Universität Zürich
Institut für Informatik

Dynamic and Distributed Information Systems

Default Inheritance for OWL-S

Extending the OWL-S (Web Ontology Language for Services) with default logic

Diploma Thesis in Informatics

Author and submitted by

- Simon Ferndriger
  Dielsdorf, Switzerland, Student ID: 02-728-384

Supervised by

- Prof. Abraham Bernstein, PhD
- A/P Dr. Jin Song Dong, PhD

Supported by

- Yuan Fang Li, PhD

Participating Universities

- University of Zurich
  Department of Informatics (IFI), Binzmühlestrasse 14, CH-8050 Zürich, Switzerland
- National University of Singapore
  Department of Computer Science, School of Computing, 21 Lower Kent Ridge Road, Singapore 119077

This thesis was submitted on October 11, 2007.

Abstract

Currently proposed Web Service Technologies allow describing services syntactically and semantically such that users and software agents are able to discover, invoke, compose and monitor these services with a high degree of automation. Thereby, the services can be connected with an ontology-based semantic description. Up to the present however, none of these standards defines a concrete and self-contained way of connecting these services among each other.

This thesis demonstrates how web service creation and web service discovery can benefit from such connections among services and how these benefits can be accomplished by introducing Inheritance Relationships (IR) for OWL-S (OWL-S: Semantic Markup for Web Services, 2004) using ideas from computer science about inheritance. For service creation, this thesis provides the possibility to share specific elements among these services. This sharing is expected to substantially reduce the amount of work necessary for creating and maintaining services. For service discovery, an interpretation of IRs among these services is provided in order to discover service substitutes. These substitutes increase the choice of a service user or the availability of a specific service.

Together with the developed prototype, the thesis demonstrates the basic feasibility of applying inheritance for OWL-S by illustrating several use cases. In addition, the thesis provides a basis for further tool development.

Contents

1 Introduction 7

1.1 Current situation 7

1.2 Motivation 7

1.3 Requirements for Inheritance Relationships (IR) 10

1.4 Assignment 11

1.5 Succinct chapter overview 11

2 Technical background 12

2.1 OWL-S: Web Ontology Language for Services 12

2.2 SWSF: Semantic Web Service Framework 15

2.3 WSDL-S 15

2.4 WSMO 16

3 Use cases 17

3.1 Strict IR 17

3.2 Normal IR 21

4 Definition of inheritance 26

4.1 Inheritance in computer science 26

4.2 Inheritance in object-oriented programming 27

4.3 Inheritance in knowledge representation 28

4.4 Inheritance for processes 28

4.5 Inheritance in OWL-S 29

5 Solution 31

5.1 Applications of IRs 31

5.2 Types of IRs 34

5.3 IR application map 38

5.4 Create Inheritance Relationships (IR) 38

5.5 Interpreting the IR 45

5.6 Validating the IR 52

5.7 Chapter summary 54

6 OWL-syntax 55

7 Dismissed approaches 56

8 Design decisions 57

9 Evaluation 58
9.1 Prototype 58
9.2 Accomplish the motivating tasks 61
9.3 Comparison with other solution approaches 75

10 Discussion 77
10.1 Multiple-inheritance 77
10.2 Effects and side effects of the solution 77
10.3 Limitations of the solution 79
10.4 Related work 80
10.5 Personal opinion 82

11 Conclusions 85
11.1 Accomplishments 85
11.2 Outlook 85

12 Acknowledgements 88

13 References 89

14 Glossary 92

15 Attachments 93
15.1 Online version 93
15.2 Abstract 93
15.3 CD-ROM 93

Tables

Table 1: Transferring inheritance into the domain of OWL-S 29
Table 2: Enhanced information reuse for OWL-S 29
Table 3: Additional information for OWL-S 30
Table 4: Transferring Overriding to the domain of OWL-S 32
Table 5: Benefits of IR for Service Customization 33
Table 6: Transferring Subtyping to OWL-S 33
Table 7: Service Enlargement, enabled by inheritance 34
Table 8: Enhanced altering within Service Manipulation 34

Figures

Figure 1-A: The comparison of Web Service Customization, Extension and Manipulation 8
Figure 1-B: The requirements list to provide IR among services 10
Figure 2-A: OWL-S enabled automation of Web Service tasks 12
Figure 2-B: OWL-S upper ontology for Web Services and their semantic descriptions 13
Figure 2-C: The OWL-S class ServiceProfile and its hierarchy 14
Figure 2-D: OWL-S class Expression is used to define logical formalism (rules) 14
Figure 3-A: Interaction between the CongoBuy service and a customer 25
Figure 5-A: Refinement in object-oriented programming for an operation 36
Figure 5-B: Refinement in OWL-S for strict IR between services 37
Figure 5-C: Applications of Inheritance Relationships 38
Figure 5-D: The proposed inheritance profile − the vocabulary to create IRs and their specification . 39
Figure 5-E: Downward propagation of changes within the inheritance chain 51
Figure 6-A: The main instances used to model the IR for the use case CharlyAir 55
Figure 9-A: Prototype architecture 58
Figure 9-B: XML Schema built-in data type hierarchy 60
Figure 9-C: Screenshot of prototype home page 61
Figure 9-D: Provide a new service name 62
Figure 9-E: Select ExpressCongoBuyService as SuperService 62
Figure 9-F: Adopting the service model from ExpressCongoBuy 63
Figure 9-G: Replacing the positive result of the ExpressCongoBuy service 63
Figure 9-H: Auxiliary shipment ontology 64
Figure 9-I: Interaction between the *EconomyCongoBuy* service and a customer 65
Figure 9-J: Discover ExpressCongoBuy service as a substitute for EconomyCongoBuy service 67
Figure 9-K: Providing a name for the new CharlyAir service 67
Figure 9-L: Select BravoAir Reservation Agent as SuperService 68
Figure 9-M: Adopting the service model from BravoAir Reservation Agent 68
Figure 9-N: Visualizing CharlyAir service 69
Figure 9-O: Visualizing BravoAir service 69
Figure 9-P: Auxiliary airline ontology 70
Figure 9-Q: Interaction between the *BravoAir* service and a customer 71
Figure 9-R: Discover DeltaAir service as a substitute for CharlyAir service 73
Figure 9-S: Validate CharlyAir service as a substitute for BravoAir service 73
Figure 9-T: Validate DeltaAir service as a substitute for CharlyAir service 74
Figure 9-U: Interaction between the CongoBuy service and a customer 75
Figure 10-A: Service Model change in a SuperService over time 77

Legend of figures

Classes

There are two different types of a visualized class: an already existing class from a known ontology or a new class defined in this document.

Existing class: This symbol represents an already existing OWL class from a known ontology – in most cases from OWL-S.

New class: This symbol represents a new OWL class defined in this document.

Property

This symbol represents an OWL object or data type property and points to a property connection.

Connections

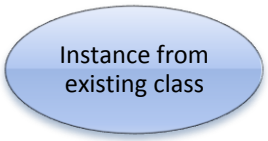
There are two different types of connections between two classes.

Inheritance connection: This symbol means that the class on the left side is an OWL super class of the class of the right side.

Property connection: This symbol means that the OWL class on the left side has an OWL property with a value from the OWL class on the right side.

Instances

This symbol represents an instance of an existing OWL class. The text in the symbol represents the name of the class the instance is generated from.

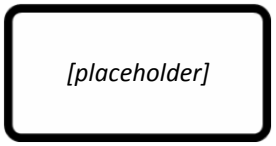


This symbol represents an instance of a new OWL class. The text in the symbol represents the name of the new class the instance is generated from.

Placeholder

This symbol represents a placeholder for either:

- One or more paraphrased OWL properties
- Concrete data, e.g. email-address



1 Introduction

This is a design science thesis, as described in (Bernstein, 2005).

1.1 Current situation

Current Web Service[1] Technologies such as WSDL (Web Services Description Language (WSDL) 1.1, 2001), UDDI (OASIS Open, 2006) and SOAP (SOAP Version 1.2, 2007) provide the means to describe the syntax of such a service. They lack, however, the capability to describe the semantics of these services which are necessary for the automation of the following service tasks: discovery, invocation, composition, interoperation and monitoring.

To that end, a number of standards such as OWL-S (OWL-S: Semantic Markup for Web Services, 2004), WSDL-S (Web Service Semantics: WSDL-S, 2005), WSMO (Web Service Modeling Ontology (WSMO), 2005), and SWSF (SWSF: Semantic Web Services Framework, 2005) have been proposed. Each of these standards allows connecting Web Services with an ontology-based semantic description. Up to the present however, only the SWSF discusses the connection of different Web Services among each other in order to reuse similar underlying elements and add additional relationship information. Furthermore, none of these standards defines a concrete and self-contained way of sharing specific elements among Web Services or a concrete way of interpreting the relationship among these services.

Theoretically, this sharing would already be possible in the OWL-S framework to some extent since it is based on OWL and OWL (Web Ontology Language, 2007) allows the sharing of its elements among different ontologies. The semantic Web Service descriptions are represented each using such an ontology and can therefore also be reused among each other by importing the according OWL ontologies. Practically, this underlying OWL connection is not sufficient for the domain of semantic Web Services in OWL-S.

In order to optimize this sharing of the underlying elements of OWL-S Web Services among them and in order to benefit from the additional information of the consequential relationships among them caused by this sharing, a concrete definition is needed for such a connection between Web Services.

This thesis proposes a solution for that need and calls it Inheritance Relationship (IR) for OWL-S.

1.2 Motivation

Shown below are the motivating tasks in *Web Service creation* and *Web Service discovery* for OWL-S which can be achieved using the proposed Inheritance Relationship (IR).

The concrete benefits of these motivating tasks are illustrated in the use cases in <Section 3: Use cases>. These benefits are mainly derived from the benefits of inheritance in computer science in general, as described in <Section 4: Definition of inheritance>. The solution which describes how these tasks can be achieved is described in <Section 5: Solution> below.

[1] The definition of a Web Service can be found in <Section 14: Glossary>.

1.2.1 Web Service creation

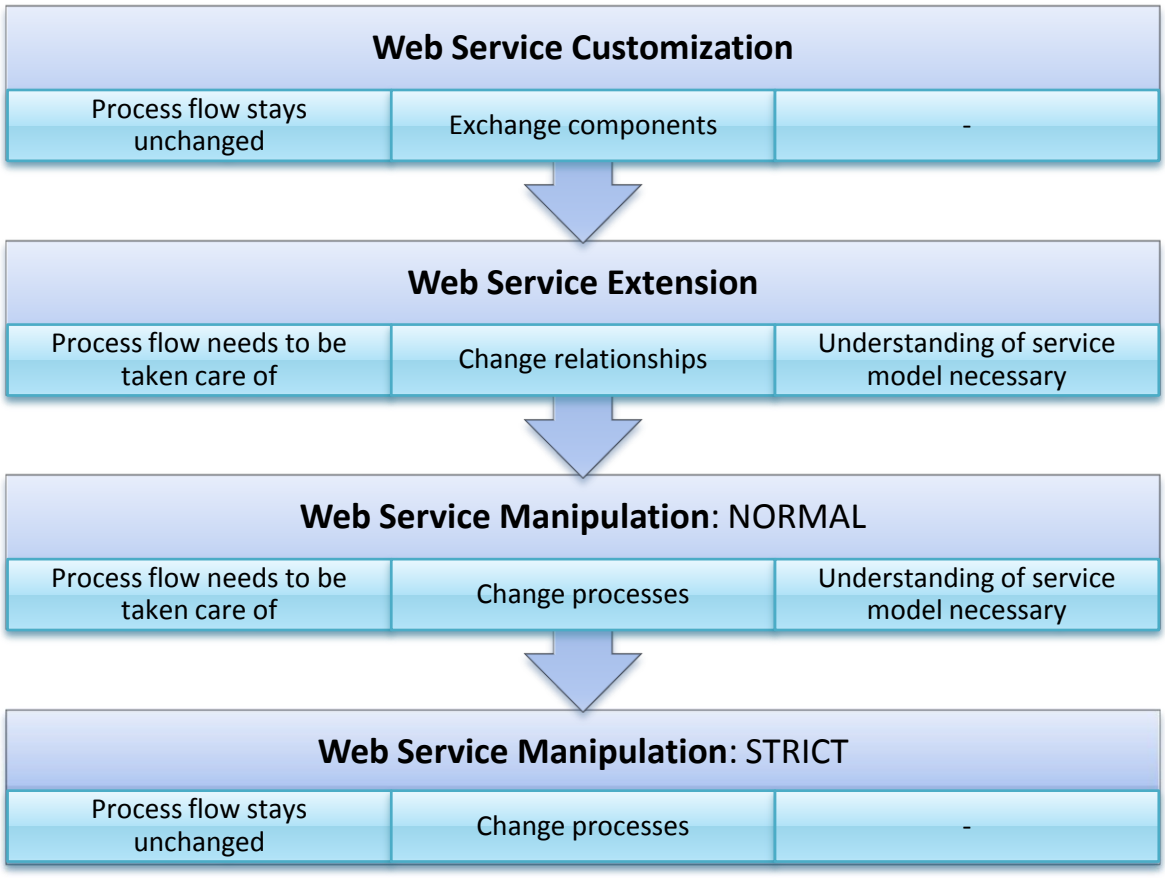
In order for the OWL-S to be successful there must be a sufficient number of OWL-S Web Services available. Since there are competitors (WSDL-S, WSMO and SWSF) for the OWL-S framework, having a critical mass of concrete Web Services created is crucial for its distribution. Typically, this Web Service creation means nowadays to begin every time from scratch.

A way to accelerate the distribution of OWL-S is to make OWL-S Web Service creation more efficient by allowing one to benefit from the work done for already existing services. Therefore, this thesis proposes three ways to do so: *Web Service Customization*, *Extension* and *Manipulation*.

For example, as described later in <Section 3.1.3: Create CharlyAir>, a new service CharlyAir can be created by reusing elements of an already existing service while spending minimal effort on modifying it.

These three applications of the proposed solution can be used complementary, but have nevertheless essential differences among each other, as illustrated in <Figure 1-A: The comparison of Web Service Customization, Extension and Manipulation>. Generally speaking, Web Service Customization allows adjusting an existing service on a high level without touching the underlying process flow, while Web Service Extension takes place on a deeper level and Web Service Manipulation on the deepest – the latter are altering the process flow of the service which needs to be taken care of. Therefore, the underlying service model needs to be understood in order to change low-level relationships or process definitions of the service model.

Figure 1-A: The comparison of Web Service Customization, Extension and Manipulation



Furthermore, Web Service Manipulation can be used in two different modes: normal and strict. The difference between these modes is explained in <Section 5.5.5: Interpreting Web Service Manipulation>.

Web Service Customization

Web Service Customization allows reusing the service model and the corresponding grounding of an existing service. Additionally, individual processes from the service model can be replaced by other processes with compatible inputs and outputs or even deleted if and only if the process flow can be sustained.

Web Service Extension

Web Service Extension allows one to add new processes to the inherited service model. In general, new processes can be added in the normal way by using regular OWL-S. In the case, however, where the new processes get inserted into an inherited process composition, it is necessary to model the connection from the new processes to the inherited ones. This can be done by allowing the definition of new input and output bindings. Furthermore, in order to provide the ability to model new process flow behavior for those new processes, one can also model new control constructs for them.

Web Service Manipulation

Finally, *Web Service Manipulation* allows one to change an inherited service model on a high detailed level. Thereby, single effects, preconditions and result conditions (i.e. OWL-S Expressions) can get replaced; and in case the service model consists only of an atomic process, the inputs and outputs of this process can be deleted or added.

However, when *Web Service Manipulation* is used, only the service model but not the complete service grounding can be inherited (i.e. reused) because of the caused inconsistency between the grounding and its corresponding manipulated service model.

1.2.2 Web Service discovery

Automated Web Service discovery is stated itself as a motivating task in future for OWL-S. The result of the service discovery, however, depends heavily on (potentially large) registries which are used to this end because there is yet no other way to discover those services otherwise.

A way to discover relevant services without the need of a registry – given a particular service – is to use the proposed Inheritance Relationship (IR) in its strict form in order to find service substitutes.

For example, as described later in <Section 3.1.4: Smooth choice increment with *CharlyAir*>, CharlyAir can be discovered as a suitable substitute for BravoAir and thereby increase the choice of a user of BravoAir.

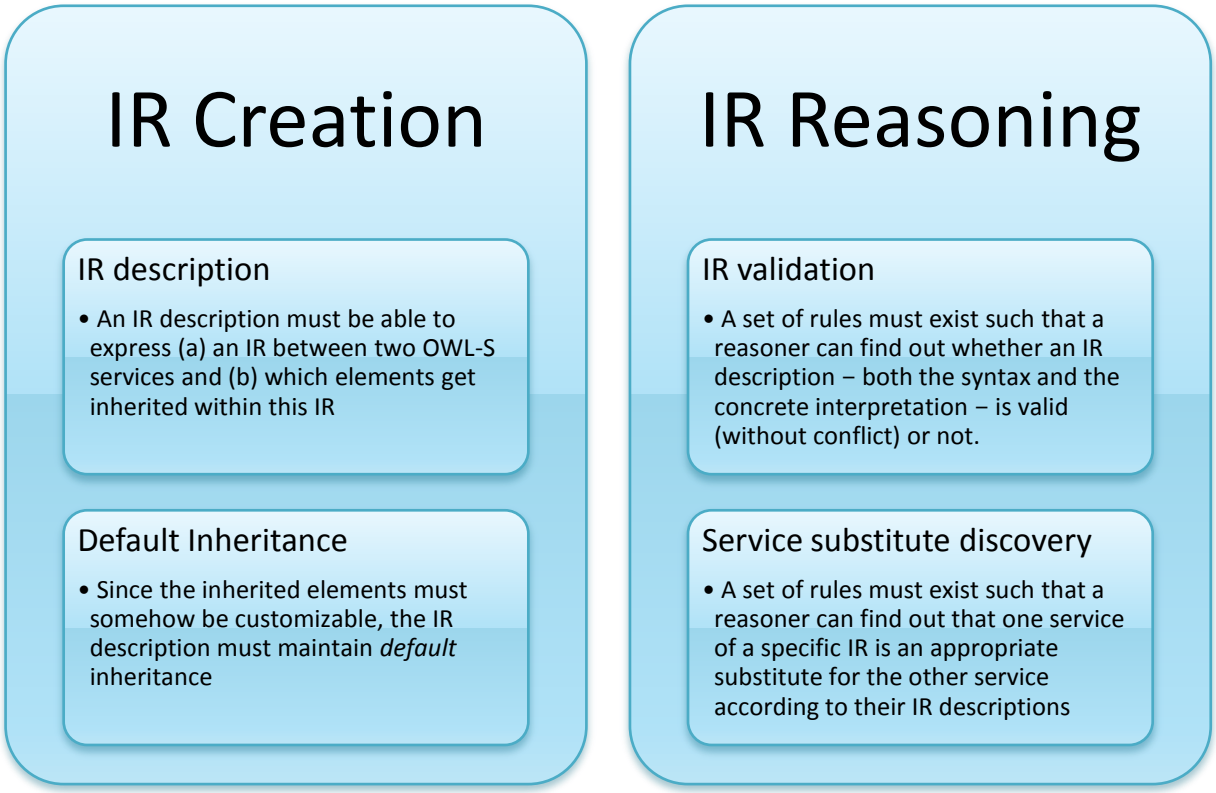
Web Service substitutes

Once a Web Service is found, service substitutes can be discovered without any registries by making use of the strict Inheritance Relationship (IR) between two services. This IR is defined such that all inheriting services can automatically be used as completely appropriate substitutes for the service they inherit from – in every situation.

1.3 Requirements for Inheritance Relationships (IR)

Given the motivating tasks from above, two main parts build the requirements for the proposed Inheritance Relationship (IR): the requirements concerning *IR Creation* and *IR Reasoning*, as illustrated in <Figure 1-B: The requirements list to provide IR among services>.

Figure 1-B: The requirements list to provide IR among services



IR Creation

First, in order to enable the motivating task *Web Service creation*, an IR must be able to express the subtasks *Web Service Customization*, *Extension* and *Manipulation*. Therefore, an IR description is needed.

Second, since those subtasks include customizability, this IR description allows default inheritance[2].

IR Reasoning

First, to ensure that an IR can be interpreted without conflicts, the IR must comply with some contracts. Therefore, it is necessary to be able to reason about whether a specific IR description is valid (i.e. complies with these contracts) or not.

Second, since it is a motivating task improving Web Service discovery by finding appropriate service substitutes without the need of a registry, rules must exist that can automatically assure whether one service is such a substitute or not – according to the corresponding IR descriptions.

[2] See <Section 14: Glossary>

1.4 Assignment

The assignment of this thesis is to extend OWL-S with the possibility to maintain Inheritance Relationships (IR) between Web Services in order to reuse elements among different Web Services and to have this additional relationship information among those services.

Since OWL-S is essentially an instance-based model that describes the semantics of Web Services, it is necessary to provide some external mechanism in order to provide these IRs. Because the SWSF presents already a use case which illustrates a basic way to inherit and override processes among services, one possibility would be to transfer the ideas from SWSF to OWL-S and elaborate them.

1.5 Succinct chapter overview

<Chapter 2: Technical background> provides the technical background for the (formal) solution of this approach.

<Chapter 3: Use cases> illustrates six possible ways how this thesis can improve motivating tasks for semantic web services.

<Chapter 4: Definition of inheritance> represents the first step into how this improvement can be accomplished, it defines the meaning of inheritance in the domain of services in OWL-S based on the current notion of inheritance in computer science.

<Chapter 5: Solution> presents the complete solution of how inheritance can be used in the domain of OWL-S: it defines and describes the vocabulary, the interpretation and the necessary conditions of this usage based on the SWSL-syntax for best readability.

<Chapter 6: OWL-syntax> presents the alternative OWL-syntax and illustrates it (using one of the use cases) by giving a concrete example.

<Chapter 7: Dismissed approaches> presents approaches to the chosen solution and explains why they have been dismissed.

<Chapter 8: Design decisions> explains all design decisions taken regarding the (formal) solution.

<Chapter 9: Evaluation> evaluates the (formal) solution of this thesis in the light of the prototype and compares it briefly with a similar approach.

<Chapter 10: Discussion> discusses the current findings and related work and holds my personal opinion which places this work into the big picture of semantic web.

<Chapter 11: Conclusions> summarizes the accomplishments of this thesis and discusses the outlook including possible future work.

Chapter summary

Currently, there exists an approach which introduces a basic way to make use of inheritance among semantic web services (SWSF). The goal of this thesis is now to elaborate this approach and transfer it to OWL-S, which is currently a promising candidate for evolving in the near future. The benefits of this thesis are an improved web service creation and discovery.

2 Technical background

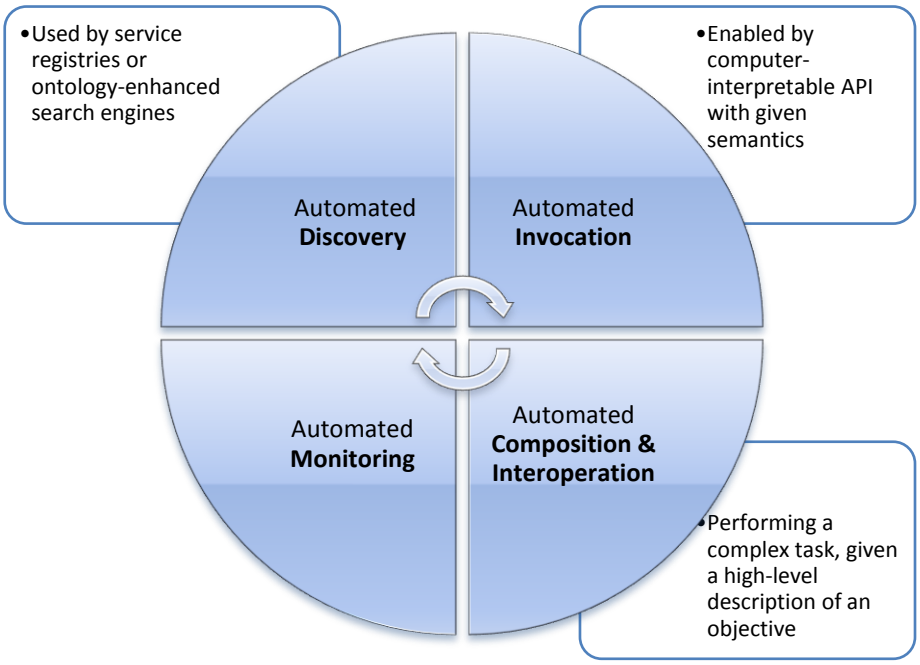
This chapter introduces the technical background of the solution of this thesis. Since the goal of this thesis is to extend the OWL-S framework, this framework is mentioned most detailed. In return, the according competitors are just mentioned by pointing out their main differences to the OWL-S.

Additionally, the work related to the approach of this thesis is presented and discussed.

2.1 OWL-S: Web Ontology Language for Services

The OWL-S (OWL-S: Semantic Markup for Web Services, 2004)[3] has been developed for an easier use of Web Services[4] by humans and by software agents. In specific, OWL-S enables the automation of the following tasks: *discovery*, *invocation*, *composition*, *interoperation* and *monitoring* of Web Services, see <Figure 2-A: OWL-S enabled automation of Web Service tasks >. This automation is achieved by providing a standard ontology (OWL-S) for declaring and describing Web Services.

Figure 2-A: OWL-S enabled automation of Web Service tasks



OWL-S is built upon OWL. Therefore, OWL-S represents a specific OWL ontology.

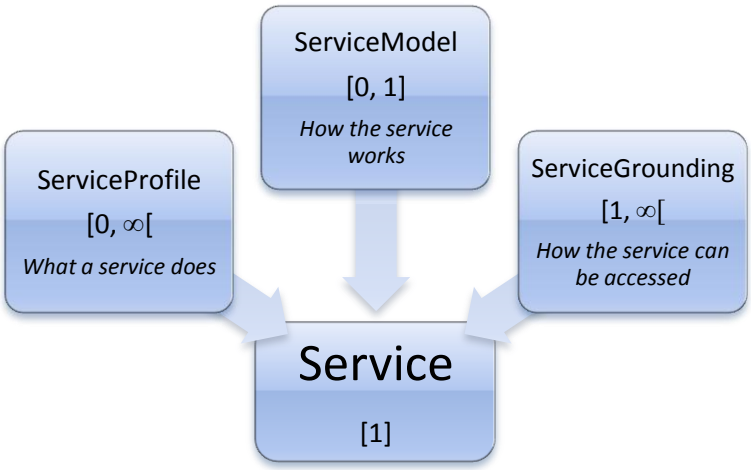
The main structure of OWL-S consists of a central service. This service can have several profiles and one service model. This service model, in turn, must have one or more service groundings. These relationships are illustrated in <Figure 2-B: OWL-S upper ontology for Web Services and their semantic descriptions>.

