details.

In order to warrant the high retrieval accuracy of our semantic matchmaking techniques, we have defined formal functional descriptions that precisely describe the solutions of goals and the executions of Web services with respect to the possible start- and end-states (see Chapter 4). Table 6.1 shows this for the goal template `gtRoot` which describes the objective of shipping a package of any weight from anywhere to anywhere in the world, and the Web service `Weasel` from the original scenario description that offers package shipment in the USA. The functional descriptions are structurally identical: the preconditions ϕ^{pre} define conditions on the location of the sender and the receiver as well as the maximal weight of the package, and the effects ϕ^{eff} state that the expected, respectively the provided output is a shipment order with respect to the inputs. The input variables *IN* occur as free variables in both ϕ^{pre} and ϕ^{eff} in order to explicitly specify their logical dependency. A goal instance formally describes a particular client request by instantiating the input variables specified for the goal template with concrete values. This input binding is also used to actually invoke a Web service in order to solve the goal instance (see Chapter 3).

Table 6.1: Examples for Functional Descriptions

	Goal Template `gtRoot`		**Web Service** `Weasel`
Ω:	location & shipment ontologies	Ω:	location & shipment ontologies
IN:	{?s,?r,?p,?w}	IN:	{?s,?r,?p,?w}
ϕ^{pre}:	$sender(?s) \wedge locatedIn(?s, world)$	ϕ^{pre}:	$sender(?s) \wedge locatedIn(?s, usa)$
	$\wedge receiver(?r) \wedge locatedIn(?r, world)$		$\wedge receiver(?r) \wedge locatedIn(?r, usa)$
	$\wedge package(?p) \wedge weight(?p, ?w)$		$\wedge package(?p) \wedge weight(?p, ?w)$
	$\wedge maxWeight(?w, heavy)$.		$\wedge maxWeight(?w, heavy)$.
ϕ^{eff}:	$\forall ?o, ?price.\ out(?o) \Leftrightarrow ($	ϕ^{eff}:	$\forall ?o, ?price.\ out(?o) \Leftrightarrow ($
	$shipmentOrder(?o, ?p)$		$shipmentOrder(?o, ?p)$
	$\wedge from(?p, ?s) \wedge to(?p, ?r)$		$\wedge from(?p, ?s) \wedge to(?p, ?r)$
	$\wedge costs(?o, ?price)$).		$\wedge costs(?o, ?price)$).

We define the relevant background knowledge in two separate domain ontologies, therewith following the principle of modular ontology design. The `location ontology` essentially provides a knowledge base of all continents, countries, and cities in the world; their geographic relationships are defined via the transitive predicate `locatedIn(?x,?y)`. The `shipment ontology` specifies the concepts package, sender, receiver, shipment order, and differentiates weight classes whose formal relationship is defined by a transitive inclusion predicate, e.g. `includedIn(light,heavy)`. Appendix C.1 provides the complete ontology specifications as well as the goal template and Web service descriptions in *WSML FOL*, which we use as the specification language in the SDC prototype (see Section 5.5).

The functional descriptions as defined here merely define constraints on the locality and the package weight. The original scenario description defines further aspects on the available Web services, e.g. the actual price for a particular shipment in dependency of the package weight and constraints on the delivery time. This information can be used to either discard a functionally usable Web service due to incompatibility with respective constraints or policies defined by the client, or to choose the best candidate on the basis of a suitability ordering. We consider this to be performed by selection and ranking mechanisms which follow *after* functional Web service discovery: the purpose of Web service discovery is to detect functionally suitable candidates out of the available Web services, while their actual usability for solving a given goal is then further investigated in subsequent processing steps (see Section 2.2.2). Thus, we consider the functional descriptions as illustrated above to be sufficient and appropriate for our purposes. The other goal templates and Web services in the shipment scenario are described analogously, differing in the conditions on the sender and receiver location and the weight of the shipped package as we shall explain below.

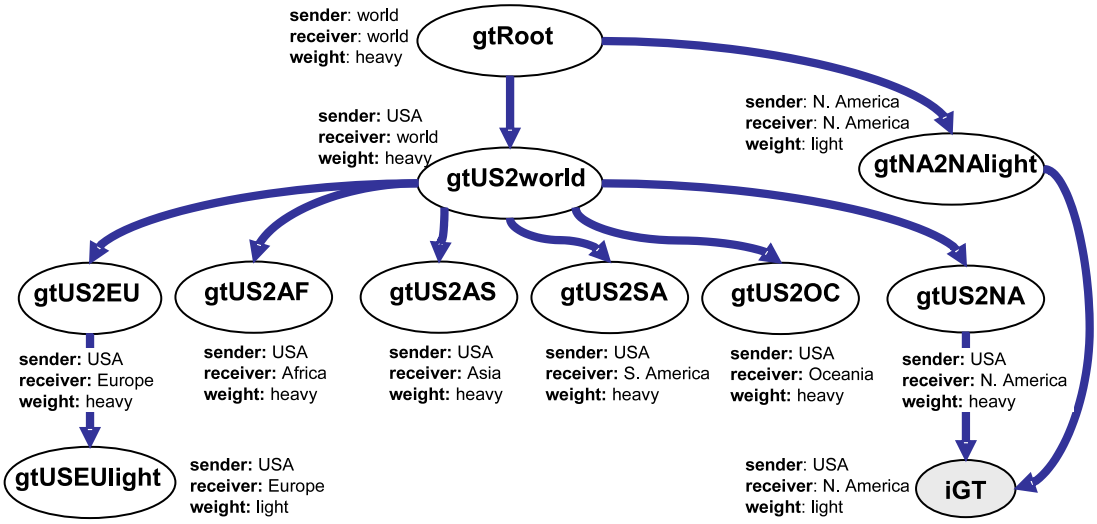
SDC Graph. We now turn towards the actual resources that are used for the evaluation. The following explains the relevant goal templates and Web services, and presents their organization in the SDC graph that we create and use in our evaluation tests.

As outlined above, we consider the five Web services defined in the original scenario description as the only available Web services for package shipment. Table 6.2 provides an overview of the provided functionalities, i.e. the conditions on the locality and the maximal weight as discussed above. We see that all offer package shipment from the USA to different destinations; we abstract them to the level of continents, and denote the specific weight restrictions by the weight classes specified in the shipment ontology (see above).

Table 6.2: Overview of Web Services in the Shipment Scenario

Web Service	Sender	Supported Destinations	Max. Weight
Muller	USA	Africa, Europe, North America, Asia	weightClass50
Racer	USA	Africa, Europe, North America, South America, Asia, Ocenania	weightClass70
Runner	USA	Europe, South America, Asia, Ocenania	heavy
Walker	USA	Africa, Europe, North America, South America, Asia, Ocenania	weightClass50
Weasel	USA	USA	heavy

We then define a set of 10 goal templates that can be solved by these Web services. Figure 6.1 below shows their organization in the SDC graph which we shall use for the performance tests. The upper part of the figure shows the goal graph, which organizes the goal templates in a subsumption hierarchy with respect to their semantic similarity. The most general goal template for the shipment scenario is `gtRoot`, which describes the objective of shipping a package of any weight from anywhere to anywhere in the world (see the functional description in Table 6.1 above); it hence is the root of the SDC graph. The goal template `gtUS2world` is for shipping packages of any weight from the USA to anywhere in the world. It holds that *subsume*(`gtRoot`, `gtUS2world`), so that `gtUS2world` is a child node of `gtRoot` in the SDC graph. Analogously, the direct children of `gtUS2world` are goal templates for package shipment from the USA to specific continents; the identifiers of the goal templates indicate the requested functionality. We further define the goal template `gtUS2EUlight` for shipping light packages from the USA to a destination in Europe; the weight class *light* includes all weights from 0 to 10 kg. We also define the goal template `gtNA2NAlight` for shipping light packages within North America. This is a child of `gtRoot`,



Usability Desgrees	Muller	Racer	Runner	Walker	Weasel
gtRoot	subsume	subsume	subsume	subsume	subsume
gtUS2world	subsume	subsume	subsume	subsume	subsume
gtNA2NAlight	intersect	intersect	X	intersect	intersect
gtUS2AF	intersect	intersect	X	intersect	X
gtUS2AS	intersect	intersect	plugin	intersect	X
gtUS2EU	intersect	intersect	plugin	intersect	X
gtUS2NA	intersect	intersect	X	intersect	subsume
gtUS2OC	X	intersect	plugin	intersect	X
gtUS2SA	X	intersect	plugin	intersect	X
gtUS2EUlight	plugin	plugin	plugin (omitted)	plugin	X
iGT	plugin	plugin	X	plugin	intersect

Figure 6.1: Overview of the SDC Graph for the Shipment Scenario

but its similarity degree with `gtUS2world` is *intersect*. As discussed in detail in Section 5.3.2, this is represented in the SDC graph by the intersection goal template `iGT`, which describes the objective of shipping light packages from the USA to North America and is allocated as a child of `gtUS2NA` and `gtNA2NAlight`.

The lower part of Figure 6.1 shows usability degrees of the five Web services for the individual goal templates. This is the knowledge captured in the discovery cache of the SDC graph, which we here represent in form of a table in order to maintain the readability. The usability degrees denote the actual matchmaking results of runtime Web service discovery

runs as we shall show below in Section 6.1.3. Let us discuss some aspects of this in order to gain a better understanding of the use case structure. The usability degree of all five Web services for `gtRoot` as well as for `gtUS2world` is *subsume* because none of the Web services offers shipment to every destination in the world. On the level of the children of `gtUS2world`, the most frequent usability is *intersect* while some of the Web services are not usable any longer. For instance, consider the goal template `gtUS2EU` for shipping packages of any weight to Europe. Here, the Web services `Muller, Racer` and `Walker` are usable under the *intersect* degree because they also support shipment to other continents but only for packages up to 70 kg, respectively 50 kg; `Runner` is usable under the *plugin* degree because it does not define a maximal weight limit and also supports shipment to other continents, and `Weasel` is not usable because it only offers package shipment within the USA. The goal templates `gtUS2EUlight` and `iGT` denote the most specialized objective descriptions. Because the Web service either cover the requested functionality completely or not at all, the most frequent usability degree on the lower levels of the goal graph is *plugin*. Here, the arc (`gtUS2EUlight`, `Runner`) is omitted in the SDC graph because its usability degree can be directly inferred from *plugin*(`gtUS2EU`, `Runner`) as defined in Section 5.2.2.

Goal Instances. The final aspect for modeling the shipment scenario within our framework is the definition of goal instances. These describe the concrete client requests, analogous to those defined in the original scenario description. We here define 10 goal instances on the basis of the goal templates specified above, which serve as the test data for the runtime discovery evaluation tests. Following the original scenario description, we define all goal instances such that the sender is located in California.

We recall that a goal instance $GI(G, \beta)$ is defined by its corresponding goal template G and an input binding β that defines a value assignment for the input variables specified for G. A goal instance must be a valid instantiation of its corresponding goal template, which is given if the functional description of G is satisfiable under the concrete input values defined in β (see Section 4.3.2). Moreover, we assume that there might exist a goal template G' which is more appropriate for a goal instance than the one initially defined by the client. This is exploited by the optimized runtime discovery algorithms in order to achieve a maximal reduction of the relevant search space (see Section 5.4).

Table 6.3 provides a concise overview of the goal instance definitions, covering all relevant information for the following discussions. From left to right, the columns define the identifier of each goal instance, the initially defined corresponding goal template G, the input binding β, the most appropriate goal template G' among the existing ones, and the set of functionally suitable Web services. Each of the goal instances defined here is a valid instantiation of its

Table 6.3: Overview of Goal Instances for the Shipment Scenario

Goal Instance	corresponding Goal Template	Input Binding			most approp. Goal Template	usable Web Services
		sender	receiver	weight		
gi1	gtUS2AF	San Francisco	Tunis	1 kg	gtUS2AF	Muller Racer Walker
gi2	gtRoot	Los Angeles	Luxembourg City	1.5 kg	gtUS2EUlight	Muller Racer Runner Walker
gi3	gtUS2world	Berkeley	Tunis	50.5 kg	gtUS2AF	Racer
gi4	gtUS2EU	Paolo Alto	Bristol	4.3 kg	gtUS2EUlight	Muller Racer Runner Walker
gi5	gtUS2NA	Los Angeles	New York City	5.5 kg	iGT	Muller Racer Walker Weasel
gi6	gtUS2world	Monterey	Berlin	60 kg	gtUS2EU	Racer Runner
gi7	gtUS2world	Santa Barbara	Sydney	17.3 kg	gtUS2OC	Racer Runner Walker
gi8	gtUS2SA	San Francisco	Quito	7.58 kg	gtUS2SA	Racer Runner Walker
gi9	gtUS2AS	Stanford	Beijing	57.8 kg	gtUS2AS	Racer Runner
gi10	gtUS2EUlight	San Francisco	Amsterdam	9.99 kg	gtUS2EUlight	Muller Racer Runner Walker

corresponding goal template. Regarding the input bindings, the table shows abstractions from the actual input bindings. These need to define the sender, the receiver, and the package to be shipped in order to properly instantiate the goal templates. We here merely show the specified locations and the package weight as the relevant information for functional Web service discovery. As above, the usable Web services for the goal instances as shown in the table denote the results of actual runtime discovery runs. In accordance to our two-phase Web service discovery model, we observe that the set of usable Web services for each goal instance is always a subset of those Web services usable for its corresponding goal template (see Section 4.1). Naturally, this holds for both the initially defined goal template G as well as for the most appropriate goal template G'.

6.1.3 Results and Discussion

The following presents the results of the evaluation tests for the design time operations and the comparison tests for runtime Web service discovery as specified above in Section 6.1.1. We provide the results of each test along with the measured times, explain where these come from, and discuss the meaning for optimizing the Web service discovery task.

The evaluation tests have been run as JUnit tests for the SDC prototype implementation on a conventional laptop with 2 GHz Intel processor and 1 GB of RAM. The following presents the evaluation results in an aggregated manner which is sufficient for the purpose of analysis and discussion. Further details on the test implementations and the obtained evaluation data are provided in Appendix C; all Java sources, the original data, and the log-files for the evaluation tests are provided on the accompanying CD-R.

SDC Graph Management

The first evaluation test is concerned with the automated creation and maintenance of the SDC graph for our use case scenario. In order to properly evaluate all relevant aspects of the design time operations in our framework, we have defined the test to consist of three parts: (1) creating the SDC graph for the shipment scenario by the subsequent insertion of all goal templates in a top-down manner, (2) creating the SDC graph by inserting the goal templates in a different order, and (3) maintenance updates of the SDC graph when goal templates and Web services are removed or added. The main interest for the evaluation is the required time for performing the respective operations as well as the operational correctness of the SDC graph at all times. The following presents and discusses the test results; additional technical details are provided in Appendix C.2.

SDC Graph Creation. We commence with the creation of the complete SDC graph for the shipment scenario as shown above in Figure 6.1 in a top-down manner. This means that we first insert the goal template `gtRoot` which will become the root node of the SDC graph, and then add the other 9 goal templates one after another. For the test we consider 50 available Web services of which only the 5 from the original scenario description offer package shipment while all others are not usable for any of the goal templates.

At first, let us examine the correctness of the automatically generated SDC graph. To verify this, Figure 6.2 shows the SDC graph that has been created by the `SDC Graph Creator` component of our prototype implementation. The graphical representation has been generated from the internal representation as a WSML knowledge base by the GraphViz tool on the basis of the DOT graph representation language (open source tool available at

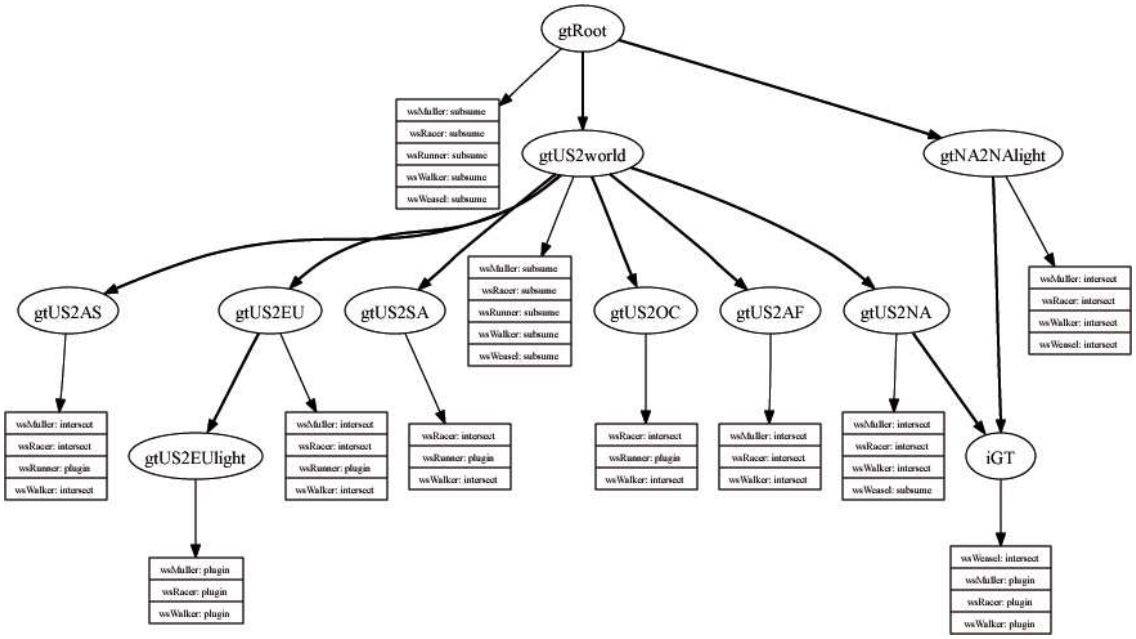


Figure 6.2: Created SDC Graph for the Shipment Scenario

`http://www.graphviz.org/`); we provide the actually created WSML knowledge base in Appendix C.2. We observe that the SDC graph correctly defines the relevant relationships of the goal templates and Web services in our test bed. The goal graph as well as the usability of the available Web services is exactly the same as discussed above in Section 6.1.2. Also the intersection goal template `iGT` is defined and allocated at the correct position, and the redundant discovery cache arc (`gtUS2EUlight`, `Runner`) is omitted.

Let us now investigate the creation of this SDC graph in more detail. We recall that the SDC graph for a given set of goal templates and Web services is created by the subsequent insertion of the goal templates (see Section 5.3.1). A newly inserted goal template G is first allocated in the goal graph such that the existing goal templates are organized in a proper subsumption hierarchy without any redundant arcs. Then, the discovery cache is created, respectively updated so that it defines the minimal set of arcs that are necessary to infer the precise usability degree of every available Web service for each existing goal template. For this, the *child node discovery* method merely inspects the usable Web services for the parents of G when this has been inserted as a child node in the goal graph; otherwise, the *root node discovery* method is applied which needs to also inspect all other available Web services. With respect to this, Table 6.4 provides an overview of the distinct insertion operations for creating the complete SDC graph along with the measured times.

Table 6.4: Operations and Times for SDC Graph Creation (Top-Down)

OPERATION		PERFORMED ACTIONS	TIME (in sec)
1	insert gtRoot	1. insert *gtRoot* as first goal template 2. create discovery cache for *gtRoot* (root node discovery)	19.88
2	insert gtUS2world	1. insert *gtUS2world* as child of *gtRoot* 2. create discovery cache for *gtUS2world* (child node discovery via *gtRoot*)	2.156
3	insert gtUS2AF	1. insert *gtUS2AF* as child of *gtUS2world* (via *gtRoot*) 2. create discovery cache for *gtUS2AF* (child node discovery via *gtUS2world*)	3.919
4	insert gtUS2AS	1. insert *gtUS2AS* as child of *gtUS2world* (via *gtRoot*) 2. create discovery cache for *gtUS2AS* (child node discovery via *gtUS2world*)	3.905
5	insert gtUS2EU	1. insert *gtUS2EU* as child of *gtUS2world* (via *gtRoot*) 2. create discovery cache for *gtUS2EU* (child node discovery via *gtUS2world*)	4.798
6	insert gtUS2EUlight	1. insert *gtUS2EUlight* as child of *gtUS2EU* (via *gtRoot, gtUS2world*) 2. create discovery cache for *gtUS2EUlight* (child node discovery via *gtUS2EU*)	4.356
7	insert gtUS2NA	1. insert *gtUS2NA* as child of *gtUS2world* (via *gtRoot*) 2. create discovery cache for *gtUS2NA* (child node discovery via *gtUS2world*)	4.998
8	insert gtUS2SA	1. insert *gtUS2SA* as child of *gtUS2world* (via *gtRoot*) 2. create discovery cache for *gtUS2SA* (child node discovery via *gtUS2world*)	6.023
9	insert gtUS2OC	1. insert *gtUS2OC* as child of *gtUS2world* (via *gtRoot*) 2. create discovery cache for *gtUS2OC* (child node discovery via *gtUS2world*)	6.646
10	insert gtNA2NAlight	1. insert *gtNA2NAlight* as child of *gtRoot* *2.* create discovery cache for *gtNA2NAlight* (child node discovery via *gtRoot)* *3.* detect *intersect(gtUS2world,gtNA2NAlight*) 4. create *iGT as child of gtUS2NA* and *gtNA2NAlight* *5. discovery cache for iGT* (child node discovery via *gtUS2NA, gtNA2NAlight)*	8.137
		TOTAL TIME:	**64.818**

We commence with inserting `gtRoot` as the first goal template into the SDC graph. We here need to perform *root node discovery* in order to determine the suitable Web services and create the discovery cache; this takes about 20 seconds because all 50 available Web services need to be inspected. Next, we insert `gtUS2world`, which becomes a child of `gtRoot`. We here can use the *child node discovery* which takes significantly less time because we only need to inspect the five Web services which are usable for `gtRoot`. Analogously, the operations 3 – 9 denote insertions of new child nodes to the SDC graph. The slightly increasing times result from the need for investigating the semantic similarity with already existing child nodes, and also from the different numbers of matchmaking operations that are necessary for the discovery cache creation. Operation 10 requires more time because in the course of inserting `gtNA2NAlight` we also need to create and insert the intersection goal template `iGT` into the SDC graph. The average time for a single matchmaking operation in this test is 110 milliseconds; several of these are needed in order to determine the actual similarity degree of two goal templates, respectively the usability degree of a Web service.

SDC Graph Creation - Other Insertion Order. In the above test, we have created the SDC graph by the subsequent insertion of the goal templates in a top-down manner. In real-world applications, we assume that the goal templates are created step-by-step and then are separately inserted into the SDC graph. To warrant the operational correctness of the SDC technique in such situations, it is necessary that the SDC graph is correct after each insertion operation regardless of the goal template insertion order.

To evaluate this, the following examines the stepwise creation of the SDC graph for the shipment by successively inserting three goal templates in an order that does not correspond to their position in the final SDC graph. We first insert the goal template `gtUS2EUlight`, which eventually will become a child node in the SDC graph. Then, we insert `gtUS2world` which becomes the new root node of the SDC graph, and finally we insert `gtUS2EU` which will be allocated between the two previously inserted goal templates in the subsumption hierarchy. This test set-up requires modifications on both the goal graph as well as the discovery cache of the SDC graph. Table 6.5 provides an overview of the insertion operations along with the measured times, and Figure 6.3 below shows the visual representation of the SDC graph created by our prototype implementation after each insertion operation.

Table 6.5: Operations and Times for Stepwise SDC Graph Creation

OPERATION		PERFORMED ACTIONS	TIME (in sec)
1	insert gtUS2EUlight	1. insert *gtUS2EUlight* as first goal template 2. create discovery cache for *gtUS2EUlight* (root node discovery)	20.201
2	insert gtUS2world	1. insert *gtUS2world* as new root node 2. re-allocate *gtUS2EUlight* to become a child of *gtUS2world* 3. create discovery cache for *gtUS2world* (root node discovery)	18.693
3	insert gtUS2EU	1. insert *gtUS2EU* as a child of *gtUS2world* 2. re-allocate *gtUS2EUlight* to become a child of *gtUS2EU* 3. create discovery cache for *gtUS2EU* (child node discovery) 4. remove redundant discovery cache arcs for *gtUS2EUlight*	2.687

Similar to the above test, the insertion of `gtUS2EUlight` as the first goal template takes about 20 seconds because we need to perform *root node discovery* for creating the discovery cache. This needs to inspect all 50 available Web services. The created SDC graph defines the correct usability degrees of all 4 Web services that are actually usable for `gtUS2EUlight` (see Figure 6.1). In the second operation, `gtUS2world` is properly allocated as a new root node in the SDC graph. We here also need to perform *root node discovery* to create the discovery cache. This however is a little bit faster than the first operation because the algorithm can make use of the knowledge captured in the previously existing

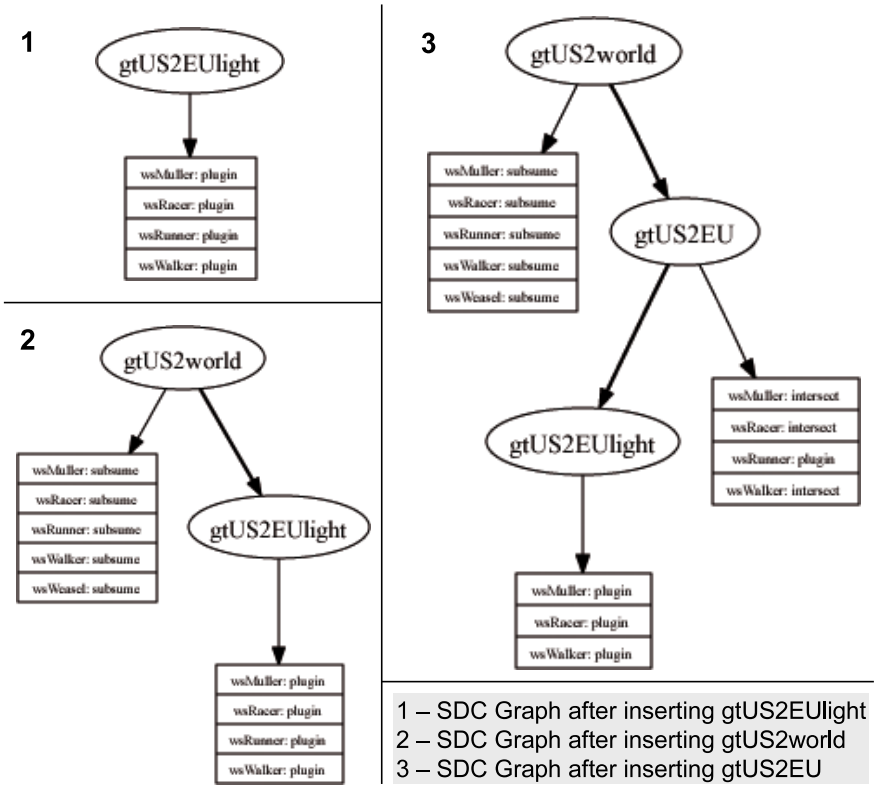


Figure 6.3: Stepwise Creation of the SDC Graph

discovery cache. In the third operation, `gtUS2EU` is properly allocated as an intermediate node between `gtUS2world` and `gtUS2EUlight` in the goal graph. Here, the previously existing goal graph arc (`gtUS2world`, `gtUS2EUlight`) is removed in order to avoid redundancy. We can use the *child node discovery* method to create the discovery cache for `gtUS2EU`, which explains the significantly faster processing time than the previous insertion operations. Finally, the algorithm correctly removes the previously existing discovery cache arc (`gtUS2EUlight`, `Runner`) which now has become redundant in the SDC graph.

SDC Graph Maintenance. The final part of the evaluation test for the SDC graph management techniques is concerned with the maintenance of the SDC graph when changes on the existing goal templates or the available Web services occur. Such changes must be properly reflected in the SDC graph in order to warrant its operational correctness in dynamically changing environments. For this, we have defined algorithms for ensuring that the SDC graph maintains its structure and properties when a goal template is removed or modified or when a Web service is added, removed, or modified (see Section 5.3.3).

Table 6.6: Actions and Times for SDC Graph Maintenance Operations

OPERATION		PERFORMED ACTIONS	TIME (in sec)
1	remove gtUS2EU	1. remove *gtUS2EU* from the SDC graph 2. re-allocate *gtUS2EUlight* to be a child node of *gtUS2world* 3. materialize previously omitted discovery cache arcs 4. remove discovery cache of *gtUS2EU*	0.016
2	remove gtNA2NAlight	1. remove *gtNA2NAlight* from the SDC graph 2. remove discovery cache of *gtNA2NAlight* 3. also remove *iGT* and its discovery cache from the SDC graph	0.025
3	remove WS Muller	remove all discovery cache arcs from the SDC graph that have Web service *Muller* as the target	0.062
4	(re-)insert WS Muller	1. determine usability degree of Web service *Muller* for *gtRoot*, add the respective discovery cache arc 2. determine the usability of WS Web service *Muller* for children of *gtRoot*, and add all non-redundant discovery cache arcs	3.891

To evaluate this, we perform the basic maintenance operations on the complete SDC graph for the shipment scenario that has been created in the first test above. We here choose four operations which are sufficient to test the correctness of the SDC graph maintenance techniques. Table 6.6 provides an overview of the operations along with the measured times for performing them with the `Evolution Manager` in the SDC prototype.

The first operation removes the goal template `gtUS2EU` from the SDC graph. Essentially, this is done by removing `gtUS2EU` and all its goal graph and discovery cache arcs. However, the resulting SDC graph must properly define the relevant relationships between the remaining goal templates and their usable Web services. For this, we need to properly re-allocate the children of the removed goal template within the SDC graph. In our test bed, `gtUS2EUlight` becomes a direct child of `gtUS2world`. We also must re-materialize the previously omitted discovery cache arc (`gtUS2EUlight`, `Runner`) because this can no longer be inferred after removing the discovery cache of `gtUS2EU`. As the second operation, we remove the goal template `gtNA2NAlight`. This requires to also remove the intersection goal template `iGT` because one of its parents does no longer exist and we hence can not perform matchmaking on the intersection goal template anymore (*cf.* Definition 5.3).

In the third operation, we remove the Web service `Muller` from the SDC graph. For this, we merely need to delete all discovery cache arcs which define this Web service as the target element. Figure 6.4 below shows the updated SDC graph after the three removal operations. We observe that all modifications have been performed correctly. We further see that each of the removal operations requires less than 100 milliseconds. The reason for this that here none of the removal operations requires matchmaking; the algorithms merely modify WSML knowledge base wherein the SDC graph is kept within the system, which

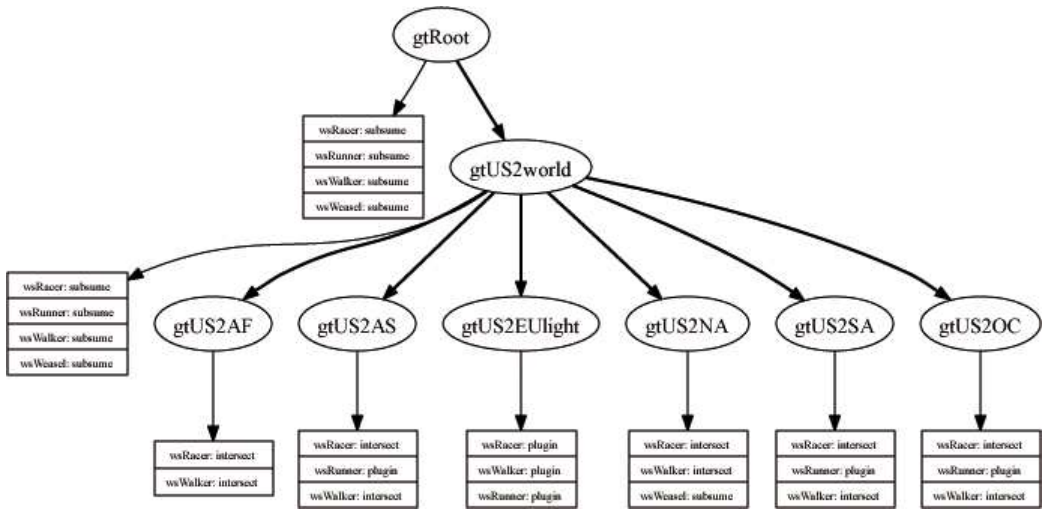


Figure 6.4: Updated SDC Graph after Removal Operations

can be done effectively via the WSMO4J API. We complete the maintenance test with the functionality for adding new Web services to the SDC graph. For this, the forth operation in Table 6.6 re-inserts the Web service `Muller`, i.e. the one that has been removed in the previous operation. The algorithm commences with inspecting the usability for `gtRoot` as the root node of the updated SDC graph, and then subsequently determines the usability for the child nodes on the basis of the already existing knowledge in the discovery cache. This takes around 4 seconds because several matchmaking operations are necessary to determine the actual usability degree of Web service `Muller` for each existing goal template.

To conclude, the tests demonstrate that our automated management techniques can create the operationally correct SDC graph for the shipment use case scenario, independent of the insertion order of the goal templates, and also that the maintenance support properly updates the SDC graph when changes on the available resources occur. This ensures the functional correctness of the SDC technique for optimizing the Web service discovery task, so that the first main aspect of this evaluation test can be judged positively. Regarding the required times managing SDC graphs, the creation of the complete SDC graph for our test bed takes about 1 minute. Once the root node of the SDC graph has been established, the average time for inserting additional goal templates as well as for other update operations is around 5 seconds; if no matchmaking is required, then the required time is negligible. The measured times are of course dependent on the application scenario. However, we assume to find similar relationships of the relevant goal templates and the available Web services in other scenarios. We shall discuss this in more detail in Section 6.2.

Runtime Web Service Discovery

We now turn towards the evaluation of the improvements for runtime Web service discovery that are achievable with the SDC technique. This is concerned with the detection of the functionally suitable Web services for a given goal instance, which we have identified as the time critical operation for the efficiency of SWS environments for solving concrete client requests. The central aim of the SDC technique is to enhance the computational performance of the runtime discovery task in order to overcome this bottleneck. To evaluate this, we have defined a comprehensive test bed for examining the behavior of our optimized runtime discovery techniques in comparison to other Web service discovery engines for goal instances (see Section 6.1.1). The following first explains the design and the analysis methodology of the evaluation test, and then discusses the results in detail.

Test Design and Analysis Methodology. The aim is to quantify the improvements in the computational performance of runtime Web service discovery that can be achieved by the SDC technique, in particular within larger search spaces of available Web services which can be expected in real-world scenarios. For this, we have defined the evaluation to consist of two comparison tests. The first one compares the behavior of the SDC-enabled runtime discovery in increasingly larger search spaces with a naive discovery engine that does not use any optimization techniques. This allows us to determine the thorough improvements that can be achieved with our technique. To also assess the improvements in comparison to less exhaustive optimization techniques, the second test compares our optimized engine with a reduced version that does not fully exploit the knowledge captured in the SDC graph.

We perform the tests with the `SDC Runtime Discoverer` component of our prototype, which implements the optimized runtime discovery algorithms specified in Section 5.4. The comparison engines are implemented as extensions of the SDC prototype, i.e. they use the same matchmaking techniques and technical infrastructure as the SDC-enabled engine. In consequence, the actually discovered Web services are identical for all engines so that we can concentrate on the performance comparison. As the test data, we use the 10 goal instances defined in Table 6.3. These describe concrete client requests for shipping a package from California to different destinations by instantiating one of the goal templates we have defined for the shipment scenario (see SDC graph in Figure 6.1). We further consider the five Web services from the original scenario description as the only available ones that offer package shipment. Hence, a subset of these is usable for each of the goal instances. Table 6.7 provides a concise overview of the relevant information for the following discussions, referring to the detailed explanations on the goal instance definitions above in Section 6.1.2.

Table 6.7: Test Data for Runtime Web Service Discovery

Goal Instance	corresp. Goal Template	Input Binding			usable Web Services	
		sender	receiver	weight	name	total
gi1	gtUS2AF	San Francisco	Tunis	1 kg	Muller Racer Walker	3
gi2	gtRoot	Los Angeles	Luxembourg City	1.5 kg	Muller Racer Runner Walker	4
gi3	gtUS2world	Berkeley	Tunis	50.5 kg	Racer	1
gi4	gtUS2EU	Paolo Alto	Bristol	4.3 kg	Racer Runner	2
gi5	gtUS2NA	Los Angeles	New York City	5.5 kg	Muller Racer Walker Weasel	4
gi6	gtUS2world	Monterey	Berlin	60 kg	Runner Runner	2
gi7	gtUS2world	Santa Barbara	Sydney	17.3 kg	Racer Runner Walker	3
gi8	gtUS2SA	San Francisco	Quito	7.58 kg	Racer Runner Walker	3
gi9	gtUS2AS	Stanford	Beijing	57.8 kg	Racer Runner	2
gi10	gtUS2EUlight	San Francisco	Amsterdam	9.99 kg	Muller Racer Runner Walker	4

In order to properly assess the achievable performance improvements, we investigate the durable behavior that is observable among several invocations of the discovery engines. For this, we have defined the comparison tests to consists of several repetitive test runs. We then prepare the test results in terms of statistical standard notions as explained below. On this basis, we analyze the evaluation results with respect to the following criteria: *efficiency* as the time required for completing a single discovery task, *scalability* as the ability to deal with a large search space of available Web services, and *stability* as a low variance of the execution time of several invocations. As discussed above, we adopt these from the standard performance measurement notions in software engineering [Ebert et al., 2004].

Table 6.8 provides an overview of the used statistical notions, following the terminology and definitions from [Crow et al., 2007]. We use them for analyzing the performance of our discovery engines as follows. The arithmetic mean μ denotes the average time for completing a single discovery task, whereby the population n are the test runs and a value x_i refers to the actual time required by a specific discovery engine. We also consider the median $\bar{x}$ as the middle value of the test runs with respect to the required time for completing a discovery task. This is less influenced by extreme values than the arithmetic mean and thus allows us to better judge the long-term average behavior of a discovery engine with respect to efficiency and scalability. We further consider the minimum and the maximum times which

Table 6.8: Overview of Used Statistical Notions

Notion	**Description**	**Formula** n = entire population x_i = value of a data item i
Arithmetic Mean μ	the average value of a data set	$\mu = \frac{1}{n} \times \sum_{i=1}^{n} x_i$
Median $\bar{x}$	the middle value of a data set such that 50 % of the values are smaller and 50 % are bigger; in contrast to μ, this is less influenced by extreme values	given a total order s.t. $x_i < x_{i+1}$: $\bar{x} = \begin{cases} x_i, i = \frac{n}{2} & \text{if } n \text{ is odd} \\ \frac{x_{i-1}+x_i}{2}, i = \frac{n}{2}+1 \end{cases}$
Minimum x_{min}	minimal value in a data set	$x_{min} \leq x_i$ for all $i \in 1, \ldots, n$
Maximum x_{max}	maximal value in a data set	$x_{max} \geq x_i$ for all $i \in 1, \ldots, n$
Standard Deviation σ	a measurement for the dispersion of the values in a data set	$\sigma = \sqrt{\frac{1}{n} \times \sum_{i=1}^{n} (x_i - \mu)^2}$
Coefficient of Variation CV	the percentaged value dispersion of a data set (allows the comparison of data sets with very different value ranges)	$CV = \frac{\sigma \ (Standard \ \ Deviation)}{\mu \ (Arithmetic \ \ Mean)}$

indicate the best, respectively the worst case for completing a discovery task. The final two notions are used to denote the stability of a discovery engine. The standard deviation σ denotes how much the values disperse from the arithmetic mean. For our purposes, this indicates to what extent the times for completing a discovery task vary from the actual average time: the higher this value is, the more unstable is the discovery engine. We further use the coefficient of variation CV which denotes the percentaged value dispersion. This allows us to compare the stability of the discovery engines in larger search spaces where longer processing times can be expected in particular from the naive discovery engine.

The following presents the comparison tests in detail. We consider both flavors of runtime discovery that we have identified to be desirable with SWS environments for automated goal solving: the discovery of a *single* Web service which can be applied when the discovery is performed in an interleaved manner with the subsequent processing steps, and the discovery of *all* functionally suitable Web services that can be applied when the further usability inspection is performed in a stepwise manner. Because of the technical design explained above, all engines used in the tests always provide the correct discovery results in accordance to Table 6.7. Additional information on the technical realization as well as further details on the test results are provided in Appendix C.3.

SDC vs. Naive Engine. We commence with the comparison test of the SDC-enabled runtime discovery with the naive discovery engine. The aim is to evaluate the overall improvements for runtime Web service discovery that is achievable with the SDC technique in comparison to not optimized discovery engines. We are in particular interested in the behavior of the compared engines within larger search spaces of available Web services.

We first discuss the test results for the discovery of a single Web service. For this, we inspect the behavior of the discovery engines within increasingly larger search spaces of 10 up to 2000 available Web services; these always contain the five Web services from the original scenario description as the only potential candidates while all others are not usable. The SDC discovery engine discovers one suitable Web service for a given goal instance by the *discoverSingleWS* algorithm specified in Section 5.4.1. We use the SDC graph from Figure 6.1 for which we have shown the correct automated creation in the above evaluation test. The naive engine inspects the available Web services in a randomized order and stops as soon as a usable Web service has been found. Table 6.9 shows the aggregated comparison test results of all test runs for all goal instances. From this, we can already make statistically firm statements on the computational performance of the discovery engines.

Table 6.9: Test Results SDC vs. Naive (Single Web Service Discovery)

SDC Runtime Discoverer

no. WS	arith. mean (in sec)	median (in sec)	min (in sec)	max (in sec)	deviation (in sec)	coefficient of variation
10	0.28	0.27	0.02	1.25	0.03	11.29%
20	0.28	0.28	0.02	1.31	0.04	12.82%
50	0.28	0.28	0.02	1.27	0.03	11.79%
100	0.29	0.28	0.02	1.25	0.03	11.53%
200	0.29	0.28	0.02	1.25	0.03	11.51%
500	0.29	0.29	0.02	1.19	0.04	13.42%
1000	0.30	0.29	0.02	1.50	0.04	14.79%
1500	0.30	0.29	0.02	1.86	0.05	17.34%
2000	0.31	0.29	0.02	1.17	0.06	18.03%

Naive Engine

no. WS	arith. mean (in sec)	median (in sec)	min (in sec)	max (in sec)	deviation (in sec)	coefficient of variation
10	0.41	0.39	0.08	1.39	0.21	51.71%
20	0.82	0.73	0.08	5.63	0.52	63.45%
50	2.00	1.80	0.09	6.05	1.29	64.59%
100	3.96	3.68	0.09	13.33	2.55	64.48%
200	7.61	6.67	0.08	26.53	5.05	66.35%
500	18.33	15.61	0.09	75.66	13.26	72.34%
1000	37.70	33.22	0.34	163.44	26.28	69.70%
1500	53.91	45.20	0.13	182.61	39.27	72.84%
2000	72.96	65.56	0.13	248.98	52.13	71.45%

The average time of the SDC discoverer for detecting one suitable Web service is 300 milliseconds, independent of the number of the search space size. In contrast, the average time required by the naive engine steadily grows with the number of the available Web services. This means that the SDC-enabled engine guarantees *scalability* because the required times remain the same for every size of the search space. We also can consider the SDC-enabled Web service discovery to be sufficiently *efficient* because an average processing time of less than 1 second for completing the discovery task appears to be expedient to ensure the effectiveness of SWS environments for automated goal solving. In addition to the fast processing times, the SCD-enabled discoverer exposes a significantly higher *stability* than the naive engine. The standard deviation of the SDC engine stays in the range of 50 milliseconds with an average variation of 13.6 %. This is a negligible variance in real world applications. Thus, we can well predict the behavior of the SDC engine, which becomes in particular important within advanced SWS techniques that employ automated Web service discovery as a heavily used component (see Section 2.2.2). This is not given for the naive engine: it exposes an average dispersion of 66.3 % where the actually required times ranges from around 100 milliseconds to over 4 minutes.

Let us now discuss where the performance improvements result from. For this, we examine the comparison tests for individual goal instances form our test set in more detail. As the first one, let us consider goal instance `gi3` which describes a client request for shipping a package of 50.5 kg from Berkeley in California to Tunis in Algeria (see Table 6.7). For this, only Web service `Racer` is usable because it supports package shipment to Africa for weights up to 70 kg; all other Web services either do not support this destination or are restricted to maximal weights of 50 kg (see Table 6.2 in Section 6.1.2). The goal instance defines `gtUS2world` as the corresponding goal template. However, the most appropriate goal template among the existing ones is `gtUS2AF` for shipping packages from the USA to Africa. Thus, the SDC discovery algorithm first refines the goal instance so that `gtUS2AF` is considered as the corresponding goal template. Then, its usable Web services are inspected as the potential candidates for `gi3`. These are `Muller`, `Racer`, and `Walker`; each of them is usable for `gtUS2AF` under the *intersect* usability degree (see Figure 6.1 Section 6.1.2). Under this degree, matchmaking is required to determine their actual suitability for the goal instance. The candidates are inspected in a unordered manner, so that the SDC discoverer requires at minimum 3 matchmaking operations: one for the initial goal instantiation check, one for the refinement step, and one for determining the usability of `Racer` when this is inspected as the first candidate. In the worst case it requires 10 matchmaking operations, which occurs when goal template `gtUS2AF` is inspected as the last child of `gtUS2world` during the refinement step and also `Racer` is inspected as the last candidate.