[3] OWL-S is currently a W3C Member Submission from the 22 November 2004.
[4] See <Section 14: Glossary>. In this document "Web Service" is also referred to as a *service*.

OWL-S class: Service

The *Service* class provides an overview and point of reference for a Web Service. Every Web Service must be defined by an instance of this central class which can be connected with instances of the classes *ServiceProfile*, *ServiceModel* and *ServiceGrounding* for describing the service in more details. Such a service instance can be thought of an API declaration for the specific Web Service, see <Figure 2-B: OWL-S upper ontology for Web Services and their semantic descriptions>.

Figure 2-B: OWL-S upper ontology for Web Services and their semantic descriptions



OWL-S class: ServiceProfile

The *ServiceProfile* class provides a bridge between service requesters and service providers. The instances are mainly meant to advertise an existing service by describing it in a general way that can be understood both by human and computer agents. It is also possible to use the service profile to advertise a needed service request.

OWL-S provides also a *Profile* class which is a subclass of *ServiceProfile*. This default class should include provider information, a functional description and host properties of the described service. It is possible to define other profile classes that specify the service characteristics more precisely. <Figure 2-D: OWL-S class Expression is used to define logical formalism (rules)> describes this more concrete.

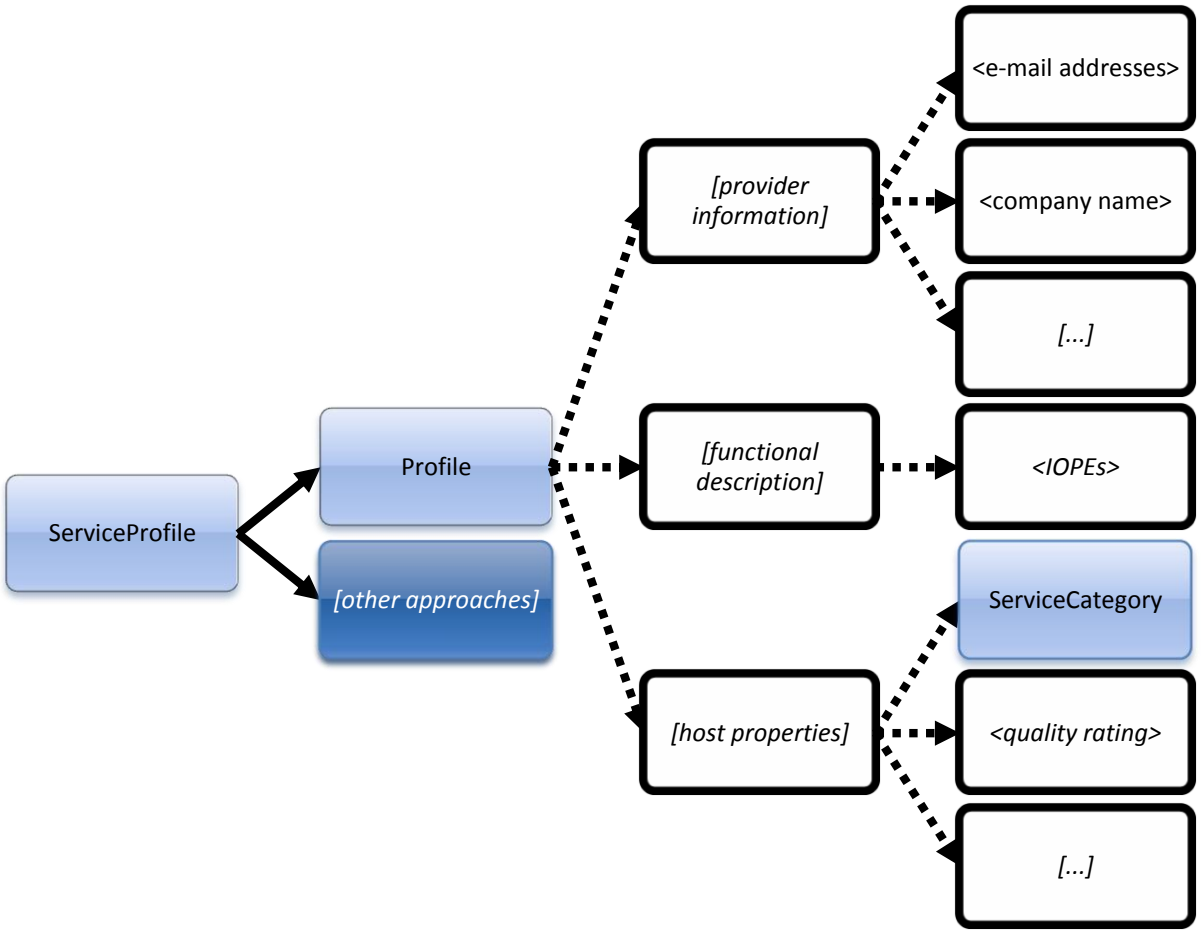
OWL-S class: ServiceModel

The *ServiceModel* class uses the subclass *Process* to provide a process view on the service. This view can be thought of as a specification of the ways a client may interact with a service. It can be used in different ways by an agent:

- **Analysis:** Performing a more in-depth analysis of whether the service meets the agent's needs.
- **Composition:** Composing service descriptions from multiple services to perform a specific task.
- **Coordination:** Coordinating the activities of different participants during the course of a service enactment.
- **Monitoring:** Monitoring the execution of the service

The OWL-S: Expression class is used to describe the preconditions and effects of processes in the process model of a service.

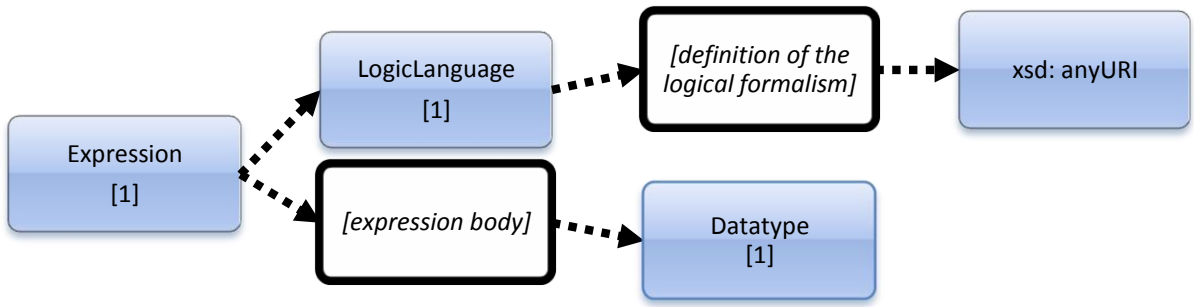
Figure 2-C: The OWL-S class ServiceProfile and its hierarchy



OWL-S class: ServiceGrounding

The ServiceGrounding class provides a concrete specification of how the service can be accessed. Of main interest here are subjects like protocol, message formats, serialization, transport and addressing. This grounding can be thought of the concrete part of the semantic Web Service description, compared to the service profile and service model which describe the service both on an abstract level.

Figure 2-D: OWL-S class Expression is used to define logical formalism (rules)



OWL-S provides an initial default grounding which uses the WSDL in a complementary way. Therefore, one can benefit from the extensive work done in WSDL for message exchanging using various protocols and transport mechanisms. OWL-S allows however to use different ground approaches.

2.2 SWSF: Semantic Web Service Framework

The Semantic Web Services Framework (SWSF: Semantic Web Services Framework, 2005) has essentially the same purpose as OWL-S: it provides semantic specifications of Web Services; namely (similarly to OWL-S) a comparable service profile, model and grounding.

Although the SWSF has some differences compared to OWL-S.

- **First-Order Logic:** FLOWS – the SWSF ontology which provides this semantic specification for Web Services – is expressed in first-order-logic. OWL-S in contrast is expressed in OWL-DL, a decidable description logic language. Therefore "FLOWS" stands for First-Order Logic Ontology.
- **Enhanced process model:** The SWSF claims to provide an enhanced process model compared to OWL-S, since it is based on the Process Specification Language (Process Specification Language (PSL), 2007). Therefore it provides Web Service specific process concepts that include not only inputs and outputs, but also messages and channels.
- **Non-monotonic language:** In addition to OWL-S, the SWSF provides SWSL-Rules, a non-monotonic language based on the logic-programming paradigm which is meant to support the use of the FLOWS in reasoning and execution environments (SWSL: Semantic Web Services Language, 2005).
- **Interoperability:** The SWSF mentions interoperability as an advantage of SWSF over OWL-S: ***«*** *An important final distinction between OWL-S and FLOWS is with respect to the role it plays. Whereas both endeavours attempt to provide an ontology for Web services, FLOWS had the additional objective of acting as a focal point for interoperability, enabling other business process modeling languages to be expressed or related to FLOWS.* ***»*** (SWSF: Semantic Web Services Framework, 2005)

2.3 WSDL-S

The Web Service Semantics Framework (Web Service Semantics: WSDL-S, 2005) has essentially the same purpose as OWL-S: it provides semantic specifications of Web Services. Since WSDL-S is a semantic extension of WSDL (Web Services Description Language (WSDL) 1.1, 2001) it can (only) be applied for Web Services that are described in the WSDL.

The WSDL-S framework claims on having the following advantages over OWL-S.

- **Language:** The semantics of Web Services cannot only be described using OWL, but also using UML or WSDL. The latter would allow the use of already established language in the domain of Web Service and the former the reuse of existing UML domain models.
- **Tool support:** Since tools for Web Service description based on WSDL are already available, semantic tool extensions for WSDL-S are likely to be built.

2.4 WSMO

The Web Service Modeling Ontology (Web Service Modeling Ontology (WSMO), 2005) has essentially the same purpose as OWL-S: it provides semantic specifications of Web Services.

Although the WSMO has some differences compared to OWL-S.

- **Expanded focus:** Contrasting to OWL-S, the WSMO does not only focus on Web Service descriptions (service profile, model and grounding), but also on goals that represent user desires and mediators which are meant to handle the aligning, merging, and transforming of imported ontologies automatically.
- **Non-monotonic language:** Similar to SWSF, the WSMO provides in addition to OWL-S a non-monotonic language, which is largely based on F-logic.

3 Use cases

The Inheritance Relationships (IR) extension for the OWL-S can be used to improve existing OWL-S service descriptions and may enable a wider use of the OWL-S itself. An overview of the different ways how this can be achieved is shown in <Section 1.2: Motivation>. This section describes concrete use cases of applying the IR extension.

Background information on the mentioned frameworks in this use case can be found in <Section 2: Technical background>.

3.1 Strict IR

This section presents four use cases that illustrate the benefit of the proposed strict Inheritance Relationships. Two use cases are about the motivating task service creation, and the other two use cases are about service discovery. Since there are two kinds of services (atomic and composite ones), one use case is provided for each kind.

3.1.1 Create EconomyCongoBuy

This use case illustrates how the atomic service *EconomyCongoBuy* can be created reusing some elements of the similar atomic service *ExpressCongoBuy* while complying with the contract of a strict inheritance by making use of the proposed *Service Customization*, see <Section 5.1.1: Web Service Customization>.

Keywords: service creation, strict inheritance, atomic service, service customization.

Situation

ExpressCongoBuy is an example service published within the OWL-S 1.1 Release (OWL-S 1.1 Release: Examples). It provides a one-step book buying service for the fictitious *Congo* shop with a standard delivery setting. In real life, however, there might also be a second version with a different default delivery setting. Since a concrete delivery is not yet defined in the example, this use case defines therefore a one-day-delivery for *ExpressCongoBuy* and creates a new service *EconomyCongoBuy* with a slower three-day-delivery.

Aim

Given the existing *ExpressCongoBuy* service, it would be convenient to benefit from the work already done when creating the other book selling service *EconomyCongoBuy* instead of beginning from scratch. Currently, there is no way to benefit from existing atomic services in order to create a new one.

Solution

The proposed Inheritance Relationship (IR), however, would make it possible to reuse in this case the service model of *ExpressCongoBuy* within a *Service Customization*.

More concretely, the new service *EconomyCongoBuy* can be created by inheriting the service model from *ExpressCongoBuy*, replacing the positive result and adding a new service profile and grounding.

The necessary statements for this strict IR are described in SWSL for the sake of human readability.

Service customization

```
Inherit[AdoptServiceModel(ExpressCongoBuy), Processes, Groundings].

Rename[ExpressCongoBuy *-> EconomyCongoBuy].

ReplaceExpressions[

       Effect(EconomyCongoBuy, ExpressCongoOrderShippedEffect) *->
       Effect(EconomyCongoBuy, EconomyCongoOrderShippedEffect)

].
```

Benefit

Basically, the general benefits of inheritance apply to this use case, mentioned in <Section 4.1: Inheritance in computer science>:

- **Efficient service creation:** First, the reuse of information facilitates the creation of the service *EconomyCongoBuy* since the service model of *ExpressCongoBuy* can be reused without the need of completely understanding the already existing *ExpressCongoBuy* service. Therefore, using Inheritance Relationships (IR) can improve the efficiency of the creation of similar services.
- **Additional relationship information:** At last, the explicit statement of this IR provides additional information about the two services. The use case in <Section 3.1.2: Smooth substitution with *ExpressCongoBuy*> describes the benefit of such an IR.

Walkthrough

A detailed walkthrough illustrates in <Section: 9.2.1: Walkthrough: Create EconomyCongoBuy> how the prototype can handle this use case.

3.1.2 Smooth substitution with *ExpressCongoBuy*

This use case illustrates the use of a strict Inheritance Relationship between the two services *ExpressCongoBuy* and *EconomyCongoBuy* which both consist each of an atomic process for service discovery.

Keywords: service discovery, strict inheritance, atomic service.

Situation

A regular customer of the Web Service *EconomyCongoBuy* wants to buy a book within this service as usual. This time however, the service is not able to serve him because the requested book is out of stock. The customer, however, is in need of that book and therefore wants to order it now.

Aim

The aim of an intelligent system which provides this service should be to suggest an appropriate alternative in order to make the book buying as smooth as possible for the customer. Therefore, a similar service is needed to be found automatically that can fulfill the customer's need without giving him any unwanted surprises.

Solution

A solution to find such an alternative automatically can be provided within the proposed strict Inheritance Relationship (IR).

In this case, it would be possible to detect that the *ExpressCongoBuy* service is such a suitable alternative for *EconomyCongoBuy* because their inputs, outputs and preconditions are equal and the effect is strengthened in the alternative service compared to the original one. In specific, both services allow to buy a book via ISBN; except the shipment of the book is faster in *ExpressCongoBuy* while the price stays moderate at the same time.

Walkthrough

A walkthrough illustrates this use case in <Section: 9.2.2: Walkthrough: Smooth substitution with ExpressCongoBuy> in more details.

Benefit

The benefit of having a strict Inheritance Relationship (IR) between two services is the ability to discover semantically related services that can improve the availability of a certain physical service (like book selling) while providing a smooth process flow at the same time because of the services' compatibility.

3.1.3 Create CharlyAir

This use case illustrates how the composite service *CharlyAir* can be created reusing some elements of the similar composite service *BravoAir* by making use of the proposed Web *Service Customization*, see <Section 5.1.1: Web Service Customization>.

Keywords: service creation, strict inheritance, composite service, service customization.

Situation

BravoAir is an example service published with the OWL-S 1.1 Release (OWL-S 1.1 Release: Examples). It provides a reservation agent for the fictitious *BravoAir* airline. In real life, however, there are not only single airlines available but rather strategic alliances which have several airlines as its members in order to benefit from each other by sharing certain things. Therefore, such a reservation agent is likely to be reused among different airlines. Even if this agent might be very similar for these airlines, there might be airline specific processes involved though.

Considering the structure of the *BravoAir* example, it would be natural to assume that the processes *GetDesiredFlightDetails*, *SelectAvailableFlight* and *LogIn* are reused among the airlines; the process *CompleteReservation* on the other hand is likely to be airline specific.

Aim

Given the existing *BravoAir* service, it would be convenient to benefit from the work already done by creating it in order to create the other airline services instead of beginning from scratch. Currently, the only way to benefit in this case from the *BravoAir* service is to reuse its grounding[5].

Solution

The proposed Inheritance Relationship (IR), however, would not only make it possible to reuse the service grounding but also the according service model.

[5] See <Section7: Dismissed approaches> for explanatory remarks

More concretely, the new service *CharlyAir* can be created by inheriting the service grounding and model from *BravoAir*, replacing the *CompleteReservation* process with a new one and creating a new service profile.

Service customization

```
Inherit[AdoptServiceModel(BravoAir_Process), Processes, Groundings].

Rename[BravoAir_Process *-> CharlyAir].

ReplaceProcess[PerformCompleteReservation *-> CompleteReservation_CharlyAir].
```

The new input bindings for the process replacement CompleteReservation_CharlyAir do not need to be stated explicitly because they can automatically be taken over from the old process perform PerformCompleteReservation since the inputs of these two processes stay completely compatible and have the same IDs.

Benefit

Basically, all the general benefits of inheritance apply to this use case, mentioned in <Section 4.1: Inheritance in computer science>:

- **Efficient service creation:** First, the reuse of information facilitates the creation of the service *CharlyAir* since the service model and groundings of *BravoAir* can be largely reused without the need of completely understanding the already existing *BravoAir* service. Therefore, using Inheritance Relationships (IR) can improve the efficiency of the creation of similar services.
- **Improved service maintenance:** Additionally to this facilitation, IR may improve service maintenance among similar services through economical data storage, complexity reduction within the similar services and the control gained over the services through the reused processes.
- **Additional relationship information:** At last, the explicit statement of this IR provides additional information about the according services. This information is of specific interest when the IR is bidirectional; in this case, the IR is made official. The use case in <Section 3.1.4: Smooth choice increment with *CharlyAir*> describes the benefit of such an official IR.

3.1.4 Smooth choice increment with *CharlyAir*

This use case illustrates the use in service discovery of a strict Inheritance Relationship (IR) between the two Web Services *BravoAir* and its SubService *CharlyAir* which both consist each of a complex process. Additionally, it illustrates the different impacts of either an official or a regular IR.

Keywords: smooth choice increment, service discovery, official strict inheritance, composite service.

Situation

A customer[6] wants to book a flight with the Web Service *BravoAir* because a close friend of his recommended him this specific airline. When it comes to the concrete booking, however, *BravoAir* has no flights available and the customer feels not comfortable with changing the airline.

[6] In general, a customer would interact with a Web Service only indirectly via a proxy. This proxy is left out in this use case for the reason of simplification

Aim

In this case, an intelligent system should be able to find automatically flights from other airlines within a similar experience (service quality, trust aspects, etc) compared to the customer's first choice *BravoAir*. Or in other words: the system should be able to give the customer more choice in a smooth way.

Solution

A solution to increment the customer's choice automatically can be provided within the proposed strict Inheritance Relationship (IR):

In this case, it would be possible to offer additionally to the empty result from *BravoAir* available flights from the official partner *CharlyAir*. The fact that *CharlyAir* can be detected as an official partner of *BravoAir* gives the customer the necessary confidence to switch to a different airline and nevertheless stay within a somehow familiar one he feels comfortable with.

Walkthrough

A walkthrough illustrates this use case in <Section: 9.2.4: Walkthrough: Smooth choice increment with *CharlyAir*> in more details.

Benefit

The benefit of having a strict Inheritance Relationship (IR) between two services is the ability to discover semantically related services that can increment the choice of a customer while providing a smooth process flow at the same time because of the service compatibility.

The additional benefit of having a bidirectional strict IR between two services is the guarantee that those two services are official partners. What this means is yet up to free interpretation. It makes sense, however, to assume that those to services are interested in providing a very similar experience to the customer in order to please them.

Furthermore, this official IR has the following beneficial side effects:

- **Strong linking:** It is guaranteed that all related services can be found that are using this IR among each other since every participating service is linked with the others. On the other hand: using registries and trying to reason about this strict IR may cause a decidability problem and is likely to be expensive to calculate.
- **Update by participation:** If all participants of such an official IR are linked among each other, it is possible to inform automatically all subservices about updates that have taken place in a SuperService.

3.2 Normal IR

This section presents two use cases that illustrate the benefit of the proposed normal Inheritance Relationships. One use case is about the motivating task service creation, and the other one about service discovery.

3.2.1 Create E-BookBuy

This use case[7] illustrates how the composite service *E-BookBuy* can be created reusing some elements of the similar composite service *FullCongoBuy* with basically the same functionality by making use of the proposed *Service Customization* and *Service Extension* which are described in <Section 5.1: Applications of IRs> below.

Keywords: service creation, normal inheritance, composite service, service customization, service extension, similar functionality.

Situation

FullCongoBuy is an example service published with the OWL-S 1.1 Release (OWL-S 1.1 Release: Examples). It provides a book search and buying service for the fictitious *Congo* shop. In real life, however, there are not only books but also electronic books (e-books). One could assume that Congo wants to create a new e-book service *E-BookBuy* which might be very similar to a regular book service, but has also some differences though.

Considering the structure of the *FullCongoBuy* example, it would be natural to assume that all the processes stay the same for *E-BookBuy;* with the exception of the processes *LocateBook* and *SpecifyDeliveryDetails*. Since in the case of an e-book there is no physical shipment necessary but rather a delivery by download, this download needs to be provided in an additional process which has the name *ProvideDownloadOptions*.

Aim

Given the existing *FullCongoBuy* service, it would be convenient to benefit from the work already done by reusing it in order to create the other book selling service *E-BookBuy* instead of beginning from scratch. Currently, the only way to benefit in this case from the *FullCongoBuy* service is to reuse its grounding[8].

Solution

The proposed Inheritance Relationship (IR), however, would not only make it possible to reuse the service grounding but also the service model.

More concretely, the new service *E-BookBuy* can be created by inheriting the service grounding and model from *FullCongoBuy*, replacing the processes *LocateBook* and *SpecifyDeliveryDetails* with new ones, adding a new process *ProvideDownloadOptions* and creating a new service profile.

[7] Since Web Service Extension is not yet implemented in the prototype, there is no walkthrough available for this use case.
[8] See <Section 7: Dismissed approaches> for explanatory remarks

Service customization

```
Inherit[AdoptServiceModel(FullCongoBuy), Processes, Groundings].

Rename[

       FullCongoBuy *-> Full_eBookCongoBuy,
       FullCongoBuyOutput *-> Full_eBookCongoBuy_Output,
       CongoBuyBook *-> CongoBuy_eBook,
       LocateBookOutput *-> Locate_eBook_Output

].

ReplaceProcesses[LocateBook *-> Locate_eBook].
```

Service extension

```
InsertProcess[

       after(BuySequence) *-> SpecifyDownloadDetailsPerform,
       after(SpecifyDownloadDetails) *-> ProvideDownloadOptionsPerform

].

DeleteProcess[SpecifyDeliveryDetailsPerform].
```

Service manipulation

```
DeleteInputsAndOutputs[

       Input(SpecifyDeliveryDetails, FullCongoBuyDeliveryAddress),
       Input(SpecifyDeliveryDetails, FullCongoBuyPackagingSelection),
       Input(SpecifyDeliveryDetails, FullCongoBuyDeliveryTypeSelection)

].
```

Benefit

Basically, all the general benefits of inheritance apply to this use case, mentioned in <Section 4.1: Inheritance in computer science>:

- **Efficient service creation:** First, the reuse of information facilitates the creation of the service *E-BookBuy* since the service model and groundings of *FullCongoBuy* can be largely reused. Therefore, using Inheritance Relationships (IR) can improve the efficiency of the creation of similar services.
  However, since the IR used is normal – and not strict – the process flow has to be taken care of by the service creator himself, and therefore, he has to be familiar with the service model, i.e. study the service, which takes time. Therefore, the savings are rather in lines of code needed for the new service than in time needed for the creation (as it is also the case for strict IRs). Consequently, this benefit is smaller compared to strict IRs.
- **Improved service maintenance:** Additionally to this facilitation, IR may improve service maintenance among similar services through economical data storage, complexity reduction within the similar services and the control gained over the services through the reused processes.

- **Additional relationship information:** At last, the explicit statement of this IR provides additional information about the according services. The use case in <Section 3.2.2: FullCongoBuy suggests E-BookBuy> describes a possible benefit of such an IR.

3.2.2 FullCongoBuy suggests E-BookBuy

This use case illustrates the use in service discovery of a normal Inheritance Relationship between the two services *FullCongoBuy* and *E-BookBuy* which both consist each of a composite process.

Keywords: service discovery, strict inheritance, composite service, unspecific customer.

Situation

A customer wants to buy a book within the *FullCongoBuy* service. The specific book, however, is very popular and thereby currently out of stock.

Aim

The aim of an intelligent system which provides this service should be to suggest an interesting alternative in order to make the book buying as smooth as possible for the customer. Therefore, a similar service is needed to be found automatically that could also fulfill the customer's need.

Solution

A solution to find such an alternative automatically can be provided within the proposed Inheritance Relationship (IR).

In this case, it is possible to detect automatically that the *E-BookBuy* service could be an interesting alternative for *FullCongoBuy* because they have a normal Inheritance Relationship between each other. In specific, both services allow buying a book while it is a physical book in *FullCongoBuy* which needs to be shipped and it is a digital one in *E-BookBuy* which needs to be downloaded.

Walkthrough

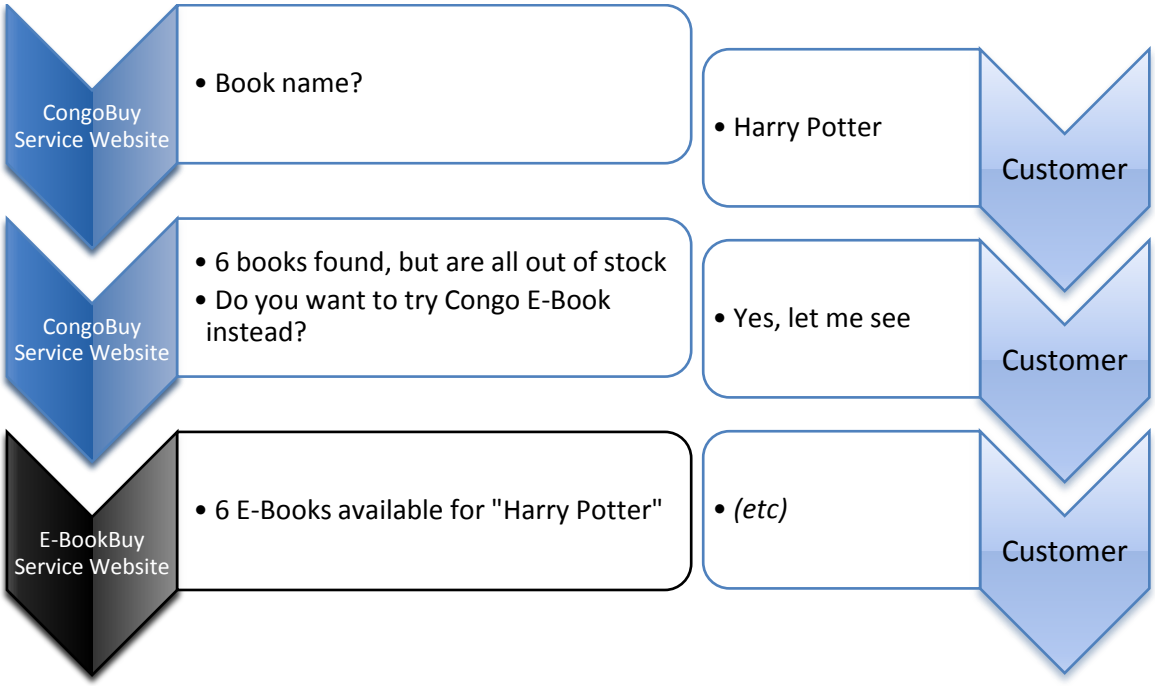
In order to illustrate this use case in more details, a concrete walkthrough is provided which covers the main interactions between the customer and the system that performs the two services *E-BookBuy* and *FullCongoBuy*, as illustrated in <Figure 3-A: Interaction between the CongoBuy service and a customer>.