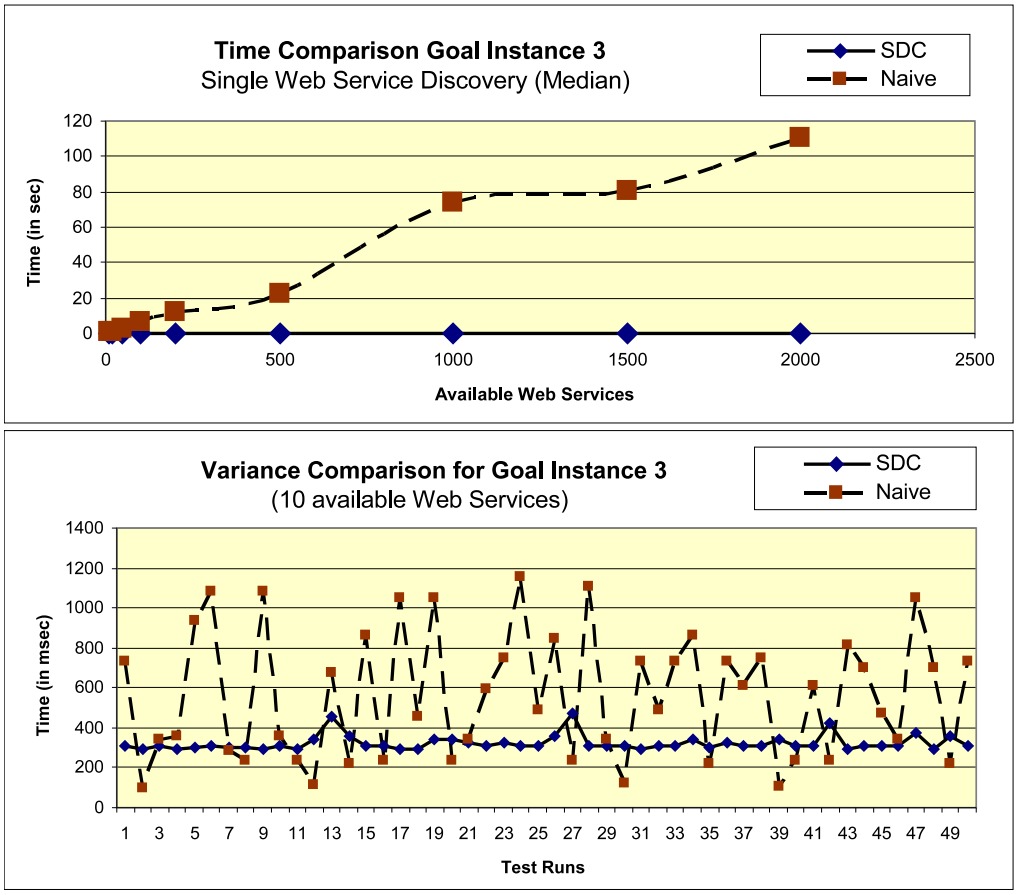


Figure 6.5: Performance Comparison Charts Goal Instance `gi3`

The naive engine inspects the available Web services in a randomized order, and checks the usability of each by the basic matchmaking condition for the given goal instance (see Definition 4.8 in Section 4.3.2). Because only one of the available Web services is usable, the success chance is 10 % for a search space of size 10 and decreases down to 0.05 % for 2000 Web services. Figure 6.5 shows the results of the comparison test for goal instance `gi3` in terms of performance charts. The results comply with the general observations discussed above. The average time for a single matchmaking operation in this test 100 milliseconds. The actually measured times for the SDC engine are $\bar{x} = 327$ msec, $x_{min} = 297$ msec, and $x_{max} = 812$ msec; for the naive engine we find $x_{min} = 94$ msec and $x_{max} = 245$ sec (i.e. 4 min and 5 sec). In order to illustrate the differences in the stability, the lower chart in the figure shows the variance among the 50 test runs for a search space of 10 available Web services: we observe that the SDC engine disperses only marginally from the actual average time while the naive engine varies heavily between less than 0.1 and over 1 second.

As another example, let us examine the comparison test for goal instance `gi10`. This instantiates the goal template `gtUS2EUlight` for shipping a package of 9.99 kg from San Francisco to Amsterdam (Netherlands). Here, the Web services `Muller`, `Racer`, `Runner` and `Walker` are usable because all support package shipment from the USA to Europe. All of them are usable for `gtUS2EUlight` under the *plugin* degree because the goal template is restricted to light packages (see Figure 6.1). Hence, the SDC discoverer detects one of the four usable Web services by the *lookup*-method which – apart from the initial goal instantiation check – does not require any matchmaking (see Section 5.4.1). The detection of the actual Web services by searching the SDC graph can then be effectively performed on the basis of the WSMO4J API. The naive engine needs to inspect every available Web service as explained above. The mere difference to the above case is that the success chance is four times higher because there are four usable Web services among the available ones instead of only one. Figure 6.6 shows the performance charts of the comparison test.

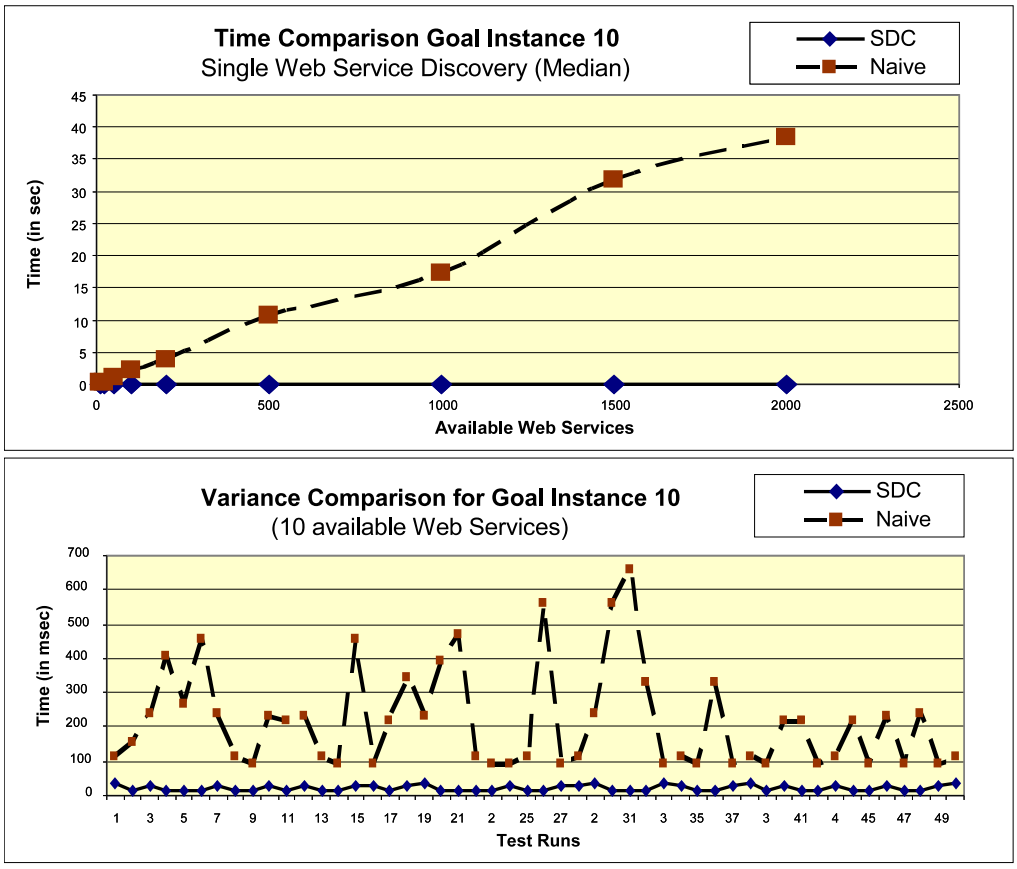


Figure 6.6: Performance Comparison Charts Goal Instance `gi10`

We see that also here the SDC engine exposes an efficient and scalable behavior. The measured median time of the SDC technique is 30 milliseconds, which remain the same for larger search spaces. The processing times of the naive engine rise steadily with the size of the search space. Due to the higher change for success, the actually measured times are about 4 times faster than the ones for `gi3`. This emphasizes our above analysis of the achievable performance improvements, and we can make similar observations for the other goal instances in our test bed.

We now turn towards the second flavor of runtime discovery, i.e. the detection of all functionally suitable Web services for a given goal instance. The SDC engine performs this discovery task by the *discoverAllWS* algorithm specified in Section 5.4.1. At first, the goal instance is refined towards the most appropriate goal template among the ones existing the SDC graph so that only the minimal set of Web services needs to be inspected as potential candidates. Then, their actual usability for the given goal instance is determined via the *lookup*-method as well as by additional matchmaking. The naive engine needs to inspect all available Web services because it can not be ensured that all functionally usable Web services have been detected until every single one has been examined.

This means that the SDC engine merely inspects the minimal set of potentially relevant candidates for the given goal instance. In contrast, the naive engine must always inspect the whole search space so that its processing times will be proportional to the number of available Web services – independent of the actually given goal instance. Because of this, it appears to be sufficient to discuss the aggregated results of the comparison test for all 10 goal instances of our test bed as shown in Table 6.10. As a sufficiently exhaustive test design to make statistically firm statements on the behavior of the discovery engines, we performed 25 repetitive runs for search spaces of 10 up to 500 available Web services of which only the five Web services from the original scenario description offer shipment services.

From this we can make the following observations about the computational performance. Considering the *efficiency*, the SDC engine is always faster than the naive engine, and the average processing time of 0.5 seconds appears to be sufficiently fast. Also for this discovery task, the SDC engine warrants *scalability* because the required times remain the same also in larger search spaces. In contrast, the processing times of the naive engine rise proportionally with the number of available Web services. Regarding the *stability*, the average variance of the SDC engine is 10.3 %, which is less than for the single Web service discovery. The actual deviation times in the range of 60 milliseconds appear to be negligible for real-world applications. The naive engine exposes an even smaller variance, which however becomes irrelevant under consideration of the significantly longer processing times.

Table 6.10: Test Results SDC vs. Naive (All Web Services Discovery)

SDC Runtime Discoverer

no. WS	arith. mean (in sec)	median (in sec)	min (in sec)	max (in sec)	deviation (in sec)	coefficient of variation
10	0.61	0.52	0.17	1.38	0.05	7.85%
20	0.54	0.52	0.17	1.55	0.06	11.21%
50	0.54	0.52	0.17	1.25	0.05	8.69%
100	0.54	0.53	0.19	1.53	0.05	9.67%
200	0.56	0.53	0.19	1.41	0.07	13.22%
500	0.55	0.53	0.19	1.20	0.06	11.02%

Naive Engine

no. WS	arith. mean (in sec)	median (in sec)	min (in sec)	max (in sec)	deviation (in sec)	coefficient of variation
10	1.26	1.23	1.11	2.36	0.09	7.20%
20	2.52	2.46	2.28	5.53	0.15	6.07%
50	6.25	6.25	5.94	7.27	0.15	2.40%
100	12.61	12.54	12.11	18.59	0.31	2.46%
200	25.22	24.97	24.25	60.46	1.11	4.39%
500	62.72	62.56	61.33	72.35	0.85	1.36%

We can summarize the central findings of the comparison test between the SDC-enabled runtime discoverer and the naive engine as follows. The first observation is that an optimization of automated Web service discovery techniques appears to be necessary in order to warrant the effectiveness of SWS techniques in larger search spaces of available Web services which can be expected in real-world scenarios. When examining the behavior of the naive engine as a representative for not optimized discovery techniques, we observe that already for 500 available Web services the actual processing times for discovering a single Web service with a success chance of about 1 % range between less than 0.1 seconds and more than 1 minute (*cf.* Table 6.9). Although a single matchmaking operation only takes about 100 milliseconds in our tests, the discovery of all usable Web services in the same setting takes longer than 1 minute because each of the available Web services must be inspected (*cf.* Table 6.10). Such long processing times and also the unpredictable behavior appear to be unacceptable for an automated discovery engine, in particular for its successful employment as a frequently used component within SWS environments.

The second central outcome is that the SDC technique can be considered as a sophisticated optimization technique for automated Web service discovery because it can achieve significant improvements in the computational performance. With respect to the relevant quality criteria, the test results show that the SDC-enabled engine is sufficiently *efficient* because it performs both flavors of runtime discovery tasks in average times of much less than 1 second. It also warrants the *scalability* because the processing times remain the same

also within larger search spaces, and it exposes a high *stability* among several invocations with marginal variations of the actual processing times. Therewith, the SDC technique overcomes the mentioned deficiencies to a substantial extent, and it further maintains the high retrieval accuracy of our semantic matchmaking techniques. A considerable part of the performance improvements result from our two-phased discovery model wherein only the usable Web services of the corresponding goal template need to be inspected as potential candidates for the given goal instance. Further improvements are gained by the exhaustive exploitation of the SDC graph, in particular by the *refinement*-method which reduces the relevant search space to a minimum by considering the most appropriate goal template as well as by the *lookup*-method for detecting usable Web services without matchmaking.

SDC$_{full}$ vs. SDC$_{light}$. We complete the performance evaluation with a comparison of our optimized runtime discoverer (SDC$_{full}$) and a reduced version that does not thoroughly utilize the knowledge captured in the SDC graph (referred to as SDC$_{light}$ in the following). While we already have shown that the SDC technique can achieve substantial improvements for optimizing automated Web service discovery, the aim of this additional comparison test is to evaluate to what extent the SDC graph as the underlying data structure for capturing design time discovery results enhances the achievable performance increase.

Let us briefly recall the relevant background in order to motivate the test design. A central part of our optimized runtime discovery algorithms is the *refinement*-method which – if possible – replaces the initially defined corresponding goal template G by a more appropriate one G' which is a child node of G in the SDC graph and for which the given goal instance is a valid instantiation (see Section 5.4). This allows us to minimize the relevant search space, because the usable Web services for G' are always a subset of those that are usable for G. Furthermore, the deeper G' is allocated in the goal graph, the higher is the possibility that some Web services are usable under the *exact* or *plugin* or degree: these can be detected by the *lookup*-method as the most efficient discovery facility.

We have implemented the SDC$_{light}$ engine to use the same matchmaking and discovery techniques as the SDC-enabled runtime discoverer but without the *refinement*-method. This means that it performs runtime discovery on the basis of the goal template that actually is defined for the given goal instance. Therewith, SDC$_{light}$ represents a simpler optimization technique for our two-phase Web service discovery which merely utilizes the knowledge on the captured design time discovery results. Thus, the comparison test allows us to precisely determine the actual effect of the SDC graph on the performance increase for runtime discovery: while SDC$_{full}$ exhaustively exploits all relevant knowledge that is provided by the SDC graph, the SDC$_{light}$ engine merely utilizes the knowledge captured in

the *discovery cache* which could technically be realized in form of a simple data base. We could also study an even simpler comparison engine that only considers the usable Web services of the corresponding goal template and then, analog to the naive engine, performs runtime discovery by the basic matchmaking condition for the given goal instance. However, considering the results of the above comparison test, it is obvious that the SDC_{full} engine would clearly dominate such a comparison; we hence do not further investigate this option.

We consider an extended version of the shipment scenario for this comparison test, because the improvement effects of the *refinement*-method expectably unfold within larger SDC graphs. In particular, we extend the original scenario setting as described above in Section 6.1.2 with 10 additional Web services that offer package shipment analog to the five existing ones but merely support specific destination countries. For example, we define `wsShipment5` to offer package shipment from North America to France and Japan. This is usable for our goal templates `gtUS2EU` and `gtUS2AS` as well as for their parents under the *subsume* degree, but it is not usable for any of the goal instances in our test bed. We also define 4 additional goal templates for shipping light packages from the USA to the distinct continents, analog to the already existing `gtUS2EUlight`; for these goal templates, the five original Web services are mostly usable under the *plugin* degree. Table 6.11 provides an overview of the additional goal templates and Web services, and Figure 6.7 below shows the SDC graph for the extended shipment scenario.

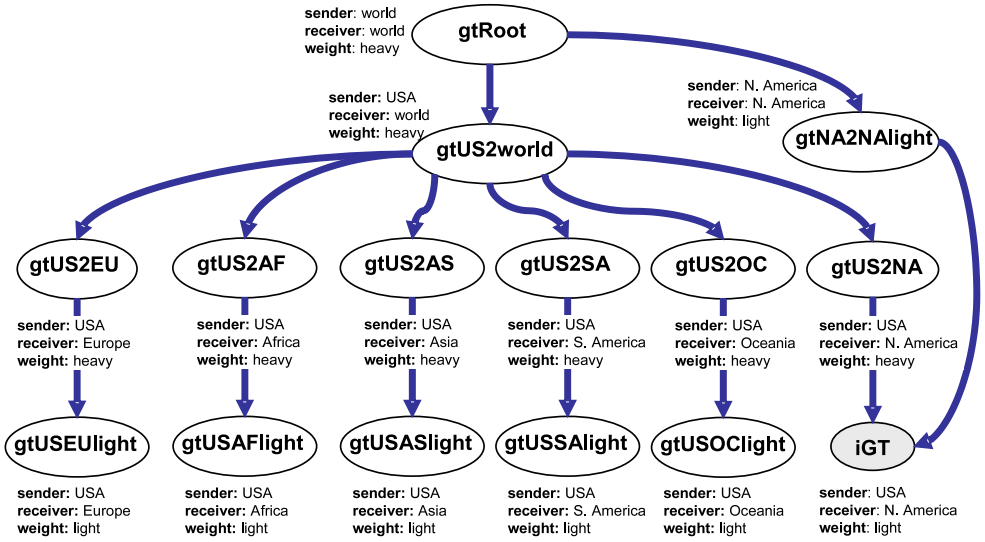
Table 6.11: Additional Goal Templates and Web Services

Additional Goal Templates

Name	Sender	Receiver	Max. Weight
gtUS2AFlight	USA	Africa	light
gtUS2ASlight	USA	Asia	light
gtUS2SA50light	USA	South America	light
gtUS2OC50light	USA	Ocenania	light

Additional Web Services

Name	Sender	Destinations	Max. Weight
wsShipment1	Europe	world	heavy
wsShipment2	Asia	world	heavy
wsShipment3	Europe	Europe	heavy
wsShipment4	Canada	Canada	heavy
wsShipment5	North America	France, Japan	weightClass50
wsShipment6	North America	Argentina, Egypt	weightClass50
wsShipment7	world	South Africa, Israel	light
wsShipment8	USA	New Zealand, Cuba	weightClass70
wsShipment9	USA	South Korea, Palau	weightClass70
wsShipment10	USA	Brazil, Belgium	weightClass50



Usability Degrees	Muller	Racer	Runner	Walker	Weasel	ship 1	ship 2	ship 3	ship 4	ship 5	ship 6	ship 7	ship 8	ship 9	ship 10
level 1															
gtRoot	subsume	subsume	subsume	subsume	subsume	subsume	subsume	subsume	subsume	subsume	subsume	subsume	subsume	subsume	subsume
level 2															
gtUS2world	subsume	subsume	subsume	subsume	subsume	X	X	X	X	subsume	subsume	subsume	subsume	subsume	subsume
gtNA2NAlight	intersect	intersect	X	intersect	intersect	X	X	X	subsume	X	X	X	X	X	X
level 3															
gtUS2AF	intersect	intersect	X	intersect	X	X	X	X	X	X	subsume	subsume	X	X	X
gtUS2AS	intersect	intersect	plugin	intersect	X	X	X	X	X	subsume	X	subsume	X	subsume	X
gtUS2EU	intersect	intersect	plugin	intersect	X	X	X	X	X	subsume	X	X	X	X	subsume
gtUS2NA	intersect	intersect	X	intersect	subsume	X	X	X	X	X	X	X	subsume	X	X
gtUS2OC	X	intersect	plugin	intersect	X	X	X	X	X	X	X	X	subsume	subsume	X
gtUS2SA	X	intersect	plugin	intersect	X	X	X	X	X	X	subsume	X	X	X	subsume
level 4															
gtUS2AFlight	plugin	plugin	X	plugin	X	X	X	X	X	X	subsume	subsume	X	X	X
gtUS2ASlight	plugin	plugin	plugin (omitted)	plugin	X	X	X	X	X	subsume	X	subsume	X	subsume	X
gtUS2OClight	X	plugin	plugin (omitted)	plugin	X	X	X	X	X	X	X	X	subsume	subsume	X
gtUS2SAlight	X	plugin	plugin (omitted)	plugin	X	X	X	X	X	X	subsume	X	X	X	subsume
gtUS2EUlight	plugin	plugin	plugin (omitted)	plugin	X	X	X	X	X	subsume	X	X	X	X	subsume
iGT	plugin	plugin	X	plugin	intersect	X	X	X	X	X	X	X	subsume	X	X

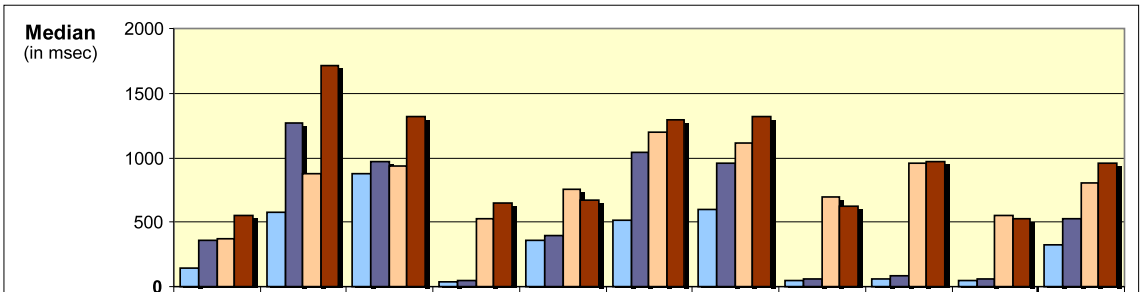
Figure 6.7: Overview of SDC Graph for Extended Shipment Scenario

The upper part of the figure shows the goal graph after inserting the 4 additional goal templates, which become child nodes of the already existing goal templates at the lowest level. Below, we show the usability of all 15 Web services as the actual design discovery results captured in the discovery cache (see Appendix C.3 for details). We see that all the additional Web services are either not usable for the existing goal templates, or merely under the *subsume* degree because they are restricted to specific countries.

On this basis, we perform both flavors of runtime discovery for each of our 10 goal instances and compare the performance of the SDC_{full} engine with the one of SDC_{light}. Because none of the additional Web services is usable for any of the goal instances, the actual discovery results for both engines are still the same as shown in Table 6.7 above. The mere difference is that for goal instance `gi1` now `gtUS2AFlight` is the most appropriate goal template, and for goal instance `gi8` this is now `gtUS2ASlight`.

Figure 6.8 shows the results of the comparison test in form of a comparison chart along with the measured median times of 100 repetitive test runs. Let us commence the analysis with some general observations. Naturally, both engines warrant the scalability because they only consider the relevant subset of the available Web services. Both engines perform the runtime discovery tasks in average times below 1 second, so that they can be considered to be sufficiently efficient for their application purpose; SDC_{full} is about 38 % faster for the single Web service discovery, and ca. 17 % faster for the discovery of all Web services (see the last column in the figure). The average standard deviation for single Web service discovery is 77 milliseconds (= 24 %) for SDC_{full} and 91 milliseconds (= 20 %) for SDC_{light}; for the discovery of all Web services we measured 128 milliseconds (= 15 %) for SDC_{full} and 203 milliseconds (= 18 %) for SDC_{light}. This means that SDC_{full} exposes a better stability among several invocations. However, the actual time variation for both engines remains in the range of milliseconds which appears to be negligible for real-world scenarios.

In the following we discuss the test results for the individual goal instances in detail. These represent different cases with respect to the actual effects of the *refinement*-method on the achievable performance increase. We commence with the single Web service discovery.



	gi1	gi2	gi3	gi4	gi5	gi6	gi7	gi8	gi9	gi10	Average
full (single)	141	578	875	31	359	516	602	47	62	47	326
light (single)	360	1265	969	47	391	1047	954	63	78	62	524
full (all)	375	875	937	532	750	1196	1109	699	953	555	798
light (all)	546.5	1718	1313	641	672	1290	1313	625	969	524	961

Figure 6.8: Test Results SDC_{full} vs. SDC_{light}

The rest results are shown in the first two data rows in Figure 6.8. There are four goal instances for which both engines require only a minimal time, namely `gi4`, `gi8`, `gi9`, and `gi10`. For each, Web service `Runner` is usable under the *plugin* degree for the initially defined corresponding goal template, so that it is immediately discovered by the *lookup*-method. SDC_{full} shows significant improvements of over 50 % in comparison to SDC_{light} for the goal instances `gi1`, `gi2`, `gi6`, and `gi7`, while it only is a little bit faster for `gi3` and `gi5`. To determine the reason for these differences, we need to examine the processing behavior of the discovery engines in more detail. For this, we investigate the necessary matchmaking operations which denote the most expensive part for performing the discovery task.

The goal instance `gi1` wants to ship a 1.0 kg package from San Francisco to Tunis, and defines `gtUS2AF` as the corresponding goal template. The Web services `Muller`, `Racer`, and `Walker` are usable for `gi1` (see Table 6.7). The most appropriate goal template in the extended SDC graph is `gtUS2AFlight`. Hence, SDC_{full} refines the goal instance accordingly, which requires 1 matchmaking operation. Then, it will detect one of the actually usable Web services by the *lookup*-method because all of them are usable for `gtUS2AFlight` under the *plugin* degree. Also `wsShipment6` and `wsShipment7` are potential candidates, but do not need to be inspected because the discovery task is already completed successfully. The SDC_{light} engine considers `gtUS2AF` as the basis for performing runtime discovery. The same 5 Web services are potential candidates, but their usability degree is either *subsume* or *intersect* so that matchmaking is necessary. Here, the success chance is 3/5 so that in average 3 matchmaking operations are necessary to complete the discovery task.

This explains the differences in the measured average times, and we can make similar observations for the other goal instances. For `gi2`, SDC_{full} refines the corresponding goal template from `gtRoot` down to `gtUS2EUlight`, which takes 5 matchmaking operations in average because the *refinement*-method inspects the child nodes in an unordered manner; then Web service `Runner` will be detected via the *lookup*-method. In contrast, SDC_{light} needs to inspect all usable Web services for `gtRoot` by matchmaking with a success chance of 4/15. For `gi6`, SDC_{full} takes 3 matchmaking operations for the refinement from `gtUS2world` to `gtUS2EU`, and then detects `Runner` via the *lookup*-method. Here, SDC_{light} needs perform matchmaking for Web services that are usable for `gtUS2world` with a success chance of 3/11. The same holds analogously for `gi7`. For `gi3`, the engines expectably require the same number of matchmaking operations and thus expose similar processing times. For `gi5`, SDC_{full} takes 1 matchmaking operation while SDC_{light} requires 2 in average; the measured average times are still nearly identical because the refinement of the corresponding goal template from `gtUS2NA` to `iGT` is a more expensive operation because we need to inspect both parents of the intersection goal template.

For the discovery of all usable Web services, the differences in the measured times are smaller because both engines always need to inspect all potential candidates for this discovery task (see the lower two rows in Figure 6.8). Also for this discovery task, the better performance of SDC_{full} results from the reduction of the necessary matchmaking operations that is achieved by the *refinement*-method. Considering goal instance `gi1`, SDC_{full} needs 3 matchmaking operations in average: 1 for the refinement to `gtUS2AFlight` so that then the three actually usable Web services can be detected by the *lookup*-method, and 2 to ensure that `wsShipment6` and `wsShipment7` are not usable; the SDC_{light} engine needs to perform matchmaking for all 5 Web services that are usable for `gtUS2AF` as the initially defined goal template because the occurring usability degrees are *subsume* and *intersect*. For goal instance `gi2`, SDC_{full} requires 7 matchmaking operations in average (5 for the refinement and 2 for the additional Web services), while SDC_{light} needs 15 in order to check all Web services that are usable for `gtRoot` under the *subsume* degree. For goal instance `gi3`, SDC_{full} needs 8 while SDC_{light} requires 11 matchmaking operations, and we can explain the differences for the other goal instances analogously. In the cases where the measured average times are nearly identical, both engines expectably require the same number of matchmaking operations (see goal instances `gi8`, `gi9`, `gi10`). A special case is goal instance `gi5` where SDC_{light} is a little bit faster. Here, SDC_{light} needs 5 matchmaking operations to check all the Web services which are usable for `gtUS2NA` as the initially corresponding goal template; SDC_{full} only needs 3 (1 for the refinement to `iGT`, and 2 for checking the usability of the Web services `Weasel` and `wsShipment8`), but all these require the more expensive matchmaking on the basis of an intersection goal template.

To conclude, this comparison test shows that the SDC technique achieves better performance enhancements for Web service discovery than closely related optimization techniques. The reason for this is the maximal reduction of the necessary matchmaking operations by considering the most appropriate goal template for performing a discovery task. This becomes in particular evident when the matchmaking requires more complex reasoning tasks. While in our comparison test a single matchmaking takes about 100 milliseconds, the usability check of a Web service for a specific goal instance in the best-restaurant-search example that we have discussed in Chapter 4 takes about 1.5 seconds. If we then save 3 matchmaking operations by the *refinement*-method, we would save about 5 seconds for the discovery task. We further observe that the actual improvement gain is dependent on the structure of the SDC graph. When comparing the test results for the goal instances `gi2`, `gi6`, and `gi7` with those of `gi3`, `gi8`, and `gi9`, we see that the performance enhancement by the *refinement*-method is higher if there are fewer direct children of the initially defined corresponding goal template than Web services which are usable under the *subsume* or *intersect*.

6.2 Practical Relevance

After having shown our Web service discovery techniques expose a high retrieval accuracy and that significant performance improvements can be achieved with the SDC technique, the second part of the evaluation is concerned with the practical relevance of the developed technology. For this, the following discusses if and to what extent this can improve the quality of SOA technologies in real-world applications.

Our technology is designed to provide an efficient and scalable Web service discovery component for Semantic Web service environments. It is based on a goal-driven approach, and we have developed semantically enabled discovery techniques with a high retrieval accuracy along with a caching-based mechanism for enhancing the computational performance. Naturally, there are application scenarios for which this appears to be desirable and wherein substantial quality improvements can be achieved, but there might also be scenarios wherein such technologies are dispensable or where only marginal improvements are achievable. In consequence, there are two central questions for evaluating the practical relevance of our Web service discovery technology:

1. when is the application of technology reasonable with respect to the goal-based approach and the need for efficient and highly accurate Web service discovery?

2. what is the expectable increase in the quality and performance of SOA technologies when our technology is employed?

In order to properly answer these questions, the following examines the applicability of our technology in real-world scenarios on the basis of a qualitative estimation model that allows us to assess both the gainable improvements in the overall operational quality of the SOA system as well as the performance increase that can be achieved by the SDC technique. We first define the estimation model, and then inspect the applicability of our technology within one of the largest actually deployed SOA systems that maintained by the US-based telecommunication provider *Verizon*, and also for prominent use case scenarios that have been investigated in the area of Semantic Web services.

6.2.1 Methodology

As the basis for the applicability study, the following specifies a model for judging the general suitability of our technology as well as the qualitative improvements that can expectably be gained for a specific application scenario. The aim is to provide a general estimation model that allows users to decide on the applicability with minimal efforts.

For this, we define the model to consist of two consecutive parts. The first one is a criteria catalogue for determining the general suitability of our technology. This is defined in form of a checklist which can be evaluated by inspecting the scenario structure and the purpose of the target SOA system. The second part is a mathematical model for estimating the performance increase that can be gained by the SDC technique, which allows users to decide on whether our complete technology framework shall be employed or if a partial solution might be sufficient. This requires a more detailed analysis of the application scenario, which naturally is only necessary when the general applicability has been evaluated positively. The following defines the estimation model in detail.

Criteria for General Applicability

We commence with the criteria for determining the overall suitability of our technology for a given application scenario. Following the design of our framework, we define decision criteria for (1) realizing the goal-driven approach for Semantic Web services, (2) the need for efficient and highly accurate automated Web service discovery, and (3) the employment of the SDC technique. We define the criteria so that users can evaluate them on the basis of a high level analysis of the given application scenario. If most aspects appear to be relevant, then the employment of our technology can be considered to be reasonable; if not, then the discrepancies can be used to identify other technical solutions.

(1) Goal-based Approach. Our technology supports a goal-based approach for Semantic Web services: clients specify the objective to be achieved in terms of a goal, and the system solves this by automatically detecting and executing the necessary Web services.

A goal formally describes an objective that a client wants to solve by using Web services, abstracting from technical details. Goals can be used in a SOA application to describe and process both end-user requests as well as specific sub-tasks for which Web services shall be employed. This enables (1) the lifting of the formulation of Web service usage request to the level of the problems that shall be solved, and (2) enhancements in the flexibility of the system because the actual Web services are determined dynamically at runtime (see Section 2.2.1). To facilitate this, we have developed a conceptual model for describing and using goals in SOA system. This distinguishes *goal templates* as generic and reusable objective descriptions that are kept in the system and *goal instances* that describe concrete client objectives by instantiating a goal template, and supports the automated invocation of Web services via *client interfaces* that facilitate the automated invocation and consumption of a Web service in order to actually solve a given goal (see Chapter 3).

This provides a conceptual model for SOA applications with sophisticated support for the client side, in particular when SWS techniques shall be employed because our goal model contains the necessary semantic descriptions. However, the realization of a real-world application requires considerable effort in addition to the provision of the Web services. We hence define the following decision criteria for the goal-based approach:

- support for problem-oriented Web service usage is desirable,
- the flexibility of the system with respect to changes on the existing Web services and their technical availability is important,
- support for request formulation by the instantiation of goal templates is sufficient;
- otherwise, a different model for handling client requests is required.

(2) Web Service Discovery. The second building block of our technology is the two-phase Web service discovery specified in Chapter 4. This separates *design time* discovery as the detection of suitable Web services for goal templates, and *runtime* discovery as the detection of the actually suitable Web services for a concrete goal instance. For this, we define sufficiently rich functional descriptions of goals and Web services, and we provide semantic matchmaking techniques that warrant a high retrieval accuracy for both phases. The following criteria allow users to judge the necessity of such a discovery technology:

- automated Web service discovery is needed, either as a stand-alone facility or as part of a comprehensive SWS system for automated goal solving;
- the target system requires high precision and recall for the discovery task in order to ensure the desired operational quality.

(3) SDC Technique. The third component of our technology is the SDC technique, the caching-based mechanism for enhancing the computational performance of Web service discovery specified in Chapter 5. This creates a *SDC graph* which organizes goal templates in a subsumption hierarchy and captures the results of design time discovery runs, and then effectively uses this knowledge for optimizing the runtime discovery task as the time critical operation by minimizing the necessary matchmaking efforts.

The need for optimized discovery techniques arises when the target system is expected to deal with larger amounts of goals and Web services. The computational performance becomes in particular relevant when advanced SWS techniques shall be used wherein an

automated discovery engine is employed as a frequently used component. We have shown that the SDC technique can achieve satisfactory improvements for this. However, the actual performance gain is dependent on the structure of the specific application scenario. In particular, the structure of the resulting SDC graph affects the achievable performance increase: in the best case, the necessary goal templates and the available Web services can be organized such that runtime discovery can be performed with minimal matchmaking effort; in the worst case this is not given, so that no performance improvements can be achieved. Besides, the SDC technique is especially designed for applications wherein larger numbers of concrete client requests need to be processed while changes on the available resources are rare, so that an efficient and stable runtime discovery facility can be provided by creating the SDC graph is created once with seldom updates. The following criteria allow users to decide on the employment of the SDC technique, for which we provide the estimation model for the achievable performance increase below:

- optimized Web service discovery techniques appear to be desirable because
  - a larger number of goals and Web services can be expected in the application
  - the target system shall employ SWS techniques wherein automated Web service discovery is a central and often performed operation
- large numbers of similar concrete client requests are expected
- a sufficient increase in the computational performance can be expected (see estimation model below).

Estimation of Achievable Performance Increase

In order to provide a more sophisticated decision criterion for the application of the SDC technique in a real world scenario, the following defines a mathematical model for estimating the expectable performance increase for the runtime discovery task.

As discussed in Section 5.4.3, the achievable performance increase is dependent on the topology of the SDC graph as the knowledge structure that is explored by our optimized discovery techniques. This is created on the basis of the existing goal templates and the usability of the available Web services, so that in fact the achievable performance increase is dependent on the specific application scenario. With respect to this, a method for determining if and to what extent the structure and relationship of the resources in a given application scenario facilitate the creation of a SDC graph that ensures significant performance improvements appears to be desirable for supporting the applicability decision.

For this, we provide a model for identifying the performance improvement that can be achieved with the SDC technique for any application scenario. The aim is to provide an indicator that is sufficiently meaningful and can be calculated with minimal efforts. Hence, we define a mathematical formula for indicating the expectable performance increase rate *PIR* as percentaged gain in comparison to a non-optimized discovery engine. The following defines this in detail for a given SDC graph, and below we provide a simplified version for estimating the expectable performance increase on the basis of approximated values.

Definition. The purpose of the performance increase rate *PIR* is to provide a simple but meaningful indicator for the expectable effectiveness of the SDC technique for optimizing Web service discovery in a given application scenario. We define this as the percentaged performance increase that can be gained by the SDC-enabled runtime discoverer in comparison to a naive engine that does not apply any optimization techniques.

The first element of the model is the *similarity ratio* $\mathcal{R}_{sim}$ that denotes to which extent the resources in the application scenario are semantically similar. This provides a general indicator for the achievable performance increase: the higher the similarity ratio, the more goal templates and Web services are organized in the SDC graph subsumption hierarchy which then can be effectively explored by the runtime discovery algorithms. We define $\mathcal{R}_{sim}$ as the number of semantically similar goal templates in relation to the existing ones, i.e. the quotient of goal templates that are organized in a connected subgraph of the SDC graph. We do not have to consider the Web services here because they are allocated as leaf nodes in the SDC graph. In order to use this as a weighting factor for overall performance increase rate, we define the formula such that $\mathcal{R}_{sim} = 1$ for the best case where all goal templates and their usable Web services can be organized in a completely connected SDC graph, and $\mathcal{R}_{sim} = 0$ for the worst case where all goal templates are disjoint.

$$\mathcal{R}_{sim} = \sum_{i=1}^{n} \frac{|G_{sim}(i)| - 1}{|G_{total}| - 1} \times \frac{1}{i} \quad \text{where i = subgraph} \tag{6.1}$$

The other elements are *cost models* for estimating the processing times for runtime Web service discovery that can be expected from the SDC-enabled discoverer and from the naive engine. Under consideration of the findings of the performance analysis above in Section 6.1, we define this with respect to the number of necessary matchmaking operations as the computationally most expensive part.

Formula 6.2 defines the cost model of the SDC-enabled engine for a given goal instance $GI(G, \beta)$ as the costs for the *refinement*-method plus those for checking the usability of the potential candidate Web services. For the former, we need to traverse the SDC graph from

the initially defined goal template G down to the most appropriate goal template G'. The costs for this is the distance of G and G' multiplied with half of the branching factor (the maximal number of direct children in the goal graph), because every goal template on the path $G \rightarrow \ldots \rightarrow G'$ needs to be inspected and for each parent the children are inspected in an unordered manner. If G is already the most appropriate goal template, then this summand becomes null because $distance(G, G) = 0$. The second summand defines the costs for discovering a single Web service as the quotient of the number of relevant candidates (i.e. usable Web services for G') and those that are actually usable for $GI(G, \beta)$. Analogously, Formula 6.3 defines the costs for discovering a single Web service with the naive engine as the quotient of all Web services and those that are usable for the given goal instance. For the discovery of all usable Web services both engines need to inspect all potential candidates, so that the cost models define $|usable(W, G')|$, respectively $|W|$ instead of the quotients.

$$Cost_{SDC} = \left(distance(G, G') \times \frac{b(SDC_{GG})}{2}\right) + \frac{|usable(W, G')|}{|usable(W, GI(G, \beta))|} \tag{6.2}$$

$$Cost_{SDC} = \frac{|W|}{|usable(W, GI(G, \beta))|} \tag{6.3}$$

We now integrate the elements into the overall formula for the performance increase rate *PIR*. As shown in Formula 6.4, we define this as the reciprocal value of the percentaged quotient of the cost models for the SDC-enabled engine and the naive engine multiplied with the similarity ratio as the general weighting factor. This indicates the performance increase that is effectively gainable by the SDC technique.

$$PIR = \mathcal{R}_{sim} \times \left(100 - \left(\frac{Cost_{SDC}}{Cost_{Naive}} \times 100\right)\right) \tag{6.4}$$

For illustration, let us exemplify the calculation and interpretation of the *PIR* for the shipment scenario as defined in Section 6.1.2 above. Here, all 10 goal templates are semantically similar, so that $\mathcal{R}_{sim} = \frac{9}{9} = 1$. Let us consider goal instance `gi6` for shipping a 60 kg package from Monterey (in California) to Berlin (in Germany). It defines `gtUS2world` as the corresponding goal template while the most appropriate one is `gtUS2EU`, and there are two usable Web services for `gi6` (see Table 6.3). We can now estimate the costs for the SDC engine as follows: $distance(G, G') = 1$ because `gtUS2EU` is a direct child of `gtUS2world`, $\frac{b(SDC_{GG})}{2} = 3$ because `gtUS2world` has 6 direct children as the maximal branching factor, and $\frac{|usable(W,G')|}{|usable(W,GI(G,\beta))|} = \frac{4}{2} = 2$ because only two of the Web services usable for `gtUS2EU` are actually usable for the goal instance. So, we obtain $Cost_{SDC} = (1 \times 3) + 2 = 5$. When

we consider a search space of 10 available Web services, then $Cost_{Naive} = \frac{10}{2} = 5$ so that $PIR_{10} = 1 \times (100 - (\frac{5}{5} \times 100)) = 0$ %. This means that no performance increase can be expected for this setting. However, if we consider 100 available Web services, we get $Cost_{Naive} = \frac{100}{2} = 50$ and $PIR_{100} = 1 \times (100 - (\frac{5}{50} \times 100)) = 90$ %, which indicates that a significant performance increase can be achieved with the SDC technique, and also that an optimization technique appears be desirable for this scenario.

Approximation. The above formulae require knowledge about the actual SDC graph for the given application scenario. In order to support the estimation for application scenarios where this is not given, we now define a calculation formula that allows users to work with approximated knowledge on the available resources and the resulting SDC graph.

For this, we simplify the formulae for the similarity ratio as well as the cost models so that they can be evaluated on the basis of an approximation of the expected SDC graph and work with estimated values (denoted by $^\sim$). For $\mathcal{R}_{sim}$, we define the approximation to merely consider the expected connectivity degree of the SDC graph while abstaining from possibly existing subgraphs. We expect this to converge towards 1 for most applications because usually the available Web services and thus the goals that can be solved by them are mostly similar. The approximation formula for $Cost_{SDC}$ merely considers the expected costs for the *refinement*-method; as a generous approximation, we assume that half of the goal graph needs to be traversed for finding the most appropriate goal template. We abandon the costs for additional matchmaking because these details are presumably not known and also only marginally influence the results. For the naive engine, we work with the expected number of available Web services $|W|^\sim$ and the expected average number of Web services that are usable for a goal instance in the application scenario. We then use formula 6.4 to calculate the expectable performance increase rate *PIR*.

$$\mathcal{R}_{sim} \approx \frac{|G_{sim}|^\sim}{|G_{total}|^\sim} \tag{6.5}$$

$$Cost_{SDC} \approx \frac{diam(SDC_{GG})^\sim}{2} \times \frac{b(SDC_{GG})^\sim}{2} \tag{6.6}$$

$$Cost_{Naive} \approx \frac{|W|^\sim}{\mu(\ |usable(W, GI(G, \beta))|^\sim\ )} \tag{6.7}$$

This allows users to evaluate the improvements that can be gained by the SDC technique on the basis of an approximated outline of the application scenario without the need of actually modeling the resources in detail. Similar models can be defined to estimate the expectable improvements in comparison to other optimization techniques.

6.2.2 The Verizon SOA System

We now examine the application of our technology for the SOA system maintained by the US-based telecommunication provider *Verzion*. This contains around 1.500 Web services that are used by over 650 client applications, and therewith represents on the worldwide largest SOA systems that is actually deployed at this point in time. The following first explains the system architecture and the applied SOA technologies, and then discusses the applicability of our technology on the basis of the evaluation model presented above. All information on the system provided here are non-confidential and have been approved for research purposes by Verizon in person of chief scientist Dr. Michael Brodie.

Overview

Verizon is one of the major telecommunication providers in the USA (see `www.verizon.com`). As an early adaptor of the SOA technology, Verizon uses Web services to provide, maintain, and integrate basic functionalities among the IT applications used in local offices and departments that are distributed over the country. The system has been in use since 2004 and is continuously extended with new Web services and functionalities.

Figure 6.9 below shows an overview of the Verizon SOA system. Its heart is the co-called IT Workbench (short: ITW) which provides the central management facilities. This is a web-based tool for registering Web services and client applications along with publication support and a discovery facility. Currently, there are about 1.500 registered Web services that provide basic facilities for product, customer, and sales management. These mostly are simple request-response services, e.g. a `getCustomerInfo` Web service for retrieving the data of a customer in a specific sales record. The Web services are used as the basic building blocks within client applications. For example, the IT system of a local Verizon service center uses the `getCustomerInfo` Web service instead of a proprietary implementation.. Currently, there are about 580 company internal and 70 external client applications registered in the system. Technically, the Web service usage is realized by the conventional technologies as described in Section 2.1.1: the Web services carry a WSDL description, and the client applications invoke them via client-stubs over SOAP messages. This system architecture complies with the idea of SOA as discussed in Section 2.1.2: the Web services provide basic functionalities that are used by several client applications in order to enhance the interoperability and reduce the maintenance costs of the IT systems in Verizon as a larger enterprise that is distributed over several locations. The following information indicates the size and the extensive usage of the Verizon SOA system.