Step 1: First, a customer looks his requested book up using the *CongoBuy* service.

Step 2: Second, the service tells the user that in general the books are available but currently out of stock. Since the *CongoBuy* service has a normal Inheritance Relationship (IR) with *E-BookBuy*, the system which is running the *CongoBuy* service can assume that *E-BookBuy* is similar to a certain degree with *CongoBuy*. Therefore, the system can suggest – by making a roughly guess – that the customer might also be interested in using the *E-BookBuy*.

Step 3: After the user agrees with trying out the suggested *E-BookBuy* service, the system switches to the other service and performs *E-BookBuy*. In this lucky case the needed input for *E-BookBuy* is also part of the needed input of *CongoBuy* and therefore the same input can just be reused by the system to search for electronic books of "Harry Potter" in *E-BookBuy*.

Figure 3-A: Interaction between the CongoBuy service and a customer



Benefit

The benefit of having a normal Inheritance Relationship (IR) for service discovery is the possibility to detect potentially other interesting services for a customer of a specific service.

However, it is not guaranteed – as it is the case using strict IR – that the detected service through the normal IR is actually of interest for the customer. On the other hand, a wider range of potentially interesting services for the customer can be discovered using normal IRs instead of strict IRs.

4 Definition of inheritance

The first part illustrates[9] inheritance from different angles. Nowadays, inheritance is probably most used in object-oriented programming and in the domain of knowledge representation, but has also a general meaning in computer science. Furthermore, inheritance has also been discovered in the context of processes.

Based on this already existing meaning of inheritance, the second part defines what inheritance means for OWL-S.

4.1 Inheritance in computer science

Inheritance in computer science in general provides the support for representation by categorization in computer languages. This categorization is very helpful in information processing by means of generalization and specialization.

- **Generalization:** An Inheritance Relationship (IR) is essentially built on an <*is-a*> relation which represents a hierarchical structure. In this hierarchy, the element on top from which the element below inherits is more general.
  - **Example:** Since an apple is a fruit (<apple> *is-a* <fruit>), the apple inherits from the fruit and therefore the fruit is meant to be a more general representation of an apple.
- **Specialization:** Of course, looking at the inheritance from the other side makes elements which inherit from a general element more specialized.

4.1.1 Types of inheritance

Inheritance can be differentiated using two perspectives. An inheritance can be asserted to only one of the categories for each perspective.

Perspective: complete or default inheritance[10]

- **Complete inheritance[11]:** When complete inheritance is used, information that is used by more than 1 element has to be stored in a more general element. This means that no redundant information is allowed and information has to be inherited down the inheritance chain: the generalization must be complete. Therefore, inherited information cannot neither be altered nor arbitrarily extended.
- **Default inheritance:** When default inheritance is used, information gets inherited from a general element by default, but this default can be arbitrarily altered and extended in the specialized elements.

Perspective: single or multiple-inheritance

- **Single inheritance:** Allows a specialized class to inherit from only one general class.
- **Multiple-inheritance:** Allows a specialized class to inherit from several general classes.

[9] Source: (Wikipedia, Inheritance (computer science), 2007)
[10] See http://de.wikipedia.org/wiki/Konstruktionsgrammatik#Default_inheritance_model (Accessed on July 12, 2007)
[11] The complete inheritance is equivalent to the differential inheritance: http://en.wikipedia.org/wiki/Differential_inheritance (Accessed on July 12, 2007)

4.1.2 Advantages of inheritance

The main advantage of inheritance is the reuse of inheritance and adding new information. The former advantage, however, leads also to some further advantages that are caused by this reusing of information.

- **Information reuse:** Shared information can be stored in a generalized element and can then be reused by more specialized elements. In addition to the direct advantage of information reuse, this can lead to the following additional advantages:
  - **Economical information storage:** The shared information only has to be stored once instead of several times for each element.
  - **Complexity reduction:** This generalization of shared information can reduce complexity.
  - **Gain control:** Once a generalization is used, every element that inherits from this general element can be controlled by it.
- **Additional information:** Since an Inheritance Relationship (IR) states that the corresponding generalization and specialization do share a certain structure, the IR provides additional information about the related Web Services. This kind of information was not possible to express before.

4.1.3 Disadvantages of inheritance

- **Yo-yo problem:** A too intensive use of inheritance can lead to a loss of overview, which is called the Yo-yo problem[12].

4.2 Inheritance in object-oriented programming

In object-oriented programming, the use of default inheritance[13] is a specific way of forming new classes by reusing already defined classes. These new classes inherit attributes and behavior of the pre-existing classes.

4.2.1 Applications of inheritance

Thereby, one application of inheritance is to reuse existing code with little or no modification for modeling new classes, as illustrated below:

- **Subtyping:** Subtyping means extending an already existing class with more specific details by creating a more specialized class with more aspects[14].
  - **Example:** From a general class <Bank Account> one could build a specialized class <Interest Bearing Account> with extensional information about the interest rate and the accrued interest.
- **Abstract classes:** Using abstract classes, means having a general, abstract class with only intended behavior. These intended behaviors (methods) represent in this case placeholders which are meant to be implemented only now by specialized, concrete classes.

[12] See http://en.wikipedia.org/wiki/Yo-yo_problem (Accessed on July 12, 2007)
[13] See <Section 14: Glossary>
[14] Aspects can be behaviors (methods) or attributes

- **Overriding:** Many object-oriented programming languages permit one not only to inherit aspects of a general class, but also to replace some of those inherited aspects [15] in the according specialized class.

Another application of inheritance in object-oriented programming is to benefit from a modeled inheritance at runtime, using polymorphic substitution:

- **Polymorphic substitution:** Using polymorphism[16], one can treat an instance of a specialized class similar as an instance of the according generalized (abstract) class. This allows one to leave the allocation to its specific class open during modeling and choose the class-specific behavior at runtime, since these instances represent substitutes.
  - **Example:** Given an abstract class <Animal> with an intended *speak* behavior, a specialized class <Dog> with a *bark* behavior and a specialized class <Cat> with a *meow* behavior. If those classes satisfy the conditions for this polymorphism, it is possible to choose just at run time of which class the actual instance is and therefore which behavior gets executed when calling the shared behavior *speak*: the *meow* behavior of <Cat> or the *bark* behavior of <Dog>.

4.3 Inheritance in knowledge representation

Inheritance is also used in the domain of knowledge representation (The Principles of Knowledge Representation and Reasoning, 1993). Two examples are semantic networks and frames.

In semantic networks, concepts can have semantic relationships among each other in order to represent knowledge. These relationships can describe inheritance relationships, i.e. the relationship "is a/is an".

Frames basically represent concepts which can be used for knowledge representation. This paradigm is similar to object-oriented programming with the difference that the classes are now called frames which have attributes, but no methods.

4.4 Inheritance for processes

The general idea does already exist that inheritance can not only be applied to objects (respectively classes), as it is done in object-oriented programming, but also to processes. Malone et al (Tools for inventing organizations: Toward a handbook of organizational processes, 1999) explain these two kinds of inheritance as the inheritance for the *nouns* (i.e. objects) and *verbs* (i.e. processes).

Thereby, the objects in object-oriented programming can be associated with actions (i.e. methods) while on the other hand the actions (i.e. processes) in this kind of inheritance may be associated with objects.

Generally speaking, instead of specializing objects, this approach aims for specializing processes by associating a specific type to them. These types can then be arranged in a hierarchy, such that every process can be decomposed in its subprocesses.

It has not yet been defined, however, what inheritance means in OWL-S.

[15] Aspects can be methods or attributes
[16] For further information visit (Harold, 1997)

4.5 Inheritance in OWL-S

Since the basic idea of inheritance has its origin in object-oriented programming, it is necessary to transfer this basic idea from object-oriented programming into the domain of OWL-S in order to know what means inheritance in OWL-S.

4.5.1 Transferring inheritance to OWL-S

How this basic idea of inheritance can be transferred from object-oriented programming into the domain of OWL-S can be established by analyzing which elements can participate in an Inheritance Relationship (IR) and which elements get inherited if such a relationship exists, see <Table 1: Transferring inheritance into the domain of OWL-S>.

In object-oriented programming	**← Inheritance →**	**In OWL-S**
Inheritance Relationship (IR) members		
class	The element to inherit from (generalization)	Instance of the OWL-S class: *Service*
	The element that inherits (specialization)	
Reusable elements		
method	Concrete aspects of an element to inherit	OWL-S instances
		WSDL grounding
attribute		WSDL operation

Table 1: Transferring inheritance into the domain of OWL-S

The essential difference between inheritance in object-oriented programming and inheritance in OWL-S is that the former has Inheritance Relationships (IR) between classes and the latter has IRs between instances. This difference has an impact on the advantage of inheritance in OWL-S.

Since the OWL-S atomic process grounding is coupled with the WSDL grounding, and the WSDL grounding itself is coupled with the underlying WSDL operation, the whole process can get inherited: starting from its OWL-S description to its very program code (e.g. java).

4.5.2 Advantage of inheritance in OWL-S

Information reuse in OWL-S		**Already possible**	**Enhanced reuse enabled by IR**
WSDL operation		Yes	-
WSDL grounding		Yes	-
OWL-S instances	unmodified	Yes	-
	modified	-	Yes

Table 2: Enhanced information reuse for OWL-S

In order to find out the advantage of Inheritance Relationships for OWL-S, it is necessary to analyze to what extend the general benefits of inheritance already apply to OWL-S itself: for the two main benefits of information reuse and additional relationship information.

Since OWL-S is based on OWL, instances and their underlying elements can already be reused to a certain degree. It is not yet possible, however, to reuse these instances in a modified way. Therefore, one benefit of IRs in OWL-S is the possibility to customize reused elements, see <Table 2: Enhanced information reuse for OWL-S>.

OWL-S service-service relationship information		Already available	Enabled by IR
Substitute relationship	Implicit (reasoning)	Yes	-
	Explicit	-	Yes
General relationship		-	Yes
Bidirectional relationship		-	Yes

Table 3: Additional information for OWL-S

The additional information about Web Services that can be given by an Inheritance Relationship (IR) is the relationship itself. Additionally, in case such a relationship is stated by both IR members, the IR can provide additional information that the IR is bidirectional. Since every Web Service can theoretically create an IR to any other service, this bidirectional relationship makes the IR official and thereby somehow stronger, see <Table 3: Additional information for OWL-S>.

Furthermore, the special relationship which declares one service as a substitute for another one can also be stated within an IR. This kind of IR can already be provided implicitly by modeling two services in a compatible way, but it is not yet possible to state this IR explicitly such that it can be discovered without the need of reasoning a registry.

Chapter summary

Inheritance in computer science has the benefit of information reuse and providing additional information about the participants. This inheritance is currently most popular in the domain of object-oriented programming and in the domain of knowledge representation. However, inheritance has also been present in the domain of (business) processes. The approach using inheritance for web services is a new development. This thesis presents a solution, how inheritance can be introduced – for the first time – in the domain of OWL-S.

Of special interest is default inheritance, since it allows one not only to reuse information, but also to alter it by overriding or deleting inherited elements.

5 Solution

5.1 Applications of IRs

Since object-oriented programming defines already a set of inheritance applications, the applications Inheritance Relationships (IR) are analyzed in order to find out if it makes sense to transfer them into the domain of OWL-S. Of course, this transfer does not represent an exact mapping, since the IRs of these two domains are somehow different; although it considers the basic underlying ideas of the specific applications.

Application transfer to OWL-S

The underlying ideas of the following applications of inheritance in object-oriented programming make sense to be transferred into the domain of OWL-S; with one exception.

- ✓ **Overriding:** The basic idea of Overriding can be transferred into the domain of OWL-S by introducing *Web Service Customization* and *Web Service Manipulation*.
- ✓ **Subtyping:** The basic idea of Subtyping can be transferred into the domain of OWL-S by introducing *Web Service Extension*.
- ✓ **Polymorphic substitution[17]:** The basic idea of polymorphic substitution (i.e. polymorphism) can be transferred into the domain of OWL-S by introducing two types which an Inheritance Relationship (IR) can have: normal IR and strict IR.
- × **Abstract classes:** One could think of transferring this approach of "having abstract classes" into "having abstract OWL-S services" and therefore having their underlying abstract Web Services. But abstract Web Services do not comply with the philosophy of OWL-S since every Web Service should be invocable (by its own) by potential customers. An abstract Web Service with only intended but not implemented behavior, however, would not be invocable and is therefore not meant to exist.
- × **Multiple-inheritance:** Multiple-inheritance can be transferred into the domain of OWL-S, as mentioned in <Section 5.2.1: Normal IR> below.

The resulting applications of inheritance for OWL-S, which can be used both by normal and strict IR, are described next in the following sections.

All three applications describe a different level of expressiveness and can be used complementary.

- **Web Service Customization:** <Section 5.1.1: Web Service Customization>
- **Web Service Extension:** <Section 5.1.2: Web Service Extension>
- **Web Service Manipulation:** <Section 5.1.3: Web Service Manipulation>

The two different types of Inheritance Relationships (IRs) are described afterwards in the two following sections:

- **Normal IR:** <Section 5.2.1: Normal IR>
- **Strict IR:** <Section 5.2.2: Strict IR>

[17] Polymorphism in object-oriented programming (Harold, 1997)

5.1.1 Web Service Customization

The basic concept of Overriding in object-oriented programming can be transferred to *Web Service Customization* in the domain of OWL-S, as illustrated in <Table 4: Transferring Overriding to the domain of OWL-S>.

Definition

Web Service Customization (WSC) is defined by the expressiveness described in the table below. WSC allows replacing atomic and composite processes in the inherited service model while this must comply with the replacement contract.

In order to provide more convenience for human readers of the service description, each inherited instance can be renamed by altering the according unique identifier (ID) while this must comply with the renaming contract.

In case of multiple-inheritance – i.e. multiple Inheritance Relationships (IR) – the corresponding IRs must comply with the multiple-inheritance contract.

Since there is yet no reason, why process replacements should be possible to prohibit, this is also not possible in the current solution.

All these contracts are defined in <Section 5.5.3: Interpreting Web Service Customization>.

Overriding **in object-oriented programming**	➔	**Service Customization** **in OWL-S**
Replacement		
Method	Inherited aspects to replace	Instance of the OWL-S class: *CompositeProcess*, *AtomicProcess*
Yes (By throwing an exception)	Deletion as a possible replacement	Yes
Contracts		
-	Contract name	Multiple-inheritance contract
		Renaming contract
		Replacement contract
Renaming		
-	Inherited aspects to rename	All inherited instances of OWL-S which have a unique identifier (ID)
Security		
Yes (In some languages)	Possibility to prohibit replacements	No

Table 4: Transferring Overriding to the domain of OWL-S

Benefit

As expected, this Service Customization can be done already to some extend in OWL-S. Using IR, however, allows a more powerful way of customizing a service, see <Table 5: Benefits of IR for Service Customization>.

Service Customization in OWL-S	Already possible	Enabled by IR
Reuse existing processes	Yes	-
Altered reuse of existing service models	-	Yes

Table 5: Benefits of IR for Service Customization

5.1.2 Web Service Extension

The basic concept of Subtyping in object-oriented programming can be transferred to *Web Service Extension* in the domain of OWL-S, as illustrated in <Table 6: Transferring Subtyping to OWL-S>.

Definition

Web Service Extension (WSE) is defined by the expressiveness described in the table below. WSE allows adding and deleting new processes in the inherited service model.

In order to provide the possibility to adjust the process flow, processes get inserted via the desired control construct. Furthermore, inserted processes can be detached from the control constructs which they belong to by default while this must comply with the control construct contract. This contract is defined in <Section 5.5.4: Interpreting Web Service Extension>.

WSE also allows deleting processes via process performance, such that different occurrences of the same process can be handled individually.

Subtyping **in object-oriented programming**	➔	**Service Extension** **in OWL-S**
Extension		
Method, attribute	Elements to add	Process (via control construct)
Contract		
-	Contract name	Control construct contract
Deletion & detachment		
-	Elements to delete	Process (via process performance)
-	Elements to detach	Process (from control construct)

Table 6: Transferring Subtyping to OWL-S

Benefit

As expected, this Service Enlargement can be done already. Using IR, however, allows a more powerful way of enlarging a service. The additional advantages of IR are pointed out in <Table 7: Service Enlargement, enabled by inheritance> below.

Web Service Extension in OWL-S	Already possible	Enabled by IR
Reuse existing process descriptions	Yes	-
Altered reuse of existing process descriptions	-	Yes

Table 7: Service Enlargement, enabled by inheritance

5.1.3 Web Service Manipulation

To enhance the altering possibilities of *Web Service Customization*, this thesis introduces also *Web Service Manipulation* which allows on a very detailed level: the preconditions and effects can be replaced in the inherited service model, and inputs and outputs can be deleted and inserted from inherited processes, see <Table 8: Enhanced altering within Service Manipulation>.

Service Customization in OWL-S	➔	**Service Manipulation** in OWL-S
Replacement		
-	Elements to replace	Expression
Deletion & insertion		
-	Elements to delete and insert	Input
		Output
Contract		
-	Contract name	Result and condition contract

Table 8: Enhanced altering within Service Manipulation

Definition

Web Service Manipulation (WSM) is defined by the expressiveness described in the table above. WSM allows replacing expressions, i.e. results and conditions, of processes in the inherited service model while this must comply with the result and condition contract. This contract is defined in <Section 5.5.5: Interpreting Web Service Manipulation>.

Additionally, WSM allows deleting and inserting inputs and outputs from, respectively into an atomic process.

5.2 Types of IRs

There are two types of Inheritance Relationships (IR): normal IR and strict IR. This section defines those two types of IR and points out briefly the differences between them.

Even though they are different as a whole, they are syntactically represented (modeled) using the same applications of IR. How this modeling of a general Inheritance Relationship (IR) and its applications can be done is shown in <Section 5.4.2: Modeling Inheritance Relationships (IR)>.

The different usage of the IR applications by normal, respectively strict IR is illustrated in <Section 5.3: IR application map>.

5.2.1 Normal IR

Since inheritance can have various appearances, it is necessary to define which kind of inheritance makes sense in the domain of OWL-S.

- ✗ **Complete inheritance:** Complete inheritance[18] is defined such that elements get inherited which can be extended afterwards while redundant information is not allowed to occur. The philosophy of Web Services does not allow complete inheritance because Web Service developers should generally be independent among each other in Web Service creation, in order to allow everyone all around the world to participate. Therefore, redundancy should be allowed to occur which contradicts with the definition of complete inheritance.
- ✓ **Default inheritance:** Default inheritance is defined such that elements get inherited by default which can be modified and extended afterwards (Laurel J. Brinton et al., 2001) such that redundant information is possible to occur. Since redundancy is possible to occur among different Web Services and allowed in default inheritance, default inheritance has been chosen. Furthermore, the advantage of default inheritance is that a specialized element inherits a default which can be altered. This possibility to alter inherited defaults enables the introduced applications in <Section 5.1: Applications of IRs> and thereby gives the proposed IR a benefit.
- ✓ **Multiple-inheritance:** There is no reason why multiple-inheritance should not be allowed for IRs in OWL-S. Multiple-inheritance has the advantage over single inheritance of providing the ability to replace processes of one inherited service with altered processes inherited from another service.

Therefore, the normal Inheritance Relationship (IR) in OWL-S is defined as a relationship between Web Services that allows default inheritance and multiple-inheritance.

5.2.2 Strict IR

In order to improve not only Web Service creation but also Web Service discovery in OWL-S, this thesis defines besides the normal also a strict Inheritance Relationships (IR). This strict IR is similar to a normal IR, but with additional restrictions and different semantics such that in every strict IR one service can automatically be used as completely appropriate substitutes for the other service – in every situation.

Polymorphism

The basic idea about this kind of substitution is adopted from the principle of polymorphism in object-oriented programming (Harold, 1997) where different kinds of specific class types can be used as substitutes for a general one.

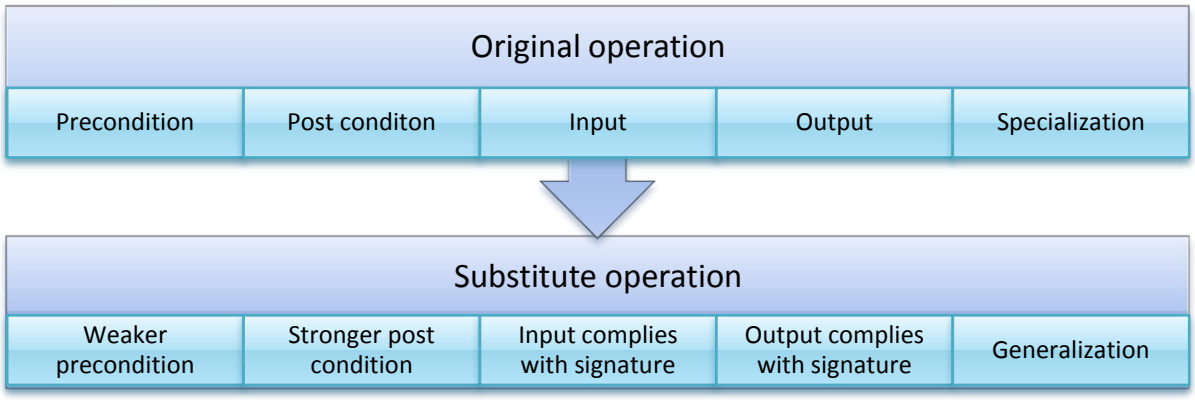
Since the functionality of an OWL-S service can be defined by four data − input, output, precondition and result − the rules needed to describe such substitutes can be created by describing the necessary relationship among these data between two services. Such relationships can be found in the domain of data refinement (De Roever & Engelhardt, 1999).

[18] See http://en.wikipedia.org/wiki/Construction_grammar (Accessed on October 8, 2007)

Refinement

Data refinement can be seen as the conversion of an abstract data model into more concrete data structures. In other words, the abstract data model is more general and occupies a wider range than the refined data structures. Therefore, whenever the refinement applies, the general data model applies also. Model transformation in object-oriented programming defines such a refinement not only for data but for a whole operation: the inputs and outputs of two different operations need to comply both with the signature (i.e. must stay compatible), the preconditions of the operations can get weakened and the post conditions strengthened in the substitute operation. Analogously, whenever the refined operation applies, the original operation applies also, see <Figure 5-A: Refinement in object-oriented programming for an operation>.

Figure 5-A: Refinement in object-oriented programming for an operation



From the other perspective, the weaker preconditions in the original operation allow more situations the original operation can be performed in, and the stronger post conditions describe its effects more precisely. Logically speaking, a condition is weakened if the original condition implies the refined condition and vice versa for the strengthening.

Weakening and strengthening

```
E1: Original element
E2: Refined element

Weakening: E1 ⇨ E2.

Strengthening: E2 ⇨ E1.
```

Refinement for services

In OWL-S, this operation can stand for a service where the inputs, outputs and preconditions match and the post conditions from the operation are mapped to the service results.

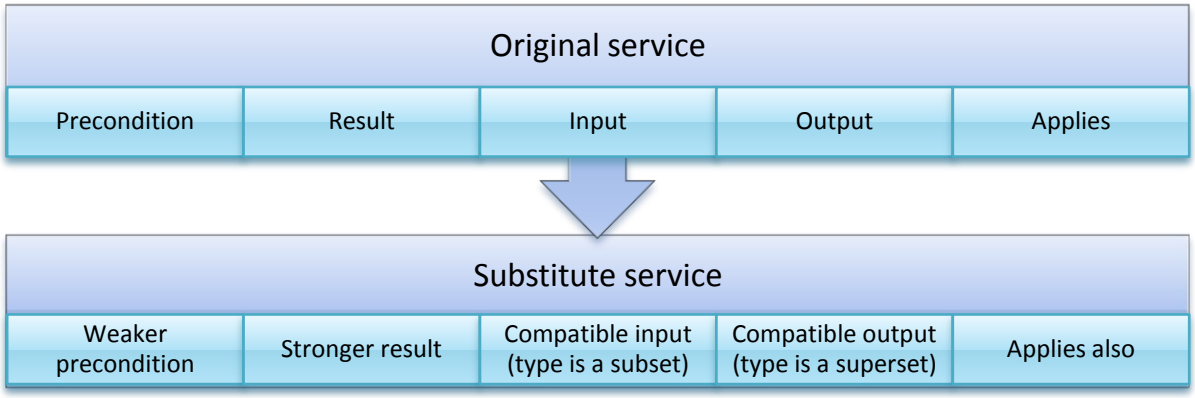
More concretely, the preconditions of a service substitute must be weaker than the ones of the corresponding original service and vice versa for the results.

According to the refinement for an operation in object-oriented programming, the inputs and outputs of those two services should both comply with some kind of signature or contract. Such a contract does not yet exist for services. But intuitively, a similar contract for OWL-S would occupy that the inputs and outputs must be from the same OWL class, respectively data type for each service. Since an OWL class belongs to a hierarchy it would also be possible, however, to apply the

idea of weakening and strengthening to the inputs and outputs. Doing so, analogously to the preconditions and results, the inputs must be weaker and the outputs stronger in the refined service.

More concretely, each input type of a service substitute must be a subset of the corresponding input type of the belonging original service and vice versa for the service output types.

Figure 5-B: Refinement in OWL-S for strict IR between services



Definition

Since such a refinement relationship between services allows exactly the desired substitutability discussed above, this thesis therefore defines strict Inheritance Relationship (IR) as the refinement relationship between a SuperService and a SubService: one service – which one it is concretely can be freely chosen – is a service substitute of the original service such that in case this original service applies to a certain service request, the belonging service substitute applies also.