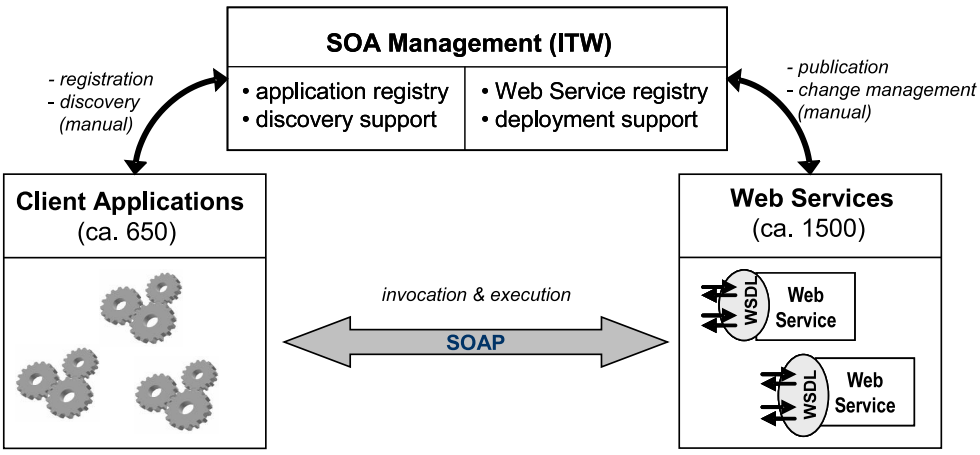


Figure 6.9: Overview of Verizon SOA System

- Transactions (i.e. Web Service Invocations): **15 million per day**
- Growth Rate (new Web Services): **23 per month**
- Number of Web Service Providers: **68**
- Service Developers: **5000+**, Service Administrators: **798**, Dashboard Users: **1600+**

The most interesting aspect for our discussion is the Web service discovery technique supported by the system. For this, the Web services are annotated on the basis of a 15 slot meta-data set that includes information about the owner, the technical access, usage rights, and also about the functionality. The latter is based on the so-called *Verizon Business Taxonomy* which consists of three categories: the *business area* as the relevant management area, the *service line* as the Verizon product line, and the *business object* that identifies the data item. Each category has a set of pre-defined values of which one or more can be used to describe a Web service. Table 6.12 provides an overview of the categorization scheme along with the number of associated Web services. We here work with a data set that has been provided in November 2006 and contains 984 Web services. The total number of 7921 shown in the table indicates that most Web services are annotated with several of the pre-defined values, which means that they can process e.g. several business objects.

These annotations provide relatively precise information about the offered functionality. For example, the annotation of the `getCustomerInfo` Web service mentioned above with the area *Pre-Order & Sales Order Management* for the service line *Long Distance Services* and the business object *Work Order* means that it can provide the customer data of a specific

Table 6.12: Web Service Descriptions in the Verizon SOA System

Business Area	
Name	No. WS
6	7921
Billing, Payment, Credit Management	844
General Management	730
Pre-Order & Sales Order Management	1230
Product, Pricing, Inventory Management	679
Provisioning, Field Fulfillment, Workforce Mgmt.	1994
Trouble, Test, Fault Management	2444

Service Line	
Name	No. WS
15	7921
Access Services	806
Broadband / Optical Services	703
Centrex Services	487
Content Distribution Services	1
Directory Services	116
DSL Services	977
Equipment Services	732
Internet Services	366
iobi Services	184
Local Services	1430
Long Distance Services	1288
Special Services	54
Video / Gaming Services	476
VOIP Services	289
Wireless Services	12

Business Object	
Name	No. WS
13	7921
Bill	125
Circuit Detail	51
Credit Status	288
Customer Service Record	886
Equipment	492
Network Service Record	961
Order or Service Request	1102
Order Status	1598
Product	217
Quotation	44
Trouble Ticket	840
Usage	189
Work Order	1128

ordering for a long distance service. The publication interface in the ITW ensures that a newly registered Web services is annotated with at least one value per category, so that the relevant meta-data are existing in the system. On this basis, client application developers can search for suitable Web services. Apart from a direct lookup for already known Web services, the ITW provides a search functionality over the meta-data annotations.

This already provides Web service discovery support with respect to the provided functionalities. Although using a keyword-based search technique, the business taxonomy appears to be sufficiently exhaustive in order to warrant relative precise search results. However, the clients need to inspect the discovered candidates manually in order to determine their actual suitability for the intended usage, and also manually integrate the finally chosen one into the client application. In our data set, the average number of Web services with the same annotations – i.e. with the same values for all three categories – is 21 with a range from 1 to 622. With respect to this, the main challenges reported by the management team of the Verizon SOA system is a more precise discovery technique, and also a better solution for handling changes in the available Web services. We shall examine in the following if and to what extent our technology can help to accomplish this.

Applicability Study

We now discuss the potential quality improvements that could be achieved by applying our technology to the Verizon SOA system. Considering the criteria catalogue specified in above Section 6.2.1, the goal-based approach appears to be suitable in order to enhance the flexibility of the system with respect to the constantly growing and changing resources. Also, a semantically enabled Web service discovery technique appears to be desirable for

enhancing the support for detecting the suitable Web services for a client application. Hence, in general the deployment of our technology appears to be reasonable. The following first illustrates the modeling as well as the adopted system design in accordance to our technology framework, and discusses the suitability of the SDC technique.

Modeling and Usage. The basis for applying our technology are the semantic descriptions of the goals and Web services. In particular, we require the formal functional descriptions as specified in Section 4.2 that precisely describe the provided and requested functionalities. The following exemplifies this for one of the existing Web services and an imaginary usage request from a client application.

We require sufficiently exhaustive domain ontology in order to properly specify the functional descriptions. For this, let us define the `Verizon Business Ontology` that essentially follows the already existing business taxonomy outlined above. This defines the three categories `area`, `line`, and `object` as top-level concepts along with the predefined values as sub-concepts which further describe the respective entity. We assume that every business object is associated with a specific management area and a particular service line. Let this be expressed by the relations $inArea(object, area)$ and the axiom $\forall ?o, ?a_1, ?a_2.\ object(?o) \wedge inArea(?o, ?a_1) \Rightarrow \neg inArea(?o, ?a_2)$, and the relation $forLine(object, line)$ along with $\forall ?o, ?l_1, ?l_2.\ object(?o) \wedge forLine(?o, ?l_1) \Rightarrow \neg forLine(?o, ?l_2)$. We further define a function $operation(?item)$ along with three sub-functions $get(?item)$, $create(?item)$, $delete(?item)$ for describing the basic management operations on data items.

We consider the existing Web service `GetCustomerInfoByBTN` that retrieves customer information for the *Long Distance Services* line, supports three of the business areas, and accepts four different business objects as input. Let us assume a client application that requires this (or a similar) functionality for processing work order items. In our approach, this is described in terms of a goal. More precisely, we define a *goal template* `GetDataItem` for getting the desired data item, and the individual usage requests are described as *goal instances* which instantiate this goal template. Table 6.13 shows the functional descriptions for the goal template and for the Web service. Both use the `Verizon Business Ontology` outlined above as the background ontology, and define four input variables IN: $?x$ denotes the requested, respectively the provided output item; the others denote the input object $?o$ with the associated business area $?a$ and the service line $?l$. The preconditions ϕ^{pre} specify the restrictions on the inputs: the goal template supports all business objects for all areas and all service lines, while the Web service is restricted to four business object types and three business areas. The effects ϕ^{eff} state that the $get(?item)$-operation shall, respectively will be executed with respect to the business object provided as input.

Table 6.13: Functional Descriptions for Verizon SOA System

Goal Template `GetDataItem`		**Web Service** `GetCustomerInfoByBTN`	
Ω:	Verizon Business Ontology	Ω:	Verizon Business Ontology
IN:	{?x,?o,?a,?l}	IN:	{?x,?o,?a,?l}
ϕ^{pre}:	$item(?x) \wedge$	ϕ^{pre}:	$customer(?x) \wedge$
	$object(?o) \wedge$		$object(?o) \wedge (LSR(?o) \vee OSR(?o)$
	$inArea(?o, ?a) \wedge area(?a) \wedge$		$\vee\, OSt(?o) \vee WO(?o)) \wedge$
	$forLine(?o, ?l) \wedge line(?l)$.		$inArea(?o, ?a) \wedge area(?a) \wedge$
ϕ^{eff}:	$get(?x) \wedge forObject(?x, ?o)$.		$(PS(?a) \vee PPI(?a) \vee PFW(?a)) \wedge$
			$forLine(?o, ?l) \wedge LDS(?l)$.
		ϕ^{eff}:	$get(?x) \wedge forObject(?x, ?o)$.

In the client applications, the concrete invocation requests are then formulated in terms of goal instances. For example, let us assume a request for the customer data of the work order item $wo123$. The goal instance specify `GetDataItem` as the corresponding goal template, and define the input binding $\beta = \{?x|customer, ?o = wo123, ?a = PS, ?l = LDS\}$. Our semantic matchmaking techniques will determine that the Web service is usable for this goal instance. However, if the requested output item would be the person in charge for the work order item, then `GetCustomerInfoByBTN` will be determined to not be usable. The reason is that here the Web service only partially covers the functionality requested by `GetDataItem`, which refers to the *subsume* match in our terminology. Under this usability degree, additional matchmaking is necessary in order to determine the functional suitability of a Web service for a specific goal instance (see Section 4.3.2).[1]

The analyzed data set contains 57 Web services with the descriptions $area = PS, line = LDS$, and $object = WO$. This would be the search result of the ITW discovery. Then, the client application developer then needs to inspect the Web services manually, and finally integrate the chosen one into the application. In contrast, our discovery techniques provide a significantly higher retrieval accuracy because they can determine the actually usable Web services for each individual invocation request. The main reason is that our functional

[1] We use the following abbreviations with respect to the readability (*cf.* Table 6.12):

- Business Areas: *Pre-Order & Sales Order Management* (PS), *Product, Pricing, Inventory Management* (PPI), *Provisioning, Field Fulfillment, Workforce Management* (PFW)
- Services Lines: *Long Distance Services* (LDS)
- Business Objects: *Customer Service Record* (CSP), *Order or Service Request* (OSR), *Order Status* (OSt), *Work Order* (WO).

descriptions are more precise than the 3-dimensional keyword-based annotations; the necessary domain ontology for this can be easily defined on the basis of the already existing business taxonomy. Also, changes on the available Web services can be handled smoothly with our 2-phase discovery model because new or modified Web services need to be inspected once at design time and then are properly integrated into the system for the successive runtime discovery operations. Moreover, the goal-based approach allows to dynamically select one of the usable candidates at runtime: e.g. if the `GetCustomerInfoByBTN` Web services is not available due to some technical problem, then a different one from the discovery result for the goal instance can be used. This ensures a much higher flexibility of the system in comparison to the currently hard-wired invocation of the target Web services out of the client applications. This shows that our technology can be considered as an appropriate solution for addressing the main challenges reported for the Verizon SOA system.

Applying the SDC Technique. We now investigate whether the employment of the SDC technique appears to be necessary and suitable in order to improve computational performance of the automated discovery process. The basic criteria for this are given because we can expect very many similar goal instances that need to be solved while only a relatively small number of changes need to be handled (see the reported numbers above). Thus, the main question is whether the existing Web services and the necessary goal templates allow the creation of a SDC graph upon which a substantial performance increase can be achieved.

For this, let us consider the expectable structure of the SDC graph for the Verizon SOA system. The vast majority of the Web services in the analyzed data set is concerned with data management, i.e. the retrieval, creation, deletion, or modification of some data item analog to the `GetCustomerInfoByBTN` Web service discussed above. We thus concentrate on these kind of functionalities. Let us assume that all relevant goal templates for data item management in the Verizon SOA system are modeled analogously to the one shown in Table 6.13. We can generate the additional goal templates by exploring the taxonomic structure of the `Verizon Business Ontology` as explained in Section 5.3.4. This results in the balanced structure of the SDC graph as illustrated in Figure 6.10: the root node is the most generic goal template for all data management operations on any item $?x$ for any business object $?o$ in any area $?a$ and for any service line $?l$. On the next level, we find the goal templates that specify the specific operation that shall be perform on the data item $?x$. The subsequent levels define restrictions on the allowed input values: on level 2, we find goal templates that are restricted to one of the 6 business areas, the ones on level 3 are further restricted to one of the 15 service lines, and finally on level 4 all inputs are restricted to a specific sub-category.

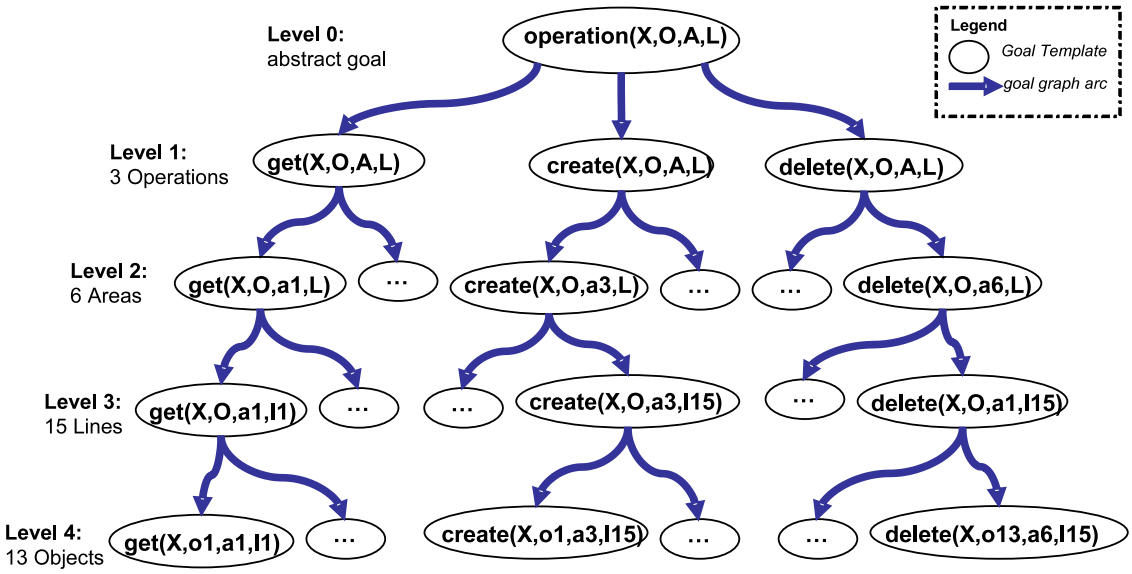


Figure 6.10: Structure of SDC Graph for Verizon SOA System

Regarding the usability of the Web services, the analyzed data set contains about 900 Web services for data management. Expectably, a third of these will be usable for each of the goal templates on level 1 which define the basic data item management operations. The number of usable Web services decreases significantly on the subsequent levels of the SDC graph due to the more specialized restrictions. As the average number of `get()` Web services with the same annotations in the data set, let us assume that there are in average 15 Web services usable for each of the most specialized goal templates on level 4. Out of these, let in average 3 be usable for a specific goal instance which also defines the type of the requested output $?x$. This corresponds to the average number of `getCustomer()` Web services with the same annotations in the given data set.

We now can estimate the achievable performance increase rate *PIR* as defined above in Section 6.2.1. The similarity ration for data management operations is 1 because all goal templates and Web services are organized in a connected SDC graph. Let us assume that each goal instance for data management instantiates one of the goal templates from level 1. The most appropriate goal template that would be used by our optimized discovery algorithms will always be allocated on level 4. We then can estimate the processing costs for runtime discovery as follows: we always need to traverse the SDC graph from level 1 to level 4 so that $distance(G, G') = 3$. The children at each level are inspected in a unordered manner, so that we can estimate 3 operations to get from level 1 to level 2, then 7.5 for the next level, and 6.5 for finding G' on level 4. Then, we have to examine the 15

candidate Web services with a success chance of 3/15, which in average requires another 5 operations to find a usable Web service for the given goal instance. So $Cost_{SDC} = 3 + 3 + 7.5 + 6.5 + 5 = 25$. A naive engine that subsequently investigates all necessary Web services needs in average 300 matchmaking operations (i.e. the 900 available ones divided by 3 as the number of expectably usable ones). Thus, we can expect a performance increase of $1 \times \left(100 - \left(\frac{25}{300} \times 100\right)\right) > 90$ % by applying the SDC technique.

This indicates that an optimized technique appears to be necessary when applying automated Web service discovery within the Verizon SOA system. Also, the expectable performance increase appears to be adequate in order to warrant an efficient and scalable runtime discovery. A problem for the actual deployment could be the relatively high number of matchmaking operations for determining the most appropriate goal template in the relatively large SDC graph. However, this could be overcome by integrating the semantically enabled discovery engine with the already existing search facilities. Instead of detecting the most appropriate goal template by the *refinement*-method as specified in our technology, we could use the ITW search facility for this. This would allow us to reduce the necessary matchmaking operations to 5 for detecting an actually usable Web services. Under the assumption that a single matchmaking operation takes about 100 milliseconds as within the performance tests discussed in Section 6.1, the average time for a runtime discovery task would be 0.5 seconds. This appears to be sufficiently fast for a real-world application with respect to the high retrieval accuracy of our discovery techniques.

6.2.3 Other Application Areas

After having outlined the beneficial employment within an existing real-world SOA application, we now examine further scenarios in order to provide a more comprehensive evaluation of the practical relevance of our technology. For this, we inspect a selection of applications that have been investigated as use case scenarios in research efforts around Semantic Web services. We first examine scenarios wherein our technology can expectably achieve substantial improvements, and then discuss scenarios for which this can not be expected.

Promising Scenarios

We commence with scenarios where significant improvements can be expected from applying our technology. With respect to our criteria catalogue, the common characteristics of such applications are that a high flexibility of the target system is desirable due to the dynamically changing environment, and that automated Web service discovery shall be employed as a heavily used facility for handling larger numbers of goals and Web services.

The Travel Scenario. One of the most prominent use case scenarios for demonstration SWS technologies is the travel scenario which we already discussed as the running example in Chapter 3. This is concerned with searching and booking items related to traveling such as flights, hotels, train tickets, etc, and appears to be a promising application scenario for our technology because several offers and requests can be expected.

Let us examine the scenario setting and the handling within our technology framework in more detail. The market for travel and tourism is one of the largest lines of industry throughout the world with very many service providers and customers, and it is highly automated in both the inter-business (B2B) and the business-to-consumer part (B2C). In general, we can distinguish two types of offers and requests in the travel domain: basic ones that are concerned with a single item such as a flight ticket for a specific journey, and more complex ones that deal with packages of several items, e.g. an all-inclusive holiday tour. This distinction will naturally be reflected in Web services for the travel domain, so we can expect to find *basic* Web services e.g. for searching and booking flight tickets with a specific airlines, and *complex* Web services that offer multi-item travel packages

In our technology framework, the goals and Web services for basic and complex functionalities are handled by different conceptual entities. For the former, the goal templates as well as the Web services have a formal functional description which is sufficient for the purpose of automated discovery. For the latter, the resources are specified as aggregations of basic functionalities. An example for this is the "Virtual Travel Agency" Web service defined in [Stollberg and Lara, 2004] that provides complex travel offers by dynamically aggregating basic services from other providers. In order to warrant its flexibility for specific requests and also with respect to the constantly changing offers, it is realized as an orchestration of goals for basic travel services. Analogously, one can define *composite goals* in order to describe complex objectives as a collection of other goal templates along with a desired workflow (see Section 3.2.3).

In order to solve a goal instance for such a composite goal – respectively to execute a Web service that is dynamically composed of other Web services – we need to perform runtime discovery for each of the subgoals. This means that the efficiency of the discovery technique becomes even more important than in the previous examples because it needs to be performed multiple times. Effectively, the overall time required for the discovery of the actual Web services is the sum of the runtime discovery times for every invocation of each subgoal. Because every composite goal eventually is an aggregation of basic goals, it appears to be sufficient to create a SDC graph that considers the basic goal templates and Web services: if we can sufficiently optimize the runtime discovery task on the level of basic goals, then we will gain the summative performance increase of the composite goal.

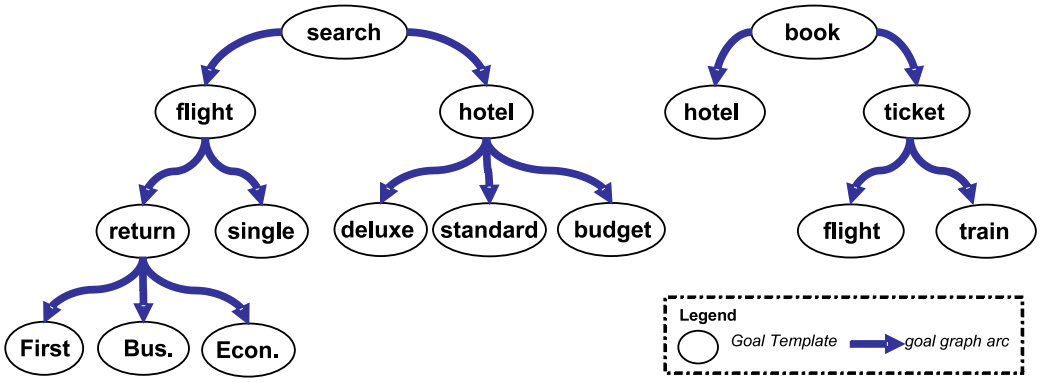


Figure 6.11: SDC Graph for Travel Scenario

For illustration, let us recall the example for a composite goal that we have outlined discussed in Section 3.2.3. This requests to book a flight and a hotel for some trip, and is aggregated of four subgoals: (1) a suitable flight shall be found at first, and then (2) a hotel for the actual duration of the stay shall be found; next, (3) the flight shall be booked, and finally (4) the hotel shall be reserved. Here, we need to perform runtime discovery at least four times in order to find the actual Web services for solving a concrete goal instance.

To estimate the performance increase by applying the SDC technique, let us assume that the necessary basic goal templates can be organized in a SDC graph as outlined in Figure 6.11 above. The goals for searching and booking are organized in two separated subgraphs because they are functionally disjoint. As the usable Web services, let us consider the numbers of publicly available ones that are found ny the SeekDa search engine (see `http://seekda.com/`): these are 95 for the keyword "flight", and 116 for the keyword "hotel". Let us assume that 10 of these are usable for each of the subgoals out of which 5 are usable for the inputs defined in the goal instance. On this basis we can estimate:

- for subgoal (1), the SDC runtime discoverer needs to traverse the SDC graph from `searchFlight` to one of its lowest children and then inspect the 10 potential candidates. Here, $distance(G, G') = 2$, $b(SDC_{GG})/2 = 1.5$, and the success chance is $10/5 = 2$ so that $Cost_{SDC}(1) = 5.5$. Analogously, we get $Cost_{SDC}(2) = 1 + 1.5 + 2 = 4.5$, $Cost_{SDC}(3) = 2 + 1 + 2 = 5$, and $Cost_{SDC}(4) = 1 + 2 = 3$. As the total time for finding all necessary Web services, we get $Cost_{SDC} = 5.5 + 4.5 + 5 + 3 = 18$;
- the naive engine always needs to subsequently inspect all available Web services until it finds one of the 5 usable ones. So, $Cost_{Naive} = (95 + 116 + 95 + 116)/5 = 84.4$;
- we obtain $PIR = \frac{16}{16} \times \left(100 - \left(\frac{18}{84.4} \times 100\right)\right) = 78.67\ \%$.

This example shows that already for a composite goal with a relatively small number of subgoals and usable Web services a not optimized discovery technique exposes an inadequate performance while the SDC technique ensures acceptable processing times. The same need for efficient and scalable discovery component naturally arises for Web service discovery techniques where – analog to the example – the suitable composition candidates need to be determined at each iteration of the composition algorithm [Bertoli et al., 2007]. We can identify several scenarios with a comparable structure and application purpose, so that similar improvements can be expected from applying our technology. The following lists some of them which have served as demonstration use cases in research projects.

Wholesaler Supply Chain Management. In a case study for the wholesaler of British Telecom (BT) the basic telecommunication services of about 150 trading partners have been integrated [Duke et al., 2005]. The wholesaler selects the most appropriate offer from the trading partners, and also aggregates them into service bundles for a specific customer demand. For this, the customer requests can be modeled as composite goals that consist of several subgoals for basic services. Due to the analogy with the travel scenario and because the trading partners usually provide several services, we can expect a similar performance increase when applying the SDC technique as in the above example.

E-Banking Services. Another prosperous application area are advanced services for e-banking. Examples for this are the mortgage comparison service and the stock market client developed within the DIP project [López-Cobo et al., 2007]. The system integrates offers from several financial service providers, which are then selected and aggregated with respect to specific client requests. Also here we can expect similar benefits as in the above example because the offers of several providers need to be matched with the end-user goals.

Business Process Management. An important application area for SOA technology is Business Process Management (BPM). The idea of semantic BPM is to enhance existing technologies on the basis of ontologies [Hepp et al., 2005]. In particular, the hard-wired invocation of Web services in the distinct activities in a business process as supported by BPEL shall be replaced by a goal-based technique in order to overcome the deficiencies in flexibility and interoperability. For this, the execution of a business process requires runtime Web service discovery, analogously to the solving of composite goals discussed above. The SUPER project (see `www.ip-super.org`) investigates large scale use cases from major European telecommunication providers whose size and setting is similar to the one of the Verizon system discussed above. Thus, we can expect substantial performance improvements when applying our optimized discovery techniques.

Scenarios where Efficient Discovery is Dispensable

We complete the applicability study with discussing scenarios where the employment of our technology appears to be not necessary. Essentially, this is given when most of the criteria of our applicability catalogue as defined in Section 6.2.1 to not apply, respectively when they are evaluated negatively. In particular, (1) the goal-based approach and automated Web service discovery techniques are not required in scenarios where only known Web services shall be used directly within the client applications, and (2) the employment of the SDC technique for optimizing the discovery task is unnecessary in applications that merely deal with a relatively small number of goals and Web services, because then only marginal improvements in the computational performance can be expected.

An example for this is the collection of e-commerce Web services that are provided by Amazon. These provide generic facilities for product, customer, shopping cart, and billing management on the basis of Amazon's own e-commerce technology, with the intention of providing off-the-shelf solutions for the development of other online shops. Although several applications have successfully utilized the Amazon Web services since the initiative commenced in 2005 [Shanahan, 2007], currently the employment of automated and optimized Web service discovery techniques appears to be superfluous. The reason is that – at the time of writing – Amazon is the sole sole provider of such advanced e-commerce solutions in form of Web services, and also that the current number of available Web services is relatively small (11 in March 2008). However, this situation will certainly change when other providers enter the market with competitive Web services.

To conclude, we have shown that the technology developed in this work can achieve significant improvements for the quality of existing SOA systems with respect to the retrieval accuracy and the computational performance of automated Web service discovery. We have discussed several scenarios where our optimized discovery techniques are beneficially applicable, in particular when larger numbers of goals and Web services need to be handled in dynamically changing environments. On the other hand, there are also application scenarios where automated and optimized discovery techniques are not needed, which mainly results from the fact that the current systems merely deal with relatively small numbers of Web services. However, this situation will expectably change when the applications grow to a larger scale in terms of the involved providers and clients.

Chapter 7

Conclusions

This chapter concludes the thesis. The aim of the work has been to develop an efficient and scalable technique for automated Web service discovery as a central and time critical operation within Semantic Web service environments. For this, we have defined a conceptual model for problem-oriented and flexible Web service usage on the basis of goals that formally describe client objectives. We have specified semantically enabled Web service discovery techniques that expose a high retrieval accuracy, and extended this with a caching-based optimization technique for enhancing the computational performance of automated Web service discovery. The techniques have been implemented as components in the open source software developed around the WSMO framework, and we have shown that significant improvements for several real-world SOA applications can be achieved.

The following summarizes the central findings of the thesis and depicts the scientific contributions. Finally, we conclude the work with general remarks on the potential of Semantic Web service technology as well as on the challenges for future developments.

7.1 Summary

In order to provide a concise overview of the thesis, the following recapitulates the line of argument and depicts the central aspects of the developed technology in a condensed manner. We already have discussed related works in detail within the preceding chapters, so that here we can concentrate on the solutions elaborated in this work.

As the general research context, in Chapter 2 we have examined Web services and Service-Oriented Architectures (SOA) as a novel design paradigm for IT systems. A Web service supports the invocation of a program over the Internet. The idea of SOA is to

use Web services as the basic building blocks within IT systems instead of proprietary solutions. The aim is to exploit the Internet as an infrastructure for computation, to reduce the development and maintenance costs by exhaustive re-use of existing implementations, and also to better integrate data and processes within and in between organizations on the basis of a generic and standardized technology. Although the initial Web service technologies around WSDL, SOAP, and UDDI support the technical realization of this vision, they limit the detection of suitable Web services to manual inspection and merely support Web service usage in form of hard-wired invocations. The emerging concept of Semantic Web services (SWS) aims at overcoming these deficiencies by developing inference-based techniques for the automated discovery, composition, mediation, and execution of Web services.

The specific research problem addressed in this work is the retrieval accuracy and computational performance of automated Web service discovery. This is concerned with the detection of the suitable Web services for a given request, and is commonly performed as the first processing step in SWS environments. Thus, discovery appears to be the bottleneck for scalability of SWS systems because it needs to inspect all potential candidates, which becomes in particular important for larger search spaces that can be expected in real-world applications. With respect to this, an automated discovery engine should (1) expose a high retrieval accuracy for the detection of suitable Web services, and (2) perform the discovery task in an efficient and scalable manner in order to serve as a reliable component within SWS environments. We further have argued that a goal-based approach appears to be suitable to better support the Web service usage by clients in a SOA system. Therein, clients describe the objective to be achieved as a goal that abstracts from technical details, and the SWS system then detects and executes the necessary Web services for solving this.

This thesis has presented a conceptual model and a technical solution for this challenge, with the aim of advancing the state-of-the-art in SWS technology developments. For this, the work consists of the following parts that we shall summarize below:

1. a model for describing and handling goals as formalized client requests in order to facilitate problem-oriented and flexible Web service usage (Chapter 3)

2. semantically enabled Web service discovery techniques that distinguish design time and runtime operations and ensure a high retrieval accuracy (Chapter 4)

3. the Semantic Discovery Caching technique as a caching-based mechanism for enhancing the computational performance of automated Web service discovery (Chapter 5)

4. a comprehensive evaluation that assesses the achievable improvements as well as the practical relevance of the developed technologies (Chapter 6).

Goal Model for Semantic Web Services

We have defined a conceptual model for describing and handling goals as formalized client objectives within SWS systems. This extends and refines the initial goal model that has been defined in the WSMO framework, and integrates several solutions that have been developed in related research and development efforts.

The purpose of the goal-based approach is to enable problem-oriented Web service usage in SOA systems, so that clients can focus on the problem to be solved while the technical details are mostly handled by the system. Also, the flexibility of the system for handling changes in the environment can be increased because the actual Web services for a concrete client request can be detected dynamically at runtime.

To facilitate this, we have defined our goal model to consist of two central aspects. We (1) explicitly distinguish *goal templates* as generic and reusable objective descriptions that are kept in the system, and *goal instances* that describe specific client requests and are defined by instantiating a goal template with concrete inputs. To support the automated consumption of Web services for solving a goal, we (2) introduce the notion of *client interfaces* that specify a compatible behavior for the one supported by the Web service. This is defined on the level of goal templates, and the inputs defined in a goal instance are used to actually invoke the Web service at runtime. We further have defined the goal model to be extensible, e.g. by the definition of *composite goals* for describing more complex client objectives or the specification of *non-functional client requirements* in a goal.

This model follows the principles of previous approaches for goal-based technologies developed in several AI disciplines. It adheres to the design of the initial WSMO goal model, which however appears to be not sufficiently elaborated to adequately realize goal-driven SWS technologies. To overcome its deficiencies, we have integrated and refined the conceptual and technical solutions that have been developed in related efforts, particularly in the context of the IRS and the WSMX systems. We have proposed possible extensions for the WSMO specification languages on the conceptual level, and we have shown that existing technologies can be used to work with the refined goal model. Summarizing, the contributions to the realization of the goal-based approach for SWS technologies are:

- the explicit distinction of goal templates and goal instances that facilitates the developed of workable, efficient, and user-friendly SWS techniques
- the concept of client interfaces that enables the automated invocation and consumption of Web services for solving a goal
- the extensibility of the goal model for additional goal types and description elements.

Two-Phase Web Service Discovery

In this work, we consider Web service discovery as the first processing step that detects suitable Web services for solving a goal with respect to the requested and the provided functionality. The actual usability of the discovered candidates is then further inspected in subsequent processing steps that consider non-functional and behavioral aspects.

In accordance to our goal model, we separate the discovery task into two phases: the suitable Web services for goal templates are discovered at design time, and the actual Web services for a goal instance are determined at runtime. We have specified semantic matchmaking techniques that work on sufficiently rich functional descriptions in order to ensure a high precision and recall for functional discovery in both phases.

We have formally defined functional descriptions for precisely describing the provided and requested functionalities with respect to the possible executions of a Web service, respectively the solutions for a goal. For this, we have defined a 3-layered abstraction model. The lowest level considers Web service executions as finite sequences of states that can be observed in the world, and the second level abstracts this to merely consider the start- and end-states. Upon this, we have defined the formal meaning of functional descriptions that formally describe the overall provided and requested functionalities in terms of preconditions and effects, following the standard approach in formal software specification. As the highest abstraction level, we have defined a representation of functional descriptions as a first-order logic structure that allows us to consider the requested and provided functionalities in terms of classical model-theoretic semantics.

On this basis, we have specified the semantic matchmaking techniques for the two-phased discovery approach. For design time discovery, the four matching degrees *exact*, *plugin*, *subsume*, and *intersect* that distinguish different situations when a Web service is functionally usable for solving a goal template, while the *disjoint* degree denotes that this is not given. A goal instance is required to be a valid instantiation of a goal template, which is given if the functional description is satisfiable under the concrete input values. Then, the solutions for a goal instance are a subset of those for the corresponding goal template. Thus, only those Web services that are usable for the goal template are potential candidates for the goal instance, and we further can use the design time discovery results to minimize the reasoning effort for the runtime discovery task. We have shown that our techniques can achieve a high retrieval accuracy for both the design- and the runtime discovery task.

We use classical first-order-logic (FOL) as the specification language for functional descriptions, and we employ the automated theorem prover VAMPIRE to implement the matchmaking techniques in form of proof obligations. The motivation for this is to address the

challenge of Web service discovery on a general level, i.e. to specify the relevant descriptions and matchmaking techniques within a sufficiently expressive language and later on define restrictions on the modeling in order to ensure desirable computational properties. FOL appears to be an adequate choice for this, because it provides a high expressivity and serves as a logical umbrella for most ontology languages. We have discussed the limitations of this approach, in particular with respect to computational complexity of FOL specifications.

The generic design of our discovery techniques allows the adaptation to other SWS framework, in particular to the ones that use the ontology languages currently recommended for the Semantic Web. To summarize, the contributions of this work for the challenge of automated Web service discovery by semantic matchmaking are:

- the explicit separation of design time and runtime operations that enables efficient and scalable Web service discovery
- the definition of sufficiently expressive functional descriptions to precisely describe Web services and goals with respect to their possible executions and solutions
- the support for functional Web service discovery on the level of goal instances
- semantic matchmaking techniques with a high retrieval accuracy for both phases.

The SDC Technique

The Semantic Discovery Caching technique (short: SDC) is a novel approach for enhancing the computational performance of automated Web service discovery, in particular for the runtime discovery tasks as the time critical operation in our two-phased approach. For this, we capture the results of runtime discovery runs and effectively utilize this knowledge to optimize the runtime discovery task. This adopts the principle of semantic caching as a successful optimization technique to the context of automated Web service discovery.

The heart of the technique is the SDC graph, a directed acyclic graph that provides an efficient search index for goal templates and Web services. Its inner layer is the goal graph that organizes the existing goal templates in subsumption hierarchies with respect to the semantic similarity of the requested functionalities. We consider goal templates to be similar if they have at least one common solution, because then mostly the same Web services are usable for them. The outer layer is the discovery cache that captures the design time discovery results by the minimal set of arcs that are necessary to determine the precise usability degree of every available Web service for each existing goal template.

The SDC graph is a formally defined knowledge structure, and for a given set of goals and Web services there is only one possible graph that properly presents the relevant relationships. The logical basis for exploiting SDC graphs are inference rules between similar goal templates and their usable Web services which result from the formal framework. SDC graphs are created automatically on the basis of semantic matchmaking, and we have defined the necessary management algorithms to ensure that its properties are maintained in all possibly occurring situations when a goal template or a Web service is added, removed, or modified. Our prototype implementation keeps SDC graphs in form of a WSML knowledge base, so that it also available as an index structure of the existing goal templates and the available Web services for other SWS techniques. As an example for this, we have presented the prototype of a goal-based browser that allows clients to inspect Web service on the level of the goals that can be solved by them.

The time critical operation in our framework is runtime Web service discovery. This needs to be performed for every goal instance in order to assure the flexibility of the system, and it should be carried out in an efficient and reliable manner in order to warrant an adequate processing of concrete client requests. For this, we have defined optimized algorithms for discovering a single Web services as well as for detecting all usable Web services for a given goal instance. These exploit the knowledge captured in the SDC graph in order to minimize the relevant search space and the number of necessary matchmaking operations. This allows us to reduce the computational costs for runtime discovery, and also to warrant the operational reliability also within larger search spaces because only the minimal set of relevant candidates needs to be inspected. The optimization strategy has been chosen with respect to the known performance deficiencies of reasoning techniques, and within the evaluation we have shown that a significant increase in the computational performance can be achieved in real-world scenarios (see below). Summarizing, the SDC technique is:

- a new approach for the performance optimization of automated Web service discovery
- the SDC graph as the underlying knowledge structure properly organizes the existing goal templates and their usable Web services in all possible situations
- the SDC graph can be generated and maintained automatically, and it provides a general purpose index structure that also can be used for other SWS techniques
- the optimization for runtime discovery is achieved by minimizing the relevant search space and the necessary matchmaking operations
- the high retrieval accuracy of our semantic matchmaking techniques is maintained.

Evaluation

The purpose of the evaluation has been to assess if and to what extent the developed technology accomplishes the research aim of developing an efficient and scalable Web service discovery technique with a high retrieval accuracy. For this, we have conducted an exhaustive performance analysis which covers all relevant aspects, and we have examined the applicability of out technology to existing real-world SOA applications.

For the former, we have in detail examined the functional correctness and the computational performance for all relevant aspects of our technology within the shipment scenario as defined in the Semantic Web Service Challenge, a widely recognized initiative for the demonstration and comparison of SWS technologies based on real-world services. This shows that our prototype implementation can create and properly maintain functionally correct SDC graphs, i.e. the design time discovery results precisely correspond to the expected accuracy, and the resulting SDC graph properly arranges the goal templates and their usable Web services. The creation of the complete SDC graph for the shipment scenario takes about 1 minute, while the update operations are in average performed within 5 seconds.

For the runtime discovery task, the comparison with a naive engine has revealed that optimization is necessary to warrant an adequate processing of concrete client requests, and that the SDC technique can achieve significant improvements in the efficiency, scalability, and stability. It performs the runtime discovery task in average times of 0.5 seconds, independent of the search space size and with a minimal variation over several invocations. The processing times of the naive engine raise proportionally with the number of available Web services to a couple of minutes, which appears to be unacceptable for real-world applications. We also have shown that the SDC-enabled engine exposes a better performance than a slightly reduced version that does not thoroughly exploit the SDC graph.

To assess the practical relevance of this work, we have examined the applicability of our technology in the Verizon SOA system wherein 1.500 Web services are used by over 650 client applications. We have outlined how the goal-based approach can considerably enhance the flexibility by replacing the hard-wired invocation of Web services, and explained how our semantic matchmaking techniques can significantly increase the retrieval accuracy for Web service discovery in comparison to the currently used keyword-based technique. Also, the size of the system requires optimized discovery techniques, and significant performance improvements can be expected from employing the SDC technique. Similar improvements can be expected in other application scenarios where larger numbers of goals and Web services need to be handled. Therewith, the evaluation has shown that the research aims of this thesis have been accomplished successfully because:

- our goal-model adequately facilitates problem-oriented Web service usage in SOA applications and can enhance the flexibility of the system
- the two-phase Web service discovery and our semantic matchmaking techniques ensure a high retrieval accuracy for automated discovery at both design- and runtime
- the SDC technique can achieve significant improvements in efficiency, scalability, and stability of automated Web service discovery
- the techniques developed in this work appear to be beneficially applicable for all scenarios where automated discovery is a frequently performed operation and a larger number of goals and Web services can be expected.

7.2 Discussion and Outlook

We conclude the thesis with general remarks on the status and future predictions for the development and the practical applicability of semantically enhanced SOA technologies. In particular, we discuss the challenge of obtaining the necessary semantic annotations, the general problem of the applicability of comprehensive SWS technologies, and the requirements that arise for a large scale adaptation in industry. Although a detailed investigation exceeds the scope of this work, these aspects appear to be relevant in order to properly judge the potential of technologies like the one developed in this work.

The pre-requisite for the operational reliability of SWS techniques is the existence of exhaustive and correct semantic descriptions of the available Web services and of all other relevant resources. Most of the techniques developed so far focus on new functionalities and the achievable benefits under the assumption that the necessary resource descriptions are given. However, this appears to not be the case, in particular when SWS techniques shall be employed within existing systems. For example, the comprehensive application of our Web service discovery techniques within the Verizon SOA system would require to correctly describe all existing Web service on the basis of an exhaustive domain ontology which needs to be developed. Thus, techniques for the semantic annotation of legacy system appear to be essential in order to assure the applicability of SWS technologies in real-world settings. This challenge as only received very little attention in the research community so far. Presumably, it is possible to adopt existing techniques for the ontology-based annotation of natural language texts for this. However, this can in general only be supported in a semi-automated manner due to the gap between syntactic and adequate semantic descriptions, and also the annotation of Web services is expectably much more complex.

The second aspect related to the general applicability of SWS techniques is the extent to which they shall be employed such that a substantial benefit can be achieved while the effort and costs remain moderate. Let us consider the state-of-the-art in conventional SOA and SWS technology with respect to the achievable degree of automation and the costs for their employment. The initial Web service technology seems to not be optimal because it limits the Web service usage to manual inspection. The SAWSDL approach appears to be only a little bit better: the semantic annotation by additional tags in WSDL documents is relatively easy to realize, but the surplus value of SWS techniques that work on this are only marginal. The OWL-S approach requires exhaustive semantic descriptions on which significant quality improvements can be achieved; however, its employment in existing systems is very expensive. The WSMO approach can achieve the highest benefits because it includes the goal-driven approach as well as mediation facilities, but its employment requires a comprehensive re-design of a SOA system. With respect to this, the aim for future research should be to identify the optimal degree of employment for which the cost-benefit-relation is minimal, and then initiate respective technology standardizations.

Another aspect for the practical employment of SWS technology in large-scale commercial applications is the provision of adequate tooling support. Although a remarkable number of graphical editors, APIs, and execution environments already exists, the technical infrastructure appears to still be not sufficient to adequately support the usage of SWS techniques within real-world SOA applications. Considering the technology developed in this work, customizable graphical user interfaces for the management of goal instances as well as sophisticated validation services for functional descriptions seem to be desirable in order to better support end-users and system developers. However, this can be considered as supplementary development effort once the underlying technology is specified.

To conclude, semantic techniques for the automated detection and usage of Web services like the one developed in this work appear to be capable and eligible to effectively support the idea of Service-Oriented Architectures. However, in order to leverage a successful deployment of such techniques within future SOA technology, it appears to be necessary to properly address the challenges outlined above.

Bibliography

[Abramowicz et al., 2007] Abramowicz, W., Haniewicz, K., Kaczmarek, M., and Zyskowski, D. (2007). Architecture for Web services Filtering and Clustering. In *Proc. of the 2nd International Conference on Internet and Web Applications and Services (ICIW 2007), Mauritius.*

[Akkiraju et al., 2005] Akkiraju, R., Farrell, J., Miller, J., Nagarajan, M., Schmidt, M.-T., Sheth, A., and Verma, K. (2005). Web Service Semantics - WSDL-S. W3C Member Submission 7 November 2005. online: `http://www.w3.org/Submission/WSDL-S/`.

[Albert et al., 2005] Albert, P., Henocque, L., and Kleiner, M. (2005). Configuration-Based Workflow Composition. In *Proc of 3rd International Conference on Web Services (ICWS-05), Orlando, Florida.*

[Alexiev et al., 2005] Alexiev, V., Breu, M., de Bruijn, J., Fensel, D., Lara, R., and Lausen, H. (2005). *Information Integration with Ontologies.* Wiley, West Sussex, UK.

[Allen et al., 1990] Allen, J., Austin, T., and Hendler, J. (1990). *Readings in Planning.* Morgan Kaufmann Publishers.

[Allen, 2006] Allen, P. (2006). *Service Orientation: Winning Strategies and Best Practices.* Cambridge University Press.

[Alonso et al., 2004] Alonso, G., Casati, F., Kuno, H., and Machiraju, V. (2004). *Web Services: Concepts, Architectures and Applications.* Data-Centric Systems and Applications. Springer, Berlin, Heidelberg.

[Anderson, 1991] Anderson, J. R. (1991). Cognitive Architectures in a Rational Analysis. In van Lehn, K., editor, *Architectures for Intelligence*, pages 1–24. Lawrence Erlbaum Associates, Hillsdale, N.J. (USA).

[Andrews et al., 2003] Andrews, T., Curbera, F., Dholakia, H., Goland, Y., Klein, J., Leymann, F., Liu, K., Roller, D., Smith, D., Thatte, S., Trickovic, I., and Weerawarana, S. (2003). Business Process Execution Language for Web Services version 1.1. Specification, IBM, BEA Systems, Microsoft, SAP AG, Siebel Systems. online: `http://www-128.ibm.com/developerworks/library/specification/ws-bpel/`.

[Anton et al., 1994] Anton, A. I., McCracken, W. M., and Potts, C. (1994). Goal Decomposition and Scenario Analysis in Business Process Reengineering. In *Proc. of the 6th Conference on Advanced Information Systems Engineering (CAiSE'94), Utrecht, The Netherlands*, pages 94–104.

[Antoniou et al., 2007] Antoniou, G., Baldoni, M., Bonatti, P. A., Nejdl, W., and Olmedilla, D. (2007). Rule-Based Policy Specification. In Yu, T. and Jajodia, S., editors, *Secure Data Management in Decentralized Systems*, volume 33 of *Advances in Information Security*. Springer.

[Astrachan and Stickel, 1992] Astrachan, O. L. and Stickel, M. E. (1992). Caching and Lemmaizing in Model Elimination Theorem Provers. In *Proc. of the 11th International Conference on Automated Deduction (CADE-11)*.

[Avancha et al., 2002] Avancha, S., Joshi, A., and Finin, T. (2002). Enhanced Service Discovery in Bluetooth. *IEEE Computer*, 35(6):96–99.

[Baader et al., 2003] Baader, F., D., C., McGuinness, D. L., Nardi, D., and Patel-Schneider, P. F. (2003). *The Description Logic Handbook*. Cambridge University Press.

[Bachlechner et al., 2006] Bachlechner, D., Siorpaes, K., Fensel, D., and Toma, I. (2006). Web Service Discovery - A Reality Check. Technical Report DERI-TR-2006-01-17, DERI.

[Baeza-Yates and Ribeiro-Neto, 1999] Baeza-Yates, R. and Ribeiro-Neto, B. (1999). *Modern Information Retrieval*. Addison Wesley.

[Bang-Jønsen and Gutin, 2000] Bang-Jønsen, J. and Gutin, G. (2000). *Digraphs: Theory, Algorithms and Applications*. Monographs in Mathematics. Springer, London.

[Battle et al., 2005] Battle, S., Bernstein, A., Boley, H., Grosof, B., Gruninger, M., Hull, R., Kifer, M., D., M., S., M., McGuinness, D., Su, J., and Tabet, S. (2005). Semantic Web Services Framework (SWSF). W3C Member Submission 9 September 2005. online: `http://www.w3.org/Submission/SWSF/`.

[Benatallah et al., 2005] Benatallah, B., Hacid, M.-S., Leger, A., Rey, C., and Toumani, F. (2005). On Automating Web Services Discovery. *VLDB Journal*, 14(1):84–96.

[Berardi et al., 2003] Berardi, D., Calvanese, D., Giacomo, G. D., Lenzerini, M., and Mecella, M. (2003). Automatic Composition of e-Services that Export their Behavior. In *Proc. of First Int. Conference on Service Oriented Computing (ICSOC)*.

[Bernays and Schönfinkel, 1928] Bernays, P. and Schönfinkel, M. (1928). Zum Entscheidungsproblem der Mathematischen Logik. *Mathematische Annalen*, 99:342 – 372.

[Berners-Lee et al., 2001] Berners-Lee, T., Hendler, J., and Lassila, O. (2001). The Semantic Web. A new form of Web content that is meaningful to computers will uleash a revolution of new possibilities. *Scientific American*, 284(5):34–43.

[Bertoli et al., 2007] Bertoli, P., Hoffmann, J., Lecue, F., and Pistore, M. (2007). Integrating Discovery and Automated Composition: from Semantic Requirements to Executable Code. In *Proc. of the IEEE 2007 International Conference on Web Services (ICWS'07), Salt Lake City, USA*.

[Bidoit et al., 1991] Bidoit, M., Kreowski, H.-J., Lescanne, P., Orejas, F., and Sannella, D. (1991). *Algebraic System Specification and Development. A Survey and Annotated Bibliography*, volume 501 of *Lecture Notes in Computer Science*. Springer.

[Birell and Nelson, 1984] Birell, A. D. and Nelson, B. J. (1984). Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2(1):39–59.

[Book, 1974] Book, R. (1974). Comparing Complexity Classes. *Journal of Computer and System Sciences*, 3(9):213–229.

[Börger et al., 1997] Börger, E., Grädel, E., and Gurevich, Y. (1997). *The Classical Decision Problem*. Perspectives in Mathematical Logic. Springer.

[Borgida, 1996] Borgida, A. (1996). On the Relative Expressiveness of Description Logics and Predicate Logics. *Artificial Intelligence*, 82(1–2):353–367.

[Born et al., 2007] Born, M., Drr, F., and Weber, I. (2007). User-friendly Semantic Annotation of Business Process Modeling. In *Proc. of the WISE 2007 Workshop on Human-friendly Service Description, Discovery and Matchmaking (Hf-SDDM), Nancy, France*.

[Bratko, 2000] Bratko, I. (2000). *Prolog Programming for Artifical Intelligence*. Longman, 3rd edition.

[Bratman, 1987] Bratman, M. E. (1987). *Intention, Plans and Practical Reason*. Harvard University Press, Cambridge, MA (USA).

[Brodie et al., 2005] Brodie, M., Bussler, C., de Brujin, J., Fahringer, T., Fensel, D., Hepp, M., Lausen, H., Roman, D., Strang, T., Werthner, H., and Zaremba, M. (2005). Semantically Enabled ServiceOriented Architectures: A Manifesto and a Paradigm Shift in Computer Science. Technical Report TR-2005-12-26, DERI.

[Bussler, 2003] Bussler, C. (2003). *B2B Integration: Concepts and Architecture*. Springer, Berlin, Heidelberg.

[Cabral and Domingue, 2005] Cabral, L. and Domingue, J. (2005). Mediation of Semantic Web Services in IRS-III. In *Proc. of the Workshop on Mediation in Semantic Web Services (MEDIATE 2005), held at the 6th International Conference on Service Oriented Computing (ICSOC 2005), Amsterdam, The Netherlands*.

[Cardoso and Sheth, 2006] Cardoso, J. and Sheth, A. (2006). *Semantic Web Services, Processes and Applications*. Semantic Web and Beyond. Springer.

[Chidlovskii et al., 1999] Chidlovskii, B., Roncancio, C., and Schneider, M.-L. (1999). Semantic Cache Mechanism for heterogeneous Web Querying. *Computer Networks (Amsterdam, Netherlands)*, 31(11–16):1347–1360.

[Cimpian and Mocan, 2005] Cimpian, E. and Mocan, A. (2005). WSMX Process Mediation Based on Choreographies. In *Proceedings of the 1st International Workshop on Web Service Choreography and Orchestration for Business Process Management at the BPM 2005, Nancy, France*.

[Cimpian et al., 2006] Cimpian, E., Mocan, A., and Stollberg, M. (2006). Mediation Enabled SemanticWeb Services Usage. In *Proc. of the 1st Asian Semantic Web Conference (ASWC 2006), Beijing, China*.

[Clancey, 1985] Clancey, W. J. (1985). Heuristic Classification. *Artificial Intelligence*, 27(3):289–350.

[Clayton et al., 2002] Clayton, R., Cleary, J. G., Pfahringer, B., and Utting, M. (2002). Tabling Structures for Bottom-Up Logic Programming. In *Proc. of 12th International Workshop on Logic Based Program Synthesis and Tranformation, Madrid, Spain*.

[Clement et al., 2004] Clement, L., Hately, A., von Riegen, C., and Rogers, T. e. (2004). UDDI Version 3. Uddi spec technical committee draft, OASIS. Available from http://uddi.org/pubs/uddi_v3.htm.

[Cohen, 2006] Cohen, F. (2006). *Fast SOA: The way to use native XML Technology to achieve Service Oriented Architecture Governance, Scalability, and Performance*. Series in Data Management Systems. Morgan Kaufmann.

[Cohen and Levesque, 1990] Cohen, P. R. and Levesque, H. J. (1990). Intention is Choice with Commitment. *Artificial Intelligence*, 42:213–261.

[Colucci et al., 2005] Colucci, S., Di Noia, T., Di Sciascio, E., Donini, F. M., and Mongiello, M. (2005). Concept abduction and contraction for semantic-based discovery of matches and negotiation spaces in an e-marketplace. *Electronic Commerce Research and Applications*, 4:345–361.

[Constantinescu et al., 2005] Constantinescu, I., Binder, W., and Faltings, B. (2005). Flexible and Efficient Matchmaking and Ranking in Service Directories. In *Proc. of the 3rd International Conference on Web Services (ICWS 2005), Florida, USA*.

[Crawford et al., 2000] Crawford, I., Wadleigh, K. R., and Wadleigh, K. (2000). *Software Optimization for High Performance Computing: Creating Faster Application*. Prentice-Hall.

[Crow et al., 2007] Crow, E. L., Davis, F. A., and Maxfield, M. W. (2007). *Statistics Manual*. Dover Publications, Inc., New York (USA).

[Davis et al., 2006] Davis, J., Studer, R., and Warren, P. (2006). *Semantic Web Technology. Trends and Research in Ontology-based System*. Wiley & Sons.

[de Bruijn, 2006] de Bruijn, J. (2006). Logics for the Semantic Web. In Cardoses, J., editor, *Semantic Web: Theory, Tools and Applications*. Idea Publishing Group.

[de Bruijn and Fensel, 2005] de Bruijn, J. and Fensel, D. (2005). Ontology Definitions. In Marcia J. Bates, Mary Niles Maack, M. D., editor, *Encyclopedia of Library and Information Science*. Tylor and Francis Books.

[de Bruijn and Heymans, 2006] de Bruijn, J. and Heymans, S. (2006). WSML Ontology Semantics. WSML Deliverable D28.3 Final Draft. available at: `http://www.wsmo.org/TR/d28/d28.3/`.

[de Bruijn et al., 2005a] de Bruijn, J., Lara, R., Arroyo, S., Gomez, J. M., Han, S.-K., and Fensel, D. (2005a). A Unified Semantic Web Services Architecture based on WSMF and UPML. *The International Journal of Web Engineering and Technology (IJWET)*, 2(2-3):148–180.

[de Bruijn et al., 2005b] de Bruijn, J., Lausen, H., Krummenacher, R., Polleres, A., Predoiu, L., Kifer, M., and Fensel, D. (2005b). The Web Service Modeling Language WSML. Deliverable D16.1 final draft 05 Oct 2005, WSML Working Group. Available at: `http://www.wsmo.org/TR/d16/d16.1/v0.2/`.

[de Leenheer and Mens, 2006] de Leenheer, P. and Mens, T. (2006). Ontology Evolution: State of the Art and Future Directions. In Hepp, de Leenheer, de Moor, and Sure, editors, *Ontology Management. Semantic Web, Semantic Web Services, and Business Applications*. Springer.

[de Nivelle and Piskac, 2005] de Nivelle, H. and Piskac, R. (2005). Verification of an Off-Line Checker for Priority Queues. In *Proc. of the 3rd IEEE International Conference on Software Engineering and Formal Methods (SEFM'2005), Koblenz, Germany.*

[Denney et al., 2006] Denney, E., Fischer, B., and Schumann, J. (2006). An Empirical Evaluation of Automated Theorem Provers in Software Certification. *International Journal on Artificial Intelligence Tools*, 15(1):81–107.

[Dickinson and Wooldridge, 2005] Dickinson, I. and Wooldridge, M. (2005). Agents are not (just) Web Services: Considering BDI Agents and Web Services. In *Proceedings of the 2005 Workshop on Service-Oriented Computing and Agent-Based Engineering (SOCABE'2005), Utrecht, The Netherlands.*

[Diestel, 2005] Diestel, R. (2005). *Graph Theory*, volume 173 of *Graduate Texts in Mathematics*. Springer, Heidelberg, 3. edition.

[Diller, 1994] Diller, A. (1994). *Z: An Introduction to Formal Methods*. John Wiley & Sons, 2 edition.

[Dimitrov et al., 2007] Dimitrov, M., Simov, A., Momtchev, V., and Konstantinov, M. (2007). WSMO Studio - A Semantic Web Services Modelling Environment for WSMO (System Description). In *Proc. of the 4th European Semantic Web Conference (ESWC)*, Innsbruck, Austria.

[Domingue et al., 2008] Domingue, J., Cabral, L., Galizia, S., Tanasescu, V., Gugliotta, A., Norton, B., and Pedrinaci, C. (2008). IRS-III: A Broker-based Approach to Semantic Web Services. *Journal of Web Semantics*. (to appear).

[Domingue et al., 2005] Domingue, J., Galizia, S., and Cabral, L. (2005). Choreography in IRS III. Coping with Heterogeneous Interaction Patterns in Web Services. In *Proc. of the 4th International Semantic Web Conference (ISWC 2005), Galway, Ireland*.

[Duke et al., 2005] Duke, A., Richardson, M., Watkins, S., and Roberts, M. (2005). Towards B2B Integration in Telecommunications with Semantic Web Services. In *Proceedings of Second European Semantic Web Conference, ESWC 2005, Heraklion, Crete, Greece*, pages 710–724.

[Duschka and Genesereth, 1997] Duschka, O. M. and Genesereth, M. R. (1997). Query Planning in Infomaster. In *Proc. of the ACM Symposium on Applied Computing*.

[Ebert et al., 2004] Ebert, C., Dumke, R., Bundschuh, M., and Schmietendorf, A. (2004). *Best Practices in Software Measurement*. Springer.

[Erl, 2005] Erl, T. (2005). *Service-Oriented Architecture (SOA). Concepts, Technology, and Design*. Prentice Hall PTR.

[Farrell and Lausen, 2007] Farrell, J. and Lausen, H. (2007). Semantic Annotations for WSDL and XML Schema. W3C Recommendation 26 January 2007. online: `http://www.w3.org/TR/sawsdl/`.

[Fensel, 1995] Fensel, D. (1995). Formal Specification Languages in Knowledge and Software Engineering. *The Knowledge Engineering Review*, 10(4):361–404.

[Fensel, 2000] Fensel, D. (2000). *Problem-Solving Methods: Understanding, Description, Development, and Reuse*. Springer, Berlin, Heidelberg.

[Fensel, 2003] Fensel, D. (2003). *Ontologies: A Silver Bullet for Knowledge Management and E-Commerce*. Springer, Berlin, Heidelberg, 2 edition.

[Fensel and Bussler, 2002] Fensel, D. and Bussler, C. (2002). The Web Service Modeling Framework WSMF. *Electronic Commerce Research and Applications*, 1(2).

[Fensel et al., 2006] Fensel, D., Lausen, H., Polleres, A., de Bruijn, J., Stollberg, M., Roman, D., and Domigue, J. (2006). *Enabling Semantic Web Services. The Web Service Modeling Ontology*. Springer, Berlin, Heidelberg.

[Fensel et al., 2003] Fensel, D., Motta, E., Benjamins, V. R., Crubezy, M., Decker, S., Gaspari, M., Groenboom, R., Grosso, W., van Harmelen, F., Musen, M., Plaza, E., Schreiber, G., Studer, R., and Wielinga, B. (2003). The Unified Problem Solving Method Development Language UPML. *Knowledge and Information Systems Journal (KAIS)*, 5(1):83–131.

[Fensel and Schönegge, 1998] Fensel, D. and Schönegge, A. (1998). Inverse Verification of Problem-Solving Methods. *International Journal of Human-Computer Studies*, 49(4):339–361.

[Fensel and Straatman, 1998] Fensel, D. and Straatman, R. (1998). The Essence of Problem-Solving Methods: Making Assumptions to Gain Efficiency. *International Journal of Human-Computer Studies*, 48(2):181–215.

[Fensel and van Harmelen, 2007] Fensel, D. and van Harmelen, F. (2007). Unifying Reasoning and Search to Web Scale. *IEEE Internet Computing*, 11(2).

[Fikes and Nilsson, 1971] Fikes, R. and Nilsson, N. (1971). STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence*, 2:189–208.

[Galizia, 2006] Galizia, S. (2006). WSTO: A Classification-Based Ontology for Managing Trust in Semantic Web Services. In *Proc. of the 3th European Semantic Web Conference (ESWC 2006), Budva, Montenegro.*

[Galizia et al., 2007] Galizia, S., Gugliotta, A., and Domingue, J. (2007). A Trust Based Methodology for Web Service Selection. In *Proc. of the 1st IEEE International Conference on Semantic Computing (IEEE-ICSC 2007) , Irvine, California, USA, September 17-19, 2007.*

[Gallier, 1986] Gallier, J. H. (1986). *Logic for Computer Science: Foundations of Automatic Theorem Proving.* John Wiley & Sons.

[Gallo and Rago, 1994] Gallo, G. and Rago, G. (1994). The Satisfiability Problem for the Schönfinkel-Bernays Fragment: Partial Instantiation and Hypergraph Algorithms. Technical Report 4/94, Dip. Informatica, Universita di Pisa.

[Gannon et al., 1994] Gannon, J. D., Purtilo, J. M., and Zelkowitz, M. V. (1994). *Software Specification. A Comparison of Formal Methods.* Ablex Publishing Co.

[Genesereth and Nilsson, 1987] Genesereth, M. J. and Nilsson, N. J. (1987). *Logical Foundations of Artificial Intelligence*. Morgan Kaufmann.

[Gerede et al., 2004] Gerede, C. E., Hull, R., Ibarra, O. H., and Su, J. (2004). Automated Composition of E-services: Lookaheads. In *Proc. of International Conference on Service Oriented Computing (ICSOC 2004), NY*.

[Ghallab et al., 2004] Ghallab, M., Nau, D., and Traverso, P. (2004). *Automated Planning. Theory & Practice*. Morgan Kaufmann.

[Giorgini et al., 2003] Giorgini, P., Mylopoulos, J., Nicchiarelli, E., and Sebastiani, R. (2003). Formal Reasoning Techniques for Goal Models. *Journal on Data Semantics I*, Lecture Notes in Computer Science, Springer(2800):1–20.

[Giunchiglia et al., 2005] Giunchiglia, F., Yatskevich, M., and Giunchiglia, E. (2005). Efficient Semantic Matching. In *Proceedings of the 2nd European Semantic Web Conference (ESWC 2005), Crete, Greece*.

[Godfrey and Gryz, 1997] Godfrey, P. and Gryz, J. (1997). Semantic Query Caching for Heterogeneous Databases. In *Proc. of 4th Knowledge Representation Meets Databases Workshop (KRDB) at VLDB'97, Athens, Greece*.

[Goméz-Peréz et al., 2003] Goméz-Peréz, A., Corcho, O., and Fernandez-Lopez, M. (2003). *Ontological Engineering. With Examples from the Areas of Knowledge Management, E-Commerce and Semantic Web*. Series of Advanced Information and Knowledge Processing. Springer, Berlin, Heidelberg.

[Gruber, 1993] Gruber, T. R. (1993). A Translation Approach to Portable Ontology Specifications. *Knowledge Acquisition*, 5:199–220.

[Gruninger and Menzel, 2003] Gruninger, M. and Menzel, C. (2003). The Process Specification Language (PSL) Theory and Applications. *AI Magazine*, 24(3):63–74.

[Haller et al., 2005] Haller, A., Cimpian, E., Mocan, A., Oren, E., and Bussler, C. (2005). WSMX - A Semantic Service-Oriented Architecture. In *Proceedings of the International Conference on Web Service (ICWS 2005), Orlando, Florida*.

[Haller et al., 2007] Haller, A., Filipowska, A., Kaczmarek, M., van Lessen, T., Nitzsche, J., and Norton, B. (2007). Process Ontology Language and Operational Semantics for Semantic Business Processes. Deliverable D1.3, SUPER.

[Haller and Scicluna, 2005] Haller, A. and Scicluna, J. (2005). WSMX Choreography. Working Draft D13.9, WSMX. 28 June 2005, online: `www.wsmo.org/TR/d13/d13.9/`.

[Handy, 1998] Handy, J. (1998). *The Cache Memory Book*. Series in Computer Architecture and Design. Morgan Kaufmann, 2 edition.

[Harth and Decker, 2005] Harth, A. and Decker, S. (2005). Optimized Index Structures for Querying RDF from the Web. In *Proc. of 3rd Latin American Web Congress, Buenos Aires, Argentina, Oct. 31 - Nov.*

[Haselwanter et al., 2006] Haselwanter, T., Kotinurmi, P., Moran, M., Vitvar, T., and Zaremba, M. (2006). WSMX: a Middleware Platform to Enact Semantic SOA in a B2B Integration Scenario. In *Proc. of the 4th International Conference on Service Oriented Computing (ICSOC 2006), Chicago, USA*.

[He et al., 2004] He, H., Haas, H., and Orchard, D. (2004). Web Services Architecture Usage Scenarios. W3C Working Group Note 11 February 2004November 2004. online: `http://www.w3.org/TR/ws-arch-scenarios/`.

[Hepp et al., 2007] Hepp, M., de Leenheer, P., de Moor, A., and Sure, Y. (2007). *Ontology Management. Semantic Web, Semantic Web Services, and Business Applications*. Semantic Web and Beyond. Springer.

[Hepp et al., 2005] Hepp, M., Leymann, F., Domingue, J., Wahler, A., and Fensel, D. (2005). Semantic Business Process Management: A Vision Towards Using Semantic Web Services for Business Process Management. In *Proc. of the IEEE International Conference on e-Business Engineering (ICEBE 2005), October 18-20, 2005, Beijing, China*.

[Hoare, 1969] Hoare, C. A. R. (1969). An Axiomatic Basis for Computer Programming. *Communincations of the ACM*, 12(10):576–580.

[Hoffmann et al., 2007] Hoffmann, J., Bertoli, P., and Pistore, M. (2007). Service Composition as Planning, Revisited: In Between Background Theories and Initial State Uncertainty. In *Proc. of the 22nd National Conference of the American Association for Artificial Intelligence (AAAI'07), Vancouver, Canada*.

[Hofstadter, 1979] Hofstadter, D. R. (1979). *Goedel, Escher, Bach: an Eternal Golden Braid*. Basic Books, New York (USA).

[Horrocks et al., 2004] Horrocks, I., Li, L., Turi, D., and Bechhofer, S. (2004). The Instance Store: Description Logic Reasoning with Large Numbers of Individuals. In *Proc. of International Workshop on Description Logics (DL 2004), British Columbia, Canada.*

[Horrocks et al., 1998] Horrocks, I., Sattler, U., and Tobies, S. (1998). A PSpace-Algorithm for Deciding $\mathcal{ALCN}\mathcal{I}_{R^+}$-Satisfiability. Technical Report LTCS-98-08, RWTH Aachen, Germany.

[Hull et al., 2006] Hull, D., Zolin, E., Bovykin, A., Horrocks, I., Sattler, U., and Stevens, R. (2006). Deciding Semantic Matching of Stateless Services. In *Proc. of the 21st National Conference on Artificial Intelligence (AAAI'2006).*

[Jones, 1990] Jones, C. B. (1990). *Systematic Software Development using VDM.* Prentice-Hall.

[Kavantzas et al., 2005] Kavantzas, N., Burdett, D., Ritzinger, G., Fletcher, T., Lafon, Y., and Barreto (eds.), C. (2005). Web Services Choreography Description Language Version 1.0. Candidate Recommendation 9 November 2005, W3C. online at: `http://www.w3.org/TR/ws-cdl-10/`.

[Keller and Basu, 1996] Keller, A. M. and Basu, J. (1996). A Predicate-based Caching Scheme for Client-Server Database Architectures. *The VLDB Journal*, 5:35–47.

[Keller et al., 2006a] Keller, U., Lara, R., Lausen, H., and Fensel, D. (2006a). Semantic Web Service Discovery in the WSMO Framework. In Cardoses, J., editor, *Semantic Web: Theory, Tools and Applications.* Idea Publishing Group.

[Keller et al., 2005] Keller, U., Lara, R., Lausen, H., Polleres, A., and Fensel, D. (2005). Automatic Location of Services. In *Proceedings of the 2nd European Semantic Web Conference (ESWC 2005), Crete, Greece.*

[Keller and Lausen, 2006] Keller, U. and Lausen, H. (2006). Functional Description of Web Services. Deliverable D28.1, WSML Working Group. Most recent version available at: `http://www.wsmo.org/TR/d28/d28.1/`.

[Keller et al., 2006b] Keller, U., Lausen, H., and Stollberg, M. (2006b). On the Semantics of Funtional Descriptions of Web Services. In *Proceedings of the 3rd European Semantic Web Conference (ESWC 2006), Montenegro.*

[Kerrigan et al., 2007] Kerrigan, M., Mocan, A., Tanler, M., and Fensel, D. (2007). The Web Service Modeling Toolkit - An Integrated Development Environment for Semantic

Web Services (System Description). In *Proc. of the 4th European Semantic Web Conference (ESWC)*, Innsbruck, Austria.

[Kifer et al., 2004] Kifer, M., Lara, R., Polleres, A., Zhao, C., Keller, U., Lausen, H., and Fensel, D. (2004). A Logical Framework for Web Service Discovery. In *Proc. of the ISWC 2004 workshop on Semantic Web Services: Preparing to Meet the World of Business Applications, Hiroshima, Japan.*

[Kiryakov et al., 2005] Kiryakov, A., Ognyanov, D., and Manov, D. (2005). A Pragmatic Semantic Repository for OWL. In *Proc. of Int. Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS 2005), NYC.*

[Klusch et al., 2006] Klusch, M., Fries, B., and Sycara, K. (2006). Automated Semantic Web Service Discovery with OWLS-MX. In *Proc. of the 5th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2006), Hakodate, Japan, May 8 - 12.*

[Kopecky, 2007] Kopecky, J. (2007). Semantic Annotations for WSDL and XML Schema. Talk at W3C track in the WWW 2007 Conference, Banff, Canada.

[Kopecký et al., 2006] Kopecký, J., Roman, D., Moran, M., and Fensel, D. (2006). Semantic Web Services Grounding. In *Proc. of the International Conference on Internet and Web Applications and Services (ICIW'06), Guadeloupe, French Caribbean.*

[Kuokka and Harada, 1996] Kuokka, D. and Harada, L. (1996). Integrating Information via Matchmaking. *Journal of Intelligent Information Systems*, 6(2–3):261–279.

[Lambert et al., 2007] Lambert, D., Galizia, S., and Domingue, J. (2007). Agile Elicitation of Semantic Goals by using Wikis. In *Proc. of the WISE 2007 Workshop on Human-friendly Service Description, Discovery and Matchmaking (Hf-SDDM), Nancy, France.*

[Lara et al., 2004] Lara, L., Roman, D., Polleres, A., and Fensel, D. (2004). A Conceptual Comparison of WSMO and OWL-S. In *Proc. of the European Conference on Web Services (ECOWS 2004), Erfurt, Germany.*

[Lara, 2006] Lara, R. (2006). Two-phased Web Service Discovery. In *Proc. of AI-Driven Technologies for Services-Oriented Computing Workshop at AAAI-06, Boston, USA.*

[Lara et al., 2006] Lara, R., Corella, M. A., and Castells, P. (2006). A Flexible Model for Web Service Discovery. In *Proc. of the 1st International Workshop on Semantic Matchmaking and Resource Retrieval: Issues and Perspectives, Seoul, South Korea.*

[Lausen et al., 2005] Lausen, H., Polleres, A., and Roman (eds.), D. (2005). Web Service Modeling Ontology (WSMO). W3C Member Submission 3 June 2005. online: `http://www.w3.org/Submission/WSMO/`.

[Lewis, 1980] Lewis, H. R. (1980). Complexity Results for Classes of Quantificational Formulas. *Journal of Computer and System Sciences (JCSS)*, 21(3):317–353.

[Li and Horrocks, 2003] Li, L. and Horrocks, I. (2003). A Software Fframework for Matchmaking based on Semantic Web Technology. In *Proceedings of the 12th International Conference on the World Wide Web, Budapest, Hungary.*

[López-Cobo et al., 2007] López-Cobo, J.-M., Losada, S., Cicurel, L., Bas, J.-L., Bellido, S., and Benjamins, R. (2007). Ontology Management in e-Banking Applications. In Hepp, M., Leenheer, P. D., de Moor, A., and Sure, Y., editors, *Ontology Management. Semantic Web, Semantic Web Services, and Business Applications.* Springer.

[Loveland, 1978] Loveland, D. W. (1978). *Automated Theorem Proving: A Logical Basis*, volume 6 of *Fundamental Studies in Computer Science.* North-Holland Publishing.

[Lu, 2005] Lu, H. (2005). Semantic Web Services Discovery and Ranking. In *Proc. of the ACM International Conference on Web Intelligence(WI 2005)*, Compiegne, France.

[Ludwig et al., 2003] Ludwig, H., Keller, A., Dan, A., King, R. P., and Franck, R. (2003). Web service level agreement (wsla). Language specification, IBM. online: `http://www.research.ibm.com/wsla/WSLASpecV1-20030128.pdf`.

[Manola and Miller, 2007] Manola, F. and Miller, E. (2007). SPARQL Query Language for RDF. W3C Candidate Recommendation 14 June 2007. online: `www.w3.org/RDF/`.

[Marks and Bell, 2006] Marks, E. A. and Bell, M. (2006). *Service-Oriented Architecture (SOA): A Planning and Implementation Guide for Business and Technology.* Wiley.

[Martens, 2003] Martens, A. (2003). On Compatibility of Web Services. *Petri Net Newletter*, 65:12–20.

[Martin, 2004] Martin, D. (2004). OWL-S: Semantic Markup for Web Services. W3C Member Submission 22 November 2004. online: `http://www.w3.org/Submission/OWL-S/`.

[Martin et al., 1999] Martin, D., Cheyer, A. J., and Moran, D. B. (1999). The Open Agent Architecture: A Framework for building Distributed Software Systems. *Applied Artificial Intelligence*, 13(1–2):91–128.

[Martin et al., 2007] Martin, D., Paolucci, M., and Wagner, M. (2007). Towards Semantic Annotations of Web Services: OWL-S from the SAWSDL Perspective. In *Proc. of the ESWC 2007 workshop OWL-S: Experiences and Directions, Innsbruck, Austria.*

[McCarthy, 1963] McCarthy, J. (1963). Situations, Actions and Causal Laws. Technical report, Stanford University.

[McCarthy and Hayes, 1969] McCarthy, J. and Hayes, P. J. (1969). Some Philosophical Problems from the Standpoint of Artificial Intelligence. *Machine Intelligence*, 4:463–502.

[McGuinness and van Harmelen, 2004] McGuinness, D. and van Harmelen, F. (2004). OWL Web Ontology Language - Overview. W3C Recommendation 10 February 2004. online: `http://www.w3.org/TR/owl-features/`.

[McIlraith et al., 2001] McIlraith, S., Cao Son, T., and Zeng, H. (2001). Semantic Web Services. *IEEE Intelligent Systems, Special Issue on the Semantic Web*, 16(2):46–53.

[McIlraith and Son, 2002] McIlraith, S. and Son, T. C. (2002). Adapting Golog for Composition of Semantic Web Services. In *roc. of the 8th International Conference on Knowledge Representation and Reasoning (KR '02), Toulouse, France.*

[Meyer, 2000] Meyer, B. (2000). *Object-Oriented Software Construction.* Professional Technical Reference. Prentice Hall, 2. edition.

[Mocan and Cimpian, 2007] Mocan, A. and Cimpian, E. (2007). An Ontology-based Data Mediation Framework for Semantic Environments. *International Journal on Semantic Web and Information Systems (IJSWIS)*, 3(2):66 – 95.

[Mocan et al., 2005] Mocan, A., Cimpian, E., Stollberg, M., Scharffe, F., and Scicluna, J. (2005). WSMO Mediators. WSMO deliverable D29 final draft 21 Dec 2005. available at: `http://www.wsmo.org/TR/d29/`.

[Motik et al., 2007] Motik, B., Shearer, R., and Horrocks, I. (2007). Optimized Reasoning in Description Logics using Hypertableaux. In *Proc. of the 21st Conference on Automated Deduction (CADE-21), Bremen, Germany, July 17-20.*

[Motta, 1999] Motta, E. (1999). *Reusable Components for Knowledge Modelling: Principles and Case Studies in Parametric Design.* IOS Press, Amsterdam.

[Newell, 1982] Newell, A. (1982). The Knowledge Level. *Artificial Intelligence*, 18:87–122.

[Newell, 1990] Newell, A. (1990). *Unified Theories of Cognition*. Harvard University Press, Cambridge, MA (USA).

[Newell and Simon, 1972] Newell, A. and Simon, H. A. (1972). *Human Problem Solving*. Prentice-Hall, Englewood Cliffs, New Jersey.

[Nieuwenhuis et al., 2003] Nieuwenhuis, R., Hillenbrand, T., Riazanov, A., and Voronkov, A. (2003). On the Evaluation of Indexing Techniques for Theorem Proving. In *Proc. of the 1st International Joint Conference on Automated Reasoning, Siena, Italy*.

[Nilsson, 1971] Nilsson, N. J. (1971). *Problem-Solving Methods in Artifuical Intelligence*. McGraw-Hill.

[Noia et al., 2003] Noia, T. D., Sciascio, E. D., Donini, F., and Mongiello, M. (2003). A System for Principled Matchmaking in an Electronic Marketplace. In *Proc. of the 12th International Conference on the World Wide Web (WWW'03), Budapest, Hungary*.

[Norton and Pedrinaci, 2006] Norton, B. and Pedrinaci, C. (2006). 3-Level Service Composition and Cashew: A Model for Orchestration and Choreography in Semantic Web Services. In *On the Move to Meaningful Internet Systems (OTM Workshops 2006), Montpellier, France, Oct 29 - Nov 03*.

[Noy, 2004] Noy, N. (2004). Semantic Integration: a Survey of Ontology-based Approaches. *ACM SIGMOD Record*, 33(4):65–70.

[Olmedilla et al., 2004] Olmedilla, D., Lara, R., Polleres, A., and Lausen, H. (2004). Trust Negotiation for Semantic Web Services. In *Proc. of the 1st International Workshop on Semantic Web Services and Web Process Composition at the ICWS 2004, SanDiego, California (USA)*.

[Oren et al., 2005] Oren, N., Preece, A., and Norman, T. (2005). Service Level Agreements for Semantic Web Agents. In *Proc. of the 2005 AAAI Fall Symposim on Agents and the Semantic Web, Arlington, Virgina (USA)*.

[Oundhakar et al., 2005] Oundhakar, S., Verma, K., Sivashanmugam, K., Sheth, A., and Miller, J. (2005). Discovery of Web Services in a Multi-Ontology and Federated Registry Environment. *International Journal of Web Services Research*, 1(3):8–39.

[Paolucci et al., 2003] Paolucci, M., Ankolekar, A., Srinivasan, N., and Sycara, K. (2003). The DAML-S Virtual Machine. In *Proc. of the 2nd International Semantic Web Conference (ISWC),Sandial Island, Florida*.

[Paolucci et al., 2002] Paolucci, M., Kawamura, T., Payne, T., and Sycara, K. (2002). Semantic Matching of Web Services Capabilities. In *Proc. of the 1st International Semantic Web Conference, Sardinia, Italy.*

[Papadimitriou, 1994] Papadimitriou, C. H. (1994). *Computational Complexity.* Addison-Wesley.

[Paurobally et al., 2005] Paurobally, S., Tamma, V., and Wooldridge, M. (2005). Cooperation and Agreement between Semantic Web Services. In *Proc. of the W3C Workshop on Frameworks for Semantics in Web Services. Innsbruck, Austria.*

[Petrie et al., 2008] Petrie, C., Lausen, H., Zaremba, M., and Margaria-Steffen, T. (2008). *Semantic Web Services Challenge*, volume 8 of *Semantic Web and Beyond.* Springer. (to appear).

[Pistore et al., 2004] Pistore, M., Barbon, F., Bertoli, P., Shaparau, D., and Traverso, P. (2004). Planning and Monitoring Web Service Composition. In *Proc. of the Workshop on Planning and Scheduling for Web and Grid Services at the ICAPS 2004, Whistler, British Columbia, Canada, June 3-7.*

[Preist, 2004] Preist, C. (2004). A Conceptual Architecture for Semantic Web Services. In *Proc. of the 2nd International Semantic Web Conference (ISWC 2004).*

[Prevosto and Waldmann, 2006] Prevosto, V. and Waldmann, U. (2006). SPASS+T. In *Proc. of the FLoC'06 Workshop on Empirically Successful Computerized Reasoning, Seattle, USA.*

[Rao and Georgeff, 1991] Rao, A. S. and Georgeff, M. P. (1991). Modeling Rational Agents within a BDI-Architecture. In *Proceedings of Knowledge Representation and Reasoning (KR&R-91)*, pages 473–484.

[Reiter, 1991] Reiter, R. (1991). The Frame Problem in the Situation Calculus: a Simple Solution (sometimes) and a Completeness Result for Goal Regression. In Lifschitz, I., editor, *Artificial Intelligence and Mathematical Theory of Computation, Papers in Honor of John McCarthy.* Academic Press.

[Riazanov, 2003] Riazanov, A. (2003). *Implementing an Efficient Theorem Prover.* PhD thesis, The University of Manchester.

[Riazanov and Voronkov, 2002] Riazanov, A. and Voronkov, A. (2002). The Design and Implementation of VAMPIRE. *AI Communications*, 15(2):91–110. Special Issue on CASC.

[Roman et al., 2006] Roman, D., Lausen, H., and Keller, U. e. (2006). Web Service Modeling Ontology (WSMO). Final Draft D2, WSMO Working Group. version v1.3, 21 October 2006, online at: `http://www.wsmo.org/TR/d2/v1.3/`.

[Roman and Scicluna, 2007] Roman, D. and Scicluna, J. (2007). Orchestration in WSMO. Working Draft D15, WSMO. April 22, 2007; online at: `www.wsmo.org/2005/d15/v0.1/`.

[Roman et al., 2007] Roman, D., Scicluna, J., and Nitzsche, J. (2007). Ontology-based Choreography. Final Draft D14, WSMO Working Group. version v1.0, 15 February 2007, online at: `http://www.wsmo.org/TR/d14/v1.0/`.

[Scharffe and de Bruijn, 2005] Scharffe, F. and de Bruijn, J. (2005). A language to specify mappings between ontologies. In *Proc. of the Internet Based Systems IEEE Conference (SITIS05)*.

[Schmid and Lindemann, 1998] Schmid, B. F. and Lindemann, M. (1998). Elements of a Reference Model for Electronic Markets. In *Proc. of the 31st Hawaiian International Conference on System Sciences, Hawaii, USA*.

[Schulz and Sutcliffe, 2005] Schulz, S. and Sutcliffe, G. (2005). TPTP Format for Interpreted Integer Arithmetic. available at: `www.cs.miami.edu/~tptp/TPTP/Proposals/IntegerArithmetic.html`.

[Serain and Craiq, 2002] Serain, D. and Craiq, I. (2002). *Middleware and Enterprise Application Integration*. Springer, 2. edition.

[Shanahan, 2007] Shanahan, F. (2007). *Amazon.com Mashups*. Wiley Publishing Inc.

[Shanahan, 1997] Shanahan, M. (1997). *Solving the Frame Problem: A Mathematical Investigation of the Common Sense Law of Inertia*. MIT Press, Cambridge, MA.

[Smullyan, 1968] Smullyan, R. M. (1968). *First Order Logic*. Springer.

[Srinivasan et al., 2004a] Srinivasan, N., Paolucci, M., and Sycara, K. (2004a). Adding OWL-S to UDDI – Implementation and Throughput. In *Proc. of the First International Workshop on Semantic Web Services and Web Process Composition at the ICWS 2004, San Diego, California, USA*.

[Srinivasan et al., 2004b] Srinivasan, N., Paolucci, M., and Sycara, K. (2004b). An Efficient Algorithm for OWL-S Based Semantic Search in UDDI. In *Proc. of the First International Workshop on Semantic Web Services and Web Process Composition at the ICWS 2004, San Diego, California, USA.*

[Stollberg, 2005] Stollberg, M. (2005). Reasoning Tasks and Mediation on Choreography and Orchestration in WSMO. In *Proceedings of the 2nd International WSMO Implementation Workshop (WIW 2005), Innsbruck, Austria.*

[Stollberg et al., 2005a] Stollberg, M., Cimpian, E., and Fensel, D. (2005a). Mediating Capabilities with Delta-Relations. In *Proceedings of the First International Workshop on Mediation in Semantic Web Services, co-located with the Third International Conference on Service Oriented Computing (ICSOC 2005), Amsterdam, the Netherlands.*

[Stollberg et al., 2006a] Stollberg, M., Cimpian, E., Mocan, A., and Fensel, D. (2006a). A Semantic Web Mediation Architecture. In *Proceedings of the 1st Canadian Semantic Web Working Symposium (CSWWS 2006), Quebec, Canada.*

[Stollberg et al., 2006b] Stollberg, M., Feier, C., Roman, D., and Fensel, D. (2006b). Semantic Web Services – Concepts and Technology. In Ide, N., Cristea, D., and Tufis, D., editors, *Language Technology, Ontologies, and the Semantic Web.* Kluwer Publishers.

[Stollberg and Kerrigan, 2007] Stollberg, M. and Kerrigan, M. (2007). Goal-based Visualization and Browsing for Semantic Web Services. In *Proc. of the WISE 2007 Workshop on Human-friendly Service Description, Discovery and Matchmaking (Hf-SDDM), Nancy, France.*

[Stollberg and Lara, 2004] Stollberg, M. and Lara, R. (2004). WSMO Use Case "Virtual Travel Agency". Deliverable D3.3, WSMO.

[Stollberg and Norton, 2007] Stollberg, M. and Norton, B. (2007). A Refined Goal Model for Semantic Web Services. In *Proc. of the 2nd International Conference on Internet and Web Applications and Services (ICIW 2007), Mauritius.*

[Stollberg and Rhomberg, 2006] Stollberg, M. and Rhomberg, F. (2006). Survey on Goal-driven Architectures. Technical Report DERI-TR-2006-06-04, DERI.

[Stollberg et al., 2005b] Stollberg, M., Roman, D., Toma, I., Keller, U., Herzog, R., Zugmann, P., and Fensel, D. (2005b). Semantic Web Fred – Automated Goal Resolution on the Semantic Web. In *Proc. of the 38th Hawaiian International Conference on System Science (HICSS-38), Big Island, Hawaii (USA).*

[Studer et al., 2007] Studer, R., Grimm, S., and Abecker, A. (2007). *Semantic Web Services. Concepts, Technologies, and Applications.* Springer.

[Sutcliffe and Suttner, 1998] Sutcliffe, G. and Suttner, C. (1998). The TPTP Problem Library: CNF Release v1.2.1. *Journal of Automated Reasoning*, 21(2):177–203.

[Sycara et al., 2003] Sycara, K., Paolucci, M., Ankolekar, A., and Srinivasan, N. (2003). Automated Discovery, Interaction and Composition of Semantic Web services. *Journal of Web Semantics*, 1(1):27–46.

[Sycara et al., 1999] Sycara, K., Widoff, S., Klusch, M., and Lu, J. (1999). Dynamic Service Matchmaking Among Agents in Open Information Environments. *Journal of the ACM SIGMOD Record, Special Issue on Semantic Interoperability in Global Information Systems*, 28(1):47–53.

[Sycara et al., 2002] Sycara, K., Widoff, S., Klusch, M., and Lu, J. (2002). LARKS: Dynamic Matchmaking Among Heterogeneous Software Agents in Cyberspace. *Autonomous Agents and Multi-Agent Systems*, 5:173–203.

[Tausch et al., 2006] Tausch, B., d'Amato, C., Staab, S., and Fanizzi, N. (2006). Efficient Service Matchmaking using Tree-Structured Clustering. In *Poster at the 5th International Semantic Web Conference (ISWC 2006), Athens, Georgia (USA).*

[Toma et al., 2007] Toma, I., Roman, D., Fensel, D., Sapkota, B., and Gómez, J. M. (2007). A Multi-criteria Service Ranking Approach Based on Non-FunctionalProperties Rules Evaluation. In *Proc. of the Fifth International Conference on Service-Oriented Computing (ICSOC 2007), Vienna, Austria, September 17-20, 2007.*

[Traverso and Pistore, 2004] Traverso, P. and Pistore, M. (2004). Automatic Composition of Semantic Web Services into Executable Processes. In *Proc. of the 3rd International Semantic Web Conference (ISWC 2004), Hiroshima, Japan.*

[Tsarkov et al., 2004] Tsarkov, D., Riazanov, A., Bechhofer, S., and Horrocks, I. (2004). Using Vampire to Reason with OWL. In *Proc. of the 3rd International Semantic Web Conference (ISWC 2004), Hiroshima, Japan.*

[Turing, 1950] Turing, A. M. (1950). Computing Machinery and Intelligence. *Mind*, 49:433–460.