Therefore, the service substitution can arise in both directions of a strict IR: either a *SuperService* is a strict substitute of the belonging strict original SubService or the *SubService* is a strict substitute of the belonging strict original SuperService.

5.2.3 Comparing strict and normal IR

The difference between strict and normal Inheritance Relationships (IR) can be explained best by looking at two different OWL-S related tasks: Web Service creation and Web Service discovery.

Modeling an IR with the goal of achieving a normal IR, every application[19] of IR can be used to its full extend. Therefore every alteration is allowed as long as it complies with the corresponding contract of the specific application.

Modeling an IR with the goal of achieving a strict IR, every application of IR however can be used as long as the alteration complies with the corresponding contract also but not arbitrary: after the IR is modeled, the two belonging services also have to comply with the refinement rules of strict IR.

These additional restrictions allow a meaningful interpretation of a strict IR for service discovery[20]. It is not yet clear, however, how a normal IR can be useful for service discovery.

[19] See <Section 14: Glossary>
[20] See <Section 1.2.2: Web Service discovery>

Conclusion

In short, strict IR is more restrictive in service creation than normal IR, but provides on the other hand a benefit for service discovery which is yet not the case for normal IR.

5.3 IR application map

This section illustrates the connection between the two kinds of Inheritance Relationships (IR) and the IR applications they include. The different applications themselves of IRs are illustrated in <Figure 5-C: Applications of Inheritance Relationships>.

The strict IR is a (stricter) subset of the normal IR. While the normal IR includes all IR applications, the strict IR only includes *Web Service Customization* and the strict mode of Web Service *Manipulation*. In general, however, all applications can be used complementary.

Additionally, both – normal and strict – IRs can be used for web service creation, while strict IRs can also be used for service discovery. The use of normal IRs for service discovery on the other hand is only meek.

The concrete vocabulary of these applications can be found in <Section 5.1: Applications of IRs>.

Figure 5-C: Applications of Inheritance Relationships

5.4 Create Inheritance Relationships (IR)

This section describes how an IR can be created using an inheritance profile as a new OWL-S profile and stating the specific IR by using either OWL or SWSL as a syntax.

5.4.1 The Inheritance Profile

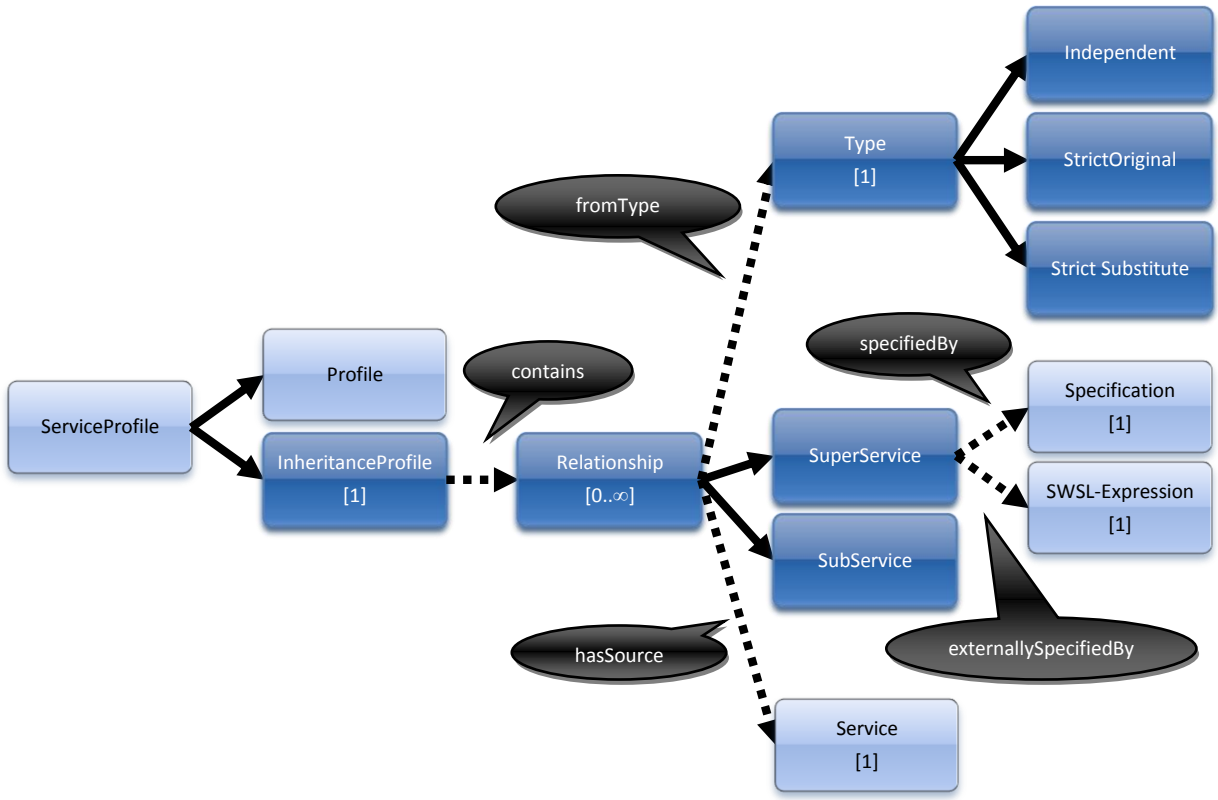
In order to create the proposed Inheritance Relationship (IR), the necessary vocabulary is provided by introducing an Inheritance Profile. This profile describes normal and strict IRs – while in the case of a strict IR, the original-substitute relationship needs to be declared – including the underlying three IR applications Web Service Customization, Extension and Manipulation which allow one to alter (i.e. specify) a normal or strict IR.

The proposed approach makes use of the possibility to provide additional profiles to the default profile as a subclass of the OWL-S *ServiceProfile*, as shown in <Figure 5-D: The proposed inheritance profile>.

Source code (OWL)

```
<owl:Class rdf:ID="InheritanceProfile">

	<rdfs:subClassOf>

		<owl:Class rdf:about="http://www.daml.org/services/owl-
	s/1.1/Service.owl#ServiceProfile"/>

	</rdfs:subClassOf>

</owl:Class>
```

Figure 5-D: The proposed inheritance profile – the vocabulary to create IRs and their specification



5.4.2 Modeling Inheritance Relationships (IR)

It is possible to point either to a SuperService or to a SubService using an instance of the proposed inheritance profile. This is done by pointing the property *contains* to an instance of either class *SuperService* or *SubService*.

Source code (OWL)

```
<owl:ObjectProperty rdf:ID="contains">

	<rdfs:range rdf:resource="#Relationship"/>
	<rdfs:domain rdf:resource="#InheritanceProfile"/>

</owl:ObjectProperty>
```

Those classes are both subclasses of the abstract class *Relationship*.

Source code (OWL)

```
<owl:Class rdf:ID="Relationship">

	<owl:unionOf>

		<owl:Class rdf:ID="SuperService"/>
		<owl:Class rdf:ID="SubService"/>

	</owl:unionOf>

</owl:Class>
```

Such an inheritance relation has to declare clearly whether the current service is either the one that inherits from another service or the one that gets inherited by another service. Therefore, an inherited related service can only be an instance of either the class *SuperService* or *SubService*, but not of both.

Source code (OWL)

```
<owl:Class rdf:about="#SuperService"/>

	<owl:disjointWith><owl:Class rdf:about="#SubService"/></owl:disjointWith>

</owl:Class>
```

Furthermore, since the IR can either be normal or strict, the type of this inheritance must be stated using the property *fromType* which points either to an instance of *Normal* or *Strict*.

Source code (OWL)

```
<owl:ObjectProperty rdf:ID="fromType">

	<rdfs:range rdf:resource="#Type"/>
	<rdfs:domain rdf:resource="#Relationship"/>

</owl:ObjectProperty>

<owl:Class rdf:ID="Type">

	<owl:unionOf>

		<owl:Class rdf:ID="Normal"/>
		<owl:Class rdf:ID="Strict"/>

	</owl:unionOf>

</owl:Class>

<owl:Class rdf:about="#Normal"/>

	<owl:disjointWith><owl:Class rdf:about="#Strict"/></owl:disjointWith>

</owl:Class>
```

In order to show which service is actually meant to be a SuperService or SubService, the property *hasSource* should be used to point to the specific service.

Source code (OWL)

```
<owl:ObjectProperty rdf:ID="hasSource">

      <rdfs:range rdf:resource="http://www.daml.org/services/owl-
      s/1.1/Service.owl#Service"/>
      <rdfs:domain rdf:resource="#Relationship"/>

</owl:ObjectProperty>
```

Specifying the IR

The proposed inheritance profile allows one to modify the default inheritance[21] by either using the property *externallySpecifiedBy* or *specifiedBy*. The former property uses an OWL-S Expression to state the alteration and the latter plain OWL for the same purpose.

How the concrete specifications can be modeled is described in the following sections:

- **Modeling Web Service Customization:** <Section 5.4.3>
- **Modeling Web Service Extension:** <Section 5.4.4>
- **Modeling Web Service Manipulation:** <Section 5.1.3>

For the sake of human readability, the OWL-S Expression with SWSL[22] as the chosen language is used from now on in order to explain modeling the IR specification in more details. The plain OWL representation of the IR specification can be found in <Section 6: OWL-syntax>.

Source code (OWL)

```
<owl:ObjectProperty rdf:ID="externallySpecifiedBy">

      <rdfs:domain rdf:resource="#SuperService"/>
      <rdfs:range rdf:resource="#SWSL-Expression"/>

</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="specifiedBy">

      <rdfs:domain rdf:resource="#SuperService"/>
      <rdfs:range rdf:resource="#Specification"/>

</owl:ObjectProperty>
```

As illustrated in <Figure 2-D>, any logical formalism can be used to state an expression in OWL-S. In order to use the SWSL-Rules as one option to define the IR applications, a new SWSL-Expression has been created in the proposed inheritance profile where the *LogicLanguage* points to the URI of SWSL where those rules are defined.

[21] See <Section 14: Glossary>
[22] In specific, the frame layer is used from this language

Source code (OWL)

```
<Expression:Expression rdf:ID=”SWSL-Expression”>
<rdfs:subClassOf><owl:Restriction>

      <owl:onProperty rdf:resource="&Expression;expressionLanguage" />
      <owl:hasValue rdf:resource="#SWSL"/>

</owl:Restriction></rdfs:subClassOf>
</Expression:Expression>

<Expression:LogicLanguage rdf:ID="SWSL">

      <Expression:refURI rdf:datatype="xsd:anyURI">
      http://www.w3.org/Submission/SWSF-SWSL/
      </Expression:refURI>

</Expression:LogicLanguage>
```

In order to connect the IR specification stated in the inheritance profile – either with an OWL-S Expression or with plain OWL - with the corresponding service model, the assumption is made that every element that is referred in the inheritance profile must have an unique identifier (i.e. a URI[23]) in the belonging OWL-S service model ontology.

Following the OWL-S example services, this assumption turns out to be reasonable, since such identifiers are available in those examples.

5.4.3 Modeling Web Service Customization

First, one can choose whether the service model and groundings get inherited or not. Additionally, one can also adopt a particular process from the SuperService as the new service model in the SubService.

Second, one can – for better human readability – rename all inherited instances from the inherited service model which have a unique identifier. Once the renaming statement has been made, only the new names can be used in other statements.

Third, one can replace inherited atomic and composite processes with either a local process or with an inherited process from another SuperService.

Last, one can delete inherited atomic and composite processes.

Vocabulary of Web Service Customization (SWSL)

```
Inherit[AdoptServiceModel(PID_INHERITED), Processes, Groundings].

Rename[ID_INHERITED *-> ID_REPLACEMENT].

ReplaceProcess[PPID_INHERITED *-> PID_REPLACEMENT:PNS_REPLACEMENT].
```

Shorthand

{x/y} stands for either x or y.

[23] See <Section 14: Glossary>

Placeholders

$PID_{INHERITED}$ stands for the ID[24] value of an inherited instance of either the OWL-S class *AtomicProcess* or *CompositeProcess* and represents generally speaking a process. Since the same process can occur several times in an OWL-S service model by using different performances of that process, when a process gets replaced the process performance gets pointed to instead. This gives one the possibility to handle multiple occurrences of the same process individually.

Therefore, $PPID_{INHERITED}$ stands for the ID value of an inherited instance of the OWL-S class *Perform* and represents generally speaking an occurrence, respectively a perform of a process.

Analogue, $ID_{INHERITED}$ stands for the ID value of an arbitrary inherited OWL-S instance.

$ID_{REPLACEMENT}$ stands for a new ID value (i.e. name) which can be chosen. This value, however, must be different from the inherited ID values in order to avoid conflict.

$PID_{REPLACEMENT}$:$PNS_{REPLACEMENT}$ stands for the URI[25] within which the new process performance can be identified which replaced the inherited one; while $PID_{REPLACEMENT}$ stands for the corresponding ID and $PNS_{REPLACEMENT}$ for the corresponding namespace. In the case where the new process performance is defined locally in the service model of the new service – and not an inherited one from another SuperService – only $PID_{REPLACEMENT}$ is required, because the namespace is already given by the new service ontology.

Statements

The molecule Inherit[] expresses which elements of the SuperService are inherited. One can inherit the service model from the SuperService by stating the method AdoptServiceModel($PID_{INHERITED}$) while $PID_{INHERITED}$ can refer to any process from the SuperService (not only to the process used for its service model). Processes must be stated in any case and expresses that the processes from the service model of the SuperService get inherited. In both cases, if one either adopts a service model or not, the service groundings can be inherited by stating Groundings.

The molecule Rename[] expresses the renaming of inherited OWL-S instances. This renaming means altering the belonging ID of these instances.

The molecule ReplaceProcess[] expresses the replacement of inherited processes performance by connecting it to a new process.

Example

An example how a concrete Web Service Customization can look like is illustrated in the use case in <Section 3.1.1: Create EconomyCongoBuy>.

5.4.4 Modeling Web Service Extension

First, one can insert new processes into the inherited process model, either before or after a specific inherited process performance.

[24] The ID is actually an RDF ID (unique identifier) from the Resource Description Framework
[25] See <Section 14: Glossary>

Second, in order to connect the new inserted processes with the inherited process model (i.e. the process flow), one can model the necessary new bindings which connect inputs and outputs among each other.

Last, in order to define the process flow for the new inserted processes one can also create new control construct for these new processes, respectively process performances. In order to provide full flexibility for choosing in which control construct the new processes should be inserted, it is possible to detach them from the current control construct they are by default member of.

Vocabulary of Web Service Extension (SWSL)

```
InsertProcess[{after/before}(PPID_INHERITED) *-> CCID_NEW:CCNS_NEW].

DetachInsertedProcessFromCurrentControlConstruct[PPID_NEW:PPNS_NEW].

DeleteProcess[PPID_INHERITED].
```

Shorthand

, … stands for similar entries that are meant to follow.

Placeholders

$PPID_{INHERITED}$ stands for an inherited process performance; $CCID_{NEW}$:$CCNS_{NEW}$ stands for a new process performance or a control construct in general which contains process performances.

Statements

The molecule InsertProcess[] expresses the insertion of a new process performances, respectively control constructs into the inherited service model either after or before a specific inherited process perform.

The molecule DetachInsertedProcessFromCurrentControlConstruct[] expresses a detaching of an inserted process performance from the current control construct. This molecule can be applied several times for the same process performance.

The molecule DeleteProcesses[] expresses the deletion of an inherited processes performance.

5.4.5 Modeling Web Service Manipulation

First, one can replace inherited OWL-S Expressions which means one can either replace inherited conditions or effects where conditions can occur as preconditions of processes or as conditions for process results. In order to address the effect, one has to provide the unique identifier for the corresponding result which the effect belongs to, because it is unlikely that the effect itself will have such an identifier.

Second, one can delete inputs and outputs of an inherited process. By doing so, it is no more allowed to inherit the service groundings for the corresponding (atomic) process because of the caused incompatibility between service model and groundings.

Last, one can also add new inputs and outputs to inherited processes. Analogue, by doing so the service groundings for the corresponding (atomic) process cannot be inherited anymore also.

Vocabulary of Web Service Manipulation (SWSL)

```
ReplaceExpressions[{

       Condition(CID) *-> Condition(CID:CNS)/

       Result(RID) *-> Result(RID:RNS)

}].

DeleteInputsAndOutputs[{Output/Input}(APID, {OID/IID})].

AddInputsAndOutputs[{Output/Input}(APID, {OID:ONS/IID:INS})].
```

STRICT and NORMAL mode

Web Service Manipulation can be used in two different modes: strict and normal. These modes correspond with the two Inheritance Relationship (IR) types (strict and normal), see <Section 5.2: Types of IRs>.

In case, Web Service Manipulation is used in *normal mode*, i.e. in a normal IR, every statement introduced in this section can be used.

In the case of the *strict mode*, i.e. when Web Service Manipulation is used in a strict IR, only the ReplaceExpressions[] statement can be used.

Placeholders

CID stands for the ID of an inherited condition ID; CID:CNS stands for an arbitrary condition identifier while a condition can either be a result condition or a precondition; RID stands for an inherited result ID value; RID:RNS stands for an arbitrary result identifier.

APID stands for the ID of an inherited atomic process.

Statements

The molecule ReplaceExpressions[] expresses the replacement of an inherited precondition, result condition or effect with a new one.

The molecule DeleteInputsAndOutputs[] expresses the deletion of either an input or output of an inherited process performance.

The molecule AddInputsAndOutputs[] expresses the insertion of either an input or output into an inherited process performance.

5.5 Interpreting the IR

This section describes how an Inheritance Relationship (IR) and the corresponding IR specification are meant to be interpreted by any software created in future.

5.5.1 SubService and SuperService

An Inheritance Relationship (IR) has exactly two members: a SubService and a SuperService. Such an IR between two services is interpreted as the intent that the SubService wants to inherit from the SuperService. Since the IR allows multiple-inheritance, a SubService can have several SuperServices

using multiple IRs. A service can also be a SuperService and a SubService at the same time but for different IRs.

5.5.2 Official IR

In case that the Inheritance Relationship (IR) is declared in both ontologies of the participating IR members, the IR is made official. However, this official IR is yet open for interpretation and does not have any effect so far.

5.5.3 Interpreting Web Service Customization

The Inheritance Relationship (IR) specification *Web Service Customization* has an effect on the SubService which inherits from a SuperService when it gets interpreted.

The interpretation of each statement is described below. Furthermore, contracts are introduced which the IR specification needs to comply with in order to be able to interpret the IR without conflicts. All the contracts refer to the OWL syntax of the IR specification.

Create a copy

The Inherit[ServiceModel, Processes, Groundings] statement gets interpreted as follows:

The term Processes must always be included in the statement. Therefore, by default, a copy of the ontology which contains the service model of the SuperService gets integrated into the service ontology of the SubService.

If Groundings is included in the statement, a copy of the ontology which contains the groundings of the SuperService gets integrated into the service ontology of the SubService. The groundings ontologies need to be copied in order to be able to extend them with new processes added to the new service model. The WSDL documents do not need to be copied and can stay referenced in the copied ontology.

If ServiceModel is included in the statement, the *presents* property value of the service instance from the SubService gets set to the new URI of the copied service model from the SuperService.

If Groundings is includes in the statement, a *supports* property value gets set for each grounding of the SuperService in the service instance from the SubService to the new URI of the corresponding copied service grounding from the SuperService.

In case any copies are made, the namespace of the copy changes to the namespace of the service ontology from the SubService for both the inherited service model and service groundings. Thereby, possibly conflicting IDs must be renamed either manually or automatically.

- **Contract to comply with:** Multiple-inheritance contract (described below)

Rename

The Rename[$ID_{INHERITED}$ *-> $ID_{REPLACEMENT}$] statement gets interpreted such that the ID value of an instance from this ontology copy which matches $ID_{INHERITED}$ gets replaced by $ID_{REPLACEMENT}$.

- **Contract to comply with:** Renaming contract (described below)

Replace processes

The ReplaceProcess[$PPID_{INHERITED}$ *-> $PID_{REPLACEMENT}$:$PNS_{REPLACEMENT}$] statement gets interpreted as follows:

First the inherited process performance instance gets identified by $PPID_{INHERITED}$. Then, the process resource value of this process performance gets replaced by the composition of $PNS_{REPLACEMENT}$ and $PID_{REPLACEMENT}$ which represent a valid URI together.

Second, the namespace of the parameter resource value of the input bindings of the identified performance gets replaced by the according namespace from the new inputs which can be found via the new process.

Last, the corresponding OWL-S result instance gets identified via the process which the identified performance belongs to. In this result, the output bindings get altered in such a way that the namespace of the parameter resource gets replaced by the according namespace from the new outputs which can be found via the new process.

- **Contract to comply with:** Replacement contract (described below)

Contracts

The necessary contracts to allow an interpretation of *Web Service Customization* without conflict are the followings:

Multiple-inheritance contract: Since an OWL-S service can only have one service mode – in case of multiple-inheritance (i.e. multiple IRs) – the service model can only be adopted once by a SubService, respectively it can only be adopted from one of all of its SuperServices.

Multiple-inheritance contract (First Order Logic – refers to OWL syntax)

$$\forall\, IHP, SS1, SS2, SP1, SP2 : O \cdot$$

$$IHP \in InheritanceProfile \wedge <IHP, SS1> \in ER(contains) \wedge <IHP, SS2> \in ER(contains) \wedge <SS1, SP1> \in ER(specifiedBy) \wedge <SS2, SP2> \in ER(specifiedBy) \wedge <SP1, SM1> \in ER(adoptServiceModel) \wedge <SP2, SM2> \in ER(adoptServiceModel) \Rightarrow$$

$$SS1 = SS2.$$

Renaming contract: The original $ID_{INHERITED}$ must exist in the inherited ontology. The new $ID_{REPLACEMENT}$ must also not conflict with the already existing or inherited ones.

Renaming contract (First Order Logic – refers to OWL syntax)

$$\forall\, SP, RN, X : O;\ \exists\, OID, XID : LV \cdot$$

$$SP \in Specification \wedge <SP, RN> \in ER(containsAltering) \wedge RN \in Renaming \wedge <RN, OID> \in ER(oldID) \wedge <X, XID> \in ER(ID) \wedge$$
$$OID = XID.$$

$$\forall\, SP, RN, X : O;\ \nexists\, NID, XID \cdot$$

$$SP \in Specification \wedge <SP, RN> \in ER(containsAltering) \wedge RN \in Renaming \wedge <RN, NID> \in ER(newID) \wedge <X, XID> \in ER(ID) \wedge$$
$$NID = XID.$$

Replacement contract (First Order Logic – refers to OWL syntax)

$$\forall\, SP, PR, OPP, OP, RP, OI : O;\ OPTc : Vc;\ OPTd : Vd;\ OIID : LV;\ \ \exists\ SRI : O;\ SRPTc : Vc;\ SRPTd : Vd;\ SRIID : LV\ \cdot$$

$$SP \in EC(Specification) \wedge < SP, PR > \in ER(containsAltering) \wedge PR \in EC(ProcessReplacement) \wedge < PR, OPP > \in ER(replaceProcess) \wedge < OPP, OP > \in ER(process) \wedge < PR, RP > \in ER(withProcess) \wedge < OP, OI > \in ER(hasInput) \wedge < RP, SRI > \in ER(hasInput) \wedge < OI, OPTc > \in ER(parameterType) \wedge < SRI, SRPTc > \in ER(parameterType) \wedge < OI, OPTd > \in ER(parameterType) \wedge < SRI, SRPTd > \in ER(parameterType) \wedge < OI, OIID > \in ER(ID) \wedge < SRI, SRIID > \in ER(ID)$$

$$\wedge\, EC(OPTc) \subseteq EC(SRPTc) \wedge EC(OPTd) \subseteq EC(SRPTd) \wedge OIID = SRIID.$$

$$\forall\, SP, PR, OPP, OP, RP, OO : O;\ OPTc : Vc;\ OPTd : Vd;\ OOID : LV;\ \exists\ SRO : O;\ SRPTc : Vc;\ SRPTd : Vd;\ SROID : LV\ \cdot$$

$$SP \in EC(Specification) \wedge < SP, PR > \in ER(containsAltering) \wedge PR \in EC(ProcessReplacement) \wedge < PR, OPP > \in ER(replaceProcess) \wedge < OPP, OP > \in ER(process) \wedge < PR, RP > \in ER(withProcess) \wedge < OP, OO > \in ER(hasOutput) \wedge < RP, SRO > \in ER(hasOutput) \wedge < OO, OPTc > \in ER(parameterType) \wedge < SRO, SRPTc > \in ER(parameterType) \wedge < OO, OPTd > \in ER(parameterType) \wedge < SRO, SRPTd > \in ER(parameterType) \wedge < OO, OIID > \in ER(ID) \wedge < SRO, SROID > \in ER(ID)$$

$$\wedge\ EC(OPTc) \subseteq EC(SRPTc) \wedge EC(OPTd) \subseteq EC(SRPTd) \wedge OOID = SROID.$$

$$\forall\, SP, PR, OPP, OP, RP, OI, OO, RI, RO : O\ \cdot$$

$$SP \in EC(Specification) \wedge < SP, PR > \in ER(containsAltering) \wedge PR \in EC(ProcessReplacement) \wedge < PR, OPP > \in ER(replaceProcess) \wedge < OPP, OP > \in ER(process) \wedge < PR, RP > \in ER(withProcess) \wedge < OP, OI > \in ER(hasInput) \wedge < RP, RI > \in ER(hasInput) \wedge < OP, OO > \in ER(hasOutput) \wedge < RP, RO > \in ER(hasOutput)$$

$$\wedge\ \#EC(OI) = \#EC(RI) \wedge \#EC(OO) = \#EC(RO).$$

$$\forall\, SP, PR, OPP, OP, RP : O;\ \exists\, OPC1 \ldots OPCx, RPC1 \ldots RPCz, OR1 \ldots ORn, RR1 \ldots RRq : \mathbb{B}\ \cdot$$

$$SP \in EC(Specification) \wedge < SP, PR > \in ER(containsAltering) \wedge PR \in EC(ProcessReplacement) \wedge < PR, OPP > \in ER(replaceProcess) \wedge < OPP, OP > \in ER(process) \wedge < PR, RP > \in ER(withProcess) \wedge < OP, OPC1 \ldots OPCx > \in ER(hasPrecondition) \wedge < RP, RPC1 \ldots RPCz > \in ER(hasPrecondition) \wedge < OP, OR1 \ldots ORn > \in ER(hasResult) \wedge < RP, RR1 \ldots RRq > \in ER(hasResult)$$

$$\wedge\ (OPC1 \wedge OPC2 \wedge \ldots OPCx \Rightarrow RPC1 \wedge RPC2 \wedge \ldots RPCz) \wedge (RR1 \wedge RR2 \wedge \ldots RRq \Rightarrow OR1 \wedge OR2 \wedge \ldots ORn).$$

Replacement contract: The inputs and outputs of the process replacement must match number and type with the inputs and outputs of the replaced process in order to maintain compatibility. Additionally, the preconditions and effects of the two processes must comply with the refinement concept <Figure 5-B: Refinement in OWL-S for strict IR between services>:

In case the input, respectively output types of the process replacement are objects, the input types must either be from the same OWL class or from an OWL superclass compared to the original ones; and the output types must either be from the same OWL class or from an OWL subclass compared to the original ones. In case the input, respectively output types of the process replacement are data

types, the condition is adequate in terms of “subset or equal” for input, respectively “superset or equal” for output types.