[Vedamuthu et al., 2007] Vedamuthu, A. S., Orchard, D., Hirsch, F., Hondo, M., Yendluri, P. Boubez, T., and Yalçinalp, Ü. (2007). Web Services Policy 1.5 – Framework. Recommendation 4 September 2007, W3C. online at: `http://www.w3.org/TR/ws-policy`.

[Verma et al., 2005] Verma, K., Sivashanmugam, K., Sheth, A., Patil, A., Oundhakar, S., and Miller, J. (2005). METEOR-S WSDI: A Scalable P2P Infrastructure of Registries for Semantic Publication and Discovery of Web Services. *Journal of Information Technology and Management*, 6(1):17–39. Special Issue on Universal Global Integration.

[Vitvar et al., 2007a] Vitvar, T., Kopecky, J., Zaremba, M., and Fensel, D. (2007a). WSMO-Lite: Lightweight Descriptions of Services on the Web. In *Proc. of 5th IEEE European Conference on Web Services (ECOWS), November, 2007, Halle, Germany.*

[Vitvar et al., 2007b] Vitvar, T., Zaremba, M., and Moran, M. (2007b). Dynamic Service Discovery through Meta-Interactions with Service Providers. In *Proc. of the 4th European Semantic Web Conference (ESWC 2007), Innsbruck, Austria.*

[Vu et al., 2005] Vu, L.-H., Hauswirth, M., and Aberer, K. (2005). QoS-Based Service Selection and Ranking with Trust and Reputation Management. In *Proc. of the OTM Confederated International Conferences CoopIS, DOA, and ODBASE 2005, Cyprus.*

[Wache et al., 2004] Wache, H., Serafini, L., and García-Castro, R. (2004). Survey of Scalability Techniques for Reasoning with Ontologies. Deliverable D2.1.1, Knowledge Web Project (FP6-507482).

[Wang et al., 2006] Wang, X., Vitvar, T., Kerrigan, M., and Toma, I. (2006). A QoS-aware Selection Model for Semantic Web Services. In *Proc. of the 4th International Conference on Service Oriented Computing (ICSOC), December, 2006, Chicago, USA.*

[Weber et al., 2007] Weber, I., Hoffmann, J., Mendling, J., and Nitzsche, J. (2007). Towards a Methodology for Semantic Business Process Modeling and Configuration. In *Proc. of the 2nd International Workshop on Business Oriented Aspects concerning Semantics and Methodologies in Service-oriented Computing (SeMSoC 2007), Vienna, Austria.*

[Weerawarana et al., 2005] Weerawarana, S., Curbera, F., Leymann, F., Storey, T., and Ferguson, D. F. (2005). *Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging, and More.* Prentice Hall PTR.

[Wessels, 2001] Wessels, D. (2001). *Web Caching.* O'Reilly & Associates Inc.

[Wetzstein et al., 2007] Wetzstein, B., Ma, Z., Filipowska, A., Kaczmarek, M., Bhiri, S., Losada, S., Lopez-Cob, J.-M., and Cicurel, L. (2007). Semantic Business Process Management: A Lifecycle Based Requirements Analysis. In *Proc. of the Workshop on Semantic Business Process and Product Lifecycle Management (SBPM-2007) at the ESWC 2007, Innsbruck, Austria*.

[Wilson and Keil, 1999] Wilson, R. A. and Keil, F. C. (1999). *The MIT Encyclopedia of the Cognitive Sciences*. MIT Press, Cambridge, MA (USA).

[Wooldridge and Rao, 1999] Wooldridge, M. and Rao, A. (1999). *Foundations of Rational Agency*. Kluwer Academic Publishers.

[Wu et al., 2003] Wu, D., Parsia, B., E., S., Hendler, J., and Nau, D. (2003). Automating DAML-S Web Services Composition Using SHOP2. In *Proceedings of 2nd International Semantic Web Conference (ISWC 2003), Sanibel Island, Florida*.

[Zaremba and Bussler, 2005] Zaremba, M. and Bussler, C. (2005). Towards Dynamic Execution Semantics in Semantic Web Services. In *Proceedings of the WWW 2005 Workshop on Web Service Semantics: Towards Dynamic Business Integration*.

[Zaremba et al., 2005] Zaremba, M., Moran, M., and Haselwanter, T. (2005). WSMX Architecture. Final Working Draft D13.4, WSMX. 13 June 2005, online: `www.wsmo.org/TR/d13/d13.4/`.

[Zaremba et al., 2006] Zaremba, M., Vitvar, T., Moran, M., and Haselwanter, T. (2006). WSMX Discovery for the SWS Challenge. In *Proc. of the Semantic Web Services Challenge Phase III Workshop at the ISWC 2006, Athens, Georgia, USA*.

[Zaremski and Wing, 1997] Zaremski, A. M. and Wing, J. M. (1997). Specification Matching of Software Components. *ACM Transactions on Software Engineering and Methodology*, 6(4):333–369.

Appendix A

Appendix for Chapter 4

A.1 Proof for Proposition 4.1

Proposition 4.1 in Section 4.2.2 defines the formal meaning of the representation of a functional description by the FOL structure $\mathcal{D}_{FOL} = (\Sigma^\star, \Omega^\star, IN, \phi^{\mathcal{D}})$ where $\phi^{\mathcal{D}}$ is a single FOL formula of the form $[\phi^{pre}]_{\Sigma_D \rightarrow \Sigma_D^{pre}} \Rightarrow \phi^{\mathit{eff}}$. It states that there is a bijection between the logical models of $\mathcal{D}_{FOL}$ and the possible executions of a Web service W with $W \models \mathcal{D}$ such that every $\mathcal{T} \in \{\mathcal{T}\}_W$ is described by a $\Sigma^\star$-interpretation that is a model of $\mathcal{D}_{FOL}$.

A functional description $\mathcal{D} = (\Sigma^\star, \Omega, IN, \phi^{pre}, \phi^{\mathit{eff}})$ describes the overall functionality that is provided by a Web service or requested by a goal, and its formal semantics is defined on the basis of a state-based model where we consider $\mathcal{T} = (s_0, s_m)$ as the abstraction of the actual executions of Web services that are observable as finite sequences of states. The purpose of the representation of functional descriptions by $\mathcal{D}_{FOL}$ is to facilitate reasoning on functional descriptions in terms of conventional model-theoretic semantics. As the basis for this, the following formally defines the correspondence between the state-based model and the first-order logic framework that does not encompass the notion of states.

We commence with the correspondence of an abstract sequence of states $\mathcal{T} = (s_0, s_m)$ as an execution of a Web service and a $\Sigma^\star$-interpretation $\mathcal{I}$ that assigns objects of the universe $\mathcal{U}$ to the non-logical symbols in a formula. We consider $\mathcal{I}$ to semantically formally describe $\mathcal{T} = (s_0, s_m)$ if $\mathcal{I}$ defines the objects that exist at both the start-state s_0 and the end-state s_m. The following defines this by introducing the notion of *semantic similarity,* denoted by $\mathcal{I} \simeq \mathcal{T}$, as the mapping between the state-based and the static formalisms. We here use the surjective function $\omega := s \rightarrow Mod_\Sigma(\Omega, \mathcal{U}) \times \mathcal{P}(\mathcal{U})$ that assigns a $\Sigma^\star$-interpretation to a particular state s of the world as defined in [Keller et al., 2006b].

Definition A.1. *Let $\mathcal{T} = (s_0, s_m)$ be the abstraction of a finite sequence of states with the start-state s_0 and the end-state s_m. Let $\Sigma^\star = \Sigma_S \cup \Sigma_D \cup \Sigma_D^{pre}$ be a signature over first-order logic that consists of static symbols Σ_S, dynamic symbols Σ_D, and their pre-variants Σ_D^{pre}. Let $\mathcal{I}$ be a $\Sigma^\star$-interpretation that assigns objects of the universe $\mathcal{U}$, and let $\omega : \mathcal{S} \rightarrow Mod_\Sigma(\Omega,\mathcal{U}) \times \mathcal{P}(\mathcal{U})$ be a surjective function that assigns such a $\Sigma^\star$-interpretation to a state s that is observable in the world.*

We define the semantic description of $\mathcal{T} = (s_0, s_m)$ by a $\Sigma^\star$-interpretation $\mathcal{I}$ as

$$
\begin{aligned}
\mathcal{I} \simeq \mathcal{T} \quad \textit{iff.} \quad & \textit{(i)} \quad \textit{for all predicate symbols } \alpha \in \Sigma^\star \textit{ with the arity } n \textit{ holds} \\
& \qquad \forall ?x_1, \ldots, ?x_n \in \mathcal{U}.\ (?x_1, \ldots, ?x_n) \in (\omega(s_0) \cup \omega(s_m))\langle\alpha\rangle \\
& \qquad\quad \Leftrightarrow (?x_1, \ldots, ?x_n) \in \mathcal{I}\langle\alpha\rangle. \\
& \textit{(ii)} \quad \textit{for all function symbols } f \in \Sigma^\star \textit{ with the arity } n \textit{ holds} \\
& \qquad \forall ?x_1, \ldots, ?x_n, ?y \in \mathcal{U}.\ (\omega(s_0) \cup \omega(s_m))\langle f(?x_1, \ldots, ?x_n) = ?y\rangle \\
& \qquad\quad \Leftrightarrow \in \mathcal{I}\langle f(?x_1, \ldots, ?x_n) = ?y\rangle.
\end{aligned}
$$

This states that a $\Sigma^\star$-interpretation $\mathcal{I}$ formally describes an abstract sequence of states $\mathcal{T} = (s_0, s_m)$ if $\mathcal{I}$ assigns all predicate and function symbols to the same objects that exist in the start-state s_0 and in the end-state s_m. On this basis, we can now define the conditions under which a FOL formula ϕ properly represents a functional description $\mathcal{D}$ whose formal semantics are defined in a state-based model. Essentially, this is given if there is a one-to-one correspondence between the models of ϕ and the executions of a Web service W that provides the functionality described by $\mathcal{D}$. The following defines this by extending the notion of *semantic simulation* to a mapping between a FOL formula ϕ and a functional description $\mathcal{D}$, denoted by $\phi \simeq \mathcal{D}$. We recall that every $\mathcal{T} \in \{\mathcal{T}\}_W$ is unique (*cf.* Definition 4.2), and that $W \models \mathcal{D}$ is given if for every $\mathcal{T} \in \{\mathcal{T}\}_W$ it holds that if $s_0 \models \phi^{pre}$ then $s_m \models \phi^{eff}$ (*cf.* Definition 4.5).

Definition A.2. *Let W be a Web service and let $\{\mathcal{T}\}_W$ be the set of all its abstract executions. Let $\mathcal{D} = (\Sigma^\star, \Omega, IN, \phi^{pre}, \phi^{eff})$ be a functional description, and let $\beta : (?i_1, \ldots, ?i_n) \rightarrow \mathcal{U}$ be an input binding for $\mathcal{D}$. Let ϕ be a first-order logic formula that is defined over $\Sigma^\star$.*

We define the semantic simulation of $\mathcal{D}$ by a first-order logic formula ϕ as

$$
\begin{aligned}
\phi \simeq \mathcal{D} \quad \textit{iff.} \quad & \textit{for all Web Services } W \textit{ with } W \models \mathcal{D} \textit{ under all input bindings } \beta\textit{:} \\
& \textit{(i)} \quad \textit{for all } \mathcal{T} \in \{\mathcal{T}\}_W \textit{ there is a } \Sigma^\star\textit{-interpretations } \mathcal{I} \textit{ such that} \\
& \qquad \textit{if } \mathcal{I} \simeq \mathcal{T} \textit{ then } \mathcal{I} \models \phi\textit{, and} \\
& \textit{(ii)} \quad \textit{for all } \Sigma\textit{-interpretations } \mathcal{I} \textit{ there is a } \mathcal{T} \in \{\mathcal{T}\}_W \textit{ such that} \\
& \qquad \textit{if } \mathcal{I} \models \phi \textit{ then } \mathcal{I} \simeq \mathcal{T}.
\end{aligned}
$$

This provides the correctness criterion for a first-order logic formula to properly represent a functional description $\mathcal{D}$ while maintaining its formal meaning. We now can use this to show that $\mathcal{D}_{FOL}$ is a FOL structure that satisfies these conditions. Recalling from Definition 4.6, $\mathcal{D}_{FOL} = (\Sigma^\star, \Omega^\star, IN, \phi^{\mathcal{D}})$ is a representation of $\mathcal{D}$ that is defined over the same signature $\Sigma^\star$ and the same background ontology Ω extended by the pre-variants of dynamic symbols, and it defines the same input variables IN that occur as free variables in both the precondition ϕ^{pre} and in the effect ϕ^{eff}. The only difference is that $\phi^{\mathcal{D}}$ is a single FOL formula of the form $[\phi^{pre}]_{\Sigma_D \to \Sigma_D^{pre}} \Rightarrow \phi^{eff}$ wherein the renaming function $[\phi]_{\Sigma_D \to \Sigma_D^{pre}}$ replaces the dynamic symbols that occur in the precondition ϕ^{pre} by their pre-variants.

Proposition A.1. *Let $\mathcal{D} = (\Sigma^\star, \Omega, IN, \phi^{pre}, \phi^{eff})$ be a functional description, and let $\mathcal{D}_{FOL} = (\Sigma^\star, \Omega^\star, IN, \phi^{\mathcal{D}})$ be a FOL structure that represents $\mathcal{D}$.*

It holds that $\mathcal{D}_{FOL} \simeq \mathcal{D}$

Proof. We need to show that clauses (i) and (ii) of Definition A.2 hold for $\mathcal{D}_{FOL}$ and $\mathcal{D}$. For this, we consider the executions of a Web service W in a not changing world under three different input bindings $\beta_1, \beta_2, \beta_3$ which cover all relevant cases:

$\langle 1 \rangle$ for $\mathcal{T}(\beta_1) = (s_0, s_m)$: $s_0, \beta_1 \models_{\mathcal{A}} \phi^{pre}$ and $s_m, \beta_1 \models_{\mathcal{A}} \phi^{eff}$

$\langle 2 \rangle$ for $\mathcal{T}(\beta_2) = (s_0, s_m)$: $s_0, \beta_2 \not\models_{\mathcal{A}} \phi^{pre}$

$\langle 3 \rangle$ for $\mathcal{T}(\beta_3) = (s_0, s_m)$: $s_0, \beta_3 \models_{\mathcal{A}} \phi^{pre}$ and $s_m, \beta_3 \not\models_{\mathcal{A}} \phi^{eff}$.

In case $\langle 1 \rangle$, the execution $\mathcal{T}(\beta_1)$ satisfies the condition for $W \models \mathcal{D}$ because the start-state s_0 satisfies the precondition ϕ^{pre} and the end-state s_m satisfies the effect ϕ^{eff}. There must be a $\Sigma^\star$-interpretation $\mathcal{I}$ under β_1 that is a model of $\mathcal{D}_{FOL}$ such that $\mathcal{I}(\beta_1) \simeq \mathcal{T}(\beta_1)$ because (i) if $\mathcal{I}, \beta \models \phi^{pre}$ and $\mathcal{I}, \beta \models \phi^{eff}$ then $\mathcal{I}, \beta \models \phi^{\mathcal{D}}$, and (ii) $\mathcal{D}_{FOL}$ defines exactly the same non-logical symbols in ϕ^{pre} and ϕ^{eff}. The renaming function $[\phi]_{\Sigma_D \to \Sigma_D^{pre}}$ ensures that the dynamic symbols and their pre-variants which occur in $\mathcal{D}$ are specified by distinct symbols; this is merely a symbol substitution, so that if $s_0, \beta \models_{\mathcal{A}} \phi^{pre}$ then also $s_0, \beta \models_{\mathcal{A}} [\phi^{pre}]_{\Sigma_D^{pre} \to \Sigma_D}$. Hence, the same objects that exist in the start-state s_0 and the end-state s_m of $\mathcal{T}(\beta_1)$ must be describable by a $\Sigma^\star$-interpretation $\mathcal{I}(\beta_1)$ such that $\mathcal{I}(\beta_1) \models \phi^{\mathcal{D}}$. Then, it holds that $\mathcal{I}(\beta_1) \simeq \mathcal{T}(\beta_1)$ in accordance to Definition A.1. This proves clause (i) of Definition A.2 hold for $\mathcal{D}_{FOL}$. This correlation holds for all executions of a Web service W with $W \models \mathcal{D}$ which are of type $\mathcal{T}(\beta_1)$. Because each of the abstract executions of a Web service are distinct, for each $\mathcal{T} \in \{\mathcal{T}\}_W$ there must be a different $\Sigma^\star$-interpretation $\mathcal{I}$ that is a model of $\mathcal{D}_{FOL}$ and for which $\mathcal{I} \simeq \mathcal{T}$ holds. Under any input binding β, if there is a $\Sigma^\star$-interpretation $\mathcal{I}(\beta)$ with $\mathcal{I}(\beta) \models \phi^{\mathcal{D}}$ but $\mathcal{I}(\beta) \not\simeq \mathcal{T}(\beta)$, then this $\mathcal{T}$ does not denote an execution of a Web service W with $W \models \mathcal{D}$; the same holds if $\mathcal{I}(\beta) \simeq \mathcal{T}(\beta)$ but $\mathcal{I}(\beta) \not\models \phi^{\mathcal{D}}$. This shows that clause (ii) of Definition A.2 holds for $\mathcal{D}_{FOL}$.

In case $\langle 2 \rangle$, the start-state of $\mathcal{T}(\beta_2)$ does not satisfy the precondition defined in $\mathcal{D}$. Formally, $\mathcal{T}(\beta_2)$ satisfies the condition for $W \models \mathcal{D}$, and also a $\Sigma^\star$-interpretation $\mathcal{I}(\beta_2)$ with $\mathcal{I}(\beta_2) \simeq \mathcal{T}(\beta_2)$ is a model of $\mathcal{D}_{FOL}$. However, we can not make any statement about the end-state s_m of $\mathcal{T}(\beta_2)$ because either $s_m, \beta_2 \models_{\mathcal{A}} \phi^{\mathit{eff}}$ or $s_m, \beta_2 \not\models_{\mathcal{A}} \phi^{\mathit{eff}}$ can hold. Because of the same reasons as above, it holds for both cases that if $\mathcal{I}(\beta_2) \simeq \mathcal{T}(\beta_2)$ then also $\mathcal{I}(\beta_2) \models \phi^{\mathcal{D}}$ and vice versa. Hence, $\mathcal{D}_{FOL} \simeq \mathcal{D}$ holds for executions of Web services of the type of $\mathcal{T}(\beta_2)$. However, such a $\mathcal{T}$ is not considered as a possible solution for a goal because the goal instantiation condition $GI(G, \beta) \models G$ requires that the precondition of $\mathcal{D}_G$ is satisfied.

Finally, case $\langle 3 \rangle$ handles the situation where the precondition is satisfied but the effect is not. Here, $\mathcal{T}(\beta_3)$ does not satisfy the condition for $W \models \mathcal{D}$, and also a $\Sigma^\star$-interpretation $\mathcal{I}(\beta_3)$ with $\mathcal{I}(\beta_3) \simeq \mathcal{T}(\beta_3)$ is a not a model of $\mathcal{D}_{FOL}$ because if $\mathcal{I}\beta \models \phi^{\mathit{pre}}$ and $\mathcal{I}\beta \not\models \phi^{\mathit{eff}}$ then $\mathcal{I}\beta \not\models \phi^{\mathcal{D}}$. This substantiates the proof as the negative case: if a $\mathcal{T} \in \{\mathcal{T}\}_W$ can not be simulated by a model of $\mathcal{D}_{FOL}$ or if a model of $\mathcal{D}_{FOL}$ does not correspond to a $\mathcal{T} \in \{\mathcal{T}\}_W$, then W does not provide the functionality described by $\mathcal{D}$.

□

In order to avoid the situations where it is not possible to make a clear statement about the behavior of a Web service, we can define a stronger representation of a functional descriptions as $\mathcal{D}_{FOL}+ = (\Sigma^\star, \Omega^\star, \mathit{IN}, \phi^{\mathcal{D}}_+)$ where $\phi^{\mathcal{D}}_+$ is a FOL formula of the form $[\phi^{\mathit{pre}}]_{\Sigma_D \rightarrow \Sigma_D^{\mathit{pre}}} \wedge \phi^{\mathit{eff}}$. This defines a logical conjunction between the precondition and the effect formulae instead of the implication defined in $\mathcal{D}_{FOL}$. We refer to this as the *conjunctive semantics* of a functional description. Here, only those sequences of states $\mathcal{T} = (s_0, s_m)$ for which both the precondition and the effect are satisfied correspond to models of $\mathcal{D}_{FOL}+$. This means that only those $\mathcal{T} \in \{\mathcal{T}\}_W$ of the type discussed in case $\langle 1 \rangle$ of the above proof are considered, while those discussed in case $\langle 2 \rangle$ are neglected because they can not be described by a model of $\mathcal{D}_{FOL}+$. In consequence, the FOL representation $\mathcal{D}_{FOL}+$ of a functional descriptions logically entails $\mathcal{D}_{FOL}$ as the one that we have discussed above.

Proposition A.2. *Let $\mathcal{D} = (\Sigma^\star, \Omega, \mathit{IN}, \phi^{\mathit{pre}}, \phi^{\mathit{eff}})$ be a functional description. Let $\mathcal{D}_{FOL} = (\Sigma^\star, \Omega^\star, \mathit{IN}, \phi^{\mathcal{D}})$ be the representation of $\mathcal{D}$ where $\phi^{\mathcal{D}} := [\phi^{\mathit{pre}}]_{\Sigma_D \rightarrow \Sigma_D^{\mathit{pre}}} \Rightarrow \phi^{\mathit{eff}}$, and let $\mathcal{D}_{FOL}+ = (\Sigma^\star, \Omega^\star, \mathit{IN}, \phi^{\mathcal{D}}_+)$ be the representation of $\mathcal{D}$ where $\phi^{\mathcal{D}}_+$ is a FOL formula of the form $[\phi^{\mathit{pre}}]_{\Sigma_D \rightarrow \Sigma_D^{\mathit{pre}}} \wedge \phi^{\mathit{eff}}$.*

It holds that $\mathcal{D}_{FOL}+ \models \mathcal{D}_{FOL}$.

A.2 Proof for Proposition 4.3

Proposition 4.3 in Section 4.3.1 defines constraints on the modeling of functional descriptions and the used domain ontologies in order to ensure the decidability of the semantic matchmaking techniques. In particular, it ensures that the proof obligations for the five matching degrees defined in Table 4.2 remain in the Bernays–Schönfinkel fragment of FOL and thus are decidable. For this, it is required that:

(i) $\Omega^\star$ as well as $\phi^{pre}, \phi^{\mathit{eff}} \in \mathcal{D}_G, \mathcal{D}_W$ do not contain any function symbols

(ii) all formulae $\phi \in \Omega^\star$ have a $\exists * \forall *$ quantifier prefix in prenex normal form

(iii) $\phi^{\mathcal{D}_G}$ and $\phi^{\mathcal{D}_W}$ do not have any existential quantifier in prenex normal form.

Proof. We need to show that the three constraints ensure that the matchmaking conditions for all five matching degrees remain in the Bernays–Schönfinkel class, which covers all FOL formulae that (1) do not contain function symbols and (2) have a $\exists * \forall *$ quantifier prefix when written in prenex normal form.

We commence with the plugin$(\mathcal{D}_G, \mathcal{D}_W)$ degree, whose condition is defined as $\Omega^\star \models \forall\beta.\ \phi^{\mathcal{D}_G} \Rightarrow \phi^{\mathcal{D}_W}$ (*cf.* Table 4.2). We can proof this by showing the unsatisfiability of the FOL formula $\bigwedge \Omega^\star \cup \neg(\forall\beta, \forall\overrightarrow{v_{other}}.\ \phi^{\mathcal{D}_G} \Rightarrow \phi^{\mathcal{D}_W})$ where $\overrightarrow{v_{other}} \in V_{free} \backslash (IN_G \cup IN_W)$. Clause (ii) requires that all formula in the domain ontology have a $\exists * \forall *$ quantifier prefix when written in prenex normal form, so that in combination with clause (i) every formula in the domain ontology Ω can be expressed in the Bernays–Schönfinkel fragment of FOL and thus $\bigwedge \Omega^\star \in FOL_{BS}$ is given. For the remaining part, clause (iii) requires that both functional descriptions $\mathcal{D}_G$ and $\mathcal{D}_W$ have only universal quantifiers when their representation as a single FOL formula with $\phi^{\mathcal{D}} = [\phi^{pre}]_{\Sigma_D \rightarrow \Sigma_D^{pre}} \Rightarrow \phi^{\mathit{eff}}$ is written in prenex normal form. Let us denote this by $\forall\overrightarrow{g}.\ \phi^{\mathcal{D}_G}(\overrightarrow{g})$, and $\forall\delta(\overrightarrow{w}).\ \phi^{\mathcal{D}_W}(\delta(\overrightarrow{w}))$ where $\delta(\overrightarrow{w})$ is a variable substitution to ensure that distinct objects are considered for the goal template and the Web service description. We then can transform the right hand side of the proof obligation as follows:

$$
\begin{aligned}
& \neg(\forall\beta, \forall\overrightarrow{v_{other}}.\ (\forall\overrightarrow{g}.\ \phi^{\mathcal{D}_G}(\overrightarrow{g}) \Rightarrow \forall\delta(\overrightarrow{w}).\ \phi^{\mathcal{D}_W}(\delta(\overrightarrow{w}))\ ) && (0)\\
\Leftrightarrow\ & \exists\beta, \exists\overrightarrow{v_{other}}.\ \neg(\forall\overrightarrow{g}.\ \phi^{\mathcal{D}_G}(\overrightarrow{g}) \Rightarrow \forall\delta(\overrightarrow{w}).\ \phi^{\mathcal{D}_W}(\delta(\overrightarrow{w}))\ ) && (1)\\
\Leftrightarrow\ & \exists\beta, \exists\overrightarrow{v_{other}}.\ \neg(\neg\forall\overrightarrow{g}.\ \phi^{\mathcal{D}_G}(\overrightarrow{g}) \vee \forall\delta(\overrightarrow{w}).\ \phi^{\mathcal{D}_W}(\delta(\overrightarrow{w}))\ ) && (2)\\
\Leftrightarrow\ & \exists\beta, \exists\overrightarrow{v_{other}}, \forall\overrightarrow{g}.\ \phi^{\mathcal{D}_G}(\overrightarrow{g}) \wedge \neg(\forall\delta(\overrightarrow{w}).\ \phi^{\mathcal{D}_W}(\delta(\overrightarrow{w}))\ ) && (3)\\
\Leftrightarrow\ & \exists\beta, \exists\overrightarrow{v_{other}}, \forall\overrightarrow{g}.\ \phi^{\mathcal{D}_G}(\overrightarrow{g}) \wedge \exists\delta(\overrightarrow{w}).\ \neg\phi^{\mathcal{D}_W}(\delta(\overrightarrow{w})) && (4)\\
\Leftrightarrow\ & \exists\beta, \exists\overrightarrow{v_{other}}, \forall\overrightarrow{g}, \exists\delta(\overrightarrow{w}).\ \phi^{\mathcal{D}_G}(\overrightarrow{g}) \wedge\ \neg\phi^{\mathcal{D}_W}(\delta(\overrightarrow{w})) && (5)\\
\Leftrightarrow\ & \exists\beta, \exists\overrightarrow{v_{other}}, \exists\delta(\overrightarrow{w}), \forall\overrightarrow{g}.\ \phi^{\mathcal{D}_G}(\overrightarrow{g}) \wedge\ \neg\phi^{\mathcal{D}_W}(\delta(\overrightarrow{w})) && (6)
\end{aligned}
$$

We see that the resulting FOL formula is in the Bernays–Schönfinkel class, because all existential quantifiers precede the universal quantifiers and clause (i) ensures that no function symbols occur in $\phi^{\mathcal{D}_G}$ or in $\phi^{\mathcal{D}_W}$. The first five steps of the transformation follow the conventional rules for moving quantifiers and resolving implications in order to transform a FOL formula into prenex normal form. In the last step, we can change the order of the quantifiers $\forall \overrightarrow{g} \exists \delta(\overrightarrow{w})$ to $\exists \delta(\overrightarrow{w}) \forall \overrightarrow{g}$ because the variable substitution $\delta(\overrightarrow{w})$ ensures that variables which possibly occur as locally bound variables in both $\phi^{\mathcal{D}_G}$ and $\phi^{\mathcal{D}_W}$ are explicitly considered as distinct objects in the proof obligation.

This shows that three constraints on the modeling of the background ontology Ω and the functional descriptions facilitates the evaluation of the condition for the $plugin(\mathcal{D}_G, \mathcal{D}_W)$ matching degree on the basis of a FOL formula that remains in the Bernays–Schönfinkel class and thus is decidable. The same holds analogously for the $subsume(\mathcal{D}_G, \mathcal{D}_W)$ degree that can be evaluated by proving the unsatisfiability of the FOL formula $\bigwedge \Omega^\star \cup \neg(\forall \beta, \forall \overrightarrow{v_{other}}.\ \phi^{\mathcal{D}_W} \Rightarrow \phi^{\mathcal{D}_G})$. Here, $\bigwedge \Omega^\star \in FOL_{BS}$ is given because of the clauses (i) and (ii), and clause (iii) ensures that the transformation of the right hand side to prenex normal formal will result in the formula $\exists \beta, \exists \overrightarrow{v_{other}}, \exists \delta(\overrightarrow{g}), \forall \overrightarrow{w}.\ \phi^{\mathcal{D}_W}(\overrightarrow{w}) \wedge\ \neg \phi^{\mathcal{D}_G}(\delta(\overrightarrow{g})) \in FOL_{BS}$. In consequence, the modeling constraints also ensure that the condition for the $exact(\mathcal{D}_G, \mathcal{D}_W)$ degree is decidable: under the assumption that the functional descriptions are consistent it holds that $exact(\mathcal{D}_G, \mathcal{D}_W) \Leftrightarrow plugin(\mathcal{D}_G, \mathcal{D}_W) \wedge subsume(\mathcal{D}_G, \mathcal{D}_W)$ (*cf.* Proposition 4.2), and if the conditions for both $plugin(\mathcal{D}_G, \mathcal{D}_W)$ and $subsume(\mathcal{D}_G, \mathcal{D}_W)$ are decidable then also their conjunction is decidable. For the overall proof obligation, we need to define three variable substitutions analogous to $\phi^{\mathcal{D}_W}$ as explained above.

The condition for the $intersect(\mathcal{D}_G, \mathcal{D}_W)$ degree requires that $\exists \beta.\ \bigwedge \Omega^\star \wedge \phi^{\mathcal{D}_G} \wedge \phi^{\mathcal{D}_W}$ is satisfiable. Here, the modeling constraints ensure that the prenex normal form of the proof obligation is a FOL formula of the form $\exists \beta, \forall \overrightarrow{v_{other}}, \forall \overrightarrow{g}, \forall \delta(\overrightarrow{w}).\ \bigwedge \Omega^\star \wedge \phi^{\mathcal{D}_G}(\overrightarrow{g}) \wedge \phi^{\mathcal{D}_W} \delta(\overrightarrow{w})$, which is in the Bernays–Schönfinkel class and therewith decidable. Analogously, the condition for the $disjoint(\mathcal{D}_G, \mathcal{D}_W)$ degree can be evaluated by the proof obligation $\forall \beta, \forall \overrightarrow{v_{other}}, \forall \overrightarrow{g}, \forall \delta(\overrightarrow{w}).\ \neg(\bigwedge \Omega^\star \wedge \phi^{\mathcal{D}_G}(\overrightarrow{g}) \wedge \phi^{\mathcal{D}_W} \delta(\overrightarrow{w})\ )$ whose prenex normal form only contains universal quantifiers, thus also is in Bernays–Schönfinkel class and therewith is decidable. This completes the proof. □

A.3 Proof for Theorem 4.1

Theorem 4.1 in Section 4.3.2 defines the integrated matchmaking conditions for Web service discovery on the goal instance level in our two-phase discovery framework. It states that a Web service W is usable under functional aspects to solve a goal instance $GI(G, \beta)$ with $GI(G, \beta) \models G$ under consideration of the matching degree between W and the corresponding goal template G if and only if:

(i)	$\texttt{exact}(\mathcal{D}_G, \mathcal{D}_W)$	or	
(ii)	$\texttt{plugin}(\mathcal{D}_G, \mathcal{D}_W)$	or	
(iii)	$\texttt{subsume}(\mathcal{D}_G, \mathcal{D}_W)$	and	$\bigwedge \Omega^\star \wedge [\phi^{\mathcal{D}_W}]_\beta$ is satisfiable, or
(iv)	$\texttt{intersect}(\mathcal{D}_G, \mathcal{D}_W)$	and	$\bigwedge \Omega^\star \wedge [\phi^{\mathcal{D}_G}]_\beta \wedge [\phi^{\mathcal{D}_W}]_\beta$ is satisfiable.

Proof. We commence with clause (iv) which defines the conditions on the functional usability of W under the *intersect* matching degree between W and the corresponding goal template G. This is the weakest degree under which the basic condition for a match on the goal template level is satisfied (*cf.* clause (i) of Definition 4.3), and it requires the complete matching on the goal instance level from Definition 4.8 to be performed at runtime. The other matching degrees under which $match(G, W)$ is given can be understood as specializations of the *intersect* degree with respect to their formal relations, *cf.* Proposition 4.2. We also recall that $GI(G, \beta) \models G$ is only given if the input binding β defined in the goal instance $GI(G, \beta)$ satisfies the functional description $\mathcal{D}_G$ of the corresponding goal template G (*cf.* Definition 4.7), and that $[\phi^{\mathcal{D}}]_\beta$ is the β-instantiation of a functional description $\mathcal{D}$ wherein every occurrence of each *IN*-variable is replaced by the concrete value assignment defined in the input binding β (*cf.* Definition 4.8).

The *intersect*$(\mathcal{D}_G, \mathcal{D}_W)$ is given if there is an input binding β under which $\bigwedge \Omega^\star \wedge \phi^{\mathcal{D}_G} \wedge \phi^{\mathcal{D}_W}$ is satisfiable, so that there is at least one $\mathcal{T}_1 \in (\{\mathcal{T}\}_G \cap \{\mathcal{T}\}_W)$ but there can also be a $\mathcal{T}_2$ where $\mathcal{T}_2 \in \{\mathcal{T}\}_G$ but $\mathcal{T}_2 \notin \{\mathcal{T}\}_W$ as well as a $\mathcal{T}_3$ where $\mathcal{T}_3 \in \{\mathcal{T}\}_W$ but $\mathcal{T}_3 \notin \{\mathcal{T}\}_G$. Thus, if *intersect*$(\mathcal{D}_G, \mathcal{D}_W)$ then W can provide at least one solution for G. For W to be suitable for a specific goal instance $GI(G, \beta)$ with $GI(G, \beta) \models G$, it must hold that $\langle 1 \rangle$ an execution of W of type $\mathcal{T}_1$ can be triggered when W is invoked with the input binding β defined in $GI(G, \beta)$, and $\langle 2 \rangle$ that this execution is a solution for the corresponding goal template G under β. This is given if the union of the formulae $\Omega^\star \cup \{[\phi^{\mathcal{D}_G}]_\beta, [\phi^{\mathcal{D}_W}]_\beta\}$ is satisfiable, i.e. if the basic matchmaking condition for the goal instance level from Definition 4.8 is satisfied. A $\Sigma^\star$-interpretation $\mathcal{I}$ for which this holds represents a $\mathcal{T}$ which is a solution for $GI(G, \beta)$ and can be provided by W if it is invoked with β (cf. Proposition 4.1). If such a common model does not exists, then there does not exists any $\mathcal{T} \in (\{\mathcal{T}\}_{\mathcal{G}(\beta_\mathcal{G})} \cap \{\mathcal{T}\}_{W(\beta_W)})$, and thus W is not usable for solving $GI(G, \beta)$ (*cf* Definition 4.3).

We now show clause (iii). The condition for *subsume*$(\mathcal{D}_G, \mathcal{D}_W)$ is defined as $\Omega^\star \models \forall\beta.\ \phi^{\mathcal{D}_G} \Leftarrow \phi^{\mathcal{D}_W}$ so that $\{\mathcal{T}\}_G \supseteq \{\mathcal{T}\}_W$ and for all input bindings β holds that if $\mathcal{T}(\beta) \in \{\mathcal{T}\}_W$ then $\mathcal{T}(\beta) \in \{\mathcal{T}\}_G$. This means that every possible execution of W is a solution for the goal template G. However, for W to be suitable for a goal instance $GI(G, \beta)$ with $GI(G, \beta) \models G$ it must hold that W can actually be invoked with the input binding β defined in $GI(G, \beta)$ – otherwise the possible solutions for $GI(G, \beta)$ are a subset of those of G which can not be provided by W. To ensure this, it must hold that $[\phi^{\mathcal{D}_W}]_\beta$ is satisfiable under consideration of the background ontology $\Omega^\star$. If this is given, then every execution of W when invoked with β is a solution of $GI(G, \beta)$, i.e. $\mathcal{T} \in \{\mathcal{T}\}_{GI(G,\beta)} \Leftrightarrow \mathcal{T} \in \{\mathcal{T}\}_{W(\beta)}$, because it holds for all $\mathcal{T} \in \{\mathcal{T}\}_{W(\beta)} \subset \{\mathcal{T}\}_W \subseteq \{\mathcal{T}\}_G$. If $[\phi^{\mathcal{D}_W}]_\beta$ is not satisfiable, then we can not make any statement of the resulting execution of W, and hence we do not consider W to be suitable for solving $GI(G, \beta)$.

For clause (ii), the condition for *plugin*$(\mathcal{D}_G, \mathcal{D}_W)$ is defined as $\Omega \models \forall\beta.\ \phi^{\mathcal{D}_G} \Rightarrow \phi^{\mathcal{D}_W}$. As the opposite of the *subsume* degree, this means that $\{\mathcal{T}\}_G \subseteq \{\mathcal{T}\}_W$ and for all input bindings β holds that $\mathcal{T}(\beta) \in \{\mathcal{T}\}_G \Rightarrow \mathcal{T}(\beta) \in \{\mathcal{T}\}_W$, so that every solution for G can be provided by W. Under this degree, W is usable for every possible goal instance $GI(G, \beta)$ with $GI(G, \beta) \models G$ because (1) $\{\mathcal{T}\}_{GI(G,\beta)} \subset \{\mathcal{T}\}_G \subseteq \{\mathcal{T}\}_W$, and (2) for each input binding β holds that if $\mathcal{T} \in \{\mathcal{T}\}_{GI(G,\beta)}$ then $\mathcal{T} \in \{\mathcal{T}\}_{W(\beta)}$. We thus do not need to perform any additional matchmaking at runtime for Web services that are usable under the *plugin* degree for the corresponding goal template of a goal instance. The same holds for the *exact* matching degree between whose condition is defined as $\Omega^\star \models \forall\beta.\ \phi^{\mathcal{D}_G} \Leftrightarrow \phi^{\mathcal{D}_W}$. Here, it holds that $\{\mathcal{T}\}_{GI(G,\beta)} \subset \{\mathcal{T}\}_G$ and $\{\mathcal{T}\}_G = \{\mathcal{T}\}_W$, and for each β holds if $\mathcal{T} \in \{\mathcal{T}\}_{GI(G,\beta)}$ then $\mathcal{T} \in \{\mathcal{T}\}_{W(\beta)}$. This proves clause (i) of the theorem.

The *disjoint*$(\mathcal{D}_G, \mathcal{D}_W)$ degree is given if $\exists\beta.\ \bigwedge \Omega^\star \wedge \phi^{\mathcal{D}_G} \wedge \phi^{\mathcal{D}_W}$ is unsatisfiable, which means that there does not exists any possible execution of W that is a solution for the goal template G. In consequence, there also can not exist any execution of W that is a solution for a goal instance $GI(G, \beta)$ with $GI(G, \beta) \models G$ because it then holds that $\{\mathcal{T}\}_{GI(G,\beta)} \subset \{\mathcal{T}\}_G$. Because $\neg intersect(\mathcal{D}_G, \mathcal{D}_W) \Leftrightarrow disjoint(\mathcal{D}_G, \mathcal{D}_W)$ and thus also $disjoint(\mathcal{D}_G, \mathcal{D}_W) => \neg(exact(\mathcal{D}_G, \mathcal{D}_W) \vee plugin(\mathcal{D}_G, \mathcal{D}_W) \vee subsume(\mathcal{D}_G, \mathcal{D}_W))$, *cf.* Proposition 4.2, clauses (i) - (iv) define all possible situations wherein W is usable for solving a goal instance $GI(G, \beta)$. This completes the proof.

□

Appendix B

Appendix for Chapter 5

B.1 Proof for Theorem 5.1

Theorem 5.1 in Section 5.2.1 defines the inference rules of the form $d(G_i, G_j) \wedge d(G_i, W) \Rightarrow d(G_j, W)$ for all possible situations that can occur between between two semantically similar goal templates and the usability degree for Web services. These rules provide the logical basis for the SDC technique, in particular for managing and exploiting the SDC graph.

The inference rules result from the formal definition of the similarity degrees $d(G_i, G_j)$ between two goal templates G_i, G_j, and the matching degrees $d(G, W)$ that denote the usability of a Web Service W for a goal template G. The conditions for both types of matching degrees are defined such they consider the relationship of the individual executions of a Web service, respectively the individual solutions for goals (see Section 4.3.1). However, for proving Theorem 5.1 it is sufficient to merely consider the set-theoretic relations. For this, the following table shows the definitions of the matching degrees between two functional descriptions $\mathcal{D}_1, \mathcal{D}_2$ and their meaning in terms of set-theoretic criteria; we define the *plugin* and the *subsume* degree to denote true subset relationships because we always use the highest possible degree in order to precisely denote the actual relationship.

Degree	**Definition**	**Meaning**
exact$(\mathcal{D}_1, \mathcal{D}_2)$	$\Omega^\star \models \forall\beta.\ \phi^{\mathcal{D}_1} \Leftrightarrow \phi^{\mathcal{D}_2}$	$\{\mathcal{T}\}_{\mathcal{D}_1} = \{\mathcal{T}\}_{\mathcal{D}_2}$
plugin$(\mathcal{D}_1, \mathcal{D}_2)$	$\Omega^\star \models \forall\beta.\ \phi^{\mathcal{D}_1} \Rightarrow \phi^{\mathcal{D}_2}$	$\{\mathcal{T}\}_{\mathcal{D}_1} \subset \{\mathcal{T}\}_{\mathcal{D}_2}$
subsume$(\mathcal{D}_1, \mathcal{D}_2)$	$\Omega^\star \models \forall\beta.\ \phi^{\mathcal{D}_1} \Leftarrow \phi^{\mathcal{D}_2}$	$\{\mathcal{T}\}_{\mathcal{D}_1} \supset \{\mathcal{T}\}_{\mathcal{D}_2}$
intersect$(\mathcal{D}_1, \mathcal{D}_2)$	$\exists\beta.\ \bigwedge \Omega^\star \wedge \phi^{\mathcal{D}_G} \wedge \phi^{\mathcal{D}_W}$ is satisfiable	$\{\mathcal{T}\}_{\mathcal{D}_1} \cap \{\mathcal{T}\}_{\mathcal{D}_2} \neq \emptyset$
disjoint$(\mathcal{D}_1, \mathcal{D}_2)$	$\exists\beta.\ \bigwedge \Omega^\star \wedge \phi^{\mathcal{D}_G} \wedge \phi^{\mathcal{D}_W}$ is unsatisfiable	$\{\mathcal{T}\}_{\mathcal{D}_1} \cap \{\mathcal{T}\}_{\mathcal{D}_2} = \emptyset$

Proof. We commence with the similarity degree $exact(G_1, G_2)$. This denotes that $\{\mathcal{T}\}_{G_1} = \{\mathcal{T}\}_{G_2}$. Here, the usability of every Web service W is the same for G_1 and G_2: if $exact(G_1, W)$, then $\{\mathcal{T}\}_{G_1} = \{\mathcal{T}\}_W = \{\mathcal{T}\}_{G_2}$ so that also $exact(G_2, W)$, if $plugin(G_1, W)$ then $\{\mathcal{T}\}_{G_1} \subset \{\mathcal{T}\}_W$ and hence also $\{\mathcal{T}\}_{G_2} \subset \{\mathcal{T}\}_W$ so that $plugin(G_2, W)$; the same trivially holds for $subsume(G_1, W)$, $intersect(G_1, W)$, and $disjoint(G_1, W)$. This proves the rules 1.1 – 1.5.