The preconditions of the replacement process must be weaker and the cumulative results stronger compared to the original ones.

Furthermore, the ID values of the inputs and outputs of those two processes must be the same in order to be able to assign them correctly.

5.5.4 Interpreting Web Service Extension

The effect on the SubService of the Inheritance Relationships (IR) specification using *Web Service Extension* is described in this section.

The interpretation of each statement is described below. Furthermore, contracts are introduced which the IR specification needs to comply with in order to be able to interpret the IR without conflicts. All the contracts refer to the OWL syntax of the IR specification.

Insert processes

The InsertProcess[{after/before}($PPID_{INHERITED}$) *-> $CCID_{NEW}$:$CCNS_{NEW}$] statement gets interpreted such that first the process perform gets identified with the ID value $PPID_{INHERITED}$.

Second, the new control construct, identified by the URI $CCID_{NEW}$:$CCNS_{NEW}$, gets inserted before, respectively after – depending on the stated function after() or before() – the identified process perform by altering the corresponding list or bag.

Change control constructs

The DetachInsertedProcessFromCurrentControlConstruct[$PPID_{NEW}$:$PPNS_{NEW}$] statement gets such interpreted that the process performance $PPID_{NEW}$:$PPNS_{NEW}$ gets detached from the control construct which it directly belongs to if that is possible.

- **Contract to comply with:** Control construct contract (described below)

Delete processes

The DeleteProcess[$PPID_{INHERITED}$] statement gets interpreted such that first the specific inherited process performance gets taken out of the control construct it is included in.

Second, the inputs, outputs, preconditions and results of the composite processes get replaced by the newly computed ones.

Contracts

The necessary contracts to allow an interpretation of *Web Service Extension* without conflict are the followings:

Control construct contract: A process performance has always to belong (directly or indirectly) to at least one control construct in order to provide a valid OWL-S service model.

Control construct contract (First Order Logic – refers to OWL syntax)

$$\forall SP, PI, IP;\ \exists PF, P, CCL, CCB, CC : O \cdot$$

$$SP \in EC(Specification) \wedge <SP, PI> \in ER(containsAltering) \wedge PR \in EC(ProcessInsertion) \wedge <PI, IP> \in ER(insert) \wedge PF \in EC(Perform) \wedge <PF, P> \in ER(process) \wedge ((CCL \in EC(ControlConstructList) \wedge <CCL, PF> \in ER(first)) \vee (CCB \in EC(ControlConstructBag) \wedge <CCB, PF> \in ER(first)) \vee (CC \in EC(ControlConstruct) \wedge <CC, PF> \in ER(then) \cup ER(else) \cup ER(whileProcess) \cup ER(untilProcess))) \wedge$$

$$IP = P\ .$$

5.5.5 Interpreting Web Service Manipulation

The effect on the SubService of the Inheritance Relationships (IR) specification using *Web Service Manipulation* is described in this section.

The interpretation of each statement is described below. Furthermore, contracts are introduced which the IR specification needs to comply with in order to be able to interpret the IR without conflicts. All the contracts refer to the OWL syntax of the IR specification.

Replace results and conditions

The ReplaceExpressions[{Condition(CID) *-> Condition(CID:CNS)/Result(RID) *-> Result(RID:RNS)}] statement gets interpreted such that first the inherited element on the left side of the *-> gets identified by its ID value CID in case of a condition, respectively RID in case of a result.

Second, a Condition(CID) *-> Condition(CID:CNS) statement gets interpreted such that the identified condition gets replaced by an either inherited or new condition identified by its ID and namespace value CID:CNS.

A Result(RID) *-> Result(RID:RNS) statement gets interpreted such that the identified inherited result gets replaced by either an inherited or new result identified by its ID and namespace value RID:RNS.

- **Contract to comply with:** Result and condition contract (described below)

Alter inputs and outputs

The DeleteInputsAndOutputs[{Output/Input}(APID, {OID/IID})] statement gets interpreted such that the instance of an OWL-S output, respectively input identified by its ID value OID respectively IID gets deleted from an inherited atomic process which can be identified by its ID value PID.

Additionally, every input and output binding gets deleted in case the source points to the deleted input, respectively output.

The AddInputsAndOutputs[{Output/Input}(APID, {OID:ONS/IID:INS})] statement gets interpreted such that new inputs, respectively outputs get inserted into the inherited atomic process which can be identified by PID. The inserted inputs, respectively outputs themselves get a reference to their identification OID:ONS, respectively IID:INS.

Finally, the inputs and outputs of the composite processes (including the one which represents the service model) get replaced by the newly computed ones.

Groundings

In case, the service groundings are not left out, i.e. get inherited, the specific groundings for every (atomic) process a Web Service Manipulation is applied on get removed from the copied ontology.

Contracts

The necessary contracts to allow an interpretation of *Web Service Extension* without conflict are the followings:

Result and condition contract: When Web Service Manipulation is used in strict mode, i.e. in a strict Inheritance Relationship (IR), the replaced results and conditions have to comply with the refinement principle for strict IRs, as described in <Section 5.2.2: Strict IR>.

Therefore, in case of strict IRs, the new conditions must be weaker and the new results stronger than the old ones.

Result and condition contract (First Order Logic – refers to OWL syntax)

$$\forall SP, RR, CR : O;\ \ OR, NR, OC, NC : \mathbb{B} \cdot$$

$$SP \in EC(Specification) \land < SP, RR > \in ER(containsAltering) \land < SP, CR > \in ER(containsAltering) \land RR \in EC(ResultReplacement) \land CR \in EC(ConditionReplacement) \land < RR, OR > \in ER(replaceResult) \land < RR, NR > \in ER(withResult) \land < CR, OC > \in ER(replaceCondition) \land < CR, NC > \in ER(withCondition) \land$$

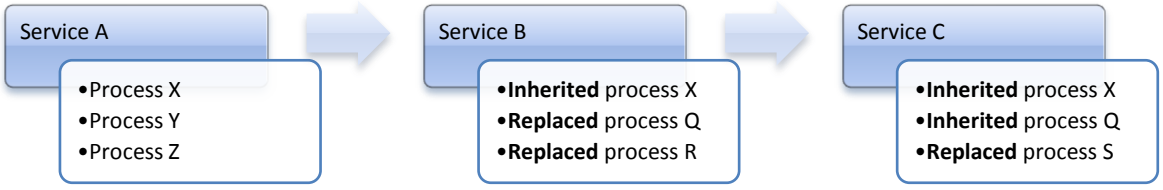
$$OC \Rightarrow NC \land NR \Rightarrow OR.$$

5.5.6 Downward propagation of IR changes

The Inherit[] statement interpretation from Web Service Customization has to be defined in the case, where a SubService of a certain Inheritance Relationship (IR) is itself also a SuperService of another IR with a third service, e.g. “C” inherits from “B” and “B” inherits from “A”.

In such a case, when “C” inherits from “B” – while “B” is a SubService of “A” and a SuperService of “C” at the same time – the Inherit[] statement in “C” must be interpreted after the Inherit[] statement in “B” is interpreted or generally speaking: the Inherit[] statement must be interpreted top down, beginning at the top of the inheritance chain. This way, all the changes made using *Web Service Customization*, *Extension* and *Manipulation* get propagated downwards within the chain – if they are not overridden themselves.

Figure 5-E: Downward propagation of changes within the inheritance chain



In <Figure 5-E: Downward propagation of changes within the inheritance chain> above where service “C” inherits from service “B” and “B” from “A”, the changes in service “B” (replace process “Y” with

"Q", and "Z" with "R") are propagated downwards to service "C", except for the one change that is itself overridden (replace process "R" with "S").

5.5.7 Service substitutes

Both the SuperService and SubService of a strict IR can be interpreted as a service substitute for the corresponding service in the IR, as it is explained in detail in <Section 5.2.2: Strict IR>.

5.6 Validating the IR

An Inheritance Relationship (IR) validation should take place during and after the modeling of the IR specification in order to ensure a conflict free interpretation of the IR by checking the corresponding conditions for *Web Service Customization*, *Extension* and *Manipulation*.

An IR is valid if all corresponding conditions of the IR interpretation are satisfied.

5.6.1 Normal IR

In general, the SubService remains in dependency with its SuperService after having modeled and interpreted an Inheritance Relationship (IR), because the SubService can indeed copy the service descriptions but not the underlying executable program itself it is reusing, respectively inheriting also. Therefore, whenever changes are made in the SuperService, the SubService has to check whether the executable program of the SuperService is still compatible, respectively without conflict with its service description.

An exception – where there generally is no need for such a validation – defines the case where only the service model but not its groundings get inherited. In this case, all the information which is subject matter of the IR can be copied into the SubService and therefore the SubService is not dependent of the SuperService after the IR is once interpreted.

Since a SuperService is probably likely to change, a solution is needed to maintain the functionality of a SubService. One solution is simply to assume that its SuperService changes (possibly) permanently and therefore the IR needs to be validated first every time before the SubService gets executed. After which change a validation would actually be necessary is described in the following. Three different kinds of changes have to be taken care of in order to maintain the functionality of the SubService using a normal Inheritance Relationship (IR): changes of (a) the service model, (b) the groundings and (c) the underlying executable program.

Changed service executables

In case the service executables change – e.g. the program code was optimized and runs now faster – but the service groundings and model stay still the same in the SuperService, no inheritance specific validation is necessary.

Verifying whether the underlying executable program corresponds to the specified OWL-S service description is a general OWL-S topic and not in the scope of this thesis.

Changed service groundings

When the service groundings change in the SuperService, the underlying executable program must also have changed such that it is in general not compatible anymore with the old service description.

In order to assure that the SubService still functions properly, the Inheritance Relationship (IR) has to be validated.

Changed service model

In the case where the service model of the SuperService changes, no validation is required because the service model does not directly influence the executable program.

Only if the changes of the service model also result in changes of the service groundings, a validation is required. Otherwise, the copy of the old service model can still be used together with the unchanged service groundings.

Solution if validation fails

If the validation fails, there are two possibilities for further proceedings:

- One could try to interpret the IR again and then try to validate the updated SubService.

5.6.2 Strict IR

In general, when using strict Inheritance Relationships (IR), the IR applications cannot be used to their fully extend in order to comply with the definition of strict IR. The different expressivenesses for normal and strict Inheritance Relationships (IR) are illustrated in <Figure 5-C: Applications of Inheritance Relationships>:

In case *Web Service Extension* or *Web Service Manipulation: NORMAL* is used in a strict IR, the IR validation fails.

Additionally, in case the service model for the SubService gets adopted from its SuperService, the adopted service model must be the service model of the SuperService (and cannot be another process in the process tree of the SuperService).

Strong dependency

Since the SubService of a strict Inheritance Relationship (IR) not only needs to comply with the underlying executable program of the SuperService but also with the service model, the strict IR dependency of a SubService from its SuperService is generally stronger compared to normal IR.

The strict IR has the same rules when a validation is required as the normal IR; except for the case when the service model changes in the SuperService.

Changed service model

Unlike for normal IR, if the service model changes it is necessary to validate the IR since the service model of the SubService has to stay in the specified strict relationship with the service model of its SuperService.

Solution if validation fails

If the validation fails, there are two possibilities for further proceedings:

- One could try to convert the strict IR into a normal IR and try to validate again.
- One could also try to interpret the IR again and then try to validate the updated SubService.

5.7 Chapter summary

This chapter explains how some applications of inheritance from object-oriented programming can be transferred into the domain of OWL-S. Thereby the applications *Web Service Customization*, *Web Service Extension* and *Web Service Manipulation* are introduced. These applications can be used in two ways: either for a normal Inheritance Relationship (IR) or for a strict IR while the latter one uses the idea of refinement in order to provide benefits for web service discovery.

Additionally, this chapter provides a concrete syntax how each of these applications can be made use of. Thereby, a first section illustrates how an IR can be modeled. For this reason, the Inheritance Profile is introduced together with two syntaxes to describe the IR specification (OWL and SWSL). Then, a second section illustrates under which conditions (i.e. contracts) and how the modeled IRs can be interpreted such that they concretely create, alter or discover an OWL-S service.

Finally, in order to ensure a conflict free interpretation of a modeled IR, a last section illustrates when a specific IR is valid.

6 OWL-syntax

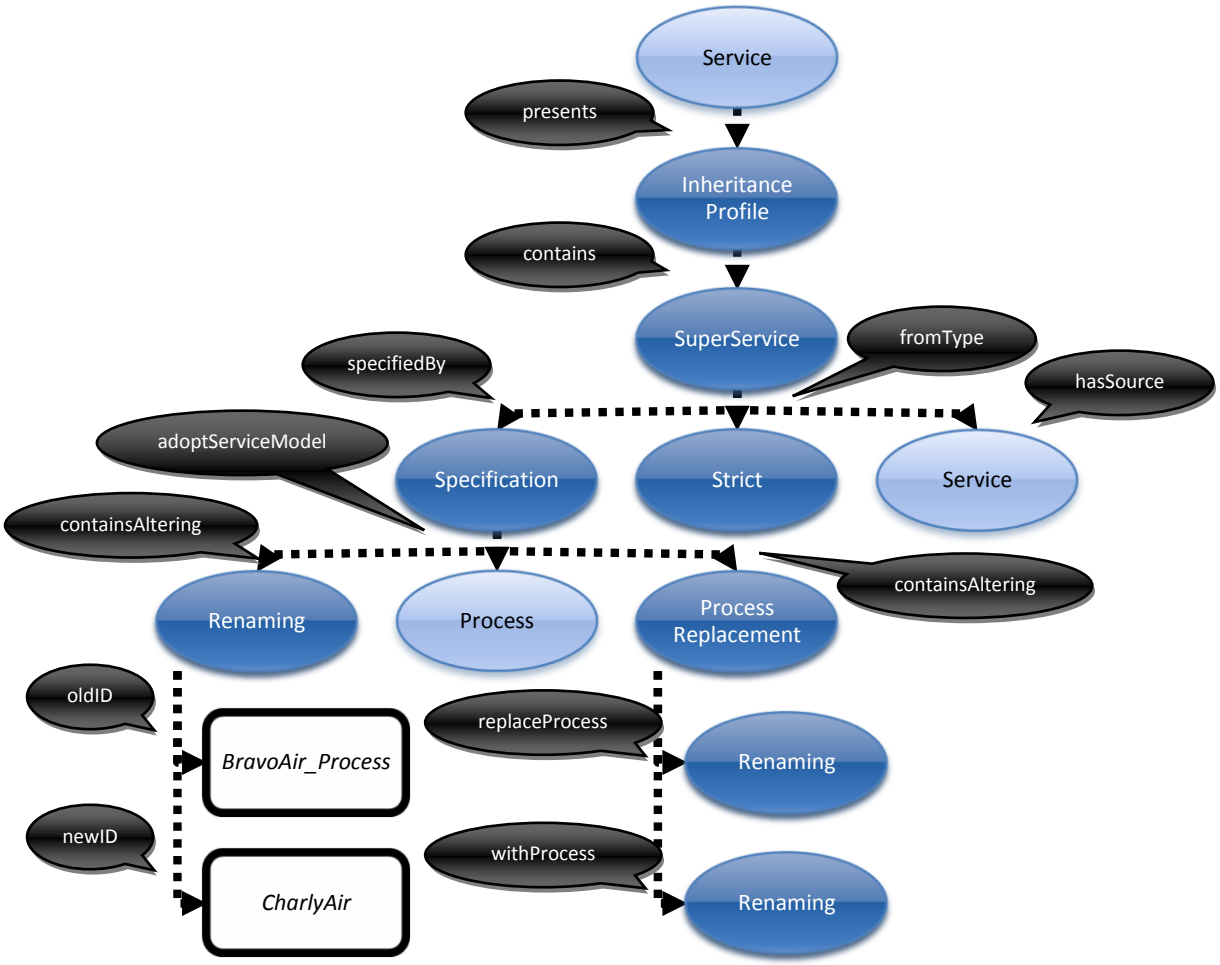
The complete vocabulary for the OWL-syntax which can also be used to represent the Inheritance Relationships (IR) specifications (as an alternative for the SWSL-syntax used before in order to provide optimal readability) is provided within the ontology that describes the Inheritance Profile. This ontology can be found in <Section 15: Attachments>.

Example

In order to illustrate a concrete Inheritance Relationship including its specification using the OWL-syntax, a visual example is provided of the CharlyAir use case from <Section 3.1.3: Create CharlyAir>.

This example illustrates from which classes instances are needed in order to describe, i.e. specify, the Inheritance Relationship (IR) between CharlyAir and BravoAir using the OWL-syntax. In the example, only the names of the classes are given, the names of the concrete instances are not, see <Figure 6-A: The main instances used to model the IR for the use case CharlyAir>.

Figure 6-A: The main instances used to model the IR for the use case CharlyAir



7 Dismissed approaches

Maintaining IR using OWL import

One idea was to maintain the IR using the import function of OWL. At first sight, this approach seems reasonable since importing one or more service ontologies offers already powerful capabilities:

a) Reuse and extension of existing services in service creation.
b) Discovery of related services by reasoning about the import statements made in service ontologies.

Taking a closer look at the approach, however, brings up a major disadvantage for point a):

- × **No altering possible of imported instances:** Once the ontologies of a desired service have been imported, the belonging OWL-S instances can indeed be reused by refereeing to them, they cannot be altered, however. Therefore, one could reuse single processes of the service model by referring to them while creating a new service model; but being not able to alter inherited instances prohibits one to inherit the service model as a whole and adjust it for the new service by altering small details – which is a major disadvantage.

Modeling IR within a new host property

It would be possible to model the Inheritance Relationship (IR) within the default service profile as a new host property, see <Figure 2-C>.

The following parameter parts for the host properties of a service profile already exist in this ontology:

- Service categories (using NAICS and UNSPSC)
- Service parameters (actual parameters: response time, duration and geographic radius)
- Quality rating (no further specification available at this time)

However, using the OWL-S service parameter to preserve the IR would hide this important part of the structural service description behind a somehow technical service aspect.

8 Design decisions

This section describes the design decisions taken for the formal part of the solution. Any design decision taken regarding the prototype is described in <Section 9.1: Prototype>.

Connecting OWL-S services

As described in the assignment in <Section 1.4: Assignment>, the essence of this thesis is to create connections, i.e. Inheritance Relationships, between OWL-S Web Services. The proposed solution models the direct connection of this relationship within an additional OWL-S service profile – as described in <Section 5.4.1: The Inheritance Profile> – using OWL. OWL has been chosen because it provides additional, directly accessible new information for the corresponding ontology without the need of a workaround using a different language (within an OWL-S Expression).

Language decision for IR specification

The Inheritance Relationship (IR) specification on the other hand provides no additional information in the philosophy of OWL because it does not describe a service in more detail, but describes the *changes* made on a service in order to obtain the result of Web Service Customization, Extension and Manipulation. Therefore, the IR specification has not primarily the character of describing a stable ontology but rather a temporary transformation.

Since there is no general benefit of OWL compared to SWSL, the proposed solution lets it yet open to be free to choose between the two languages OWL and SWSL for describing the IR specification, i.e. both syntaxes are supported in the proposed formal solution.

IR application bundles

The Inheritance Relationship (IR) applications are organized in the three main bundles *Web Service Customization (WSC)*, *Web Service Extension (WSE)* and *Web Service Manipulation (WSM)* because they all have different properties. WSE can only be used for normal IRs, while WSC and WSM can be used for both normal and strict IRs. WSM allows changes on an (atomic) process such that the corresponding groundings must be excluded, while WSC and WSE do not produce any incompatibility between service model and service groundings.

All operations offered by WSM produce an incompatibility between the service model and the service groundings. Some of them, however, can preserve the process flow of the service while the others cannot. This is the reason why WSM is additionally organized in two different parts: STRICT mode and NORMAL mode while the STRICT mode preserves the process flow and the NORMAL mode does not necessarily.

9 Evaluation

This section evaluates the proposed solution by illustrating how the motivating tasks in <Section 1.2: Motivation> can be accomplished using the prototype.

For that purpose the prototype is introduced in the first part by presenting its architecture and its release notes. The second part discusses the motivating tasks via the corresponding use cases in the light of the prototype and compares these tasks with their current existing alternative.

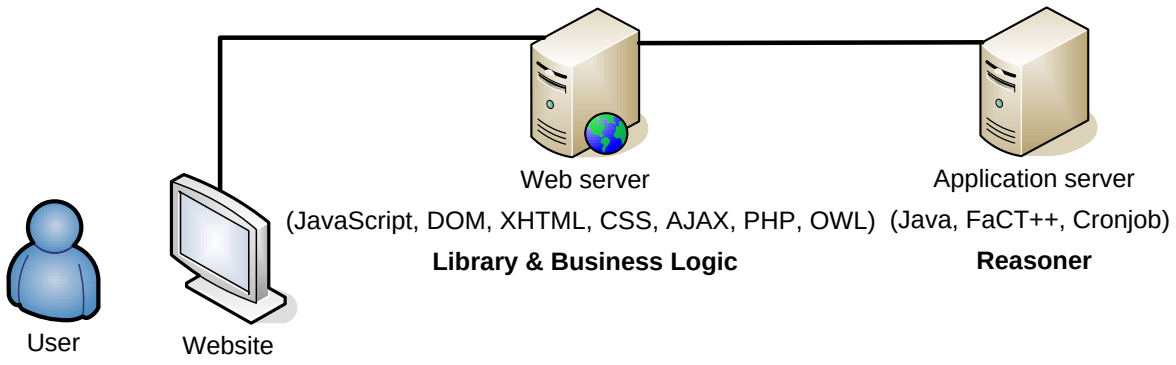
9.1 Prototype

9.1.1 Architecture

The architecture of the prototype consists of three main components: the website, the web server and the application server, see <Figure 9-A: Prototype architecture>.

The website consists of the graphical user interface and handles the human-computer interaction. The website is hosted by the web server. The hosting includes also the scripts which contain the function library and the main part of the business logic. The other part is located on the application server as a Java application, together with the reasoner FaCT++ to perform the reasoning tasks.

Figure 9-A: Prototype architecture



Metaphorically speaking, the brain and body of the prototype is the web server, the face is the website, and the application server is one supporting hand.

Design decisions

XHTML and CSS have been chosen for creating the graphical user-interface (GUI) because they are aligned to build GUIs.

JavaScript has been chosen for creating the business logic and the library because it fits very well together with XHTML and CSS. The richest JavaScript functionality today is provided by the web browser Firefox, therefore the prototype makes use of its DOM specification[26] in order to parse and create OWL files.

[26] Gecko DOM Reference: http://developer.mozilla.org/en/docs/Gecko_DOM_Reference (Friday, September 21, 2007)

Since the prototype should be able not only to work with existing data, but also with online data from arbitrary sources, AJAX together with PHP have been chosen to provide simple file reading and writing functionality for JavaScript.

Java has been chosen to create the needed functions in order to perform the needed OWL reasoning tasks because the most evolved API's to access an OWL reasoner is provided for Java at the moment. For the performing reasoning tool FaCT++ was chosen, since it seems to be best in performance. The reasoner can be replaced at any time with any other DIG (Turi, 2004) compatible reasoner[27].

Since the current Java Protégé API[28] does not yet allow constructing a stable running reasoning application, a UNIX cronjob[29] has been additionally chosen which starts the Java (reasoning) application anew every time the application is invoked and lets it terminate after having answered a request.