We now discuss the inference rules for the similarity degree $plugin(G_1, G_2)$. This denotes the situation where $\{\mathcal{T}\}_{G_1} \subset \{\mathcal{T}\}_{G_2}$. For the usability of a Web service W for G_2 it holds:
$\langle 1 \rangle$ every Web service W that is usable for G_1 is also usable for G_2 because if $\exists \mathcal{T}.\ \mathcal{T} \in (\{\mathcal{T}\}_{G_1} \cap \{\mathcal{T}\}_W)$ then also $\exists \mathcal{T}.\ \mathcal{T} \in (\{\mathcal{T}\}_{G_2} \cap \{\mathcal{T}\}_W)$, i.e. whenever the usability degree of W for G_1 is either *exact*, *plugin*, *subsume*, or *intersect* so that the condition for $match(G_1, W)$ is satisfied then also $match(G_2, W)$ holds because $\{\mathcal{T}\}_{G_1} \subset \{\mathcal{T}\}_{G_2}$.
$\langle 2 \rangle$ $exact(G_1, W)$ defines that $\{\mathcal{T}\}_{G_1} = \{\mathcal{T}\}_W$. Because of $\{\mathcal{T}\}_{G_1} \subset \{\mathcal{T}\}_{G_2}$ it holds that $\{\mathcal{T}\}_{G_2} \supset \{\mathcal{T}\}_W$, and thus $subsume(G_2, W)$ is the only possible usability degree of W for G_2. There can not be any $\mathcal{T} \in \{\mathcal{T}\}_W$ and $\mathcal{T} \notin \{\mathcal{T}\}_{G_2}$ such that $intersect(G_2, W)$ can not hold; also, $disjoint(G_2, W)$ can not hold because of $\langle 1 \rangle$. This proves rule 2.1.
$\langle 3 \rangle$ $subsume(G_1, W)$ defines that $\{\mathcal{T}\}_{G_1} \supset \{\mathcal{T}\}_W$. Because then $\{\mathcal{T}\}_W \subset \{\mathcal{T}\}_{G_1} \subset \{\mathcal{T}\}_{G_2}$, the only possible usability degree of W for G_2 is $subsume(G_2, W)$; this shows rule 2.2.
$\langle 4 \rangle$ $plugin(G_1, W)$ defines that $\{\mathcal{T}\}_{G_1} \subset \{\mathcal{T}\}_W$. Here, the usability degree of W for G_2 can be either *exact* if $\{\mathcal{T}\}_{G_2} = \{\mathcal{T}\}_W$, *plugin* if $\{\mathcal{T}\}_{G_2} \subset \{\mathcal{T}\}_W$, *subsume* if $\{\mathcal{T}\}_{G_2} \supset \{\mathcal{T}\}_W$, or *intersect* if there exists a $\mathcal{T}$ such that $\mathcal{T} \in \{\mathcal{T}\}_W$ and $\mathcal{T} \notin \{\mathcal{T}\}_{G_2}$; it can not be *disjoint* because of $\langle 1 \rangle$. This proves the rules under 2.3.
$\langle 5 \rangle$ $intersect(G_1, W)$ defines that $\{\mathcal{T}\}_{G_1} \cap \{\mathcal{T}\}_W \neq \emptyset$. Because $\{\mathcal{T}\}_{G_1} \subset \{\mathcal{T}\}_{G_2}$, it also holds that $\{\mathcal{T}\}_{G_2} \cap \{\mathcal{T}\}_W \neq \emptyset$. Here, the only possible usability degrees of W for G_2 are *subsume* if $\{\mathcal{T}\}_{G_2} \supset \{\mathcal{T}\}_W$, or *intersect* otherwise. It can not be *exact* or *plugin* because then it would hold that $\mathcal{T} \in (\{\mathcal{T}\}_{G_1} \subseteq \{\mathcal{T}\}_W)$ – which contradicts the matching condition for $intersect(G_1, W)$. This proves rule 2.4.
$\langle 6 \rangle$ $disjoint(G_1, W)$ defines that $\{\mathcal{T}\}_{G_1} \cap \{\mathcal{T}\}_W = \emptyset$. Under this similarity degree, we only know that W can not provide any solution for G_1. However, W might be able to provide a solution for G_2 such that $\exists \mathcal{T} \in \{\mathcal{T}\}_W \cap \{\mathcal{T}\}_{G_2}$, but we can not infer the usability degree of such a W from $plugin(G_1, G_2)$ and $disjoint(G_1, W)$ This relates to clause 2.5.

Next, we discuss the rules under $subsume(G_1, G_2)$ which denotes the situation where $\{\mathcal{T}\}_{G_1} \supset \{\mathcal{T}\}_{G_2}$. Here, the following holds for inferring the usability of W for G_2:
$\langle 1 \rangle$ if $exact(G_1, W)$ such that $\{\mathcal{T}\}_{G_1} = \{\mathcal{T}\}_W$ it holds that $\{\mathcal{T}\}_W \supset \{\mathcal{T}\}_{G_2}$. Hence, the only possible usability degree of W for G_2 is $plugin(G_2, W)$; it can neither be *exact*, *subsume*, nor *intersect* because $\{\mathcal{T}\}_W = \{\mathcal{T}\}_{G_1}, \{\mathcal{T}\}_{G_1} \supset \{\mathcal{T}\}_{G_2}$, and thus also not *disjoint*. Analog,

if $plugin(G_1, W)$ then $\{\mathcal{T}\}_W \supset \{\mathcal{T}\}_{G_1} \supset \{\mathcal{T}\}_{G_2}$ so that also here $plugin(G_2, W)$ is the only possible usability degree. This proves the rules 3.1 and 3.2.
$\langle 2 \rangle$ $subsume(G_1, W)$ defines that $\{\mathcal{T}\}_{G_1} \supseteq \{\mathcal{T}\}_W$. Because $\{\mathcal{T}\}_W$ can be any subset of $\{\mathcal{T}\}_{G_1}$, here all five usability degrees are possible for W and G_2. In particular, W can be not usable for G_2 if $\neg\exists\mathcal{T} \in (\{\mathcal{T}\}_{G_2} \cap \{\mathcal{T}\}_W)$. This shows rule 3.3.
$\langle 3 \rangle$ $intersect(G_1, W)$ denotes that $\exists\mathcal{T}_1 \in (\{\mathcal{T}\}_{G_1} \cap \{\mathcal{T}\}_W)$ but there can be a $\mathcal{T}_2 \in \{\mathcal{T}\}_{G_1}$ but $\mathcal{T}_2 \notin \{\mathcal{T}\}_W$ as well as a $\mathcal{T}_3 \notin \{\mathcal{T}\}_{G_1}$ but $\mathcal{T}_3 \in \{\mathcal{T}\}_W$. The possible degrees under which W is usable for G_2 are $plugin(G_2, W)$ if $\{\mathcal{T}\}_{G_2} \subset \{\mathcal{T}\}_W$, or $intersect(G_2, W)$ if $\exists\mathcal{T} \in (\{\mathcal{T}\}_{G_2} \cap \{\mathcal{T}\}_W)$. W might also be not usable, i.e. $disjoint(G_2, W)$, if the condition for the *intersect* degree is not satisfied. However, the usability can not be *subsume* and hence also not *exact* because this would require $\{\mathcal{T}\}_W \subseteq \{\mathcal{T}\}_{G_2} \subseteq G_1$ – which contradicts the condition of $intersect(G_1, W)$ under $subsume(G_1, G_2)$. This proves rule 3.4.
$\langle 4 \rangle$ if $disjoint(G_1, W)$ then also $disjoint(G_2, W)$ because $\{\mathcal{T}\}_{G_1} \cap \{\mathcal{T}\}_W = \emptyset$ and thus also $\{\mathcal{T}\}_{G_2} \cap \{\mathcal{T}\}_W = \emptyset$ because $\{\mathcal{T}\}_{G_1} \supset \{\mathcal{T}\}_{G_2}$. This proves rule 3.5.

We now turn towards the similarity degree $intersect(G_1, G_2)$ which denotes that $\{\mathcal{T}\}_{G_1} \cap \{\mathcal{T}\}_{G_2} \neq \emptyset$ and that there are non-common solution for G_1 and G_2:
$\langle 1 \rangle$ if $exact(G_1, W)$, then the only possible usability degree of W for G_2 is $intersect(G_2, W)$ because $\exists\mathcal{T}.\mathcal{T} \in (\{\mathcal{T}\}_{G_1} \cap \{\mathcal{T}\}_{G_2})$ and $\{\mathcal{T}\}_{G_1} = \{\mathcal{T}\}_W$. This proves rule 4.1.
$\langle 2 \rangle$ if $plugin(G_1, W)$ such that $\{\mathcal{T}\}_{G_1} \subset \{\mathcal{T}\}_W$, then the only possible usability degrees of W for G_2 are $plugin(G_2, W)$ if $\{\mathcal{T}\}_{G_2} \subset \{\mathcal{T}\}_W$ or $intersect(G_1, W)$ otherwise. It can not be $subsume(G_2, W)$ and hence not $exact(G_2, W)$ because this would require $\{\mathcal{T}\}_{G_2} \supset \{\mathcal{T}\}_W$ which contradicts $\{\mathcal{T}\}_{G_1} \subset \{\mathcal{T}\}_W$ under $intersect(G_1, G_2)$ because there exists a $\mathcal{T}_2$ such that $\mathcal{T}_2 \in \{\mathcal{T}\}_{G_1}$ but $\mathcal{T}_2 \notin \{\mathcal{T}\}_{G_2}$. This proves rule 4.2
$\langle 3 \rangle$ if $subsume(G_1, W)$ such that $\{\mathcal{T}\}_{G_1} \supset \{\mathcal{T}\}_W$, then W can be usable for G_2 under the *subsume* degree if $\{\mathcal{T}\}_{G_2} \supset \{\mathcal{T}\}_W$, or under the *intersect* degree if there is a $\mathcal{T}$ such that $\mathcal{T} \in \{\mathcal{T}\}_W$ but $\mathcal{T} \in \{\mathcal{T}\}_{G_2}$; otherwise, W is not usable so that $disjoint(G_2, W)$. However, the usability degree of W for G_2 can not be *plugin* and hence not *exact* because this would require that $\{\mathcal{T}\}_{G_2} \subset \{\mathcal{T}\}_W$ which contradicts $subsume(G_1, W)$ under the intersect similarity degree. This proves rule 4.3.
$\langle 4 \rangle$ under the *intersect* degree for both the similarity of G_1 and G_2 as well as for the usability of W for G_1, all five usability degrees are possible for W and G_2. In particular, W might not be usable for G_2 if $\neg\exists\mathcal{T} \in (\{\mathcal{T}\}_{G_2} \cap \{\mathcal{T}\}_W)$. This relates to rule 4.4.
$\langle 5 \rangle$ $disjoint(G_1, W)$ defines that $\{\mathcal{T}\}_{G_1} \cap \{\mathcal{T}\}_W = \emptyset$. However, W might be able to provide a solution for G_2 such that $\exists\mathcal{T} \in \{\mathcal{T}\}_W \cap \{\mathcal{T}\}_{G_2}$, but we can not infer the usability degree of such a W for G_2 from $intersect(G_1, G_2)$ and $disjoint(G_1, W)$. This relates to rule 4.5.

We finally discuss the implications of similarity degree $disjoint(G_1, G_2)$ which denotes the situation where G_1 and G_2) do not have a common solution, i.e. $\{\mathcal{T}\}_{G_1} \cap \{\mathcal{T}\}_{G_2} = \emptyset$. If the usability of W for G_1 is either *subsume* or *exact*, then $\{\mathcal{T}\}_{G_1} \supseteq \{\mathcal{T}\}_W$. As $disjoint(G_1, G_2)$ defines that $\{\mathcal{T}\}_{G_1} \cap \{\mathcal{T}\}_{G_2} = \emptyset$, W can not be usable in these cases. This proves the rules 5.1 and 5.2. Under all other usability degrees of W for G_1, we can not make any statement about the usability of W for G_2 because there can always be a $\mathcal{T} \in (\{\mathcal{T}\}_{G_2} \cap \{\mathcal{T}\}_W)$. This means that W might be usable for G_2 if it is not usable for G_1. This relates to rules 5.3 – 5.5 and completes the proof. □

B.2 Proof for Theorem 5.2

Theorem 5.2 in Section 5.2.3 states that the SDC graph $SDC = (V_{\mathcal{G}} \cup V_{\mathcal{W}}, E_{sim} \cup E_{use})$ over a set of goal templates $\mathcal{G}$ and a set of Web services $\mathcal{W}$ is inferentially complete and minimal because:

(i) $d(x, y) \in cl^{\star}(A_{SDC})$ if and only if $d(x, y)$ is `true` for $\mathcal{G} \times \mathcal{W}$, and

(ii) for all $d(x, y) \in A_{SDC} : cl^{\star}(A_{SDC} \setminus d(x, y)) \subset cl^{\star}(A_{SDC})$

where

- $A_{SDC} = \{d(x, y) | d \in \{\mathsf{exact}, \mathsf{plugin}, \mathsf{subsume}, \mathsf{intersect}\}, x \in \mathcal{G}, y \in (\mathcal{G}, \mathcal{W}), (x, y) \in (E_{sim}, E_{use})\}$ is the set of atoms defined in SDC
- $cl^{\star}(A_{SDC}) = \{d(x, y) | A_{SDC} \wedge \mathcal{IR} \models d(x, y)\}$ is the deductive closure of A_{SDC} over $\mathcal{IR}$ as the set of all inference rules defined in Theorem 5.1.

Proof. We commence with clause (i) which defines the inferential completeness of the SDC graph. By definition, every atom $d(x, y) \in A_{SDC}$ is `true` for $\mathcal{G} \times \mathcal{W}$ because every arc in SDC represents the result of semantic matchmaking on formal functional descriptions: E_{sim} defines the subsumption hierarchy of $\mathcal{G}$ in a redundance-free manner (*cf.* clause (iii) in Definition 5.2), and E_{use} is the minimal set of arcs to denote all situations where $d(G, W) \neq$ *disjoint* (*cf.* clause (iv) in Definition 5.2). The deductive closure $cl^{\star}(A_{SDC})$ is the set of all atoms $d(x, y) \in A_{SDC} \cup \{d^{\star}(x, y)\}$ where $\{d^{\star}(x, y)\}$ is the set of atoms that are not explicitly defined in SDC but can be deduced from inference rules defined in Theorem 5.1.

Such an atom $d^{\star}(x, y)$ is `true` for $\mathcal{G} \times \mathcal{W}$ when all $\mathcal{IR}$ rules hold under $d^{\star}(x, y)$; then, it must describe a correct relationship between a goal template and a Web service. For the situation where a goal template G_1 is a parent of G_2 and there is a Web service W whose usability degree for G_1 is *plugin*, the following atoms are defined in SDC graph: $subsume(G_1, G2), plugin(G_1, W) \in A_{SDC}$. Rule 3.2 from Theorem 5.1 states that $subsume(G_1, G_2) \wedge plugin(G_1, W) \Rightarrow plugin(G_2, W)$, so that the atom $plugin(G_2, W) \in cl^{\star}(A_{SDC})$, and this is `true` for $\mathcal{G} \times \mathcal{W}$ because it describes the correct usability degree of W for G_2. For this, also the opposite rule $plugin(G_2, G_1) \wedge plugin(G_2, W) \Rightarrow plugin(G_1, W)$ is `true`, so that clause (i) holds. In the situations where the $\mathcal{IR}$ rules do not allow us to infer the precise usability degree of W for G_2, this is by definition explicitly defined in the SDC graph. For example, when the usability degree of W for G_1 is *intersect* and G_1 is a parent of G_2 in the SDC graph, then the possible usability degrees of W for G_2 are *plugin*, *intersect*, or *disjoint* (*cf.* rule 3.4 in Theorem 5.1). If this is not *disjoint*, then there must a discovery cache arc (G_2, W) that defines the precise usability degree (*cf.* clause (iv) in Definition 5.2). Let us assume that this is *intersect*, so that the SDC graph defines the atoms

$subsume(G_1, G_2), intersect(G_1, W), intersect(G_2, W) \in A_{SDC}$. Given this, both relevant inference rules are true: (3.4) $subsume(G_1, G_2) \wedge intersect(G_1, W) \Rightarrow plugin(G_2, W) \vee intersect(G_2, W) \vee disjoint(G_2, W)$, and also (2.4) $plugin(G_2, G_1) \wedge intersect(G_2, W) \Rightarrow subsume(G_1, W) \vee intersect(G_2, W)$ as the opposite one. These rules would allow us to define the atoms $plugin(G_2, W), disjoint(G_2, W), subsume(G_1, W) \in cl^\star(A_{SDC})$. However, non of these atoms is `true` for $\mathcal{G} \times \mathcal{W}$ because we always use the highest possible degree to denote the usability of W for a goal template G in accordance to Proposition 4.2.

The same holds analogously for the other situations where the $\mathcal{IR}$ rules do not allow us to infer the precise usability degree. Thus, $cl^\star(A_{SDC})$ defines the total set of atoms which are `true` for $\mathcal{G} \times \mathcal{W}$: if there is a $d^\star(x, y)$ which is `true` for $\mathcal{G} \times \mathcal{W}$, then by definition there must be an arc in SDC from which this atom can be inferred, and any $d^\star(x, y)$ that is `false` for $\mathcal{G} \times \mathcal{W}$ will violate a $\mathcal{IR}$ rule. This shows that the SDC graph correctly keeps all knowledge that is relevant for inferring the precise usability of every Web service for each existing goal template, which we have defined as the requirement for the inferential completeness.

We now turn towards clause (ii) which defines the inferential minimality of the SDC graph. It states that if a single arc is removed from SDC, then its inferential completeness is disabled because then the respective atom $d(x, y)$ does not exist in A_{SDC}, and in consequence all the deducible atoms $d^\star(x, y)$ do not exist in $cl^\star(A_{SDC})$. If we remove a goal graph arc $(G_i, G_j) \in E_{sim}$, then we loose information on the semantic similarity of G_i and G_j and also can no longer determine the usability of all Web services W for G_j for which the discovery cache arcs are omitted in the SDC graph; if we remove a discovery cache $(G, W) \in E_{use}$, then we loose all relevant knowledge about the usability degree of W for G. On the other hand, if we add an arc that represents an atom $d^\star(x, y)$ that can be deduced from the $\mathcal{IR}$ rules and is `true` for $\mathcal{G} \times \mathcal{W}$, then by definition this atom is already in the deductive closure of the initial SDC graph so that $cl^\star(A_{SDC} \cup d(x, y)) = cl^\star(A_{SDC})$; thus, the additional arc is redundant. This shows that the SDC graph defines all relevant knowledge in a redundance-free manner, which we have defined as the requirement for the inferential minimality. □

B.3 Documentation of the SDC Prototype

The following provides the detailed technical documentation of the SDC prototype implementation presented in Section 5.5.

In accordance to the general usage policy of the WSMX system, the SDC prototype is implemented as open-source software. Technically, it consists of two Java packages: the first one provides the implementations for the central components of the SDC technique, and the second one provides the implementation of the used matchmaking techniques. The following provides general information on the availability of the SDC prototype software, and then explains the technical realization in more detail. We also refer to the actual implementation and the additional documentation available on the accompanying CD-R.

Availability

SDC Homepage: `http://www.michael-stollberg.de/phd/`

Owner & Contact: Michael Stollberg, `http://www.michael-stollberg.de`

Nature: Java Application

Platform: JDK 1.5

Licensing: GNU Lesser General Public License (LGPL)

Copy Right: DERI Innsbruck 2007

Version: 1.3, date: 19 October 2007

Download: `http://www.michael-stollberg.de//phd/prototype/SDCprototype.zip`

Source Control: CVS of the WSMX project (`http://wsmx.cvs.sourceforge.net`)

Required Libraries: all included in the "SDCstandalone.zip" archive

- WSMO4J – the WSMO API for Java (`wsmo4j.sourceforge.net`)
- Apache AXIS 2 – an open source SOAP engine (`http://ws.apache.org/axis2`)
- Apache Log4J – an open source API for inserting logging statements into Java code (`http://logging.apache.org/log4j`).

Package: `org.deri.wsmx.discovery.caching`

This package contains the main classes of the SDC prototype, in particular the implementations of the three main components `SDC Runtime Discoverer`, `SDC Graph Creator`, and `Evolution Manager` as defined in Figure 5.12 (see Section 5.5.1). Figure B.1 provides comprehensive overview in form of a UML class diagram; it consists of the following classes:

SDCResourceManager loads and stores WSML descriptions (ontology, goal, and Web service descriptions, as well as the SDC Graph Ontology); uses the WSMX Resource Manager for storage in a file system on the local machine

SDCGraphManager provides the basic facilities for managing the SDC Graph Ontology, including ontology instance management for SDC graph elements (goal templates, intersection goal templates, goal graph arcs, discovery cache arcs) and routines for handling the SDC Graph knowledge base

SDCGraphCreator implements the `SDC Graph Creator` component with the algorithm specified in Section 5.3.1

SDCGraphCreatorHelper provides helper methods for the `SDC Graph Creator`

SDCGraphEvolutionManager implements the `Evolution Manager` component with the algorithms specified in Section 5.3.3

GoalInstanceManager creation, management, and validation service for goal instances

GoalInstanceSDCDiscoverer implements `SDC Runtime Discoverer` component with the algorithms specified in Section 5.4.

Package: `org.deri.wsmx.discovery.caching.matchmaking`

This package contains the implementation of the matchmaking techniques used by the SDC prototype. Figure B.2 provides the UML class diagram, consisting of the following classes:

Matchmaker defines all semantic matchmaking needed for the SDC technique

- uses the Web service *VampireInvoker* to perform the matchmaking (see below)
- uses the *POGenerator* for generating the proof obligations (see below)
- the interface *Matchmaker* defines the method skeletons for all matchmaking operations needed for the SDC technique; other implementations of this interface may use different reasoning environments

POGenerator generates the TPTP proof obligations on the client side

- a proof obligation is a logical statement which defines a particular matchmaking operation; this is to be proved by VAMPIRE (see Section 4.4)
- the interface *POGenerator* defines the method skeletons for all types of proof obligations needed for the SDC technique
- the class *POGenerator* implements the interface methods and defines the mapping between the WSML specifications and the pre-defined TPTP descriptions

VampireInvoker Web service implementation class for invoking VAMPIRE on a remote web server; this is a generic facility for invoking VAMPIRE for any proof obligation

- intermediately stores the TPTP proof obligation, invokes VAMPIRE for proving it, and returns the result (as a boolean)
- the Web service is publicly available at `http://138.232.65.138:8080/axis2/services/VampireInvoker?wsdl`; Listing B.1 shows the WSDL description that has been generated with the Java2WSDL tool provided by Apache AXIS2

VampireInvokerStub client stub for the Web service (generated by Apache AXIS2).

```
< wsdl:definitions
xmlns:axis2 ="http://matchmaking.caching.discovery.wsmx.deri.org"
  xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
  xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
  xmlns:ns0="http://matchmaking.caching.discovery.wsmx.deri.org/xsd"
  xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/"
  xmlns:ns1="http://org.apache.axis2/xsd"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  targetNamespace="http://matchmaking.caching.discovery.wsmx.deri.org">
<wsdl:documentation>runs a given proof obligation</wsdl:documentation><wsdl:types>
 <xs:schema xmlns:ns="http://matchmaking.caching.discovery.wsmx.deri.org/xsd"
    attributeFormDefault="qualified" elementFormDefault="qualified"
    targetNamespace="http://matchmaking.caching.discovery.wsmx.deri.org/xsd">
  <xs:element name="check"> <xs:complexType> <xs:sequence>
     <xs:element name="poContent" nillable="true" type="xs:string"/>
 </xs:sequence> </xs:complexType> </xs:element>
 <xs:element name="checkResponse"> <xs:complexType> <xs:sequence>
    <xs:element name="return" nillable="true" type="xs:boolean"/>
 </xs:sequence> </xs:complexType> </xs:element>
 </xs:schema></wsdl:types>
 <wsdl:message name="checkMessage"> <wsdl:part name="part1" element="ns0:check"/></wsdl:message>
 <wsdl:message name="checkResponse">
```

```
  <wsdl:part name="part1" element="ns0:checkResponse" /></wsdl:message>
<wsdl:portType name="VampireInvokerPortType">
 <wsdl:operation name="check">
   <wsdl:input xmlns:wsaw="http://www.w3.org/2006/05/addressing/wsdl"
        message="axis2:checkMessage" wsaw:Action="urn:check" />
   <wsdl:output message="axis2:checkResponse" />
  </wsdl:operation>
</wsdl:portType>
<wsdl:binding name="VampireInvokerSOAP11Binding" type="axis2:VampireInvokerPortType">
 <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document" />
 <wsdl:operation name="check">
   <soap:operation soapAction="urn:check" style="document" />
   <wsdl:input><soap:body use="literal" /></wsdl:input>
   <wsdl:output><soap:body use="literal" /></wsdl:output>
 </wsdl:operation>
</wsdl:binding>
<wsdl:binding name="VampireInvokerSOAP12Binding" type="axis2:VampireInvokerPortType">
 <soap12:binding transport="http://schemas.xmlsoap.org/soap/http"
     style="document" />
<wsdl:operation name="check">
 <soap12:operation soapAction="urn:check" style="document" />
  <wsdl:input> <soap12:body use="literal" /> </wsdl:input>
   <wsdl:output> <soap12:body use="literal" /> </wsdl:output>
 </wsdl:operation>
</wsdl:binding>
<wsdl:binding name="VampireInvokerHttpBinding" type="axis2:VampireInvokerPortType">
  <http:binding verb="POST" />
  <wsdl:operation name="check">
    <http:operation location="check" />
    <wsdl:input><mime:content type="text/xml" /> </wsdl:input>
    <wsdl:output><mime:content type="text/xml" /> </wsdl:output>
  </wsdl:operation>
</wsdl:binding>
<wsdl:service name="VampireInvoker">
 <wsdl:port name="VampireInvokerSOAP11port_http"
    binding="axis2:VampireInvokerSOAP11Binding">
   <soap:address location="http://138.232.65.138:8080/axis2/services/VampireInvoker" />
 </wsdl:port>
 <wsdl:port name="VampireInvokerSOAP12port_http"
    binding="axis2:VampireInvokerSOAP12Binding">
   <soap12:address location="http://138.232.65.138:8080/axis2/services/VampireInvoker" />
  </wsdl:port>
   <wsdl:port name="VampireInvokerHttpport1"
    binding="axis2:VampireInvokerHttpBinding">
      <http:address location="http://138.232.65.138:8080/axis2/rest/VampireInvoker" />
  </wsdl:port>
</wsdl:service>
</ wsdl:definitions >
```

Listing B.1: WSDL Description of VAMPIRE Invoker Web Service

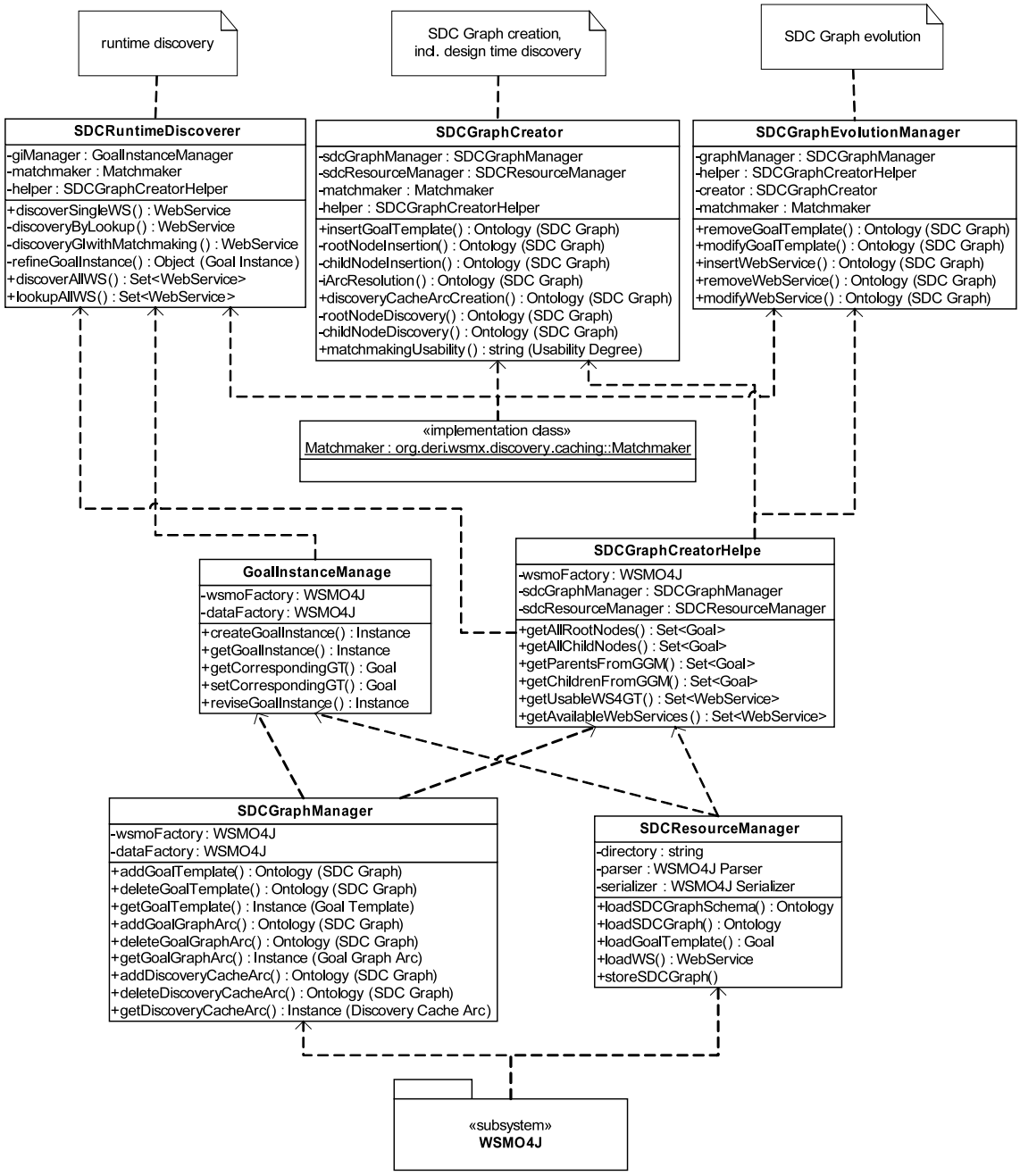


Figure B.1: UML Class Diagram of SDC Prototype

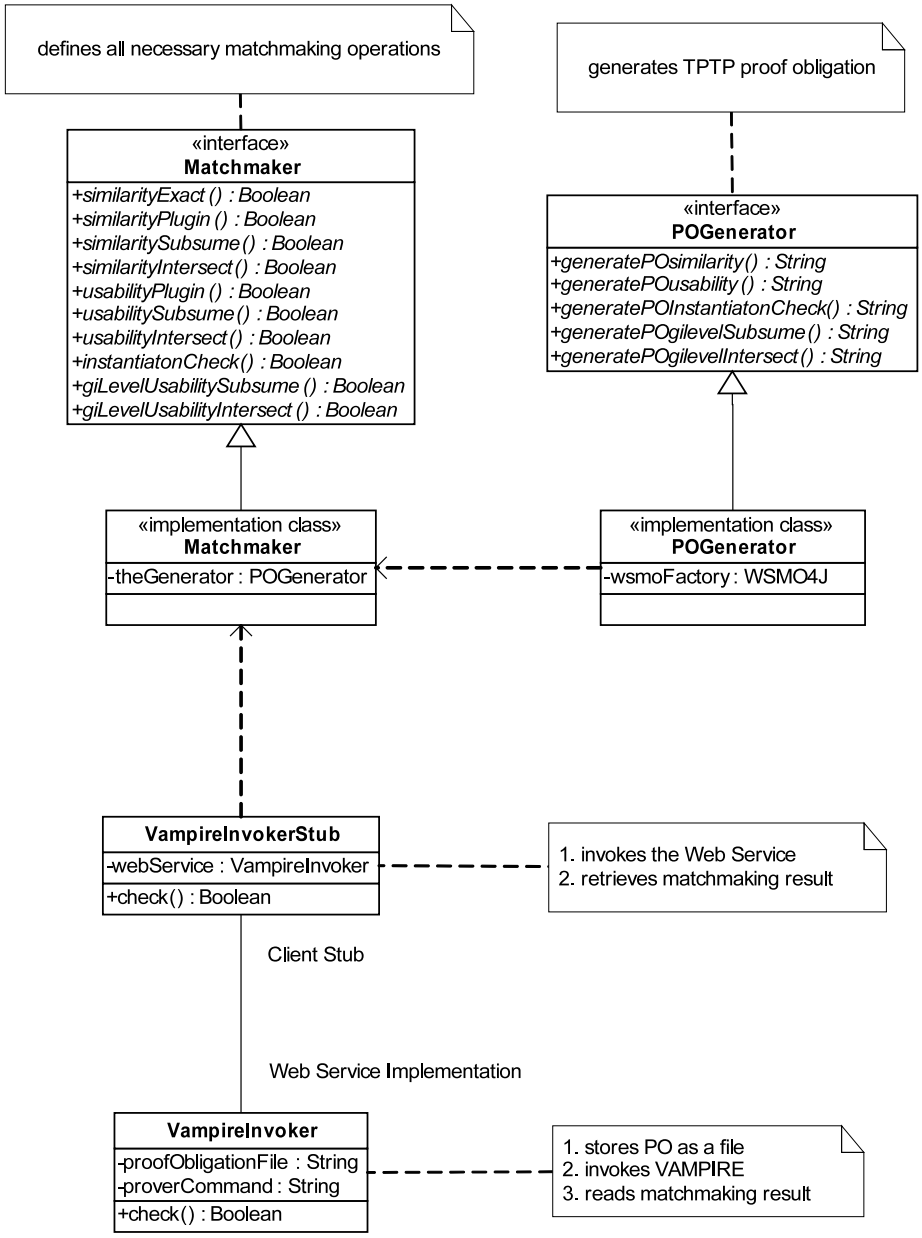


Figure B.2: UML Class Diagram of Matchmaker used in SDC Prototype

Appendix C

Appendix for Chapter 6

C.1 Resource Descriptions in WSML

The following provides the complete specifications of the domain ontologies as well as the descriptions of the goal templates and the Web services for the shipment scenario, which serves as the use case for the performance analysis presented in Section 6.1.

We use *WSML FOL* as the ontology specification language supported by the SDC prototype implementation. The functional descriptions of goals and Web services are here defined as WSMO capabilities using *WSML FOL*; we refer to Section 5.5.1 for the translation to the TPTP representation that is used by VAMPIRE for the matchmaking.

Domain Ontologies

The following specifies the two domain ontologies used for describing the goals and Web services in the shipment scenario. Listing C.1 shows the `location ontology` which describes continents, countries, and cities; below, Listing C.2 shows the `shipment ontology`.

```
wsmlVariant _"http://www.wsmo.org/wsml/wsml-syntax/wsml-full"
namespace {_"http://members.deri.org/~michaels/sdc/swsc-shipment#",
  dc _"http://purl.org/dc/elements/1.1#",
  wsml _"http://www.wsmo.org/wsml/wsml-syntax#" }
ontology _"http://members.deri.org/~michaels/sdc/swsc-shipment/ontologies/location.wsml"
  nonFunctionalProperties
    dc#title hasValue "Location Ontology"
    dc#contributor hasValue {"Holger Lausen", "Adina Sirbu","Michael Stollberg"}
    dc#date hasValue _date(2007,03,08)
  endNonFunctionalProperties
```

```
/** CONCEPTS **/

 concept Location
    nfp
        dc#description hasValue "a location is a physical defined by longitude, latitude, and altitude."
    endnfp
    locatedIn impliesType GeographicalArea
 concept GeographicalArea
    nfp
        dc#description hasValue "a geographical area is a larger area such as a state or country."
    endnfp
    name impliesType _string
    containsLocation impliesType Location
 concept Continent subConceptOf GeographicalArea
 concept Region subConceptOf GeographicalArea
 concept Country subConceptOf GeographicalArea
 concept State subConceptOf GeographicalArea
 concept City subConceptOf GeographicalArea

/** AXIOMS **/

axiom transitivityLocatedIn
  nfp
      dc#description hasValue "defines the transitivity of the locatedIn attribute"
  endnfp
  definedBy forAll (?L1,?L2,?L2).
    ?L1[locatedIn hasValue ?L2] and ?L2[locatedIn hasValue ?L3] implies ?L1[locatedIn hasValue ?L3].

axiom locationsInContinentsAreDistinct
  nfp
      dc#description hasValue "locations in different continents are disjoint"
  endnfp
  definedBy forAll (?C1,?C2,?L).
    ?C1 memberOf continent and ?C2 memberOf continent and L[locatedIn hasValue ?C1]
    implies neg ?L[locatedIn hasValue ?C2].

axiom countryConstraint
  nfp
      dc#description hasValue "each country is located in exactly one continent or region (constraint)"
  endnfp
  definedBy forAll (?C,?X,?Y).
    ?C memberOf country and ?C[locatedIn hasValue ?X] and ?C[locatedIn hasValue ?Y] implies ?X = ?Y.

axiom cityConstraint
  nfp
      dc#description hasValue "each city is located in exactly one country (constraint)"
  endnfp
  definedBy forAll (?C,?X,?Y).
    ?C memberOf city and ?C[locatedIn hasValue ?X] and ?X memberOf country and
    ?C[locatedIn hasValue ?Y] and ?Y memberOf country implies ?X = ?Y.
```

```
/** CONTINENTS & REGIONS **/

 instance world memberOf Region
 instance africa memberOf Continent
    name hasValue "Africa"
    locatedIn hasValue world
 instance antarctica memberOf Continent
    name hasValue "Antarctica"
    locatedIn hasValue world
 instance asia memberOf Continent
    name hasValue "Asia"
    locatedIn hasValue world
 instance australia memberOf Continent
    name hasValue "Australia"
    locatedIn hasValue world
 instance europe memberOf Continent
    name hasValue "Europe"
    locatedIn hasValue world
 instance northAmerica memberOf Continent
    name hasValue "North America"
    locatedIn hasValue world
 instance southAmerica memberOf Continent
    name hasValue "South America"
    locatedIn hasValue world
 instance oceania memberOf Region
  name hasValue "Oceania"
  locatedIn hasValue world
  containsLocation hasValue australia

/** COUNTIES & STATES (we only list those ones used within the running examples) **/

 instance usa memberOf Country
    name hasValue "United States of America"
    locatedIn hasValue northAmerica
 instance california memberOf State
    name hasValue "California"
    locatedIn hasValue usa
 instance germany memberOf Country
    name hasValue "Germany"
    locatedIn hasValue europe
 instance austria memberOf Country
    name hasValue "Austria"
    locatedIn hasValue europe
 instance switzerland memberOf Country
    name hasValue "Switzerland"
    locatedIn hasValue europe
 instance unitedKingdom memberOf Country
    name hasValue "United Kingdom"
    locatedIn hasValue europe
 instance netherlands memberOf Country
```

```
    name hasValue "Netherlands"
    locatedIn hasValue europe
instance luxembourg memberOf Country
    name hasValue "Luxembourg"
    locatedIn hasValue europe
instance algeria memberOf Country
    name hasValue "Alegria"
    locatedIn hasValue africa
instance ecuador memberOf Country
    name hasValue "Ecuador"
    locatedIn hasValue southAmerica
instance china memberOf Country
    name hasValue "China"
    locatedIn hasValue asia

/** CITIES (we only list those ones used within the running examples) **/

instance sanFrancisco memberOf City
    name hasValue "sanFrancisco"
    locatedIn hasValue california
instance newYorkCity memberOf City
    name hasValue "New York City"
    locatedIn hasValue usa
instance berlin memberOf City
    name hasValue "Berlin"
    locatedIn hasValue germany
instance bristol memberOf City
    name hasValue "Bristol"
    locatedIn hasValue unitedKingdom
instance amsterdam memberOf City
    name hasValue "Amsterdam"
    locatedIn hasValue netherlands
instance luxembourgCity memberOf City
    name hasValue "Luxembourg"
    locatedIn hasValue luxembourg
instance sydney memberOf City
    name hasValue "Sydney"
    locatedIn hasValue australia
instance tunis memberOf City
    name hasValue "Tunis"
    locatedIn hasValue algeria
instance quito memberOf City
    name hasValue "Quito"
    locatedIn hasValue ecuador
instance beijing memberOf City
    name hasValue "Bejing"
    locatedIn hasValue china
```

Listing C.1: Location Ontology

```
wsmlVariant _"http://www.wsmo.org/wsml/wsml-syntax/wsml-full"
namespace {_"http://members.deri.org/~michaels/sdc/swsc-shipment#",
  dc _"http://purl.org/dc/elements/1.1#",
  loc _"http://members.deri.org/~michaels/sdc/swsc-shipment/ontologies/location.wsml#",
  wsml _"http://www.wsmo.org/wsml/wsml-syntax#" }
ontology _"http://members.deri.org/~michaels/sdc/swsc-shipment/ontologies/shipment.wsml"
 importsOntology {_"http://members.deri.org/~michaels/sdc/swsc-shipment/ontologies/location.wsml"}
  nonFunctionalProperties
    dc#title hasValue "Shipment Ontology"
    dc#contributor hasValue {"Michael Stollberg"}
    dc#date hasValue _date(2007,03,08)
  endNonFunctionalProperties

/** CONCEPTS **/

 concept Package
    weight impliesType WeightClass
    size impliesType Size
 concept Sender
    address impliesType loc#Location
 concept Receiver
    address impliesType loc#Location
 concept shipmentOrder
    from impliesType Sender
    to impliesType Receiver
    item impliesType Package
    price impliesType Price
 concept Price
    amount impliesType wsml#float
    currency impliesType Currency
 concept Currency
    name impliesType wsml#string
    symbol impliesType wsml#string
 concept WeightClass
    nfp
        dc#description hasValue "weights are distinguished in classes of 10 kg. This is a sufficient
            abstraction for the intended usage."
    endnfp
    includedIn impliesType WeightClass

/** AXIOMS **/

 axiom transitiveWeightClassInclusion
    nfp
        dc#description hasValue "defines the relationship of weight classes"
    endnfp
    definedBy forAll (?W1,?W2,?W3).
        ?W1 memberOf WeightClass and ?W2 memberOf WeightClass and ?W3 memberOf WeightClass
        and ?W1[includedIn hasValue ?W2] and ?W2[includedIn hasValue ?W3]
        implies ?W1[includedIn hasValue ?W3].
```

```
/** INSTANCES **/

 instance USD memberOf Currency
    name hasValue "US Dollar"
    symbol hasValue "USD"

 instance heavy memberOf WeightClass
    includedIn hasValue heavy
 instance weightClass70kg memberOf WeightClass
    includedIn hasValue heavy
 instance weightClass60kg memberOf WeightClass
    includedIn hasValue weightClass70kg
 instance weightClass70kg memberOf WeightClass
    includedIn hasValue weightClass60kg
 instance weightClass40kg memberOf WeightClass
    includedIn hasValue weightClass50kg
 instance weightClass30kg memberOf WeightClass
    includedIn hasValue weightClass40kg
 instance weightClass20kg memberOf WeightClass
    includedIn hasValue weightClass30kg
 instance light memberOf WeightClass
    includedIn hasValue weightClass20kg
    includedIn hasValue light
```

Listing C.2: Shipment Ontology

Goal Templates

The following shows the functional descriptions of goal templates in the form of WSMO capability descriptions for goals. We here content ourselves with three goal templates; the other ones used for the performance analysis tests are defined analog.

```
wsmlVariant _"http://www.wsmo.org/wsml/wsml-syntax/wsml-full"
namespace {_"http://members.deri.org/~michaels/sdc/swsc-shipment#",
  dc _"http://purl.org/dc/elements/1.1#",
  wsml _"http://www.wsmo.org/wsml/wsml-syntax/#",
  sho _"http://members.deri.org/~michaels/sdc/swsc-shipment/ontologies/shipment.wsml#",
  loc _"http://members.deri.org/~michaels/sdc/swsc-shipment/ontologies/location.wsml#" }
goal _"http://members.deri.org/~michaels/sdc/swsc-shipment/goals/goaltemplates/gtRoot.wsml"
 nfp
    dc#title hasValue "Goal Template gtRoot"
    dc#description hasValue "package shipment from anywhere to anywhere with any weight"
    dc#relation hasValue _"http://sws-challenge.org/wiki/index.php/Scenario:_Shipment_Discovery"
    dc#contributor hasValue {"Michael Stollberg"}
 endnfp
 importsOntology {_"http://members.deri.org/~michaels/sdc/swsc-shipment/ontologies/shipment.wsml#",
    _"http://members.deri.org/~michaels/sdc/swsc-shipment/ontologies/location.wsml#"}
```

```
capability gtRootCapability
 sharedVariables {?SendLoc,?RecLoc,?Package,?Weight}
 precondition definedBy
  forAll (?Sender,?Receiver).
   ?Sender[address hasValue ?SendLoc] memberOf sho#Sender and
       ?SendLoc[locatedIn hasValue loc#world] memberOf loc#Location and
   ?Receiver[address hasValue ?RecLoc] memberOf sho#Receiver and
       ?RecLoc[locatedIn hasValue loc#world] memberOf loc#Location and
   ?Package[weight hasValue ?Weight] memberOf sho#Package and
   ?Weight[includedIn hasValue sho#heavy].
 postcondition definedBy
  forAll (?O). ?O [ from hasValue ?SendLoc,
                    to hasValue ?RecLoc,
                    item hasValue ?Package,
                    price hasValue thePrice] memberOf sho#shipmentOrder.
```