9.1.2 Release notes: version 1.0

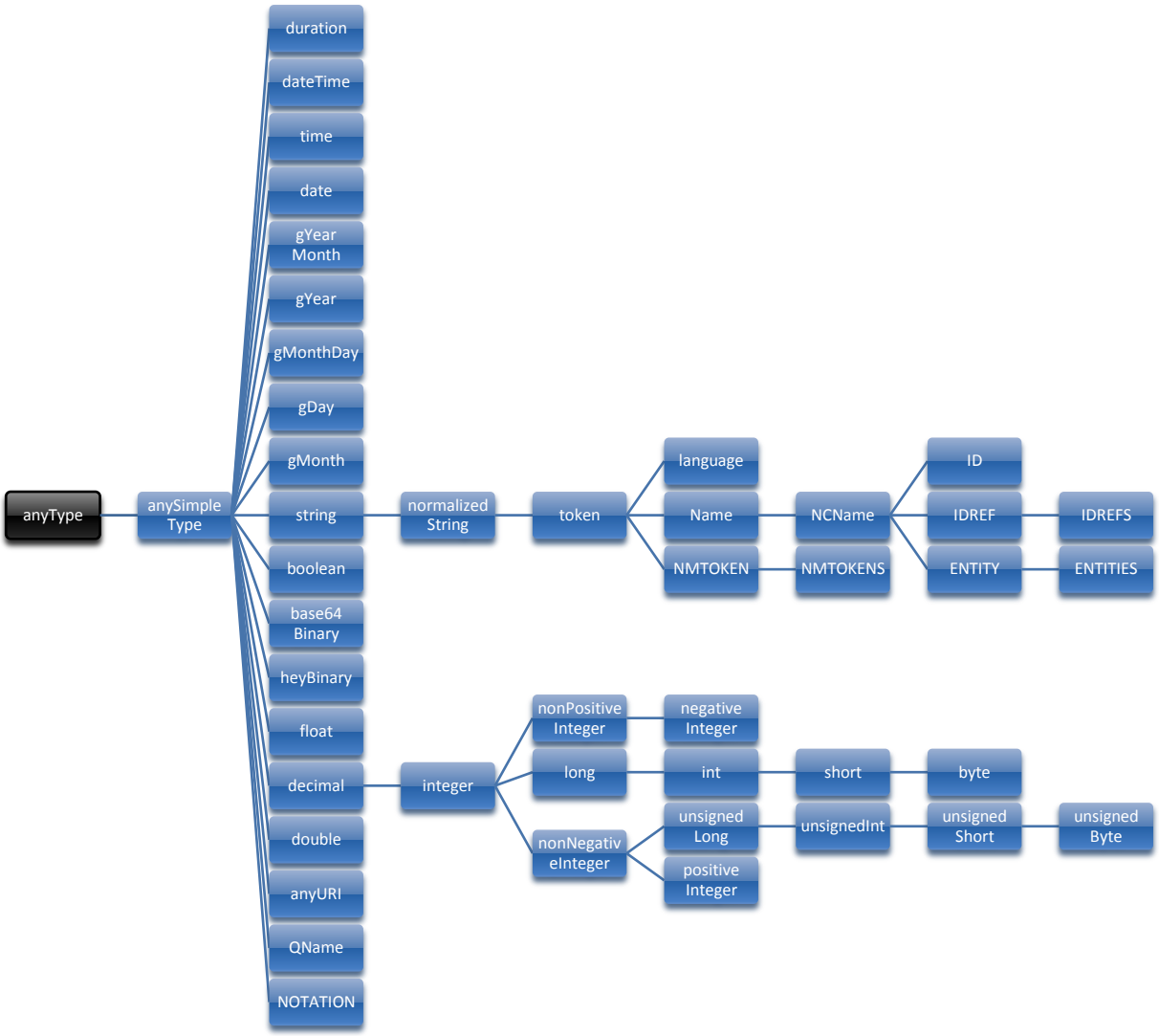
- **OWL import:** Since JavaScript does support XML, but not directly OWL, there is no OWL import available for JavaScript. The solutions provided in the prototype to handle OWL imports manually are the following:
  - **Passive import handling:** OWL imports are only considered, in case the current ontology references somewhere to a specific resource from the corresponding imported ontology (e.g. with the RDF property "resource" or "about"). Within this approach, however, statements about the current ontology which are made in an imported ontology are not considered. In most cases, and also in the case of the used example ontologies, this passive import handling is sufficient for the parsing tasks of this prototype.
  - **Active import handling:** The active import handling on the other hand considers OWL imports – whether there are somewhere referenced in the current ontology or not. Since this approach turned out to be very slow, it is disabled in the current release of the prototype.
- **Inheritance type support:** Single inheritance is fully supported by the prototype. Multiple-inheritance is not implemented yet.
- **IR application support:** Web Service Customization and Web Service Manipulation: STRICT are fully supported in the prototype. Web Service Extension is not implemented yet and Web Service Manipulation: NORMAL is only available in case the adopted service model is an atomic process. Otherwise it would be necessary to compute the new resulting inputs and outputs for the belonging composite processes which is yet still a very complex task.
  However, in order to illustrate the potential of Web Service Manipulation, Web Service Manipulation: STRICT is available also for composite adopted service models; the belonging conditions and results do not get newly computed and updated though.
- **Grounding reuse support:** The prototype supports the reuse of the service groundings. A partial reuse of the service groundings, however, is not yet supported. Therefore, Web

[27] Theoretically, there would be no need for Java, since the DIG interface could be used directly (which is XML based and therefore independent of any programming language). The current release DIG 1.1, however, does not provide sufficient functionality for direct use. The scheduled future release DIG 2.0 (DL Implementation Group, 2006) although plans to provide such functionality and could therefore be used in future versions of this prototype instead of Java.
[28] See http://protege-owl.sourceforge.net/javadoc/index.html (Accessed on: Friday, September 21, 2007)
[29] See http://en.wikipedia.org/wiki/Crontab (Accessed on: Friday, September 21, 2007)

Service Manipulation is only allowed in case the service groundings are left out completely in the Inheritance Relationship – i.e. excluding only specific processes is not supported.

- **Service creation:** For the convenience of the user of the prototype, the new services created within the prototype get automatically saved on a web server.
- **OWL syntax:** OWL (not SWSL) is used for the specification of the Inheritance Relationship since the prototype already needs to understand OWL because of the OWL-S services.
- **Downward propagation support:** The prototype supports the downward propagation of changes within the inheritance chain, as described in <Section 5.5.6: Downward propagation of IR changes>. In order to illustrate this support in the prototype, the DeltaAir service inherits from the CharlyAir service, and the CharlyAir service inherits from the BravoAir service.
- **Reinterpretation support:** The prototype supports the reinterpretation of the Inheritance Profile of a service. Therefore, the modeled Inheritance Relationship (IR) gets interpreted anew every time the corresponding service is used within the prototype.

Figure 9-B: XML Schema built-in data type hierarchy



Data types

The data types illustrated in <Figure 9-B: XML Schema built-in data type hierarchy> are used in the prototype in order to check the compatibility among inputs, respectively outputs of a process replacement.

9.1.3 Special feature

Automatic process structure generation: The prototype is able to present the process structure from the service model of a Web Service automatically by parsing the corresponding service ontology. This can be a big help in finding out whether a service could be reused in service creation or not because this process structure offers a very succinct and at the same time relevant insight, respectively overview for a service.

9.1.4 Application

The actual prototype, including its source code, can be found in <Section 15: Attachments>. The online version of the prototype is available at:

http://www.fo-ss.ch/simon/DiplomaThesis/IR_prototype/.

<Figure 9-C: Screenshot of prototype home page> shows a screenshot from the start screen of the prototype.

Figure 9-C: Screenshot of prototype home page

9.2 Accomplish the motivating tasks

This section illustrates how the motivating tasks in <Section 1.2: Motivation> can be accomplished in the light of the prototype by providing a detailed walkthrough for each use case.

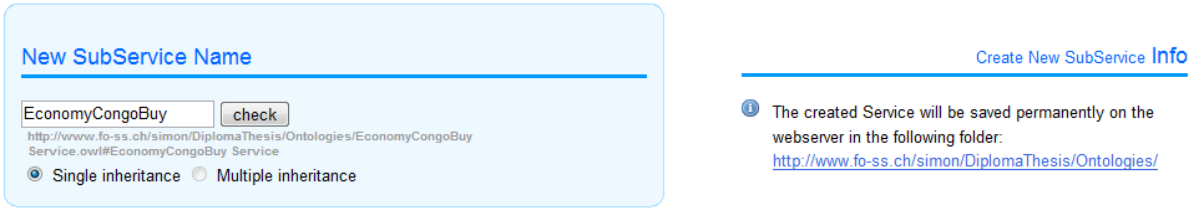
9.2.1 Walkthrough: Create EconomyCongoBuy

This walkthrough illustrates, how the prototype can handle the use case in <Section 3.1.1: Create EconomyCongoBuy>. Using the Inheritance Relationships, the new service *EconomyCongoBuy* can be created within the following steps.

Step 1: Providing a new service name

A new service name needs to be given in order to create a new and unique service identifier, see <Figure 9-D: Provide a new service name>.

Figure 9-D: Provide a new service name



Step 2: Selecting SuperService

The SuperService needs to be selected from which the new service wants to inherit. In this case, the SuperService is the *ExpressCongoBuy* service, see <Figure 9-E: Select ExpressCongoBuyService as SuperService>.

Figure 9-E: Select ExpressCongoBuyService as SuperService

Step 3: Adopting the service model

The SuperService groundings need to be excluded from the Inheritance Relationship (IR) since Web Service Manipulation[30] (on the only process the service consists of) is needed when replacing the positive result. Because “strict substitute” has been selected in the previous step, the service model from the SuperService gets automatically adopted, see <

Figure 9-F: Adopting the service model from ExpressCongoBuy>.

Step 4: Replacing result

The positive result needs to be replaced by a new one provided in the confirm window. After that, the new service can be created by clicking the finishing button, see <Figure 9-G: Replacing the positive result of the ExpressCongoBuy service>.

[30] Web Service Manipulation is only allowed if the service groundings are not inherited, as described in <Section 5.5.4: Interpreting Web Service Manipulation>

Figure 9-F: Adopting the service model from ExpressCongoBuy

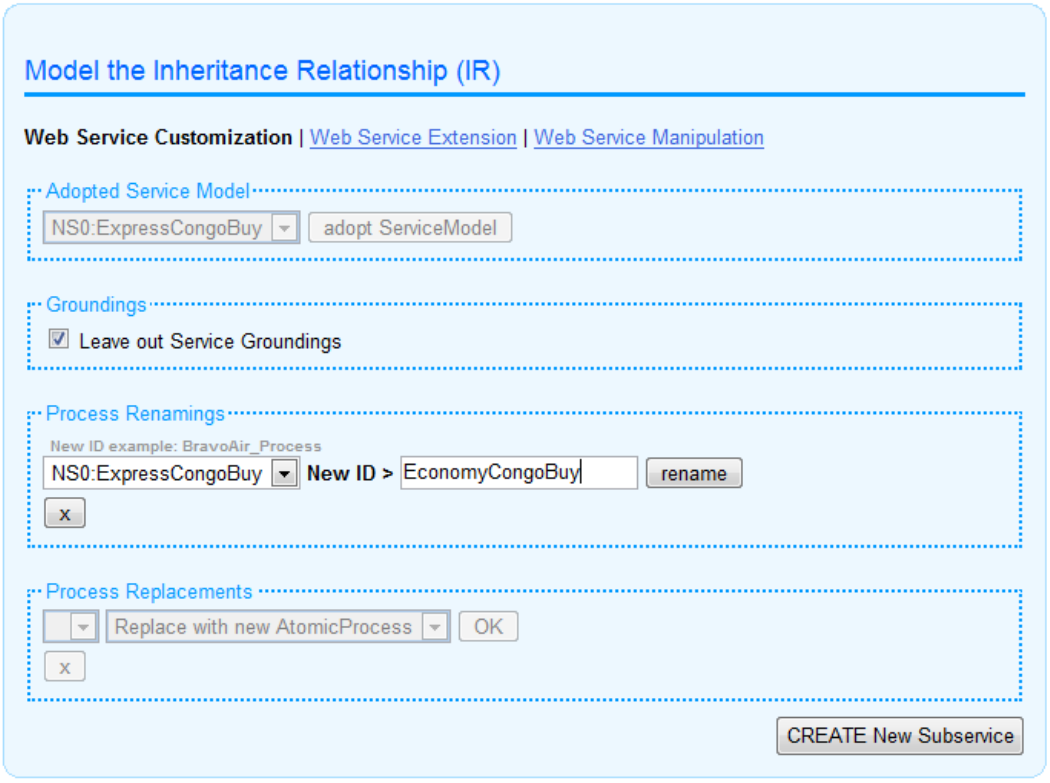
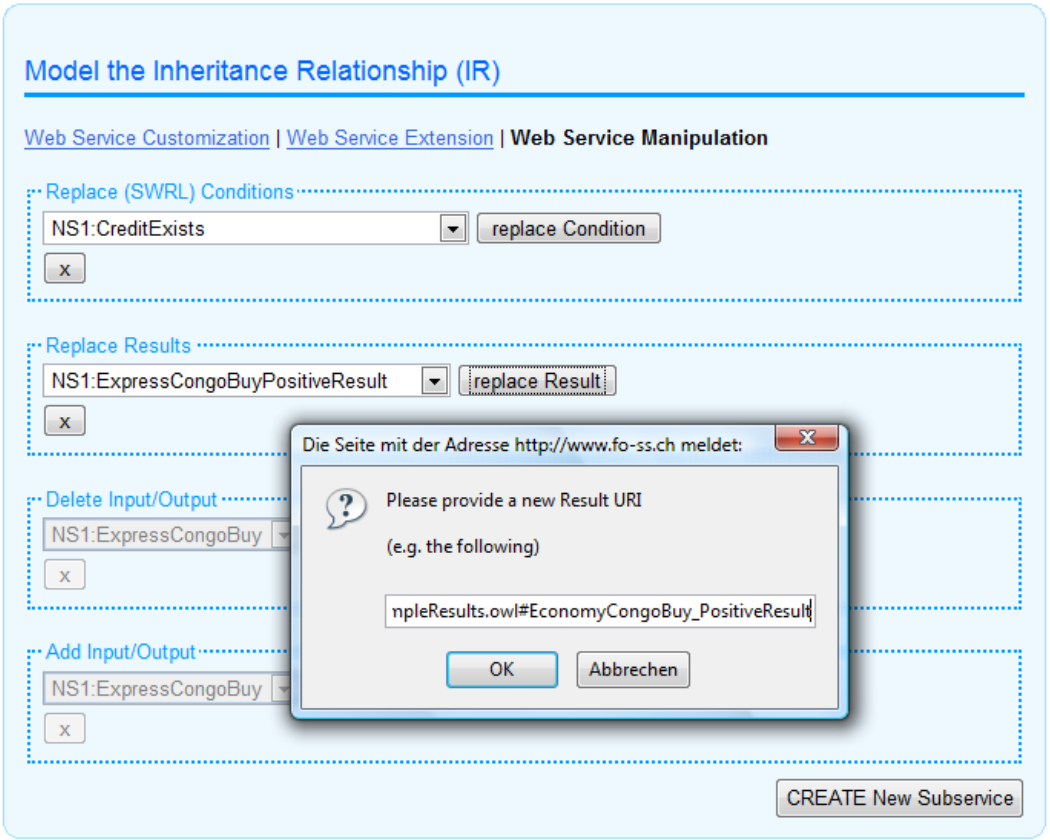


Figure 9-G: Replacing the positive result of the ExpressCongoBuy service



Step 5: The new service

The new service gets saved on a web server and is ready to be interpreted. Since the groundings do not get inherited, they need now to be created and added to the new web service. Additionally, a new service profile should be added to the new service.

Now the new service consists of an instance of the OWL-S Service which is connected with a complete inheritance profile. This profile contains all the previous adjustments made. When the new service needs to be read as a regular OWL-S Service, the inheritance profile needs to be interpreted first.

Comparison to current alternative

Currently, the new service EconomyCongoBuy needs to be built anew, the service model could not be reused. The only way to profit from the existing ExpressCongoBuy would be to point to already existing instances (conditions, results) while creating the new service model.

9.2.2 Walkthrough: Smooth substitution with ExpressCongoBuy

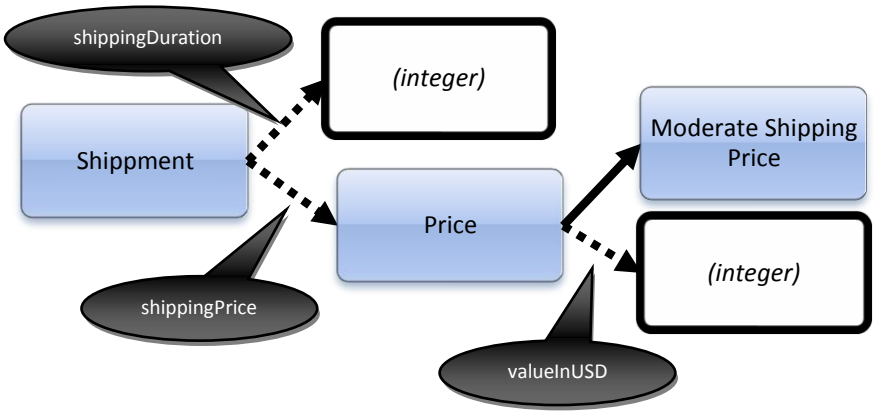
In order to illustrate the use case in <Section 3.1.2: Smooth substitution with *ExpressCongoBuy*> in more detail, a concrete walkthrough is provided which covers the main interactions between the customer and the system that performs the two services *EconomyCongoBuy* and *ExpressCongoBuy*[31] in the light of the prototype.

The customer's demand can be formulated as a logical expression, stating that he wants to buy a book which should be shipped for less than $15 USD and within one week, using the ontology illustrated in <Figure 9-H: Auxiliary shipment ontology>. Since the OWL-S profile can be used for both advertising and requesting a service, this expression could be stated as an effect in an OWL-S service demand profile.

Service demand profile: the customer's need described as an effect (SWRL)

```
Shipment(?s)  ∧  shippingDuration(?s, ?days) ∧  ?days < 7  ∧  shippingPrice(?s, ?p)
∧  valueInUSD(?p, ?value)  ∧  ?value < 15.
```

Figure 9-H: Auxiliary shipment ontology



Crawling through different service profiles, a service registry would now find the *EconomyCongoBuy* service among others since the effect from its result is matching the customer's demand. More

[31] In general, a customer would interact with a Web Service only indirectly via a proxy. This proxy is left out in this use case for the reason of simplification

concretely, the *EconomyCongoBuy's* positive result states that the service sells books and ships them in less than four days for a moderate shipping price which is below $20 USD[32] if the book is in stock.

***EconomyCongoBuy* service profile: condition implies effect for the positive result (SWRL)**

```
hasISBN(?EconomyCongoBuyBook)  ∧  InStockBook(?EconomyCongoBuyBook)  ⇨

shippingDuration(?EconomyCongoBuyShipment, ?days)  ∧  ?days < 4  ∧
shippingPrice(?EconomyCongoBuyShipment, ?EconomyCongoBuyShipmentPrice)  ∧
ModerateShippingPrice(?EconomyCongoBuyShipmentPrice).
```

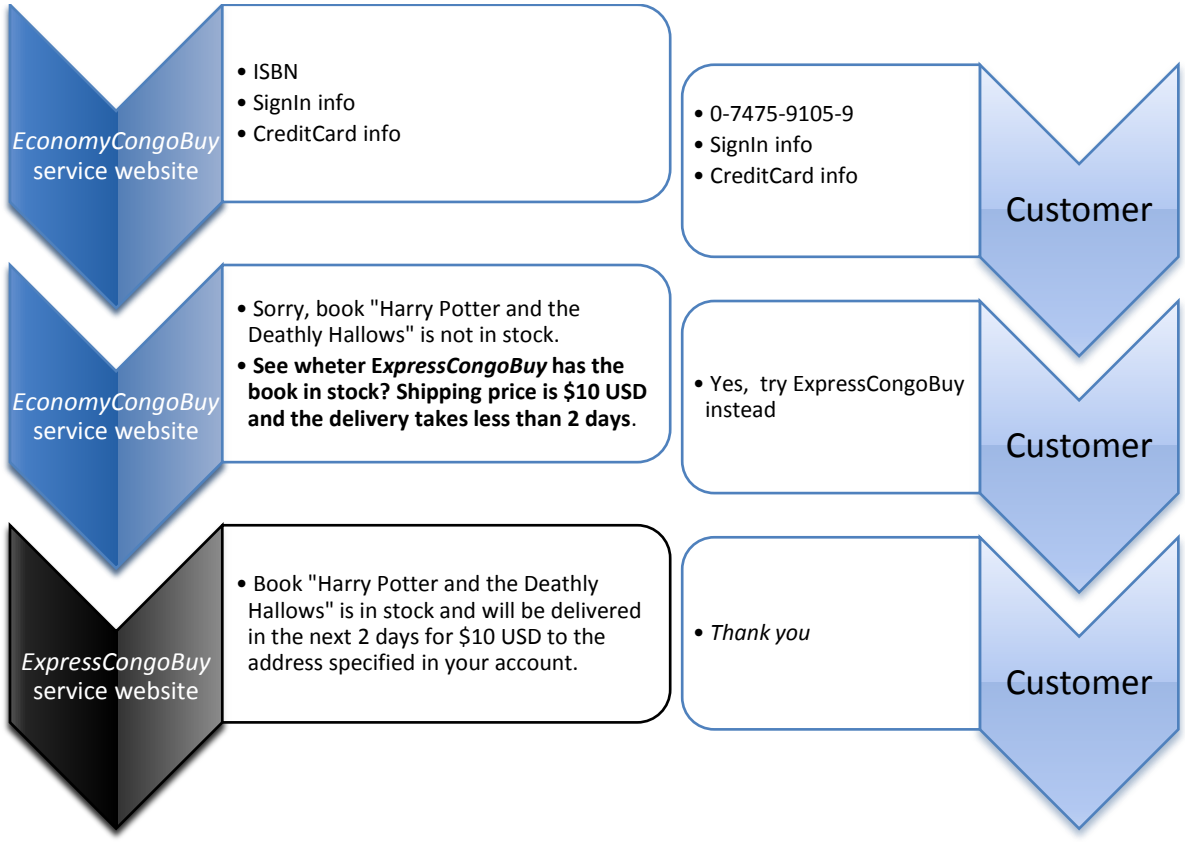
This economical shipping price itself then can be described in an additional auxiliary shipping ontology which can be part of the *EconomyCongoBuy* service ontology.

Auxiliary shipment ontology: the definition of a moderate shipping price (SWRL)

```
Shipment(?s)  ∧  shippingPrice(?s, ?p)  ∧  valueInUSD(?p, ?value)  ∧  ?value < 20
⇨  ModerateShippingPrice(?p).
```

Having the services listed that sell books, the customer sticks to his habit and picks the *EconomyCongoBuy* service out of the result list displayed by the registry. After this service is initiated, the following interaction between the customer and the service occurs in three steps. On the left side is the input request from the service website and on the right side is the response input given by the customer.

Figure 9-I: Interaction between the *EconomyCongoBuy* service and a customer



[32] The concrete shipping price itself is only available during a concrete service perform.

Step 1: The interaction begins with the regular input requirements from *EconomyCongoBuy*. In response, the customer specifies his book order by providing the specific input. Unfortunately, after the service looks up the book in the inventory, it finds out that this book is out of stock. Therefore, the service is not able to deliver.

Step 2: The service knows, however, that there exists an *ExpressCongoBuy* service which it can also try using the same input since those two services are connected with each other using a strict Inheritance Relationship (IR); this strict IR claims that *ExpressCongoBuy* has compatible inputs and outputs, weaker preconditions and stronger results compared to *EconomyCongoBuy*.

***ExpressCongoBuy* service profile: condition implies effect for the positive result (SWRL)**

```
hasISBN(?ExpressCongoBuyBook)  ∧  InStockBook(?ExpressCongoBuyBook)  ⇨

shippingDuration(?ExpressCongoBuyShipment, ?days)  ∧  ?days < 2  ∧
shippingPrice(?ExpressCongoBuyShipment, ?ExpressCongoBuyShipmentPrice)  ∧
ModerateShippingPrice(?ExpressCongoBuyShipmentPrice).
```

In this case, the inputs, outputs and preconditions from *EconomyCongoBuy* and *ExpressCongoBuy* are the same while the total result of *ExpressCongoBuy* is stronger than the result of *EconomyCongoBuy*. Therefore, whenever *EconomyCongoBuy* matches a certain request, *ExpressCongoBuy* can match the same request also.

The total result itself is composed of the logical conjunction of the positive and the negative result while each result itself is composed of the result's condition implying the result's effect. The expressions used in the short proof below are the following:

A(E) represents the value of the attribute A from the element E. MSP stands for a moderate shipping price, NR stands for the negative result, PRwDaP stands for the positive result without shipping duration and price, TRwDaP stands for the total result without shipping duration and price, price stands for the shipping price and days stands for the shipping duration.

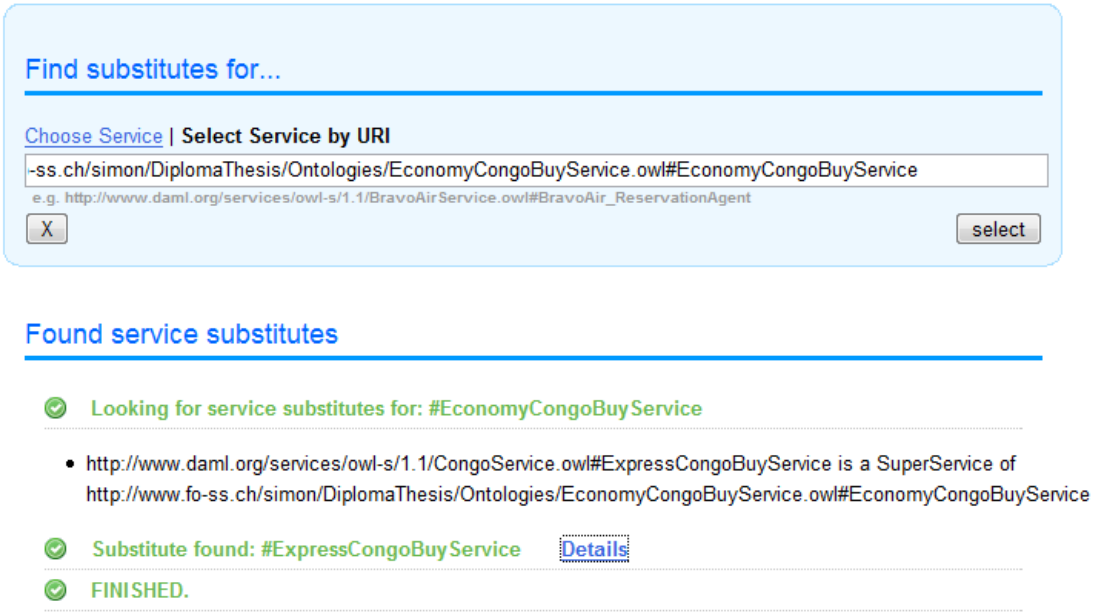
Short proof

```
(Preconditions(EconomyCongoBuy) ≡ Preconditions(ExpressCongoBuy))  ⇨
(Preconditions(EconomyCongoBuy) ⇨ Preconditions(ExpressCongoBuy)).

MSP == price < 20.

NR == Condition(NR) ⇨ Effect(NR).

PRwDaP == Condition(PRwDaP) ⇨ Effect(PRwDaP) ∧  MSP.

TotalResult(EconomyCongoBuy) ≡ (PRwDaP ∧  days < 4) ∧ NR ≡ TRwDaP ∧  days < 4.
TotalResult(ExpressCongoBuy) ≡ (PRwDaP ∧  days < 2) ∧ NR ≡ TRwDaP ∧  days < 2.

(TRwDaP ∧  days < 4 ⇨ TRwDaP ∧  days < 2)  ⇨
(Results(ExpressCongoBuy) ⇨ Results(EconomyCongoBuy)).
```

Prototype

Using the prototype, the service discovery of ExpressCongoBuy as a substitute for EconomyCongoBuy can be accomplished, as it is illustrated in <Figure 9-J: Discover ExpressCongoBuy service as a substitute for EconomyCongoBuy service>.

Figure 9-J: Discover ExpressCongoBuy service as a substitute for EconomyCongoBuy service



Step 3: Luckily, *ExpressCongoBuy* has the book in stock and therefore delivers it to the customer's address specified in his account within a shipping price of $10 in less than two days[33] thereby the customer's need can be satisfied smoothly with the alternative *ExpressCongoBuy*.