Listing C.3: WSMO Description of Goal Template `gtRoot`

```
wsmlVariant _"http://www.wsmo.org/wsml/wsml-syntax/wsml-full"
namespace {_"http://members.deri.org/~michaels/sdc/swsc-shipment#",
  dc _"http://purl.org/dc/elements/1.1#",
  wsml _"http://www.wsmo.org/wsml/wsml-syntax/#",
  sho _"http://members.deri.org/~michaels/sdc/swsc-shipment/ontologies/shipment.wsml#",
  loc _"http://members.deri.org/~michaels/sdc/swsc-shipment/ontologies/location.wsml#" }
goal _"http://members.deri.org/~michaels/sdc/swsc-shipment/goals/goaltemplates/gtUS2EU.wsml"
 nfp
    dc#title hasValue "Goal Template gtUS2EU"
    dc#description hasValue "package shipment from USA to Europe with any weight"
    dc#relation hasValue _"http://sws-challenge.org/wiki/index.php/Scenario:_Shipment_Discovery"
    dc#contributor hasValue {"Michael Stollberg"}
 endnfp
 importsOntology {_"http://members.deri.org/~michaels/sdc/swsc-shipment/ontologies/shipment.wsml#",
    _"http://members.deri.org/~michaels/sdc/swsc-shipment/ontologies/location.wsml#"}
 capability gtUS2worldCapability
  sharedVariables {?SendLoc,?RecLoc,?Package,?Weight}
  precondition definedBy
   forAll (?Sender,?Receiver).
    ?Sender[address hasValue ?SendLoc] memberOf sho#Sender and
        ?SendLoc[locatedIn hasValue loc#usa] memberOf loc#Location and
    ?Receiver[address hasValue ?RecLoc] memberOf sho#Receiver and
        ?RecLoc[locatedIn hasValue loc#europe] memberOf loc#Location and
    ?Package[weight hasValue ?Weight] memberOf sho#Package and
    ?Weight[includedIn hasValue sho#heavy].
  postcondition definedBy
   forAll (?O). ?O[ from hasValue ?SendLoc,
                    to hasValue ?RecLoc,
                    item hasValue ?Package,
                    price hasValue thePrice] memberOf sho#shipmentOrder.
```

Listing C.4: WSMO Description of Goal Template `gtUS2EU`

```
wsmlVariant _"http://www.wsmo.org/wsml/wsml-syntax/wsml-full"
namespace {_"http://members.deri.org/~michaels/sdc/swsc-shipment/#",
  dc _"http://purl.org/dc/elements/1.1#",
  wsml _"http://www.wsmo.org/wsml/wsml-syntax/#",
  sho _"http://members.deri.org/~michaels/sdc/swsc-shipment/ontologies/shipment.wsml#",
  loc _"http://members.deri.org/~michaels/sdc/swsc-shipment/ontologies/location.wsml#" }
goal _"http://members.deri.org/~michaels/sdc/swsc-shipment/goals/goaltemplates/gtUS2EUlight.wsml"
 nfp
    dc#title hasValue "Goal Template gtUS2EUlight"
    dc#description hasValue "package shipment from USA to Europe with light weight"
    dc#relation hasValue _"http://sws-challenge.org/wiki/index.php/Scenario:_Shipment_Discovery"
    dc#contributor hasValue {"Michael Stollberg"}
 endnfp
 importsOntology {_"http://members.deri.org/~michaels/sdc/swsc-shipment/ontologies/shipment.wsml#",
    _"http://members.deri.org/~michaels/sdc/swsc-shipment/ontologies/location.wsml#"}
 capability gtUS2worldCapability
  sharedVariables {?SendLoc,?RecLoc,?Package,?Weight}
  precondition definedBy
   forAll (?Sender,?Receiver).
    ?Sender[address hasValue ?SendLoc] memberOf sho#Sender and
        ?SendLoc[locatedIn hasValue loc#usa] memberOf loc#Location and
    ?Receiver[address hasValue ?RecLoc] memberOf sho#Receiver and
        ?RecLoc[locatedIn hasValue loc#europe] memberOf loc#Location and
    ?Package[weight hasValue ?Weight] memberOf sho#Package and
    ?Weight[includedIn hasValue sho#light].
  postcondition definedBy
   forAll (?O). ?O[ from hasValue ?SendLoc,
                    to hasValue ?RecLoc,
                    item hasValue ?Package,
                    price hasValue thePrice] memberOf sho#shipmentOrder.
```

Listing C.5: WSMO Description of Goal Template `gtUS2EUlight`

Web Services

The following defines the WSMO capability descriptions of the five Web services in accordance to the the original scenario description of the SWS challenge shipment scenario.

```
wsmlVariant _"http://www.wsmo.org/wsml/wsml-syntax/wsml-full"
namespace {_"http://members.deri.org/~michaels/sdc/swsc-shipment#",
  dc _"http://purl.org/dc/elements/1.1#",
  wsml _"http://www.wsmo.org/wsml/wsml-syntax/#",
  sho _"http://members.deri.org/~michaels/sdc/swsc-shipment/ontologies/shipment.wsml#",
  loc _"http://members.deri.org/~michaels/sdc/swsc-shipment/ontologies/location.wsml#" }
webService _"http://members.deri.org/~michaels/sdc/swsc-shipment/webservices/wsMuller.wsml"
 nfp
    dc#title hasValue "Muller Shipment Web Service"
```

```
    dc#relation hasValue _"http://sws-challenge.org/wiki/index.php/Scenario:_Shipment_Discovery#Muller"
    dc#contributor hasValue {"Michael Stollberg"}
 endnfp
 importsOntology {_"http://members.deri.org/~michaels/sdc/swsc-shipment/ontologies/shipment.wsml#",
    _"http://members.deri.org/~michaels/sdc/swsc-shipment/ontologies/location.wsml#"}
 capability wsMullerCapability
  sharedVariables {?SendLoc,?RecLoc,?Package,?Weight}
  precondition definedBy
   forAll (?Sender,?Receiver).
    ?Sender[address hasValue ?SendLoc] memberOf sho#Sender and
        ?SendLoc[locatedIn hasValue loc#usa] memberOf loc#Location and
    ?Receiver[address hasValue ?RecLoc] memberOf sho#Receiver and
        (?RecLoc[locatedIn hasValue loc#africa] memberOf loc#Location or
         ?RecLoc[locatedIn hasValue loc#europe] memberOf loc#Location or
         ?RecLoc[locatedIn hasValue loc#northAmerica] memberOf loc#Location or
         ?RecLoc[locatedIn hasValue loc#asia] memberOf loc#Location) and
    ?Package[weight hasValue ?Weight] memberOf sho#Package and
    ?Weight[includedIn hasValue sho#w50lq].
  postcondition definedBy
   forAll (?O). ?O[from hasValue ?SendLoc,
                   to hasValue ?RecLoc,
                   item hasValue ?Package,
                   price hasValue thePrice] memberOf sho#shipmentOrder.
```

Listing C.6: WSMO Description of Web Service `Muller`

```
wsmlVariant _"http://www.wsmo.org/wsml/wsml-syntax/wsml-full"
namespace {_"http://members.deri.org/~michaels/sdc/swsc-shipment#",
  dc _"http://purl.org/dc/elements/1.1#",
  wsml _"http://www.wsmo.org/wsml/wsml-syntax/#",
  sho _"http://members.deri.org/~michaels/sdc/swsc-shipment/ontologies/shipment.wsml#",
  loc _"http://members.deri.org/~michaels/sdc/swsc-shipment/ontologies/location.wsml#" }
webService _"http://members.deri.org/~michaels/sdc/swsc-shipment/webservices/wsRacer.wsml"
 nfp
    dc#title hasValue "Racer Shipment Web Service"
    dc#relation hasValue _"http://sws-challenge.org/wiki/index.php/Scenario:_Shipment_Discovery#Racer"
    dc#contributor hasValue {"Michael Stollberg"}
 endnfp
 importsOntology {_"http://members.deri.org/~michaels/sdc/swsc-shipment/ontologies/shipment.wsml#",
    _"http://members.deri.org/~michaels/sdc/swsc-shipment/ontologies/location.wsml#"}
 capability wsRacerCapability
  sharedVariables {?SendLoc,?RecLoc,?Package,?Weight}
  precondition definedBy
   forAll (?Sender,?Receiver).
    ?Sender[address hasValue ?SendLoc] memberOf sho#Sender and
        ?SendLoc[locatedIn hasValue loc#usa] memberOf loc#Location and
    ?Receiver[address hasValue ?RecLoc] memberOf sho#Receiver and
        (?RecLoc[locatedIn hasValue loc#africa] memberOf loc#Location or
         ?RecLoc[locatedIn hasValue loc#europe] memberOf loc#Location or
         ?RecLoc[locatedIn hasValue loc#northAmerica] memberOf loc#Location or
```

```
        ?RecLoc[locatedIn hasValue loc#southAmerica] memberOf loc#Location or
        ?RecLoc[locatedIn hasValue loc#asia] memberOf loc#Location or
        ?RecLoc[locatedIn hasValue loc#oceania] memberOf loc#Location) and
    ?Package[weight hasValue ?Weight] memberOf sho#Package and
    ?Weight[includedIn hasValue sho#w70lq].
  postcondition definedBy
   forAll (?O). ?O[from hasValue ?SendLoc,
                   to hasValue ?RecLoc,
                   item hasValue ?Package,
                   price hasValue thePrice] memberOf sho#shipmentOrder.
```

Listing C.7: WSMO Description of Web Service `Racer`

```
wsmlVariant _"http://www.wsmo.org/wsml/wsml-syntax/wsml-full"
namespace {_"http://members.deri.org/~michaels/sdc/swsc-shipment#",
  dc _"http://purl.org/dc/elements/1.1#",
  wsml _"http://www.wsmo.org/wsml/wsml-syntax/#",
  sho _"http://members.deri.org/~michaels/sdc/swsc-shipment/ontologies/shipment.wsml#",
  loc _"http://members.deri.org/~michaels/sdc/swsc-shipment/ontologies/location.wsml#" }
webService _"http://members.deri.org/~michaels/sdc/swsc-shipment/webservices/wsRunner.wsml"
  nfp
    dc#title hasValue "Runner Shipment Web Service"
    dc#relation hasValue _"http://sws-challenge.org/wiki/index.php/Scenario:_Shipment_Discovery#Runner"
    dc#contributor hasValue {"Michael Stollberg"}
 endnfp
 importsOntology {_"http://members.deri.org/~michaels/sdc/swsc-shipment/ontologies/shipment.wsml#",
    _"http://members.deri.org/~michaels/sdc/swsc-shipment/ontologies/location.wsml#"}
 capability wsRunnerCapability
  sharedVariables {?SendLoc,?RecLoc,?Package,?Weight}
  precondition definedBy
   forAll (?Sender,?Receiver).
    ?Sender[address hasValue ?SendLoc] memberOf sho#Sender and
        ?SendLoc[locatedIn hasValue loc#usa] memberOf loc#Location and
    ?Receiver[address hasValue ?RecLoc] memberOf sho#Receiver and
        (?RecLoc[locatedIn hasValue loc#africa] memberOf loc#Location or
         ?RecLoc[locatedIn hasValue loc#europe] memberOf loc#Location or
         ?RecLoc[locatedIn hasValue loc#northAmerica] memberOf loc#Location or
         ?RecLoc[locatedIn hasValue loc#southAmerica] memberOf loc#Location or
         ?RecLoc[locatedIn hasValue loc#asia] memberOf loc#Location or
         ?RecLoc[locatedIn hasValue loc#oceania] memberOf loc#Location) and
    ?Package[weight hasValue ?Weight] memberOf sho#Package and
    ?Weight[includedIn hasValue sho#heavy].
  postcondition definedBy
   forAll (?O). ?O[from hasValue ?SendLoc,
                   to hasValue ?RecLoc,
                   item hasValue ?Package,
                   price hasValue thePrice] memberOf sho#shipmentOrder.
```

Listing C.8: WSMO Description of Web Service `Runner`

```
wsmlVariant _"http://www.wsmo.org/wsml/wsml-syntax/wsml-full"
namespace {_"http://members.deri.org/~michaels/sdc/swsc-shipment#",
  dc _"http://purl.org/dc/elements/1.1#",
  wsml _"http://www.wsmo.org/wsml/wsml-syntax/#",
  sho _"http://members.deri.org/~michaels/sdc/swsc-shipment/ontologies/shipment.wsml#",
  loc _"http://members.deri.org/~michaels/sdc/swsc-shipment/ontologies/location.wsml#" }
webService _"http://members.deri.org/~michaels/sdc/swsc-shipment/webservices/wsWalker.wsml"
 nfp
    dc#title hasValue "Walker Shipment Web Service"
    dc#relation hasValue _"http://sws-challenge.org/wiki/index.php/Scenario:_Shipment_Discovery#Walker"
    dc#contributor hasValue {"Michael Stollberg"}
 endnfp
 importsOntology {_"http://members.deri.org/~michaels/sdc/swsc-shipment/ontologies/shipment.wsml#",
    _"http://members.deri.org/~michaels/sdc/swsc-shipment/ontologies/location.wsml#"}
 capability wsWalkerCapability
  sharedVariables {?SendLoc,?RecLoc,?Package,?Weight}
  precondition definedBy
   forAll (?Sender,?Receiver).
    ?Sender[address hasValue ?SendLoc] memberOf sho#Sender and
        ?SendLoc[locatedIn hasValue loc#usa] memberOf loc#Location and
    ?Receiver[address hasValue ?RecLoc] memberOf sho#Receiver and
        (?RecLoc[locatedIn hasValue loc#africa] memberOf loc#Location or
         ?RecLoc[locatedIn hasValue loc#europe] memberOf loc#Location or
         ?RecLoc[locatedIn hasValue loc#northAmerica] memberOf loc#Location or
         ?RecLoc[locatedIn hasValue loc#southAmerica] memberOf loc#Location or
         ?RecLoc[locatedIn hasValue loc#asia] memberOf loc#Location or
         ?RecLoc[locatedIn hasValue loc#oceania] memberOf loc#Location) and
    ?Package[weight hasValue ?Weight] memberOf sho#Package and
    ?Weight[includedIn hasValue sho#w50lq].
  postcondition definedBy
   forAll (?O). ?O[from hasValue ?SendLoc,
                   to hasValue ?RecLoc,
                   item hasValue ?Package,
                   price hasValue thePrice] memberOf sho#shipmentOrder.
```

Listing C.9: WSMO Description of Web Service `Walker`

```
wsmlVariant _"http://www.wsmo.org/wsml/wsml-syntax/wsml-full"
namespace {_"http://members.deri.org/~michaels/sdc/swsc-shipment#",
  dc _"http://purl.org/dc/elements/1.1#",
  wsml _"http://www.wsmo.org/wsml/wsml-syntax/#",
  sho _"http://members.deri.org/~michaels/sdc/swsc-shipment/ontologies/shipment.wsml#",
  loc _"http://members.deri.org/~michaels/sdc/swsc-shipment/ontologies/location.wsml#" }
webService _"http://members.deri.org/~michaels/sdc/swsc-shipment/webservices/wsWeasel.wsml"
 nfp
    dc#title hasValue "Weasel Shipment Web Service"
    dc#relation hasValue _"http://sws-challenge.org/wiki/index.php/Scenario:_Shipment_Discovery#Weasel"
    dc#contributor hasValue {"Michael Stollberg"}
 endnfp
```

```
importsOntology {_"http://members.deri.org/~michaels/sdc/swsc-shipment/ontologies/shipment.wsml#",
    _"http://members.deri.org/~michaels/sdc/swsc-shipment/ontologies/location.wsml#"}
 capability wsWeaselCapability
  sharedVariables {?SendLoc,?RecLoc,?Package,?Weight}
  precondition definedBy
   forAll (?Sender,?Receiver).
    ?Sender[address hasValue ?SendLoc] memberOf sho#Sender and
        ?SendLoc[locatedIn hasValue loc#usa] memberOf loc#Location and
    ?Receiver[address hasValue ?RecLoc] memberOf sho#Receiver and
        ?RecLoc[locatedIn hasValue loc#usa] and
    ?Package[weight hasValue ?Weight] memberOf sho#Package and
    ?Weight[includedIn hasValue sho#heavy].
  postcondition definedBy
   forAll (?O). ?O[from hasValue ?SendLoc,
                   to hasValue ?RecLoc,
                   item hasValue ?Package,
                   price hasValue thePrice] memberOf sho#shipmentOrder.
```

Listing C.10: WSMO Description of Web Service `Weasel`

C.2 SDC Graph Management Evaluation

The following provides information on the technical realization of the evaluation tests for the SDC graph management techniques presented in Section 6.1.3.

The tests have been performed with the SDC prototype implementation. In particular, we tested the `SDC Graph Creator` component and the `Evolution Manager` component (see Appendix B.3 for technical details). The following lists the implementations of the individual tests, the actually generated SDC graphs, and the log-files of the test runs; all these resources are available on the accompanying CD-R.

SDC Graph Creation - Top-Down

- Implementation Class (JUnit Test): `SDCGraphCreationTestTopDown.java`
- created SDC graph (WSML Knowledge Base): `sdcGraphOntology-complete.wsml`
- log-file: `SDCgraphCreation.log`

SDC Graph Creation - Other Insertion Order

- Implementation Class (JUnit Test): `SDCGraphCreationTestOtherOrder.java`
- stepwise created SDC graphs (WSML Knowledge Base): `sdcGraph-partial-1.wsml`, `sdcGraph-partial-2.wsml`, `sdcGraph-partial-3.wsml`
- log-file: `SDCgraphCreationOtherOrder.log`

SDC Graph Maintenance

- Implementation Class (JUnit Test): `SDCGraphMaintenanceTest.java`
- updated SDC graphs (WSML Knowledge Base):
  `sdcGraphOntology-afterGTremoval.wsml`,
  `sdcGraphOntology-afterGTremoval2.wsml`,
  `sdcGraphOntology-afterWSremoval.wsml`,
  `sdcGraphOntology-afterWSinsertion.wsml`
- log-file: `SDCgraphMaintenance.log`

Below, Listing C.11 provides the complete SDC graph ontology that has been generated for the SDC graph management tests discussed in Section 6.1.3. This represents the SDC graph in form of a WSML knowledge base on the basis of the SDC Graph Ontology specified as specified in Listing 5.19 (see Section 5.5.1).

```
wsmlVariant _"http://www.wsmo.org/wsml/wsml-syntax/wsml-core"

namespace {_"http://members.deri.org/~michaels/sdc/swsc-shipment#",
      sdc _"http://members.deri.at/~michaels/ontologies/SDContology.wsml#" ,
      goalTemplates _"http://members.deri.org/~michaels/sdc/swsc-shipment/goals/goaltemplates/",
      webServices _"http://members.deri.org/~michaels/sdc/swsc-shipment/webservices/",
      dc _"http://purl.org/dc/elements/1.1#",
      wsml _"http://www.wsmo.org/wsml/wsml-syntax#" }

ontology _"http://members.deri.org/~michaels/sdc/swsc-shipment/SDCgraphOntology.wsml"
      nonFunctionalProperties
            dc#description hasValue "SDC Graph Ontology for SWSC shipment scenario"
            dc#creator hasValue "generated by org.deri.wsmx.discovery.caching.SDCGraphCreator"
            dc#date hasValue _date(2007,10,11)
      endNonFunctionalProperties

 instance goalTemplate1 memberOf sdc#goalTemplate
       description hasValue goalTemplates#gtRoot
 instance dcArc1 memberOf sdc#discoveryCacheArc
      sourceGT hasValue goalTemplates#gtRoot
      targetWS hasValue webServices#wsMuller
       usability hasValue sdc#subsume
 instance dcArc2 memberOf sdc#discoveryCacheArc
      sourceGT hasValue goalTemplates#gtRoot
      targetWS hasValue webServices#wsRacer
       usability hasValue sdc#subsume
 instance dcArc3 memberOf sdc#discoveryCacheArc
      sourceGT hasValue goalTemplates#gtRoot
      targetWS hasValue webServices#wsRunner
       usability hasValue sdc#subsume
 instance dcArc4 memberOf sdc#discoveryCacheArc
      sourceGT hasValue goalTemplates#gtRoot
      targetWS hasValue webServices#wsWalker
       usability hasValue sdc#subsume
 instance dcArc5 memberOf sdc#discoveryCacheArc
      sourceGT hasValue goalTemplates#gtRoot
      targetWS hasValue webServices#wsWeasel
       usability hasValue sdc#subsume

 instance goalTemplate2 memberOf sdc#goalTemplate
       description hasValue goalTemplates#gtUS2world
 instance ggArc1 memberOf sdc#goalGraphArc
      sourceGT hasValue goalTemplates#gtRoot
      targetGT hasValue goalTemplates#gtUS2world
 instance dcArc6 memberOf sdc#discoveryCacheArc
      sourceGT hasValue goalTemplates#gtUS2world
      targetWS hasValue webServices#wsMuller
       usability hasValue sdc#subsume
 instance dcArc7 memberOf sdc#discoveryCacheArc
      sourceGT hasValue goalTemplates#gtUS2world
```

```
      targetWS hasValue webServices#wsRacer
       usability  hasValue sdc#subsume
instance dcArc8 memberOf sdc#discoveryCacheArc
      sourceGT hasValue goalTemplates#gtUS2world
      targetWS hasValue webServices#wsRunner
       usability  hasValue sdc#subsume
instance dcArc9 memberOf sdc#discoveryCacheArc
      sourceGT hasValue goalTemplates#gtUS2world
      targetWS hasValue webServices#wsWalker
       usability  hasValue sdc#subsume
instance dcArc10 memberOf sdc#discoveryCacheArc
      sourceGT hasValue goalTemplates#gtUS2world
      targetWS hasValue webServices#wsWeasel
       usability  hasValue sdc#subsume

instance goalTemplate3 memberOf sdc#goalTemplate
       description  hasValue goalTemplates#gtUS2AF
instance ggArc2 memberOf sdc#goalGraphArc
      sourceGT hasValue goalTemplates#gtUS2world
      targetGT hasValue goalTemplates#gtUS2AF
instance dcArc11 memberOf sdc#discoveryCacheArc
      sourceGT hasValue goalTemplates#gtUS2AF
      targetWS hasValue webServices#wsMuller
       usability  hasValue sdc#intersect
instance dcArc12 memberOf sdc#discoveryCacheArc
      sourceGT hasValue goalTemplates#gtUS2AF
      targetWS hasValue webServices#wsRacer
       usability  hasValue sdc#intersect
instance dcArc13 memberOf sdc#discoveryCacheArc
      sourceGT hasValue goalTemplates#gtUS2AF
      targetWS hasValue webServices#wsWalker
       usability  hasValue sdc#intersect

instance goalTemplate4 memberOf sdc#goalTemplate
       description  hasValue goalTemplates#gtUS2AS
instance ggArc3 memberOf sdc#goalGraphArc
      sourceGT hasValue goalTemplates#gtUS2world
      targetGT hasValue goalTemplates#gtUS2AS
instance dcArc14 memberOf sdc#discoveryCacheArc
      sourceGT hasValue goalTemplates#gtUS2AS
      targetWS hasValue webServices#wsMuller
       usability  hasValue sdc#intersect
instance dcArc15 memberOf sdc#discoveryCacheArc
      sourceGT hasValue goalTemplates#gtUS2AS
      targetWS hasValue webServices#wsRacer
       usability  hasValue sdc#intersect
instance dcArc16 memberOf sdc#discoveryCacheArc
      sourceGT hasValue goalTemplates#gtUS2AS
      targetWS hasValue webServices#wsRunner
       usability  hasValue sdc#plugin
instance dcArc17 memberOf sdc#discoveryCacheArc
```

sourceGT **hasValue** goalTemplates#gtUS2AS
targetWS **hasValue** webServices#wsWalker
usability **hasValue** sdc#intersect

instance goalTemplate5 **memberOf** sdc#goalTemplate
description **hasValue** goalTemplates#gtUS2EU
instance ggArc4 **memberOf** sdc#goalGraphArc
sourceGT **hasValue** goalTemplates#gtUS2world
targetGT **hasValue** goalTemplates#gtUS2EU
instance dcArc18 **memberOf** sdc#discoveryCacheArc
sourceGT **hasValue** goalTemplates#gtUS2EU
targetWS **hasValue** webServices#wsMuller
usability **hasValue** sdc#intersect
instance dcArc19 **memberOf** sdc#discoveryCacheArc
sourceGT **hasValue** goalTemplates#gtUS2EU
targetWS **hasValue** webServices#wsRacer
usability **hasValue** sdc#intersect
instance dcArc20 **memberOf** sdc#discoveryCacheArc
sourceGT **hasValue** goalTemplates#gtUS2EU
targetWS **hasValue** webServices#wsRunner
usability **hasValue** sdc#plugin
instance dcArc21 **memberOf** sdc#discoveryCacheArc
sourceGT **hasValue** goalTemplates#gtUS2EU
targetWS **hasValue** webServices#wsWalker
usability **hasValue** sdc#intersect

instance goalTemplate6 **memberOf** sdc#goalTemplate
description **hasValue** goalTemplates#gtUS2EUlight
instance ggArc5 **memberOf** sdc#goalGraphArc
sourceGT **hasValue** goalTemplates#gtUS2EU
targetGT **hasValue** goalTemplates#gtUS2EUlight
instance dcArc22 **memberOf** sdc#discoveryCacheArc
sourceGT **hasValue** goalTemplates#gtUS2EUlight
targetWS **hasValue** webServices#wsMuller
usability **hasValue** sdc#plugin
instance dcArc23 **memberOf** sdc#discoveryCacheArc
sourceGT **hasValue** goalTemplates#gtUS2EUlight
targetWS **hasValue** webServices#wsRacer
usability **hasValue** sdc#plugin
instance dcArc24 **memberOf** sdc#discoveryCacheArc
sourceGT **hasValue** goalTemplates#gtUS2EUlight
targetWS **hasValue** webServices#wsWalker
usability **hasValue** sdc#plugin

instance goalTemplate7 **memberOf** sdc#goalTemplate
description **hasValue** goalTemplates#gtUS2NA
instance ggArc6 **memberOf** sdc#goalGraphArc
sourceGT **hasValue** goalTemplates#gtUS2world
targetGT **hasValue** goalTemplates#gtUS2NA
instance dcArc25 **memberOf** sdc#discoveryCacheArc
sourceGT **hasValue** goalTemplates#gtUS2NA

```
      targetWS hasValue webServices#wsMuller
       usability  hasValue sdc#intersect
instance dcArc26 memberOf sdc#discoveryCacheArc
      sourceGT hasValue goalTemplates#gtUS2NA
      targetWS hasValue webServices#wsRacer
       usability  hasValue sdc#intersect
instance dcArc27 memberOf sdc#discoveryCacheArc
      sourceGT hasValue goalTemplates#gtUS2NA
      targetWS hasValue webServices#wsWalker
       usability  hasValue sdc#intersect
instance dcArc28 memberOf sdc#discoveryCacheArc
      sourceGT hasValue goalTemplates#gtUS2NA
      targetWS hasValue webServices#wsWeasel
       usability  hasValue sdc#subsume

instance goalTemplate8 memberOf sdc#goalTemplate
       description  hasValue goalTemplates#gtUS2SA
instance ggArc7 memberOf sdc#goalGraphArc
      sourceGT hasValue goalTemplates#gtUS2world
      targetGT hasValue goalTemplates#gtUS2SA
instance dcArc29 memberOf sdc#discoveryCacheArc
      sourceGT hasValue goalTemplates#gtUS2SA
      targetWS hasValue webServices#wsRacer
       usability  hasValue sdc#intersect
instance dcArc30 memberOf sdc#discoveryCacheArc
      sourceGT hasValue goalTemplates#gtUS2SA
      targetWS hasValue webServices#wsRunner
       usability  hasValue sdc#plugin
instance dcArc31 memberOf sdc#discoveryCacheArc
      sourceGT hasValue goalTemplates#gtUS2SA
      targetWS hasValue webServices#wsWalker
      usability  hasValue sdc#intersect

instance goalTemplate9 memberOf sdc#goalTemplate
       description  hasValue goalTemplates#gtUS2OC
instance ggArc8 memberOf sdc#goalGraphArc
      sourceGT hasValue goalTemplates#gtUS2world
      targetGT hasValue goalTemplates#gtUS2OC
instance dcArc32 memberOf sdc#discoveryCacheArc
      sourceGT hasValue goalTemplates#gtUS2OC
      targetWS hasValue webServices#wsRacer
       usability  hasValue sdc#intersect
instance dcArc33 memberOf sdc#discoveryCacheArc
      sourceGT hasValue goalTemplates#gtUS2OC
      targetWS hasValue webServices#wsRunner
       usability  hasValue sdc#plugin
instance dcArc34 memberOf sdc#discoveryCacheArc
      sourceGT hasValue goalTemplates#gtUS2OC
      targetWS hasValue webServices#wsWalker
       usability  hasValue sdc#intersect
```

```
instance goalTemplate10 memberOf sdc#goalTemplate
    description hasValue goalTemplates#gtNA2NAlight
instance ggArc9 memberOf sdc#goalGraphArc
    sourceGT hasValue goalTemplates#gtRoot
    targetGT hasValue goalTemplates#gtNA2NAlight
instance dcArc39 memberOf sdc#discoveryCacheArc
    sourceGT hasValue goalTemplates#gtNA2NAlight
    targetWS hasValue webServices#wsMuller
    usability hasValue sdc#intersect
instance dcArc40 memberOf sdc#discoveryCacheArc
    sourceGT hasValue goalTemplates#gtNA2NAlight
    targetWS hasValue webServices#wsRacer
    usability hasValue sdc#intersect
instance dcArc41 memberOf sdc#discoveryCacheArc
    sourceGT hasValue goalTemplates#gtNA2NAlight
    targetWS hasValue webServices#wsWalker
    usability hasValue sdc#intersect
instance dcArc42 memberOf sdc#discoveryCacheArc
    sourceGT hasValue goalTemplates#gtNA2NAlight
    targetWS hasValue webServices#wsWeasel
    usability hasValue sdc#intersect

instance goalTemplate11 memberOf sdc#intersectionGoalTemplate
    parent1 hasValue goalTemplates#gtNA2world
    parent2 hasValue goalTemplates#gtNA2NAlight
instance ggArc11 memberOf sdc#goalGraphArc
    sourceGT hasValue goalTemplates#gtNA2NAlight
    targetGT hasValue goalTemplate11
instance ggArc12 memberOf sdc#goalGraphArc
    sourceGT hasValue goalTemplates#gtUS2NA
    targetGT hasValue goalTemplate11
instance dcArc35 memberOf sdc#discoveryCacheArc
    sourceGT hasValue goalTemplate11
    targetWS hasValue webServices#wsWeasel
    usability hasValue sdc#intersect
instance dcArc36 memberOf sdc#discoveryCacheArc
    sourceGT hasValue goalTemplate11
    targetWS hasValue webServices#wsMuller
    usability hasValue sdc#plugin
instance dcArc37 memberOf sdc#discoveryCacheArc
    sourceGT hasValue goalTemplate11
    targetWS hasValue webServices#wsRacer
    usability hasValue sdc#plugin
instance dcArc38 memberOf sdc#discoveryCacheArc
    sourceGT hasValue goalTemplate11
    targetWS hasValue webServices#wsWalker
    usability hasValue sdc#plugin
```

Listing C.11: Complete SDC Graph for Shipment Scenario (WSML Knowledge Base)

C.3 Runtime Web Service Discovery Evaluation

This appendix provides information of the technical realization of the evaluation test for the runtime Web service discovery presented in Section 6.1.3 as well as the statistically prepared results of the comparison tests. The software as well as the original test data are provided on the accompanying CD-R.

The evaluation tests have been performed with the `SDC Runtime Discoverer` component of the SDC prototype implementation that implements the optimized discovery algorithms as specified in Section 5.4 (see Appendix B.3 for the technical specification). The following lists the implementations of the comparison engines and of the comparison tests as well as the respective log-files. Below, we provide the detailed results of each test.

SDC vs. Naive, Single Web Service Discovery:

- Naive Discovery Engine Implementation: `SDCNaiveComparison.java`
- Test Impl. (JUnit Test): `RuntimeDiscoverySingleWSComparisonTester.java`
- log-file: `RunTimeDiscoverySingleWStest.log`

SDC vs. Naive, All Web Service Discovery:

- Naive Discovery Engine Implementation: `SDCNaiveComparisonAllWS.java`
- Test Impl. (JUnit Test): `RuntimeDiscoveryAllWSComparisonTester.java`
- log-file: `RunTimeDiscoveryAllWStest.log`

SDC$_{full}$ vs. SDC$_{light}$:

- SDC$_{light}$ Implementation: `SDCRuntimeDiscoverer.java`
- Test Impl. (JUnit Test): `ComparisonSDClightExtendedScenario.java`
- log-file: `ComparisonTest-SDCvsSDClight.log`

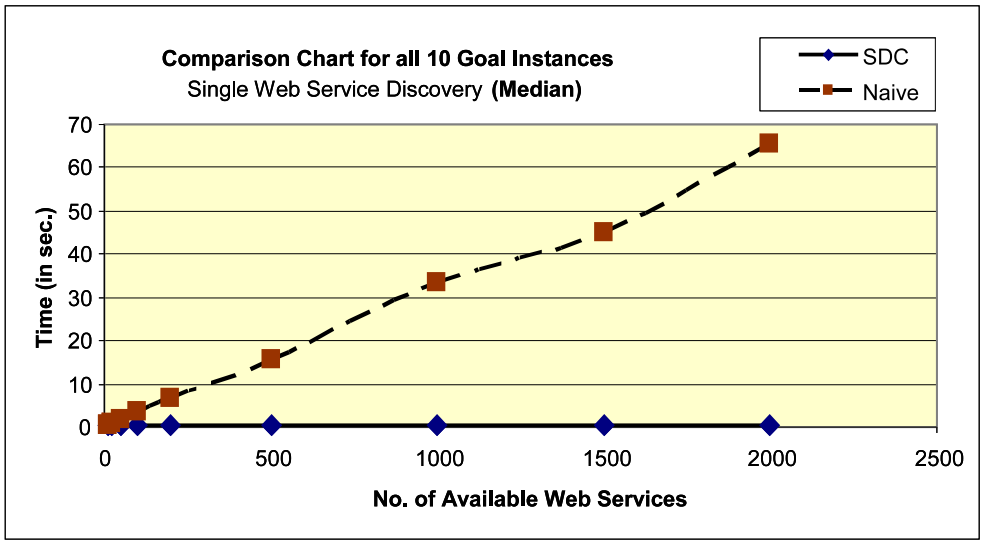
The creation and management of goal instances in handled by the `Goal Instance Manager` component of the SDC prototype. Although goal instances as the runtime data are not stored in our system, we have defined the 10 goal instances that serve as the test data for the runtime comparison tests in form of a WSML knowledge base (filename: `goalInstances4SWSC.wsml`). From this, we load the goal instances into the test implementations in order to actually run the comparison tests. Listing C.12 below shows the goal instance knowledge base, which extends the SDC graph ontology with a concept for goal instance in accordance to the conceptual model presented in Section 3.2.1.

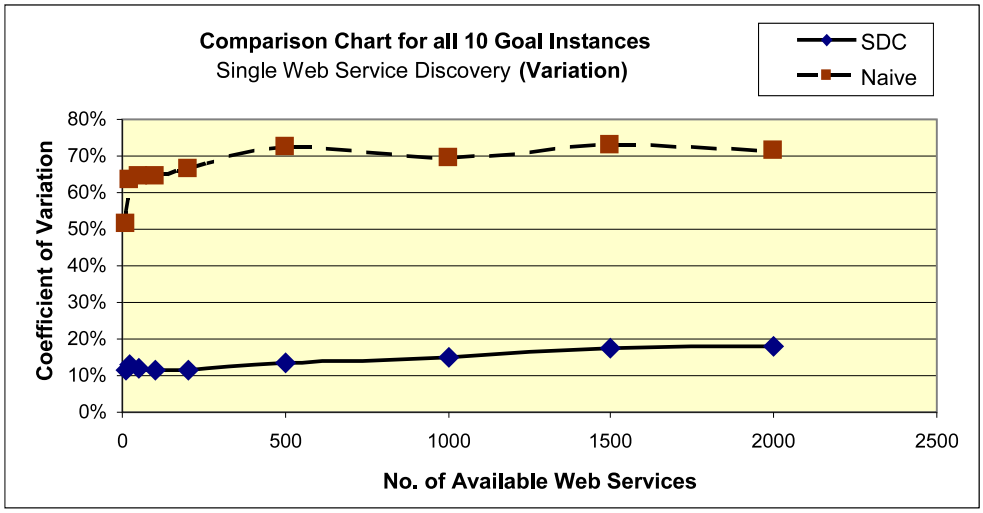
```
wsmlVariant _"http://www.wsmo.org/wsml/wsml-syntax/wsml-core"
namespace {_"http://members.deri.org/~michaels/sdc/swsc-shipment#",
      loc _"http://members.deri.org/~michaels/sdc/swsc-shipment/ontologies/location.wsml#",
      sdc _"http://members.deri.at/~michaels/ontologies/SDContology.wsml#" ,
      goalTemplates _"http://members.deri.org/~michaels/sdc/swsc-shipment/goals/goaltemplates/",
      dc _"http://purl.org/dc/elements/1.1#",
      wsml _"http://www.wsmo.org/wsml/wsml-syntax#" }
ontology _"http://members.deri.org/~michaels/sdc/swsc-shipment/goalinstances.wsml"
      nonFunctionalProperties
            dc#description hasValue "Goal Instances for SWS Shipment Scenario"
            dc#creator hasValue "generated by org.deri.wsmx.discovery.caching.GoalInstanceManager"
            dc#date hasValue _date(2007,03,04)
      endNonFunctionalProperties

 concept goalInstance
      correspondingGoalTemplate impliesType goalTemplate
      inputs impliesType wsml#datatype

 instance goalInstance1 memberOf goalInstance
      correspondingGoalTemplate hasValue goalTemplates#gtUS2AF
      inputs hasValue {loc#california, loc#tunis, 1.0}
 instance goalInstance2 memberOf goalInstance
      correspondingGoalTemplate hasValue goalTemplates#gtRoot
      inputs hasValue {loc#california, loc#luxembourgCity, 1.5}
 instance goalInstance3 memberOf goalInstance
      correspondingGoalTemplate hasValue goalTemplates#gtUS2world
      inputs hasValue {loc#california, loc#tunis, 50.5}
 instance goalInstance4 memberOf goalInstance
      correspondingGoalTemplate hasValue goalTemplates#gtUS2EU
      inputs hasValue {loc#california, loc#bristol, 4.3}
 instance goalInstance5 memberOf goalInstance
      correspondingGoalTemplate hasValue goalTemplates#gtUS2NA
      inputs hasValue {loc#california, loc#newYorkCity, 5.5}
 instance goalInstance6 memberOf goalInstance
      correspondingGoalTemplate hasValue goalTemplates#gtUS2world
      inputs hasValue {loc#california, loc#berlin, 60}
 instance goalInstance7 memberOf goalInstance
      correspondingGoalTemplate hasValue goalTemplates#gtUS2world
      inputs hasValue {loc#california, loc#sydney, 17.3}
 instance goalInstance8 memberOf goalInstance
      correspondingGoalTemplate hasValue goalTemplates#gtUS2SA
      inputs hasValue {loc#california, loc#quito, 7.5}
 instance goalInstance9 memberOf goalInstance
      correspondingGoalTemplate hasValue goalTemplates#gtUS2AS
      inputs hasValue {loc#california, loc#beijing, 57.8}
 instance goalInstance10 memberOf goalInstance
      correspondingGoalTemplate hasValue goalTemplates#gtUS2EUlight
      inputs hasValue {loc#california, loc#amsterdam, 9.99}
```

Listing C.12: Goal Instances for Shipment Scenario (WSML Knowledge Base)

SDC vs. Naive (Single Web Service Discovery)

The following provides the evaluation data of the comparison test between the `SDC Runtime Discoverer` and the naive engine for the discovery of a single Web service. We already provided the aggregated test results in Table 6.9 (see Section 6.1.3). As the most relevant information for this comparison test, the following shows the performance comparison charts for all 10 goal instances with respect to the median time and the coefficient of variation. Below, we provide the statistically prepared test results for the individual goal instances.