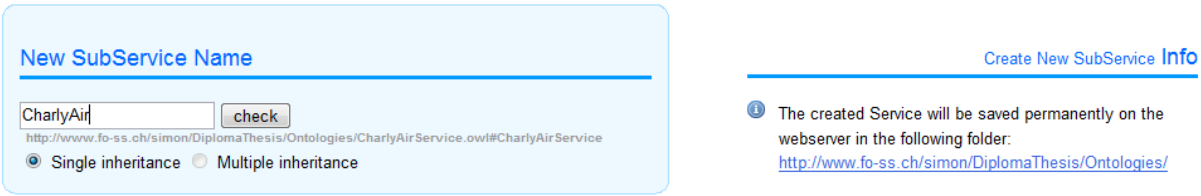
Comparison to current alternative

Currently, in case the EconomyCongoBuy Service would get executed as described in <Figure 9-I: Interaction between the *EconomyCongoBuy* service and a customer>, a potential registry could lookup all its entries and perform the same logical comparisons as they are used for this Inheritance Relationship (IR). This lookup however, may be very time consuming. Additionally, if the substitute service would not be listed in the registry, the registry would be unable to find it without interpreting the IRs.

9.2.3 Walkthrough: Create CharlyAir

This walkthrough illustrates, how the prototype can handle the use case in <Section 3.1.3: Create CharlyAir>. The new service *CharlyAir* can be created within the following steps.

Figure 9-K: Providing a name for the new CharlyAir service



Step 1: Providing a new service name

A new service name needs to be given in order to create a new and unique service identifier, see <Figure 9-K: Providing a name for the new CharlyAir service>.

[33] For simplicity reasons the price of the book itself is assumed to be the same in both services

Figure 9-L: Select BravoAir Reservation Agent as SuperService

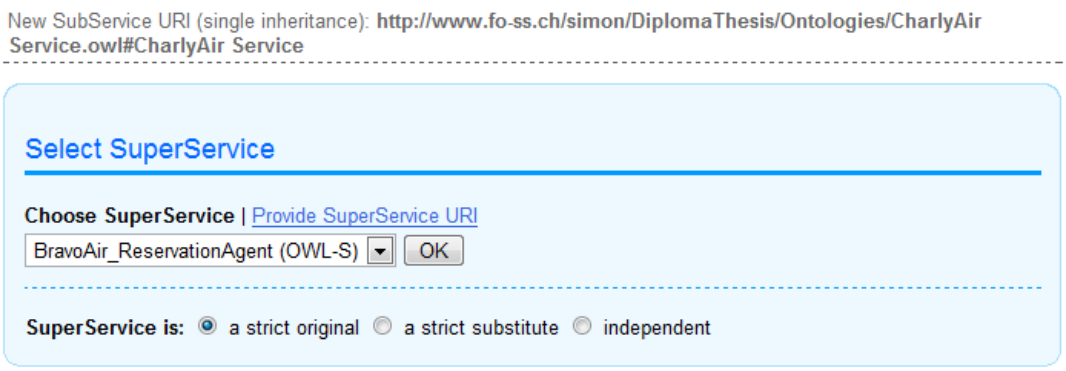
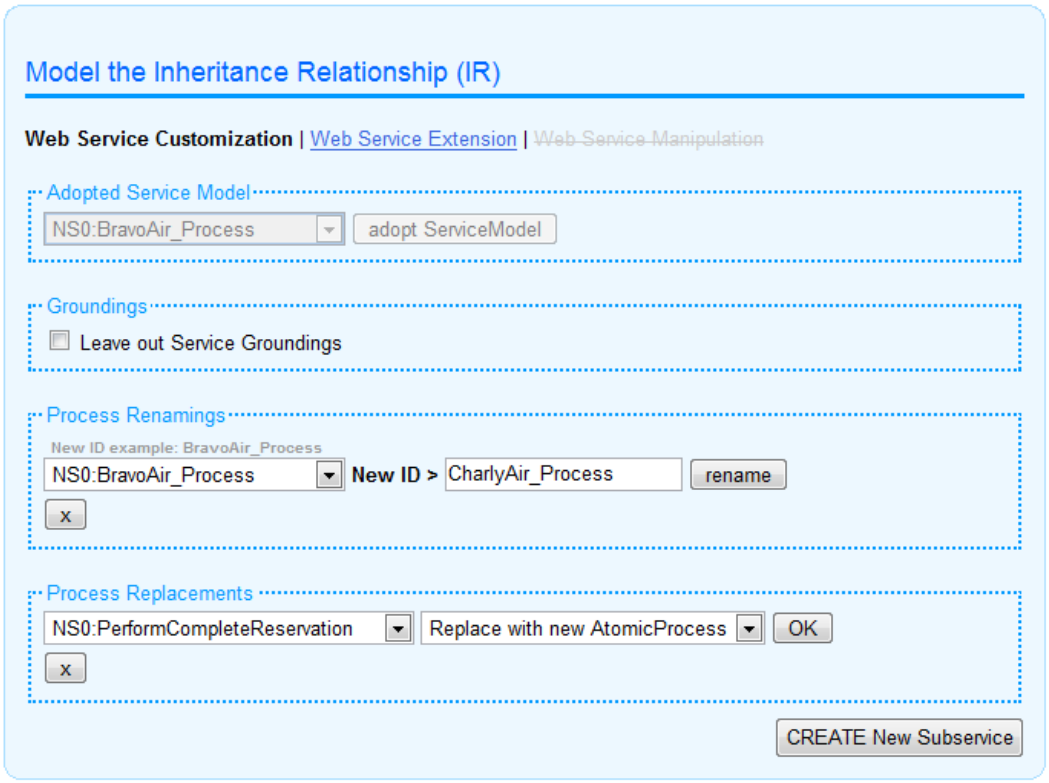


Figure 9-M: Adopting the service model from BravoAir Reservation Agent



Step 2: Selecting SuperService

The SuperService needs to be selected from which the new service wants to inherit. In this case, the SuperService is the *BravoAir Reservation Agent* service, see <Figure 9-L: Select BravoAir Reservation Agent as SuperService>.

Step 3: Adopting the service model

Because "strict original" has been selected in the previous step, the service model from the SuperService gets automatically adopted, see <Figure 9-M: Adopting the service model from BravoAir Reservation Agent>.

Step 4: Replacing process

Third, the process for completing the reservation needs to be replaced by a new one. After that, the new service can be created by clicking the finishing button, see <Figure 9-M: Adopting the service model from BravoAir Reservation Agent>.

Step 5: The new service

The service now gets saved on a web server and is ready to be interpreted, see <Figure 9-N: Visualizing CharlyAir service>. Since a process gets replaced by a new one, the groundings for the new process need to be created be created and added to the inherited groundings. Additionally, a new service profile should be added to the new service.

Now the new service consists of an instance of the OWL-S Service which is connected with a complete inheritance profile. This profile contains all the previous adjustments made. When the new service needs to be read as a regular OWL-S Service, the inheritance profile needs to be interpreted first.

<Figure 9-N: Visualizing CharlyAir service> and <Figure 9-O: Visualizing BravoAir service> Illustrate the original BravoAir service and the new created CharlyAir service.

Comparison to current alternative

Currently, the new service CharlyAir needs to be built anew, the service model could not be reused. The only way to profit from the existing BravoAir would be to point to already existing instances (processes, conditions, etc) while creating the new service model.

Figure 9-N: Visualizing CharlyAir service

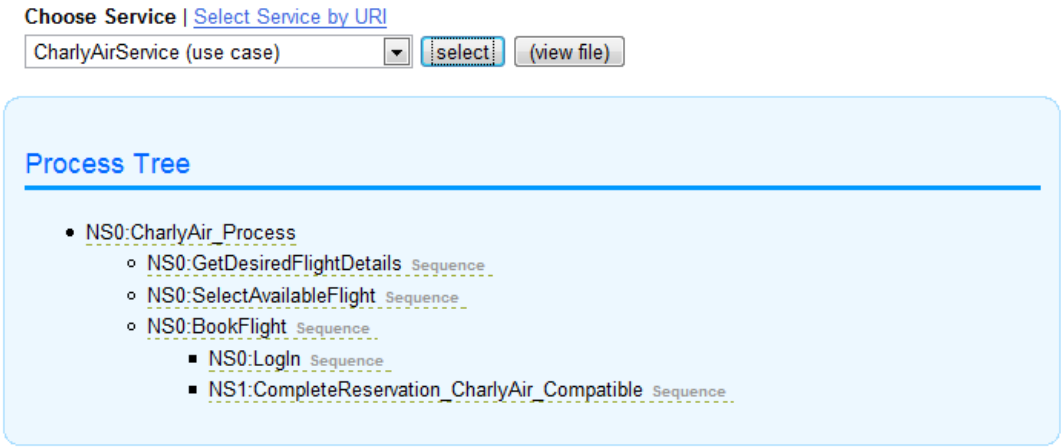
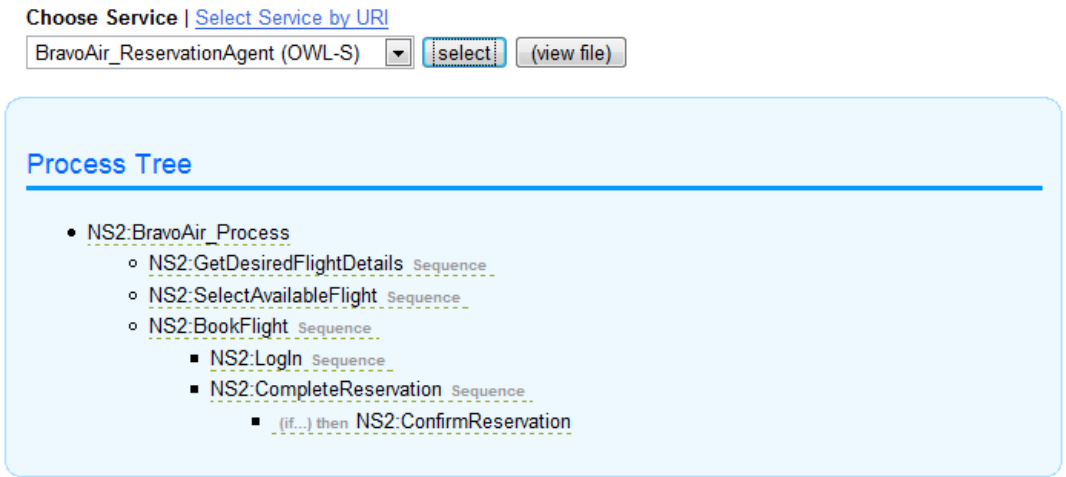


Figure 9-O: Visualizing BravoAir service



9.2.4 Walkthrough: Smooth choice increment with *CharlyAir*

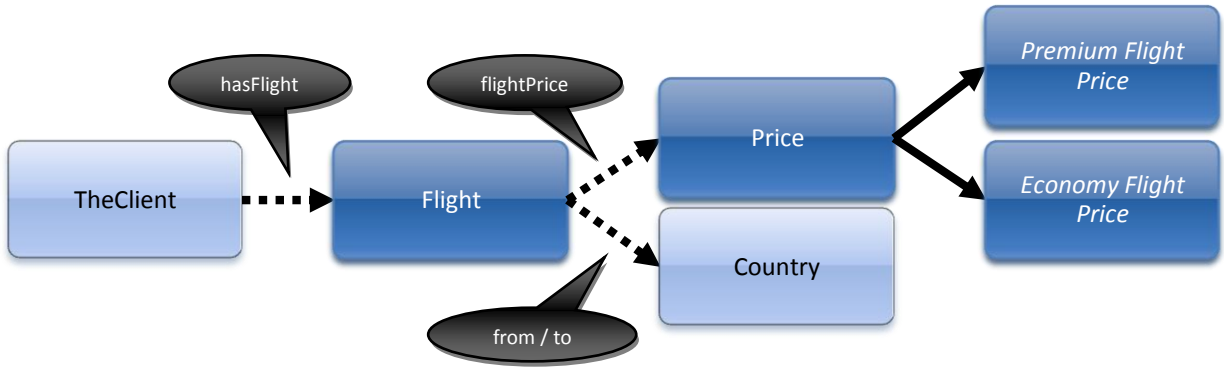
In order to illustrate the use case in <Section 3.1.4: Smooth choice increment with *CharlyAir*> in more detail, a concrete walkthrough is provided which covers the main interactions between the customer and the system that performs the two services *BravoAir* and *CharlyAir* in the light of the prototype.

The customer's demand can be formulated as a logical expression, stating that he wants a flight from Switzerland to Singapore for less than $2500 USD, using the ontology illustrated in <Figure 9-P: Auxiliary airline ontology>. Since the OWL-S profile can be used for both advertising and requesting a service, this expression could be stated as an OWL-S effect in a service demand profile.

Service demand profile: the customer's need described as an effect (SWRL)

```
Flight(?f)  ∧  from(?f, Switzerland)  ∧  to(?f, Singapore)  ∧  flightPrice(?f, ?p)
∧  valueInUSD(?p, ?value)  ∧  ?value < 2500.
```

Figure 9-P: Auxiliary airline ontology



Crawling through different service profiles, a service registry would now find the *BravoAir* service among others since the effect from its result is matching the customers demand. More concretely, the *BravoAir*'s effect[34] states that the service provides flights from Europe to Asia within an economical flight price which is below $2000 USD[35].

***BravoAir* service profile: empty condition implies effect for the service's result (SWRL)**

```
⇨

hasFlightItinerary(?TheClient, ?PreferredFlightItinerary)  ∧
hasFlight(?TheClient, ?BravoAirFlight)  ∧

from(BravoAirFlight, ?from)  ∧  EuropeanCountry(?from)  ∧
to(BravoAirFlight, ?to)  ∧  AsianCountry(?to)  ∧

flightPrice(?BravoAirFlight, ?p)  ∧  EconomyFlightPrice(?p).
```

This economical flight price itself then can be described in an additional auxiliary airline ontology which can be part of the *BravoAir* service ontology.

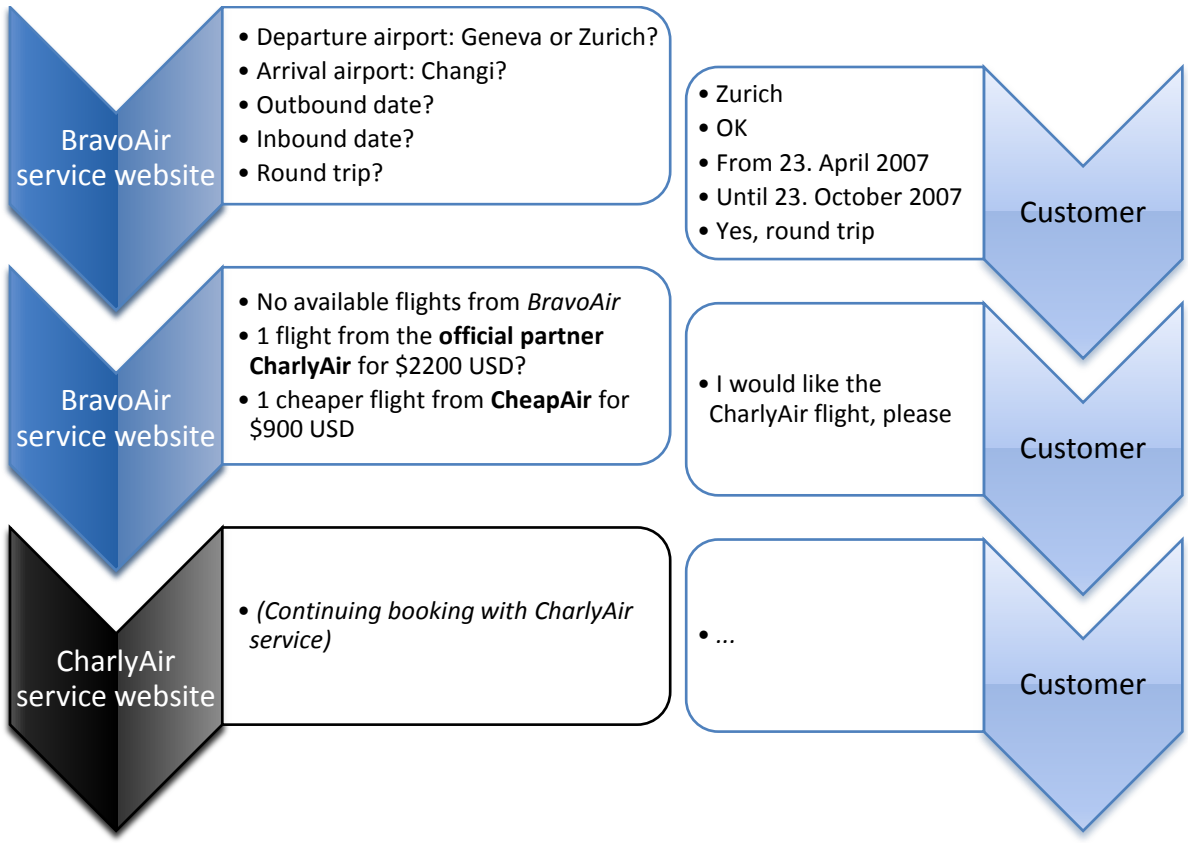
[34] The effects from the original OWL-S examples have been extended in order to support the use cases
[35] The concrete flight price itself is only available during a concrete service perform.

Auxiliary airline ontology: the definition of an economical flight price (SWRL)

```
Flight(?f) ∧ flightPrice(?f, ?p) ∧ valueInUSD(?p, ?value) ∧ ?value < 2000 ⇨
EconomyFlightPrice(?p).
```

Having the services listed that provide the specified flights, the customer sticks to his friend's recommendation and picks the *BravoAir* service out of the result list displayed by the registry. After this service is initiated, the following interaction between the customer and the service occurs in three steps. On the left side is the input request from the service website and on the right side is the response input given from the customer.

Figure 9-Q: Interaction between the *BravoAir* service and a customer



Step 1: The interaction begins with the regular input requirements from *BravoAir*. In response, the customer specifies his flight by providing the specific input. Unfortunately, after the service tries to match his input with concrete flights, there are no flights available from *BravoAir*.

Step 2: The service knows, however, that there exists a *CharlyAir* service which it can also try to get flight results by using the same input since those two services are connected with each other using a strict Inheritance Relationship (IR); this IR claims that *CharlyAir* has compatible inputs and outputs, weaker preconditions and stronger results compared to *BravoAir*.

CharlyAir service profile: empty condition implies effect for the service's result (SWRL)

```
⇨

hasFlightItinerary(process:TheClient, PreferredFlightItinerary)  ∧
hasFlight(process:TheClient, CharlyAirFlight)  ∧

from(BravoAirFlight, ?from)  ∧  EuropeanCountry(?from)  ∧
to(BravoAirFlight, ?to)  ∧  AsianCountry(?to)  ∧

flightPrice(CharlyAirFlight, ?p)  ∧
PremiumFlightPrice(?p)  ∧  EconomyFlightPrice(?p).
```

In this case, the inputs and outputs from *CharlyAir* and *BravoAir* are the same while they both have no precondition; the result of *CharlyAir* is stronger than the result of *BravoAir*. Therefore, whenever *BravoAir* matches a certain request, *CharlyAir* can match the same request also. The expressions used in the short proof below are the following:

A(E) represents the value of the attribute A from the element E. EBA stands for the effect of the result of BravoAir and ECA stands for the effect of the result of CharlyAir.

Short proof

```
(Preconditions(CharlyAir) ≡ Preconditions(BravoAir) ≡ {})  ⇨
(Preconditions(BravoAir) ⇨ Preconditions(BravoAir)).

EBA == Effect(BravoAir).

ECA == Effect(CharlyAir).

Result(BravoAir) ≡ ({} ⇨ EBA) ∧ ({}⇨ ECA) ≡ EBA ∧ ECA.

Result(CharlyAir) ≡ {} ⇨ ECA ≡ ECA.

(EBA ∧ ECA ⇨ ECA)  ⇨  (Results(CharlyAir) ⇨ Results(BravoAir)).
```

Luckily, *CharlyAir* has not only flights in the economy but also in the premium class while one of the latter matches the customer's request[36] with a price of $2200 USD.

Auxiliary airline ontology: the definition of a premium flight price (SWRL)

```
Flight(?f)  ∧  flightPrice(?f, ?p) ∧  valueInUSD(?p, ?value)  ∧  ?value > 2000  ⇨
PremiumFlightPrice(?p).
```

In the case of *CharlyAir*, this strict IR is bidirectional: *BravoAir* states that it has *CharlyAir* as an IR partner and the other way round. The important thing here is that both parties agree officially to their IR connection. Therefore, the system can suggest the flight results from *CharlyAir* while the service can be titled as an official partner of *BravoAir*.

In order to illustrate the difference between bidirectional and one-dimensional IR, a third service called *CheapAir* gets discovered which satisfies the requirements for a strict IR as well as *CharlyAir* does; with the exception, however, that this IR is stated only in the *CheapAir* but not in the *BravoAir*

[36] In general, the premium class would not match the customer's demand of a price below $2500 USD. But since *CharlyAir* does provides both classes, the service taken as a whole does match the customer's demand.

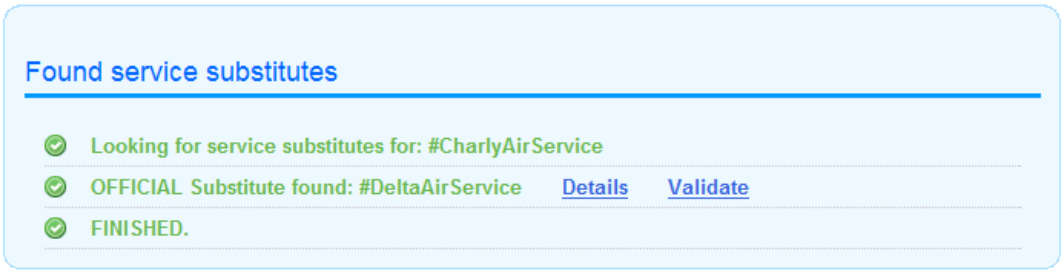
service. Therefore this IR is not officially confirmed from both parties. The interpretation of this unofficial versus official IR is yet up to the customer. However, it makes sense that the customer can expect a very similar experience from *CharlyAir* as he can from *BravoAir* because this is also in the interest of those two official partners since they agreed implicitly to exchange customers among each other whom they want to be satisfied.

Step 3: According to this implicit agreement between *BravoAir* and *CharlyAir*, the customer can change the airline while remaining within a very similar experience by selecting the flight from *CharlyAir* for $2200 USD. Furthermore, because those two services have a strict IR, their process models are compatible. This compatibility allows the system which performs the *BravoAir* service to perform the *CharlyAir* service with the same inputs given already which results in a smooth change.

Prototype

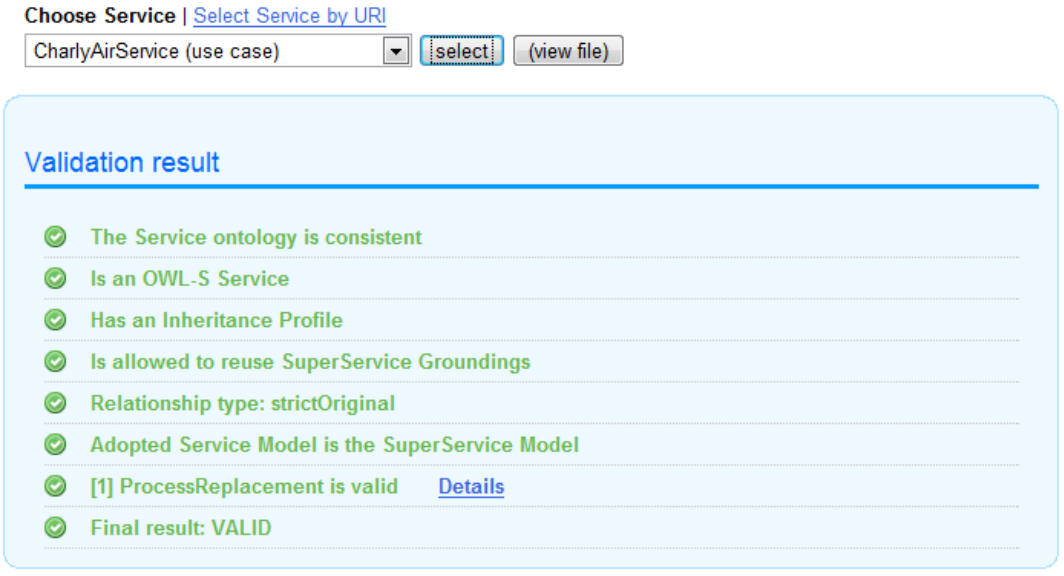
The prototype is able to discover and validate such substitutes: in this case, for the analogue services CharlyAir and DeltaAir. Since BravoAir cannot be altered in order to insert the inheritance relationship because it is an OWL-S example service, the prototype uses the two analogue services CharlyAir and DeltaAir instead, see <Figure 9-R: Discover DeltaAir service as a substitute for CharlyAir service> and <Figure 9-S: Validate CharlyAir service as a substitute for BravoAir service>.

Figure 9-R: Discover DeltaAir service as a substitute for CharlyAir service



Because the prototype can only model SuperService relationships currently, the SubService relationship from CharlyAir to DeltaAir in order to make the Inheritance Relationship (IR) official was modeled externally with an OWL editor.

Figure 9-S: Validate CharlyAir service as a substitute for BravoAir service



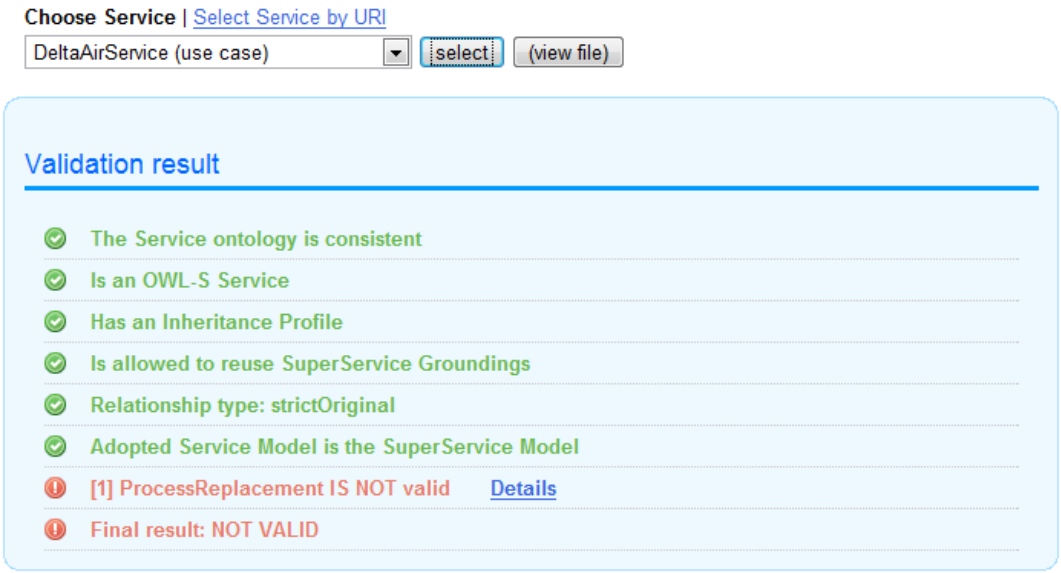
Additionally, in order to demonstrate that the prototype is also capable of detecting invalid IRs, an invalid process replacement has been implemented in the SubService DeltaAir, see <Figure 9-T: Validate DeltaAir service as a substitute for CharlyAir service>.