Goal Instance 1 (discover one of 3 usable Web services from search space)

SDC Runtime Discoverer

no. WS	arith. mean (in sec)	median (in sec)	min (in sec)	max (in sec)	variance (in msec^2)	stand. deviation (in sec)	coefficient of variation
10	0.11285714	0.109	0.093	0.188	314.6122449	0.017737312	15.72%
20	0.1115	0.109	0.093	0.203	323.13	0.017975817	16.12%
50	0.11246	0.109	0.093	0.203	234.2084	0.015303869	13.61%
100	0.11616	0.11	0.109	0.156	118.5744	0.010889187	9.37%
200	0.125	0.11	0.109	0.219	803.72	0.028349956	22.68%
500	0.12626	0.125	0.109	0.219	478.1124	0.021865781	17.32%
1000	0.12966	0.125	0.109	0.234	274.8644	0.016579035	12.79%
1500	0.14556	0.14	0.125	0.297	935.6464	0.030588338	21.01%
2000	0.15252	0.141	0.125	0.453	3213.8096	0.056690472	37.17%

Naive Engine

no. WS	arith. mean (in sec)	median (in sec)	min (in sec)	max (in sec)	variance (in msec^2)	stand.deviation (in sec)	coefficient of variation
10	0.29976	0.25	0.078	0.797	35511.9024	0.18844602	62.87%
20	0.6147	0.508	0.078	1.938	208179.49	0.456266907	74.23%
50	1.77864	1.7105	0.093	4.031	1384656.55	1.176714303	66.16%
100	3.0873	2.391	0.094	8.593	5279921.09	2.297807888	74.43%
200	5.64736	3.0545	0.078	19.608	27426398.71	5.237021931	92.73%
500	17.12608	15.0305	0.094	46.044	146719239.3	12.11277174	70.73%
1000	31.25436	30.1855	0.344	92.165	566908810.1	23.80984691	76.18%
1500	45.28732	30.7865	0.828	147.067	1445426988	38.01877153	83.95%
2000	56.24786	39.583	1.422	175.487	2542395428	50.42217199	89.64%

Goal Instance 2 (discover one of 4 usable Web services from search space)

SDC Runtime Discoverer

no. WS	arith. mean (in sec)	median (in sec)	min (in sec)	max (in sec)	variance (in msec^2)	stand. deviation (in sec)	coefficient of variation
10	0.67094	0.656	0.625	0.781	1046.844648	0.032354979	4.82%
20	0.68506	0.672	0.625	1.047	4271.8564	0.06535944	9.54%
50	0.67156	0.656	0.625	0.797	1659.7264	0.04073974	6.07%
100	0.67366	0.6565	0.64	0.797	1177.7044	0.034317698	5.09%
200	0.68524	0.672	0.64	0.828	1599.3024	0.039991279	5.84%
500	0.67462	0.672	0.64	0.766	816.8756	0.028581036	4.24%
1000	0.6834	0.672	0.624	0.844	1852.08	0.043035799	6.30%
1500	0.70454	0.672	0.641	1.407	14129.2884	0.118866683	16.87%
2000	0.68118	0.672	0.64	0.797	1156.9076	0.034013344	4.99%

Naive Engine

no. WS	arith. mean (in sec)	median (in sec)	min (in sec)	max (in sec)	variance (in msec^2)	stand.deviation (in sec)	coefficient of variation
10	0.88336	0.8515	0.703	1.39	28242.5504	0.1680552	19.02%
20	1.18666	1.1165	0.718	2.266	136750.5044	0.369797924	31.16%
50	1.76084	1.485	0.734	3.937	732013.8144	0.855578059	48.59%
100	3.38882	3.242	0.75	8.156	3757901.788	1.938530832	57.20%
200	5.02378	4.703	0.766	12.717	10643375.65	3.262418681	64.94%
500	10.48922	7.734	0.75	33.014	69717564.21	8.349704439	79.60%
1000	28.59172	22.0375	0.828	106.57	659784393.8	25.68626858	89.84%
1500	33.92656	23.023	2.052	107.751	781996608.8	27.96420227	82.43%
2000	51.03844	33.076	2.031	162.254	1768918426	42.05851193	82.41%

Goal Instance 3 (discover one of 1 usable Web services from search space)

SDC Runtime Discoverer

no. WS	arith. mean (in sec)	median (in sec)	min (in sec)	max (in sec)	variance (in msec^2)	stand. deviation (in sec)	coefficient of variation
10	0.32526531	0.313	0.297	0.469	1378.399	0.037126796	11.41%
20	0.31928	0.313	0.297	0.422	478.9616	0.021885191	6.85%
50	0.33648	0.313	0.297	0.688	3580.8096	0.059839866	17.78%
100	0.32798	0.328	0.312	0.422	462.0196	0.021494641	6.55%
200	0.33442	0.328	0.312	0.438	980.2436	0.031308842	9.36%
500	0.3472	0.328	0.312	0.812	4866.6	0.069761021	20.09%
1000	0.34028	0.329	0.312	0.437	410.9616	0.020272188	5.96%
1500	0.35066	0.344	0.328	0.532	1559.2644	0.039487522	11.26%
2000	0.36634	0.344	0.328	0.704	3561.5044	0.059678341	16.29%

Naive Engine

no. WS	arith. mean (in sec)	median (in sec)	min (in sec)	max (in sec)	variance (in msec^2)	stand.deviation (in sec)	coefficient of variation
10	0.55926	0.5395	0.094	1.157	98894.4724	0.314474915	56.23%
20	1.27992	1.3845	0.109	2.408	448213.9536	0.669487829	52.31%
50	2.72448	2.6195	0.109	5.912	2945607.13	1.716277113	62.99%
100	6.74048	6.799	0.703	12.363	12168460.69	3.488332078	51.75%
200	13.18636	12.649	1.391	26.533	50271689.63	7.090253143	53.77%
500	27.5141	22.812	0.937	75.655	357211973.1	18.9000522	68.69%
1000	71.98234	74.3735	7.203	121.857	1009630366	31.7746812	44.14%
1500	93.33886	80.9435	2.078	181.808	2691333380	51.87806261	55.58%
2000	117.40564	110.901	0.391	244.951	5246537195	72.43298416	61.69%

Goal Instance 4 (discover one of 2 usable Web services from search space)

SDC Runtime Discoverer

no. WS	arith. mean (in sec)	median (in sec)	min (in sec)	max (in sec)	variance (in msec^2)	stand. deviation (in sec)	coefficient of variation
10	0.01559184	0.015	0	0.125	309.2619742	0.017585846	112.79%
20	0.0162	0.016	0	0.125	273.16	0.016527553	102.02%
50	0.01658	0.016	0	0.125	461.7236	0.021487755	129.60%
100	0.01746	0.016	0	0.125	270.4484	0.016445315	94.19%
200	0.01654	0.016	0	0.047	33.4084	0.00578	34.95%
500	0.01816	0.016	0	0.141	591.1744	0.024314078	133.89%
1000	0.01746	0.0155	0	0.125	514.4084	0.022680573	129.90%
1500	0.01904	0.016	0	0.141	665.5184	0.025797643	135.49%
2000	0.02312	0.016	0	0.328	2362.7456	0.048608082	210.24%

Naive Engine

no. WS	arith. mean (in sec)	median (in sec)	min (in sec)	max (in sec)	variance (in msec^2)	stand.deviation (in sec)	coefficient of variation
10	0.3892	0.352	0.093	0.969	61639.32	0.248272673	63.79%
20	0.74056	0.711	0.093	1.954	262068.3664	0.511926134	69.13%
50	2.5007	2.4695	0.234	5.689	2555616.17	1.598629466	63.93%
100	4.64586	3.947	0.109	13.333	9679545.68	3.111196824	66.97%
200	7.75394	6.08	0.234	20.49	33590662.5	5.795745206	74.75%
500	18.48602	14.121	0.609	57.283	223885820.5	14.96281459	80.94%
1000	44.49564	42.294	3.361	92.841	713497450.4	26.71137305	60.03%
1500	59.13076	46.3745	0.328	164.794	2319156593	48.15762237	81.44%
2000	77.48424	75.3945	9.187	224.395	2923191810	54.06654982	69.78%

Goal Instance 5 (discover one of 4 usable Web services from search space)

SDC Runtime Discoverer

no. WS	arith. mean (in sec)	median (in sec)	min (in sec)	max (in sec)	variance (in msec^2)	stand. deviation (in sec)	coefficient of variation
10	0.22895918	0.219	0.203	0.563	2485.957518	0.049859377	21.78%
20	0.23834	0.234	0.218	0.641	3404.5844	0.058348817	24.48%
50	0.2428	0.234	0.218	0.594	3278.76	0.057260458	23.58%
100	0.24152	0.234	0.218	0.36	607.6896	0.024651361	10.21%
200	0.24244	0.235	0.218	0.281	169.0064	0.013000246	5.36%
500	0.25106	0.25	0.234	0.36	560.6564	0.023678184	9.43%
1000	0.25108	0.25	0.234	0.312	198.1936	0.014078125	5.61%
1500	0.25904	0.25	0.25	0.328	243.7184	0.015611483	6.03%
2000	0.26252	0.265	0.25	0.344	284.5296	0.016868005	6.43%

Naive Engine

no. WS	arith. mean (in sec)	median (in sec)	min (in sec)	max (in sec)	variance (in msec^2)	stand.deviation (in sec)	coefficient of variation
10	0.23112	0.2105	0.093	0.703	25191.6256	0.1587187	68.67%
20	0.47244	0.3755	0.093	1.422	120317.8064	0.346868572	73.42%
50	1.49648	1.445	0.094	4.422	963954.8896	0.981812044	65.61%
100	1.91464	1.6165	0.093	7.468	2735509.83	1.653937674	86.38%
200	5.52624	4.726	0.094	21.139	20944864.22	4.576555935	82.82%
500	11.78104	10.6245	0.218	39.769	88498622.12	9.407370627	79.85%
1000	23.13188	18.397	1.688	73.95	300887400.4	17.3461062	74.99%
1500	40.1553	33.6255	0.453	144.159	1247036569	35.31340495	87.94%
2000	46.35394	42.0245	1.39	143.83	1493547346	38.64644027	83.37%

Goal Instance 6 (discover one of 2 usable Web services from search space)

SDC Runtime Discoverer

no. WS	arith. mean (in sec)	median (in sec)	min (in sec)	max (in sec)	variance (in msec^2)	stand. deviation (in sec)	coefficient of variation
10	0.5404898	0.516	0.5	0.937	4610.290712	0.067899122	12.56%
20	0.53162	0.531	0.5	0.672	930.3556	0.030501731	5.74%
50	0.53156	0.531	0.5	0.656	677.4464	0.026027801	4.90%
100	0.53562	0.531	0.515	0.672	888.8356	0.029813346	5.57%
200	0.54974	0.532	0.515	0.89	3528.5524	0.05940162	10.81%
500	0.54064	0.531	0.515	0.656	612.9104	0.024757027	4.58%
1000	0.5522	0.5465	0.515	0.922	3210.8	0.056663922	10.26%
1500	0.58092	0.547	0.516	0.922	6373.1936	0.079832284	13.74%
2000	0.60094	0.563	0.531	0.953	10790.2164	0.103875966	17.29%

Naive Engine

no. WS	arith. mean (in sec)	median (in sec)	min (in sec)	max (in sec)	variance (in msec^2)	stand.deviation (in sec)	coefficient of variation
10	0.5185	0.5705	0.094	1.094	80855.13	0.284350365	54.84%
20	1.1366	1.008	0.094	2.5	460362.4	0.678500111	59.70%
50	3.0282	2.867	0.219	5.844	2645127.48	1.626384788	53.71%
100	5.74968	5.7185	0.203	11.61	11150342.74	3.339212892	58.08%
200	12.9583	13.234	0.359	23.703	40064712.45	6.329669221	48.85%
500	31.36966	32.9465	2.922	60.095	292608442.9	17.10580144	54.53%
1000	56.52352	50.5715	3.563	163.44	1514250992	38.91337805	68.84%
1500	83.5551	77.142	5.875	182.612	2676860466	51.73838484	61.92%
2000	131.31316	150.894	3.719	248.975	5970983802	77.27214118	58.85%

Goal Instance 7 (discover one of 3 usable Web services from search space)

SDC Runtime Discoverer

no. WS	arith. mean (in sec)	median (in sec)	min (in sec)	max (in sec)	variance (in msec^2)	stand. deviation (in sec)	coefficient of variation
10	0.84122449	0.828	0.796	1.25	4922.745523	0.07016228	8.34%
20	0.8419	0.828	0.735	1.313	6316.01	0.079473329	9.44%
50	0.85138	0.8365	0.797	1.265	4709.6756	0.068627076	8.06%
100	0.8665	0.844	0.812	1.25	6655.49	0.081581187	9.42%
200	0.86126	0.829	0.797	1.25	8959.1524	0.094652799	10.99%
500	0.8581	0.844	0.797	1.187	5358.37	0.073200888	8.53%
1000	0.8822	0.844	0.812	1.5	12153.68	0.11024373	12.50%
1500	0.86792	0.844	0.812	1.86	21356.3536	0.146138132	16.84%
2000	0.86184	0.844	0.812	1.172	3834.3344	0.061922003	7.18%

Naive Engine

no. WS	arith. mean (in sec)	median (in sec)	min (in sec)	max (in sec)	variance (in msec^2)	stand.deviation (in sec)	coefficient of variation
10	0.29532	0.25	0.094	0.875	33619.3776	0.183355877	62.09%
20	0.72626	0.594	0.094	5.625	636375.8724	0.797731705	109.84%
50	1.67348	1.0465	0.094	5.297	2124020.09	1.457401828	87.09%
100	3.46646	3.125	0.14	12.172	7054502.088	2.656031266	76.62%
200	5.6289	4.555	0.109	19.454	19586851.53	4.425703507	78.62%
500	16.04202	11.5005	0.234	54.939	168858004.1	12.99453747	81.00%
1000	26.03778	18.8285	1.359	83.877	426512815.8	20.65218671	79.32%
1500	48.69554	45.392	1.172	133.191	1122212051	33.49943359	68.79%
2000	58.73366	43.642	3.281	188.926	2382483636	48.81069182	83.11%

Goal Instance 8 (discover one of 3 usable Web services from search space)

SDC Runtime Discoverer

no. WS	arith. mean (in sec)	median (in sec)	min (in sec)	max (in sec)	variance (in msec^2)	stand. deviation (in sec)	coefficient of variation
10	0.02587755	0.031	0.015	0.047	75.41357768	0.008684099	33.56%
20	0.0269	0.0235	0.015	0.172	518.25	0.022765105	84.63%
50	0.03184	0.031	0.015	0.188	971.4944	0.031168805	97.89%
100	0.03034	0.031	0.015	0.172	888.1844	0.029802423	98.23%
200	0.02744	0.031	0.015	0.187	580.0064	0.024083322	87.77%
500	0.02438	0.031	0.015	0.032	61.5556	0.007845738	32.18%
1000	0.04302	0.031	0.015	0.422	5552.0596	0.074512144	173.20%
1500	0.03002	0.031	0.015	0.188	576.2196	0.024004575	79.96%
2000	0.02784	0.031	0.015	0.062	69.7744	0.008353107	30.00%

Naive Engine

no. WS	arith. mean (in sec)	median (in sec)	min (in sec)	max (in sec)	variance (in msec^2)	stand.deviation (in sec)	coefficient of variation
10	0.26652	0.2265	0.093	0.625	25349.0096	0.159213723	59.74%
20	0.63098	0.469	0.093	1.954	191615.2596	0.437738803	69.37%
50	1.59292	1.219	0.093	4.829	1377464.434	1.173654308	73.68%
100	3.20008	2.8755	0.109	10.485	4498163.754	2.120887492	66.28%
200	6.945	6.0625	0.25	16.001	19457440.68	4.411058907	63.51%
500	15.64352	12.071	0.125	50.892	166101445.2	12.88803496	82.39%
1000	35.34498	30.5165	1.578	107.69	664823392.7	25.78416942	72.95%
1500	40.12298	32.384	0.125	129.847	795477792.9	28.20421587	70.29%
2000	52.08724	39.298	0.485	187.817	2136735218	46.22483335	88.75%

Goal Instance 9 (discover one of 2 usable Web services from search space)

SDC Runtime Discoverer

no. WS	arith. mean (in sec)	median (in sec)	min (in sec)	max (in sec)	variance (in msec^2)	stand. deviation (in sec)	coefficient of variation
10	0.02553061	0.031	0.015	0.047	67.14702207	0.008194329	32.10%
20	0.0321	0.031	0.015	0.172	914.69	0.030243842	94.22%
50	0.0275	0.031	0.015	0.047	54.65	0.007392564	26.88%
100	0.03152	0.031	0.015	0.157	711.3296	0.026670763	84.62%
200	0.02818	0.031	0.015	0.047	58.9876	0.007680339	27.25%
500	0.03058	0.031	0.015	0.14	299.8036	0.017314838	56.62%
1000	0.0385	0.031	0.015	0.406	3220.81	0.056752181	147.41%
1500	0.02872	0.031	0.015	0.062	110.3616	0.010505313	36.58%
2000	0.05164	0.031	0.015	0.954	17670.9104	0.132931977	257.42%

Naive Engine

no. WS	arith. mean (in sec)	median (in sec)	min (in sec)	max (in sec)	variance (in msec^2)	stand.deviation (in sec)	coefficient of variation
10	0.47244	0.484	0.109	1.172	81797.7664	0.286003088	60.54%
20	0.95972	0.828	0.125	2.25	381264.5616	0.617466243	64.34%
50	2.19534	2.0315	0.109	6.047	2036616.264	1.42710065	65.01%
100	5.18928	4.938	0.235	12	11579663.4	3.402890448	65.58%
200	8.79054	7.7345	0.109	23.532	32325589.53	5.685559737	64.68%
500	21.79904	18.4925	0.625	58.861	259218407.9	16.10026111	73.86%
1000	35.66096	27.721	0.468	108.314	951496707.5	30.84634026	86.50%
1500	60.73036	50.386	0.594	168.088	2425576340	49.25014051	81.10%
2000	90.47256	82.2385	0.125	229.966	3473247853	58.93426722	65.14%

Goal Instance 10 (discover one of 4 usable Web services from search space)

SDC Runtime Discoverer

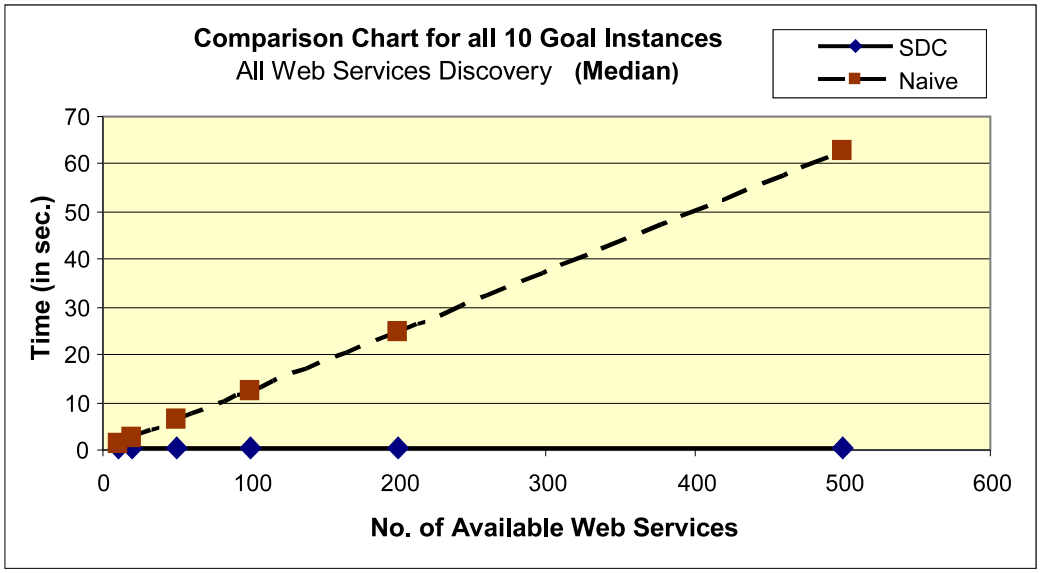
no. WS	arith. mean (in sec)	median (in sec)	min (in sec)	max (in sec)	variance (in msec^2)	stand. deviation (in sec)	coefficient of variation
10	0.02238776	0.016	0.015	0.032	58.93127863	0.007676671	34.29%
20	0.0278	0.031	0.015	0.157	397.88	0.01994693	71.75%
50	0.02432	0.031	0.015	0.032	60.0576	0.007749684	31.87%
100	0.03468	0.031	0.015	0.422	3111.6176	0.055781875	160.85%
200	0.03092	0.031	0.015	0.172	878.9536	0.029647152	95.88%
500	0.0513	0.031	0.015	0.609	10198.37	0.100986979	196.86%
1000	0.02906	0.031	0.015	0.188	568.9364	0.023852388	82.08%
1500	0.0322	0.031	0.015	0.203	1065.04	0.032634951	101.35%
2000	0.03214	0.031	0.015	0.187	822.4404	0.028678222	89.23%

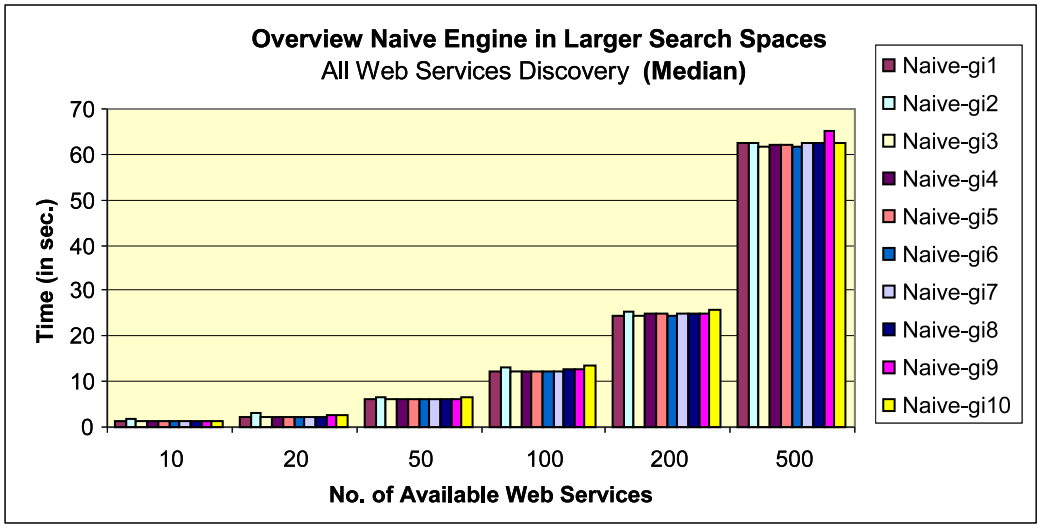
Naive Engine

no. WS	arith. mean (in sec)	median (in sec)	min (in sec)	max (in sec)	variance (in msec^2)	stand.deviation (in sec)	coefficient of variation
10	0.21504	0.1875	0.093	0.656	21003.9184	0.144927287	67.40%
20	0.40372	0.3355	0.093	1.391	82090.5616	0.286514505	70.97%
50	1.25356	1.086	0.109	4.532	825091.3264	0.908345378	72.46%
100	2.24352	2.133	0.094	8.703	2374782.89	1.541033059	68.69%
200	4.62438	3.8985	0.094	12.282	13436127.68	3.665532386	79.27%
500	13.09228	10.782	0.5	36.658	96212509.92	9.808797578	74.92%
1000	23.94354	17.2895	0.734	86.723	450853532.6	21.23331186	88.68%
1500	34.15452	31.9395	1.094	128.445	821473537.8	28.66135966	83.92%
2000	48.4988	38.5105	1.625	123.101	1054707261	32.47625688	66.96%

SDC vs. Naive (All Web Service Discovery)

The following provides the evaluation data of the comparison test between the `SDC Runtime Discoverer` and the naive engine for the discovery of all usable Web services. We already provided the aggregated test results in Table 6.10 (see Section 6.1.3). As the most relevant information for this comparison test, the following shows the overall performance comparison chart. We also provide a comprehensive overview of the naive engine which shows that its behavior is actually independent of the actual goal instance because it always needs to inspect all available Web services. Below, we provide the statistically prepared test results for the individual goal instances.





Goal Instance 1 (discover all 3 usable Web services from search space)

SDC Runtime Discoverer

no. WS	arith. mean (in sec)	median (in sec)	min (in sec)	max (in sec)	variance (in msec^2)	stand. deviation (in sec)	coefficient of variation
10	0.33983333	0.297	0.281	0.5	4291.555556	0.065509965	19.28%
20	0.31688	0.297	0.281	0.454	2489.3856	0.049893743	15.75%
50	0.31188	0.297	0.281	0.453	1700.1056	0.041232337	13.22%
100	0.30752	0.297	0.281	0.453	1578.7296	0.03973323	12.92%
200	0.31248	0.297	0.281	0.672	5473.8496	0.073985469	23.68%
500	0.32072	0.297	0.282	0.484	2406.7616	0.049058757	15.30%

Naive Engine

no. WS	arith. mean (in sec)	median (in sec)	min (in sec)	max (in sec)	variance (in msec^2)	stand.deviation (in sec)	coefficient of variation
10	1.23316	1.156	1.109	2.36	59361.9744	0.243643129	19.76%
20	2.40876	2.375	2.297	2.516	5845.7824	0.076457716	3.17%
50	6.16756	6.109	6.016	7.266	54467.6064	0.233382961	3.78%
100	12.24712	12.219	12.11	12.531	10677.7856	0.103333371	0.84%
200	26.3744	24.563	24.251	60.455	51876569.2	7.20253908	27.31%
500	62.52396	62.517	61.673	65.033	431599.5584	0.656962372	1.05%

Goal Instance 2 (discover all 4 usable Web services from search space)

SDC Runtime Discoverer

no. WS	arith. mean (in sec)	median (in sec)	min (in sec)	max (in sec)	variance (in msec^2)	stand. deviation (in sec)	coefficient of variation
10	0.71624	0.688	0.656	0.844	4215.0624	0.064923512	9.06%
20	0.70376	0.687	0.656	0.844	2639.3824	0.05137492	7.30%
50	0.70688	0.688	0.672	0.86	2254.5056	0.047481634	6.72%
100	0.71376	0.703	0.687	0.875	2213.7024	0.047049999	6.59%
200	0.7158	0.703	0.687	0.891	2293.76	0.047893215	6.69%
500	0.74004	0.703	0.687	0.891	5031.3184	0.070931787	9.58%

Naive Engine

no. WS	arith. mean (in sec)	median (in sec)	min (in sec)	max (in sec)	variance (in msec^2)	stand.deviation (in sec)	coefficient of variation
10	1.79316	1.781	1.718	1.953	3770.1344	0.06140142	3.42%
20	3.01312	2.984	2.906	3.141	6248.7456	0.079049008	2.62%
50	6.70768	6.719	6.501	6.829	5586.9376	0.07474582	1.11%
100	12.94832	12.953	12.782	13.141	11192.9376	0.10579668	0.82%
200	25.42416	25.438	24.953	26.125	64034.8544	0.25305109	1.00%
500	62.77916	62.423	61.845	71.767	3478501.974	1.865074254	2.97%

Goal Instance 3 (discover the only usable Web service from search space)

SDC Runtime Discoverer

no. WS	arith. mean (in sec)	median (in sec)	min (in sec)	max (in sec)	variance (in msec^2)	stand. deviation (in sec)	coefficient of variation
10	0.39056	0.391	0.375	0.407	59.1264	0.007689369	1.97%
20	0.40504	0.391	0.375	0.579	2702.7584	0.05198806	12.84%
50	0.39944	0.391	0.375	0.579	1420.9664	0.037695708	9.44%
100	0.4126	0.391	0.39	0.594	2750	0.052440442	12.71%
200	0.40128	0.406	0.39	0.422	130.5216	0.011424605	2.85%
500	0.41816	0.406	0.39	0.594	3137.5744	0.056014055	13.40%

Naive Engine

no. WS	arith. mean (in sec)	median (in sec)	min (in sec)	max (in sec)	variance (in msec^2)	stand.deviation (in sec)	coefficient of variation
10	1.22944	1.188	1.172	1.375	4906.8864	0.070049171	5.70%
20	2.44252	2.406	2.359	2.579	6721.2096	0.081982984	3.36%
50	6.17392	6.172	5.984	6.797	20469.1936	0.14307059	2.32%
100	12.29532	12.297	12.125	12.485	10611.3376	0.103011347	0.84%
200	24.74364	24.703	24.516	25.469	41332.2304	0.203303297	0.82%
500	61.89524	61.829	61.47	62.658	103721.7824	0.322058663	0.52%

Goal Instance 4 (discover all 2 usable Web services from search space)

SDC Runtime Discoverer

no. WS	arith. mean (in sec)	median (in sec)	min (in sec)	max (in sec)	variance (in msec^2)	stand. deviation (in sec)	coefficient of variation
10	0.51004	0.5	0.484	0.688	1391.1584	0.037298236	7.31%
20	0.54112	0.515	0.5	0.703	3854.1056	0.062081443	11.47%
50	0.52328	0.516	0.5	0.672	1119.0816	0.033452677	6.39%
100	0.53432	0.516	0.5	0.687	3049.1776	0.055219359	10.33%
200	0.52436	0.516	0.5	0.688	1257.2704	0.035458009	6.76%
500	0.55188	0.516	0.5	1.172	17221.5456	0.131230887	23.78%

Naive Engine

no. WS	arith. mean (in sec)	median (in sec)	min (in sec)	max (in sec)	variance (in msec^2)	stand.deviation (in sec)	coefficient of variation
10	1.20064	1.172	1.156	1.391	4901.0304	0.07000736	5.83%
20	2.41696	2.391	2.344	2.578	5615.2384	0.074934894	3.10%
50	6.16196	6.187	6.016	6.25	5446.2784	0.073798905	1.20%
100	12.38836	12.329	12.25	13.016	22116.3104	0.148715535	1.20%
200	24.75756	24.719	24.516	25.422	42576.9664	0.206341868	0.83%
500	61.9808	61.939	61.595	62.781	103622.24	0.321904085	0.52%

Goal Instance 5 (discover all 4 usable Web services from search space)

SDC Runtime Discoverer

no. WS	arith. mean (in sec)	median (in sec)	min (in sec)	max (in sec)	variance (in msec^2)	stand. deviation (in sec)	coefficient of variation
10	0.47432	0.453	0.437	0.64	3389.1776	0.058216644	12.27%
20	0.46692	0.453	0.437	0.625	2311.4336	0.048077371	10.30%
50	0.4832	0.453	0.437	0.657	4769.12	0.069058816	14.29%
100	0.46808	0.468	0.437	0.625	1165.9936	0.034146648	7.30%
200	0.4956	0.468	0.453	0.938	10554.16	0.102733441	20.73%
500	0.46688	0.469	0.453	0.485	84.1056	0.009170911	1.96%

Naive Engine

no. WS	arith. mean (in sec)	median (in sec)	min (in sec)	max (in sec)	variance (in msec^2)	stand.deviation (in sec)	coefficient of variation
10	1.14876	1.125	1.109	1.328	4573.1424	0.067625013	5.89%
20	2.41352	2.359	2.281	3.015	20821.3696	0.144296118	5.98%
50	6.08824	6.125	5.937	6.188	6571.4624	0.081064557	1.33%
100	12.38576	12.375	12.156	12.937	33475.8624	0.182964101	1.48%
200	25.00168	24.735	24.469	29.407	868604.6976	0.931989645	3.73%
500	62.35452	62.204	61.907	63.329	139178.6496	0.373066548	0.60%

Goal Instance 6 (discover all 2 usable Web services from search space)

SDC Runtime Discoverer

no. WS	arith. mean (in sec)	median (in sec)	min (in sec)	max (in sec)	variance (in msec^2)	stand. deviation (in sec)	coefficient of variation
10	0.92004	0.891	0.875	1.094	3568.1184	0.059733729	6.49%
20	0.95936	0.891	0.875	1.546	20374.3104	0.142738609	14.88%
50	0.92688	0.906	0.875	1.094	3611.0656	0.060092143	6.48%
100	0.94932	0.907	0.875	1.531	16737.5776	0.12937379	13.63%
200	0.94828	0.922	0.891	1.109	5431.2416	0.073696958	7.77%
500	0.93688	0.906	0.89	1.125	4807.9456	0.069339351	7.40%

Naive Engine

no. WS	arith. mean (in sec)	median (in sec)	min (in sec)	max (in sec)	variance (in msec^2)	stand.deviation (in sec)	coefficient of variation
10	1.21636	1.188	1.172	1.453	5103.5104	0.071438858	5.87%
20	2.40812	2.39	2.359	2.563	3572.5056	0.059770441	2.48%
50	6.13872	6.172	5.953	6.828	30300.6816	0.17407091	2.84%
100	12.25392	12.235	12.14	12.438	5754.3136	0.075857192	0.62%
200	24.67192	24.596	24.407	25.377	68226.3136	0.261201672	1.06%
500	61.88704	61.861	61.33	62.517	111151.4784	0.333393879	0.54%

Goal Instance 7 (discover all 3 usable Web services from search space)

SDC Runtime Discoverer

no. WS	arith. mean (in sec)	median (in sec)	min (in sec)	max (in sec)	variance (in msec^2)	stand. deviation (in sec)	coefficient of variation
10	1.03384	1	0.984	1.375	7617.4944	0.087278258	8.44%
20	1.02424	1	0.984	1.172	3042.5024	0.055158883	5.39%
50	1.02308	1	0.984	1.25	3406.8736	0.05836843	5.71%
100	1.03876	1.015	0.984	1.203	4555.6224	0.067495351	6.50%
200	1.05252	1.016	0.984	1.406	9049.6096	0.095129436	9.04%
500	1.03372	1.015	0.984	1.203	3187.8816	0.056461328	5.46%

Naive Engine

no. WS	arith. mean (in sec)	median (in sec)	min (in sec)	max (in sec)	variance (in msec^2)	stand.deviation (in sec)	coefficient of variation
10	1.18392	1.157	1.14	1.376	4857.9136	0.069698735	5.89%
20	2.44812	2.39	2.343	3.016	21333.3056	0.146059254	5.97%
50	6.16576	6.172	5.937	6.344	9197.0624	0.095901316	1.56%
100	12.44592	12.406	12.188	13.016	45387.2736	0.213042891	1.71%
200	24.996	24.953	24.61	25.61	85638.88	0.292641214	1.17%
500	62.87256	62.392	61.673	72.345	4208540.326	2.051472721	3.26%

Goal Instance 8 (discover all 3 usable Web services from search space)

SDC Runtime Discoverer

no. WS	arith. mean (in sec)	median (in sec)	min (in sec)	max (in sec)	variance (in msec^2)	stand. deviation (in sec)	coefficient of variation
10	0.30812	0.297	0.281	0.484	1353.3856	0.036788389	11.94%
20	0.3308	0.313	0.297	0.641	5325.36	0.072975064	22.06%
50	0.32688	0.312	0.297	0.516	3216.2656	0.056712129	17.35%
100	0.33244	0.313	0.297	0.5	2625.2064	0.051236768	15.41%
200	0.33248	0.313	0.297	0.516	2769.4496	0.05262556	15.83%
500	0.32396	0.313	0.297	0.516	1634.8384	0.040433135	12.48%

Naive Engine

no. WS	arith. mean (in sec)	median (in sec)	min (in sec)	max (in sec)	variance (in msec^2)	stand.deviation (in sec)	coefficient of variation
10	1.17376	1.157	1.125	1.36	3243.4624	0.056951404	4.85%
20	2.5912	2.406	2.343	5.531	386378.96	0.621593887	23.99%
50	6.19248	6.172	5.968	6.844	35215.5296	0.187658012	3.03%
100	12.52412	12.5	12.188	13.141	39081.7856	0.197691137	1.58%
200	25.088	25.032	24.751	25.703	72919.2	0.270035553	1.08%
500	62.60312	62.658	61.845	63.236	162737.5456	0.403407419	0.64%

Goal Instance 9 (discover all 2 usable Web services from search space)

SDC Runtime Discoverer

no. WS	arith. mean (in sec)	median (in sec)	min (in sec)	max (in sec)	variance (in msec^2)	stand. deviation (in sec)	coefficient of variation
10	1.2246	0.469	0.453	0.5	100.3424	0.010017105	0.82%
20	0.48384	0.469	0.468	0.657	1329.5744	0.03646333	7.54%
50	0.47868	0.484	0.468	0.485	55.3376	0.007438925	1.55%
100	0.4806	0.484	0.469	0.5	100.24	0.010011993	2.08%
200	0.57864	0.485	0.468	1.125	40299.0304	0.200746184	34.69%
500	0.50876	0.5	0.484	0.688	1474.1824	0.038395083	7.55%

Naive Engine

no. WS	arith. mean (in sec)	median (in sec)	min (in sec)	max (in sec)	variance (in msec^2)	stand.deviation (in sec)	coefficient of variation
10	1.2246	1.188	1.172	1.828	16658.56	0.129068044	10.54%
20	2.47132	2.422	2.375	2.64	10028.2176	0.100140989	4.05%
50	6.23748	6.203	5.953	7.047	58252.8896	0.241356354	3.87%
100	12.60428	12.47	12.142	15.172	332393.6416	0.576535898	4.57%
200	25.13124	24.955	24.532	26.188	219573.0624	0.468586238	1.86%
500	65.4042	65.064	63.564	69.784	1597720.96	1.264009873	1.93%

Goal Instance 10 (discover all 4 usable Web services from search space)

SDC Runtime Discoverer

no. WS	arith. mean (in sec)	median (in sec)	min (in sec)	max (in sec)	variance (in msec^2)	stand. deviation (in sec)	coefficient of variation
10	0.19944	0.187	0.172	0.391	2757.8464	0.052515202	26.33%
20	0.19572	0.188	0.172	0.375	1407.4016	0.037515352	19.17%
50	0.20884	0.188	0.172	0.407	3216.2144	0.056711678	27.16%
100	0.20188	0.188	0.187	0.39	1532.2656	0.039144164	19.39%
200	0.20444	0.203	0.187	0.407	1766.5664	0.042030541	20.56%
500	0.22696	0.203	0.187	0.609	7737.0784	0.087960664	38.76%

Naive Engine

no. WS	arith. mean (in sec)	median (in sec)	min (in sec)	max (in sec)	variance (in msec^2)	stand.deviation (in sec)	coefficient of variation
10	1.19936	1.172	1.125	1.437	4603.8304	0.067851532	5.66%
20	2.55	2.5	2.437	3.078	20259.2	0.142334817	5.58%
50	6.47824	6.453	6.265	7.172	37430.1024	0.193468608	2.99%
100	13.97968	13.578	13.048	18.594	1942337.818	1.393677803	9.97%
200	26.0306	25.969	24.516	28.36	950322.8	0.974845013	3.74%
500	62.887	62.673	61.517	65.267	853912.8	0.924074023	1.47%

$\mathbf{SDC}_{full}$ vs. $\mathbf{SDC}_{light}$

The following provides additional information for the comparison test between the full `SDC Runtime Discoverer` and the SDC_{light} engine.

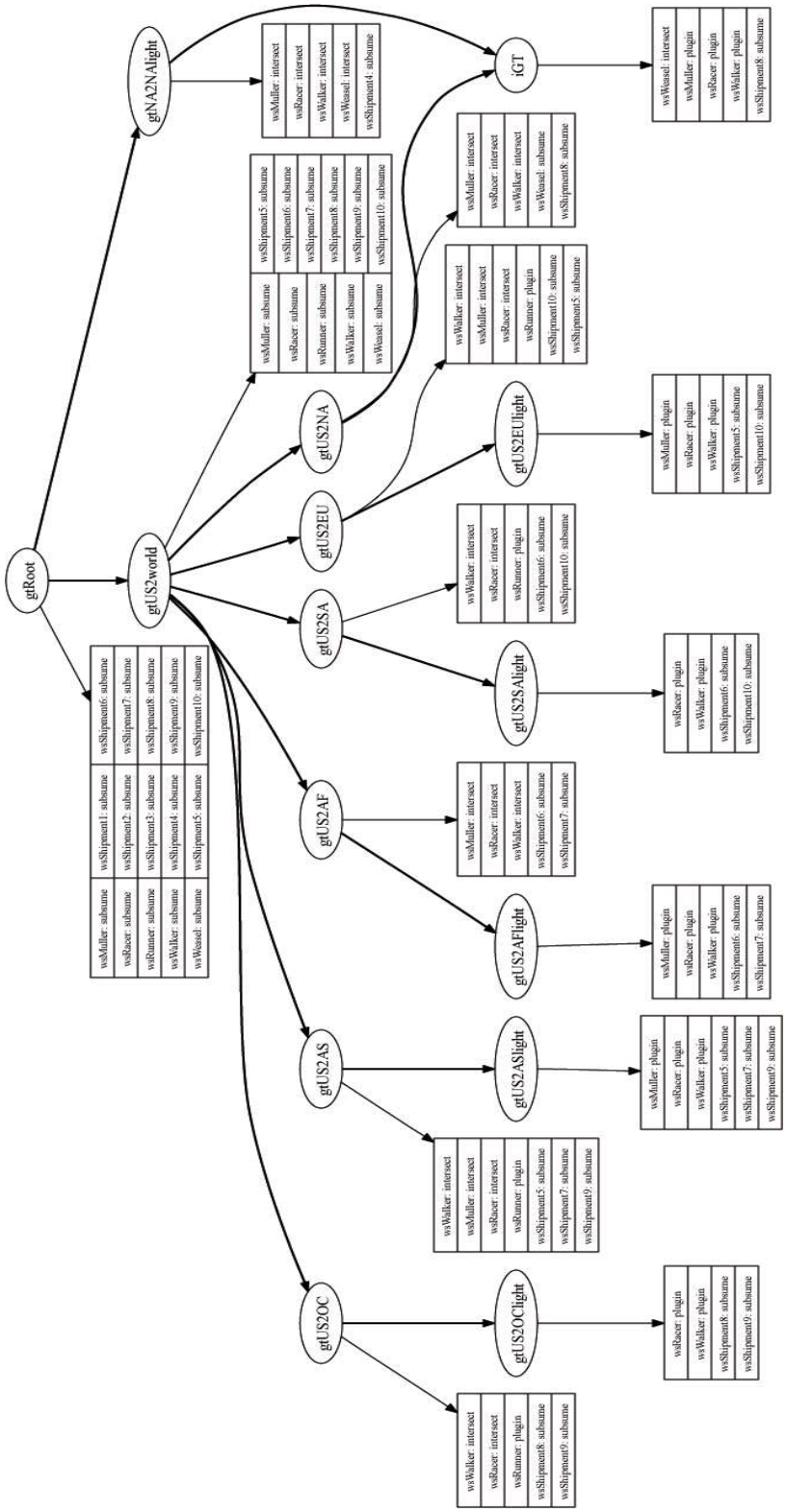
SDC_{light} Specification. Listing C.13 shows the runtime discovery algorithms which are implemented in the SDC_{light} engine. Essentially, these are exactly the same as the ones used by the `SDC Runtime Discoverer` but without the *refinement*-method. We refer to Section 5.4.1 for the detailed specification of the SDC runtime discovery algorithms. The SDC_{light} light engine is implemented as an extension of the `SDC Runtime Discoverer` component in the SDC prototype implementation.

```
discoverSingleWS−light(GI(G,b)) {
    if ( instantiationCheck (GI(G,b)) = false ) then return " invalid goal instantiation ";
    if ( lookup(G) = W ) then return W;
    if ( checkOtherWS(G,b) = W ) then return W;
    else { return "no Web service found"; }
}
discoverAllWS−light(GI(G,b)) {
    if ( instantiationCheck (GI(G,b)) = false ) then return " invalid goal instantiation ";
    result = lookupAllWS(G);
    forall ( (G,W) in discoverycache and degree(G,W) = subsume ) {
        if ( satisfiable (W,b) ) then result = + W; }
    forall ( (G,W) in discoverycache and degree(G,W) = intersect ) {
        if ( satisfiable (G,W,b) ) then result = + W; }
    if ( result = empty ) then return "no Web service found";
    else return result ;
}
```

Listing C.13: Discovery Algorithms for SDC_{light}

Extended Shipment Scenario. For the comparison test, we use an extended version of the shipment scenario for the comparison test with defines 10 additional Web services and 4 additional goal templates (see Table 6.11 in Section 6.1.3). We have created the SDC graph for this with the `SDC Graph Creator` component of our prototype. The below figure shows the graphical representation of the created SDC graph, which correctly defines the relevant relations in accordance to Figure 6.7. The relevant technical sources are:

- SDC Graph Creation Class: `SDCGraphGenerationExtendedSWSCscenario.java`
- created SDC graph (WSML Knowledge Base): `sdcGraph4extendedSWSC.wsml`
- log-file: `SDCGraphGenerationShipmentSceanrioExtended.log`


gtRoot
gtNA2NAlight
gtUS2world
gtUS2NA
gtUS2EU
gtUS2SA
gtUS2AF
gtUS2AS
gtUS2OC
iGT
gtUS2EUlight
gtUS2SAlight
gtUS2AFlight
gtUS2ASlight
gtUS2OClight
wsMuller: subsume
wsRacer: subsume
wsRunner: subsume
wsWalker: subsume
wsWeasel: subsume
wsShipment1: subsume
wsShipment2: subsume
wsShipment3: subsume
wsShipment4: subsume
wsShipment5: subsume
wsShipment6: subsume
wsShipment7: subsume
wsShipment8: subsume
wsShipment9: subsume
wsShipment10: subsume
wsMuller: intersect
wsRacer: intersect
wsWalker: intersect
wsWeasel: intersect
wsShipment4: subsume
wsRunner: plugin
wsMuller: plugin
wsRacer: plugin
wsWalker: plugin
wsShipment5: subsume
wsShipment6: subsume
wsShipment7: subsume
wsShipment8: subsume
wsShipment9: subsume
wsShipment10: subsume

Goal Instance	corresponding Goal Template	Input Binding			most approp. Goal Template	usable Web Services
		sender	receiver	weight		
gi1	gtUS2AF	San Francisco	Tunis	1 kg	gtUS2AFlight	Muller Racer Walker
gi2	gtRoot	Los Angeles	Luxembourg City	1.5 kg	gtUS2EUlight	Muller Racer Runner Walker
gi3	gtUS2world	Berkeley	Tunis	50.5 kg	gtUS2AF	Racer
gi4	gtUS2EU	Paolo Alto	Bristol	4.3 kg	gtUS2EUlight	Muller Racer Runner Walker
gi5	gtUS2NA	Los Angeles	New York City	5.5 kg	iGT	Muller Racer Walker Weasel
gi6	gtUS2world	Monterey	Berlin	60 kg	gtUS2EU	Racer Runner
gi7	gtUS2world	Santa Barbara	Sydney	17.3 kg	gtUS2OC	Racer Runner Walker
gi8	gtUS2SA	San Francisco	Quito	7.58 kg	gtUS2SAlight	Racer Runner Walker
gi9	gtUS2AS	Stanford	Beijing	57.8 kg	gtUS2AS	Racer Runner
gi10	gtUS2EUlight	San Francisco	Amsterdam	9.99 kg	gtUS2EUlight	Muller Racer Runner Walker

The above table shows the 10 goal instances that we use for the comparison test. These are the same as defined in Table 6.3 (see Section 6.1.2), i.e. the ones that we also used for the comparison test of the `SDC Runtime Discoverer` with the naive discovery engine. We have defined the additional Web services such that none of them is actually usable for any of the goal instances In consequence, also the usable Web services for each goal instance are exactly the same as before; the mere difference is that for goal instances `gi1` and `gi8` now the most appropriate corresponding goal template is one of the additionally defined ones.

Comparison Test Results. Below, we provide the the statistically prepared results of 100 repetitive test runs for both flavors of runtime Web service discovery. We here show the arithmetic mean and the median times, the standard deviation and variation coefficient measured for the individual goal instance as well as the aggregated data for all of them.

Goal Instance	Statistcal Notion	Single WS Discovery		All WS Discovery	
		SDC-full (in msec)	SDC-light (in msec)	SDC-full (in msec)	SDC-light (in msec)
gi1	arithm. mean	149.6836735	379.88	393.13	555.54
	median	141	360	375	546.5
	stand. deviation	40.70641451	62.27251079	68.48775876	58.12562602
	coeff. of variation	27.19%	16.39%	17.42%	10.46%
gi2	arithm. mean	598.74	1305.07	931.84	1821.48
	median	578	1265	875	1718
	stand. deviation	66.52302158	107.1120213	241.2075753	573.2451566
	coeff. of variation	11.11%	8.21%	25.89%	31.47%
gi3	arithm. mean	904.03	1015.53	961.2	1389.97
	median	875	969	937	1313
	stand. deviation	103.0761325	245.9792859	101.2392216	326.479416
	coeff. of variation	11.40%	24.22%	10.53%	23.49%
gi4	arithm. mean	33.02	46.4	553.64	665.77
	median	31	47	531.5	641
	stand. deviation	7.62755531	8.435638684	63.3994511	68.85301083
	coeff. of variation	23.10%	18.18%	11.45%	10.34%
gi5	arithm. mean	376.76	400.76	782.69	705.75
	median	359	391	750	672
	stand. deviation	79.3653728	56.98019305	82.64510814	101.8225294
	coeff. of variation	21.07%	14.22%	10.56%	14.43%
gi6	arithm. mean	555.43	1085.28	1205.3	1365.67
	median	516	1047	1195.5	1289.5
	stand. deviation	219.9001708	300.2151588	200.5148124	527.2860524
	coeff. of variation	39.59%	27.66%	16.64%	38.61%
gi7	arithm. mean	617.18	977.34	1081.48	1338.09
	median	601.5	954	1109	1313
	stand. deviation	209.9271483	71.41753566	231.1231914	75.53967103
	coeff. of variation	34.01%	7.31%	21.37%	5.65%
gi8	arithm. mean	52.07	67.2	678.01	630.15
	median	47	63	698.5	625
	stand. deviation	14.97147621	15.02198389	114.2068733	114.4463521
	coeff. of variation	28.75%	22.35%	16.84%	18.16%
gi9	arithm. mean	56.29	76.33	969.84	962.65
	median	62	78	953	969
	stand. deviation	14.07074625	17.78822925	96.49473768	84.93390077
	coeff. of variation	25.00%	23.30%	9.95%	8.82%
gi10	arithm. mean	41.7	65.27	567.7	529.21
	median	47	62	554.5	523.5
	stand. deviation	8.91908067	22.89185663	80.30909039	97.98900908
	coeff. of variation	21.39%	35.07%	14.15%	18.52%

TOTAL (aggregated in msec)		Single WS Discovery		All WS Discovery	
		SDC-full	SDC-light	SDC-full	SDC-light
all gi	arithm. mean	338.4903673	541.906	812.483	996.428
all gi	median	325.75	523.6	797.9	961.05
all gi	stand. deviation	76.5087119	90.8114414	127.962782	202.8720724
all gi	coeff. variation	24.26%	19.69%	15.48%	17.99%