Comparison to current alternative

Currently, in case the BravoAir Service would get executed as described in <Figure 9-Q: Interaction between the *BravoAir* service and a customer>, a potential registry could lookup all its entries and perform the same logical comparisons as they are used for this Inheritance Relationship (IR). This lookup however, may be very time consuming. Additionally, if the substitute service would not be listed in the registry, the registry would be unable to find it without interpreting the IRs.

Furthermore, a registry would not be able to detect an official relationship among those two services, since there is yet no other way to declare it.

Figure 9-T: Validate DeltaAir service as a substitute for CharlyAir service



9.2.5 Walkthrough: FullCongoBuy suggests E-BookBuy

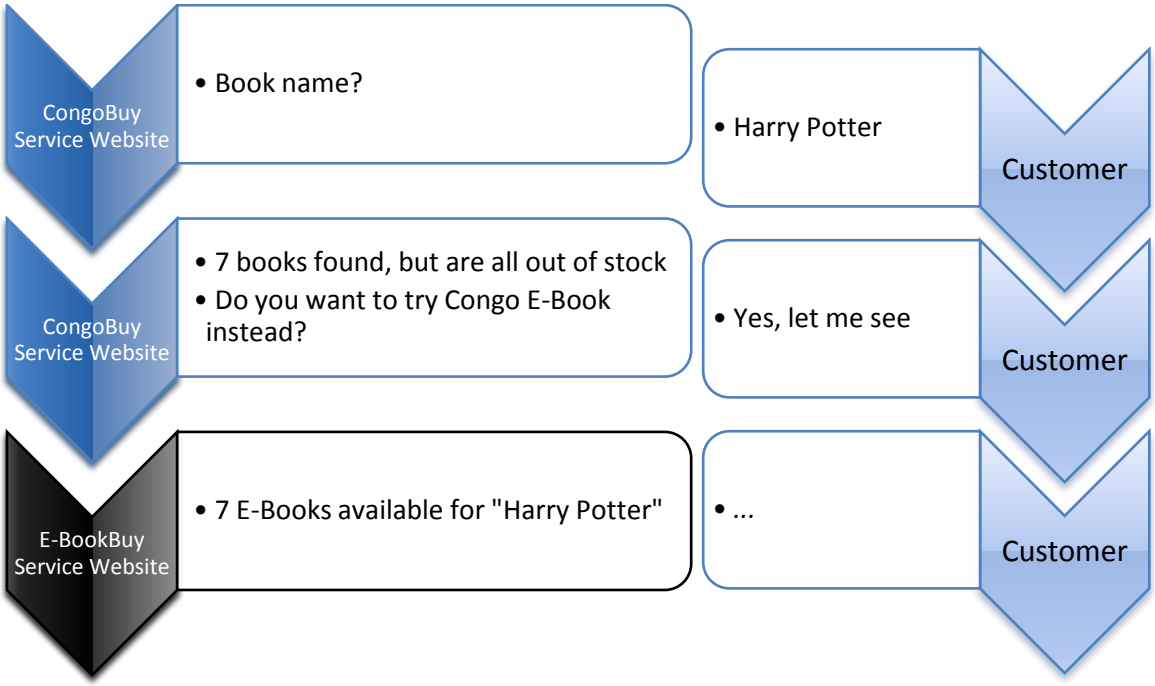
In order to illustrate the use case in <Section 3.2.2: FullCongoBuy suggests E-BookBuy> in more details, a concrete walkthrough is provided – see <Figure 9-U> – which covers the main interactions between the customer and the system that performs the two services *E-BookBuy* and *FullCongoBuy*.

Step 1: First, a customer looks his requested book up using the *CongoBuy* service.

Step 2: Second, the service tells the user that in general the books are available but currently out of stock. Since the *CongoBuy* service has a normal Inheritance Relationship (IR) with *E-BookBuy*, the system which is running the *CongoBuy* service can assume that *E-BookBuy* is similar to a certain degree with *CongoBuy*. Therefore, the system can suggest – by making a roughly guess – that the customer might also be interested in using the *E-BookBuy*.

Step 3: After the user agrees with trying out the suggested *E-BookBuy* service, the system switches to the other service and performs *E-BookBuy*. In this lucky case the needed input for *E-BookBuy* is also part of the needed input of *CongoBuy* and therefore the same input can just be reused by the system to search for electronic books of “Harry Potter” in *E-BookBuy*.

Figure 9-U: Interaction between the CongoBuy service and a customer



9.3 Comparison with other solution approaches

Since the idea to introduce inheritance for semantic web services is very new, there exists only one other approach which could be compared to the one proposed in this thesis: the use case of the SWSF which describes how SWSL-Rules can be used to implement inheritance for domain specific ontologies with default flavor.

The relationship to this approach is described in <Section 10.4: Related work>.

9.3.1 SWSF: Using Defaults in Domain-Specific Service Ontologies

Although the approach illustrated in the SWSF framework of introducing inheritance for semantic web services is based on a different framework, it covers almost the same subject matters as the approach proposed in this thesis for the OWL-S framework.

However, it seems that the focus of the SWSF approach is on service creation, while the focus of this thesis is both on service creation and service discovery, i.e. discover service substitutes. The SWSF use case does not explicitly discuss the latter.

Furthermore, it seems that the OWL-S framework is yet more likely to evolve than its competitors are, i.e. the SWSF.

Chapter summary

This chapter evaluates the solution of this thesis in a first part in the light of the prototype. Thereby, the prototype is explained in detail regarding to its architecture and specifications. This is done by discussing the use cases introduced earlier in more details. Additionally, the prototype helps to illustrate concrete benefits of Inheritance Relationships (IR) for OWL-S by guiding the uses cases step

by step. In order to stress out the benefits of IRs, the solution within using IRs gets compared with the current alternative without using IRs − for each use case.

In a second part, the solution of this thesis is briefly compared to the current most similar approach introduced in the SWSF in order to evaluate the additional value of this solution.

10 Discussion

This chapter summarizes the accomplishments of this thesis, points out specific findings of the proposed solution, presents my personal opinion on the domain of semantic web in general and provides an outlook by introducing ideas about future work.

10.1 Multiple-inheritance

Since a SubService can only adopt one service model, as described in <Section 5.5.3: Interpreting Web Service Customization>, one might ask himself what the reasons for multiple-inheritance are.

One reason is the additional information provided by stating the Inheritance Relationship (IR) itself, e.g. a strict IR. Since the benefit of this information seems to be bigger for strict IRs compared to normal IRs, multiple-inheritance according to this reason might be especially interesting for strict IRs.

Another reason is that the service model for a new service can be adopted from one SuperService, while processes from this service model can be replaced by altered processes inherited from another SuperService. Since process alterations are mostly only available for normal IRs – e.g. the deletion of a process component – multiple-inheritance might be especially interesting for normal IRs according to this reason.

10.2 Effects and side effects of the solution

10.2.1 Reinterpretation of the IR

It might be necessary in certain situations to reinterpret an Inheritance Relationship (IR) after the corresponding SuperService changed in such a way that the IR is no longer valid: as described in <Section 5.6: Validating the IR>.

Figure 10-A: Service Model change in a SuperService over time

Another reason to reinterpret an IR is to benefit from possible side effects of such a reinterpretation. In case a SuperService changes not its process groundings, but its process composition, such a change would not affect the corresponding SubService as long as the IR stays valid and does not get interpreted again.

For example, the SuperService could have a “Search Flight” process which changes from being atomic to being composite in order to make it more efficient. Since this change happens in the service model which gets copied to the SubService during the interpretation of the IR, the service model of the

SubService needs first to be updated by reinterpreting the IR in order to adopt this change, see <Figure 10-A: Service Model change in a SuperService over time>.

10.2.2 Meaningful top-down modeling approach

Currently, there are in general no abstract OWL-S web services meant to be, as it is not part of the philosophy of OWL-S. Therefore, there will not be any repository which helps finding appropriate (abstract) services from which one can inherit in order to create a new service.

Therefore, it might be meaningful to use the idea of inheritance in service creation currently in a top-down approach, where one models the SuperService first in order to profit from it when modeling the SubService afterwards instead of modeling a SubService first by looking for an appropriate (already existing) SuperService.

10.2.3 No abstract services in OWL-S?

As mentioned a few times in this thesis, a Web Service must be invocable on its operation level, according to the philosophy of OWL-S. Therefore, abstract services are not meant to be described using the OWL-S framework. Although, the basic idea of having abstract services – or in other words, service templates – is somehow compatible with OWL-S when looking at the idea of extending the OWL-S profile hierarchy as described in <Section 11.2.1: Future work>: instead of having abstract services, one could have service classes which have an impact on the service model.

10.2.4 Strict IR

Substitutes by construction

Since the strict Inheritance Relationship (IR) is based on the ideas of refinement[37], in every strict IR, one service is always an accurate substitute for the other one. Therefore, service substitutes follow automatically strict IR constructions.

Better change for substitute decidability by construction

Theoretically, service substitutes can not only be found via strict Inheritance Relationships (IR), but also via ordinary reasoning over a service registry, using the same formalism[38] as it is used for defining a strict IR. Without the relationship, however, this reasoning is likely to be expensive in time, since every service has to be considered as a potential substitute.

Furthermore, it might be the case that such reasoning over a whole service is not decidable. Using the strict IRs, however, a valid service substitute can be modeled by construction when following the corresponding contracts. The compliance testing with the contracts on the other hand only regards parts of the service and not the service as a whole[39]. Therefore the validation of a service substitute when modeling a strict IR is more likely to be decidable compared to the reasoning situation without the strict IR.

[37] See <Section 5.2.2: Strict IR>
[38] See <Section 5.2.2: Strict IR>
[39] As long as the SuperService does not change such that a new validation is required, as described in <Section 5.6: Validating the IR>

10.2.5 Others

ID values

In order to reference inserted or new created instances using the SWSL syntax, one must provide ID values for them. This is not always the case, however, when the OWL syntax is used since the instances can get referenced directly.

Update propagation

In the case, where the Inheritance Relationship (IR) is interpreted anew every time the belonging SubService gets performed, updates from its SuperService concerning the service model and service groundings could potentially be propagated down to the belonging Subservices.

Since in case of such an update, the semantics of a SubService would change automatically, it is not yet clear however to what extend such an update is desirable or not. This is also an issue proposed for future work, as discussed in <Section 11.2.1: Future work>.

Using plain OWL

Since the IR applications can be modeled using OWL, it is possible to model the whole inheritance profile (i.e. every information about the Inheritance Relationship) within OWL. This plain OWL representation has the advantage that potential reasoners only need to understand OWL and not an additional language which is used when modeling the IR alteration with the OWL-S Expression.

Faster growth of available Web Services

As explained in <Section 1.2: Motivation>, the solution of this thesis could improve the growth of available semantic web services because they can be created more easily and can also be constructed in advance for the concrete purpose of having service substitutes.

Broader use of Web Services

Having the possibility of finding service substitutes relatively easy – as described in <Section 5.5.7: Service substitutes> – can improve the availability of a specific web service. Therefore, web services could be broader used because of the better availability by using strict Inheritance Relationships.

10.3 Limitations of the solution

10.3.1 Requirement of IDs

For the SWSL-syntax, and in case the SuperService and its SubService are not modeled in the same ontology also for the OWL-syntax, the resources which are referenced in the Inheritance Relationship (IR) specification must have an RDF ID in order to be able to identify them, i.e. they cannot be anonymous. Such a resource is for example the process perform element of the SuperService in case of a process replacement. If the necessary IDs are not provided and cannot be added, it would not be possible to model the corresponding IR specification.

10.3.2 Changes in the SuperService

In order to maintain the validity of an Inheritance Relationship (IR), it is yet necessary to assume that the SuperService changes potentially permanently. This assumption is likely to be time expensive,

since it requires a validation checking every time before an IR related service gets accessed – as described in <Section 5.6: Validating the IR> – as a direct consequence.

It might be more efficient to be able to tell whether a specific service actually has changed over time or not. Currently, there is not yet an obvious solution available. A potential solution could follow the direction of the idea of the versioning metadata introduced in OWL.

10.4 Related work

10.4.1 Profile-based class hierarchies

Both frameworks (SWSF: Semantic Web Services Framework, 2005) and (Tools for inventing organizations: Toward a handbook of organizational processes, 1999) present a way to integrate services, respectively processes in a class hierarchy. This categorization can ease the organization of different services by their process structure.

OWL-S itself offers a similar approach, the so called profile-based class hierarchies[40]. This class hierarchy can be used to categorize the corresponding service by using the OWL-S profile hierarchy class instead of the regular OWL-S profile class. As an example, the ExpressCongoBuy book service[41] can be categorized in its service profile as a book selling service which delivers to the United States of America.

Profile hierarchy example from the ExpressCongoBuy service profile

```
<profileHierarchy:BookSelling rdf:ID="Profile_Congo_BookBuying_Service">

      <profileHierarchy:deliveryRegion rdf:resource="http://...#UnitedStates">

...
```

However, the profile-based class hierarchy has no influence of how a concrete service is actually modeled (i.e. which processes it is composed of). Therefore, in contrast to the two frameworks mentioned above, there is no organization possible among these categorized services according to their process structure, but only according to their general purpose for the end consumer.

10.4.2 SWSF: Using Defaults in Domain-Specific Service Ontologies

One use case of the SWSF describes how SWSL-Rules can be used to implement inheritance for domain specific ontologies with default flavor: inherited information can be overridden or left out.

More concretely, this use case illustrates how more specific services can be created by inheriting from a general one with the possibility to override or leave out some of its components, i.e. subtasks (respectively processes).

As an example, the Sell Product case of (Bernstein, Abraham ; Grosof, Benjamin, 2003) is illustrated in SWSL-Rules by using the Frame syntax. The service "Sell in retail store" can be created by inheriting from the general service "Sell Product", overriding some of its tasks and canceling one.

[40] In OWL-S 1.1, described at http://www.daml.org/services/owl-s/1.1/ProfileHierarchy.html
[41] Example service of the OWL-S 1.1 release

Example taken from (SWSF: Semantic Web Services Framework, 2005) – slightly modified

```
SellProduct[

      identifyCustomers *-> genericFindCust,
      informCustomers *-> genericInformCust,
      obtainOrder *-> genericGetOrder,
      deliverProduct *-> genericDeliver,
      receivePay *-> genericGetPay

  ].

  SellInRetail::SellProduct[

       identifyCustomers *-> attractToBrickAndMortar,
       obtainOrder *-> getOrderAtRegister,
       receivePay *-> getPayAtRegister,
      informCustomers *-> null

  ].
```

The services are modeled as a class, where SellInRetail is a subclass of SellProduct. Since the SellProduct class is modeled with inheritable attributes using the *-> arrow, the attributes from SellProduct get automatically inherited by SellInRetail, except the ones which get overridden, i.e. identifyCustomers, obtainOrder, receivePay and informCustomers.

This approach seems very reasonable for the SWSF, since in the service ontology SWSO which is one part of this framework, a service is also defined as a class (Service) with processes (from type Process) as its attributes. Therefore, the inheritance statements shown above can perfectly enrich a SWSO service ontology.

However, this approach presents just a basic way of using inheritance among services. How the dependencies among the inherited and overridden processes are taken into account is not explicitly defined yet.

Definition taken from (SWSF: Semantic Web Services Framework, 2005)

```
Service[process *=> Process].
```

Relationship to this thesis

Since OWL-S does not use the SWSL language, nor does it use the Frame syntax to describe services, this approach cannot be used for OWL-S as it is. More concretely, a service in OWL-S is represented as an OWL instance, and can therefore not be treated as a class which has the power to inherit its properties to its members or subclasses – as it is done in this approach.

This thesis therefore shows a different approach, based on an OWL instance level, in order to provide a specific inheritance mechanism for OWL-S. Furthermore, this thesis provides a solution how the (sequence) dependencies among the inherited processes can be handled.

10.4.3 Toward a handbook of organizational processes

The paper (Tools for inventing organizations: Toward a handbook of organizational processes, 1999) presents a way to describe and organize processes of organizations. For that purpose, it introduces

two dimensions to do so: notions of specializing processes and concepts of managing dependencies among processes.

The specialization of processes makes use of the concept of inheritance, as described in <Section 4.4: Inheritance for processes>. For the issue of managing dependencies, the paper introduces three kinds of them: fit, flow and sharing. Fit dependency between processes means, that these processes fit together in order to produce a single resource collectively. Flow dependency means, that one process produces a resource which is used by another process. Sharing dependency means, that multiple processes use the same resource.

Relationship to this thesis

Generally speaking, the idea of describing processes in this approach via their dependencies is very similar to the idea of OWL-S or semantic web service descriptions in general. To that end, the second dimension of specializing processes by using the concept of inheritance is very similar to the solution proposed in this thesis.

The main difference, however, is the level of concreteness of the processes (i.e. services).

The processes (i.e. services) in this framework are completely integrated in one big organizational process hierarchy in order to analyze and optimize them among different companies. For that purpose, the framework introduces the so called “process handbook”. Consequently, the processes in this handbook will get more and more abstract on a higher level, e.g. from a concrete process “Sell automotive components” to the abstract process “Sell something”.

OWL-S in contrast provides a description for already existing services, respectively for concrete services which will be created. Since the purpose of OWL-S is to connect a semantic service description with an executable software which can be accessed over the internet (via automated service discovery, invocation, monitoring and composition), an OWL-S service is not meant to be abstract. Therefore, such a hierarchy is not meant to exist among OWL-S services.

Chapter summary

OWL-S is an OWL-based framework that provides the semantics to describe a web service with a profile, process model and process grounding. The competitors of OWL-S are WSMO, SWSF and WSDL-S while the latter is based on the already established non-semantic WSDL framework. Closest to the approach presented in this thesis, however, are the SWSF and the handbook of organizational processes since they both introduce the notion of inheritance in their approach.

10.5 Personal opinion

10.5.1 Web applications could become more attractive

The semantic web – together with semantic web services – has the potential to make web applications more attractive for both suppliers and demanders. This could happen in two ways: on the side of the semantic web, the supply-demand matching process through the web is likely to become more human friendly; on the other side, semantic web services could provide more and better matching supplies.

Human friendly matching process

Because of the semantic web, web applications have now the potential to semantically understand what their users want, for the first time. Search engines and web catalogues have now the chance to match the demand of a user with their available supply not only just by keyword or categorization, but by their knowledge about what the user really wants and what they really have to offer – enabled by the semantic web.

In this matching process, the interaction between a human user and a web application can become more human friendly. As a metaphor, one can imagine to interact as a user with a (semantic) web application, i.e. website, as one would interact with a sales agent when calling the hotline in order to book a hotel, for example. A web site which is providing a semantic web application could behave to the user in a similar way and guide him like a sales agent would because this website is potentially able to *understand* what a specific user wants during the matching process and consequently also to provide a better matching supply (if available) compared to the situation today.

More and better matching supplies

Since a website has the potential to know very precise what a user is looking for, i.e. the user's demand can be described using concepts with underlying semantics (e.g. a user might be planning an excursion with secondary school students in the mountains for a certain budget in a specific time period where the area is possibly not occupied by many tourists), this website could create the content and the possibilities to interact with it very flexible, according to the specific request instead of just providing the default content and interface for every user.

Because a website could be able to understand its users by the means of the semantic description of the demand, web service discovery – one of the motivating tasks of OWL-S – would enable a just-in-time search and integration of additional web services. This website could provide these to the human end-user in order to not only try to sell its own goods but to try to satisfy the user's needs as good possible. This sharing of web services not only enables more and better matching demand with supply, but is also a great way of creating business partnerships across companies. In the end, the reusing of software applications is the main goal of the concept of Web Services, as described in (Matthew W. Guah; Wendy L. Currie, 2006). Therefore, web applications have a great chance to become more attractive not only for demanders, but also for suppliers.

Shifting transactions towards the internet

This human friendly interaction – together with the rich and matching supply enabled by semantic web services – is likely to shift transactions which are currently done by phone (what is in many cases still preferred to the pure online transaction), face-to-face or not at all, towards the internet, since the internet gets now a chance to serve humans better than ever.

10.5.2 Ontology visualization

As promising as the semantic web and semantic web services are, they still lack of good tool support. An example is that there are in my personal opinion yet no reasonable tools to visualize ontologies. Visualizing ontologies, however, seems to be critical in order to keep the overview over the knowledge gathered in such ontologies.

A good visualization would also enable an easier creation of new ontologies, and which is likely to accelerate the number of ontologies created which are needed in order to let the semantic web evolve.

10.5.3 Ontology reasoning

It seems that the tool support for ontology reasoning needs further improvements. The example services of the OWL-S framework can illustrate this need. Currently, the reasoner Pellet[42] is able to classify the taxonomy of the Congo service ontology[43], but fails (has errors) when classifying the taxonomy of the BravoAir service ontology[44]. On the other hand, FaCT++[45] is able to classify the taxonomy of the BravoAir service ontology, but fails (has errors) when trying to classify the taxonomy of the Congo service ontology.

[42] Pellet version 1.5.0, see http://pellet.owldl.com/ (Accessed on October 8, 2007)
[43] http://www.daml.org/services/owl-s/1.1/CongoService.owl (Accessed on October 8, 2007)
[44] http://www.daml.org/services/owl-s/1.1/BravoAirService.owl (Accessed on October 8, 2007)
[45] FaCT++ version 1.1.8, see http://owl.man.ac.uk/factplusplus/ (Accessed on October 8, 2007)

11 Conclusions

11.1 Accomplishments

The initial goal of this thesis was to extend OWL-S with the possibility to maintain Inheritance Relationships (IR) between OWL-S Web Services. This thesis shows that this is possible by providing a new and innovative solution to accomplish such IRs.

The concrete accomplishments of the thesis are the following:

The thesis has illustrated the advantages of using inheritance for OWL-S Web Services, in specific for service creation and discovery and provides a concrete way to achieve them. For service creation, the thesis provides Web Service Customization, Extension and Manipulation for sharing and modifying specific elements among these services while looking at such a service from its OWL-S description to its underlying operation (including dependencies within a service). This sharing is expected to substantially reduce the amount of work necessary for creating and maintaining services. For service discovery, the thesis provides a solution to find service substitutes for the developed strict IRs, based on the idea of refinement. These substitutes increase the choice of a service user or the availability of a specific service.

Furthermore, the thesis has illustrated that the level of detail needed to create a new service using the proposed IRs is quite low in case of Web Service Customization. The corresponding service (i.e. SuperService) can be chosen on the basis of its profile and modified by replacing only the processes that need to be done differently. Thereby, a new service (i.e. SubService) can be created without touching the others or any dependency among the processes. Thereby, one can create a new service from an existing one without the need of knowing the existing service on a detailed level because the semantics provided by the OWL-S description guides the selection and modification of the service.

Additionally, the thesis has illustrated that IRs link similar services are very strongly among each other, especially in case the IRs are bidirectional. Thereby, for service discovery, it is possible to find all possible substitutes in case IRs are widely used – the IRs themselves connect a potential large amount of services and thereby build a somehow strong network without the need of a central service registry.

Together with the developed prototype, the thesis demonstrates the basic feasibility of applying inheritance for OWL-S by illustrating several use cases.

Finally, the thesis provides a basis for further tool development because the developed solution covers not only the vocabulary needed in order to specify IRs, but also the interpretation of these IRs as well as a way to validate the IRs.

11.2 Outlook

Of course, since this is the first approach introducing inheritance to OWL-S services, the suggested IR applications are likely to be refined in future, as the approach gets used for further and more practical services. The services used in this thesis are not fully practical, since the services yet

available are mainly exemplary, e.g. the OWL-S service example or the OWL-S Service Retrieval Test Collection[46].

Nevertheless, the thesis demonstrates the potential of inheritance for web service creation and discovery. This potential is very likely to help semantic web services evolving – and in the end also semantic web in general. This will hopefully lead to an improved and more useful internet.

11.2.1 Future work

Transfer possible to other frameworks

Since there are competitors for the OWL-S framework[47], a future work would be to transfer this approach into these other frameworks.

Loosen contracts for normal IR

The contracts introduced in <Section 5.5: Interpreting the IR> are required in order to interpret an Inheritance Relationship (IR). These contracts are primary aligned with the strict IR. Therefore, in the case of the normal IR, some of these contracts could eventually be loosened.

For example, the contract for a process replacement requires that the IOPEs of the corresponding process stay completely compatible – independent of the context of the service model – in order to maintain the process flow. In the case of normal IR, however, the process flow can still be maintained when the outputs of the process are not compatible but also not further used by other processes as an input (this output would only be forwarded to the service model, i.e. main process).

Future work could discover the possibilities of the loosening of contracts when using normal IR.

OWL-S process hierarchy

Analogue to the OWL-S profile hierarchy[48], one could create also a hierarchy for OWL-S processes which could ease the search for process alternatives when making a process replacement[49] in a concrete OWL-S service. Thereby one could benefit from this process hierarchy as it is shown analogue for the "process handbook"[50].

Extended OWL-S profile hierarchy

The OWL-S profile hierarchy[51] could be extended such that the classification of a specific service has a direct influence on which processes (i.e. from the process hierarchy above) are recommended or allowed to use in the corresponding service model. This could help service creators to optimize or invent new services, as it is the main purpose of the "process handbook".

Reinterpretation of the IR

In <Section 10.2.1: Reinterpretation of the IR>, possible scenarios are discussed which could lead to a reinterpretation of an already interpreted Inheritance Relationship (IR). It is yet up to future work, however, to elaborate these scenarios in more detail, i.e. the possibilities of automation while taking the proper functioning of the service into account.

---

[46] See http://projects.semwebcentral.org/projects/owls-tc/ (Accessed on October 9, 2007)
[47] See <Section 1.1: Current situation>
[48] See <Section 10.4.1: Profile-based class hierarchies>
[49] Such a process replacement is described in <Section 5.1.1: Web Service Customization>.
[50] See (Tools for inventing organizations: Toward a handbook of organizational processes, 1999)
[51] See <Section 10.4.1: Profile-based class hierarchies>

Exploration of multiple-inheritance

In the current solution of this thesis, multiple-inheritance is yet very restricted. As described in the multiple-inheritance contract in <Section 5.5.3: Interpreting Web Service Customization>, it is not yet possible to adopt several service models. However, using normal Inheritance Relationships, it would be generally feasible to do so. Multiple service models could be combined into one. To what extend this approach would be possible and make sense is yet up to be explored in future work.

More use cases for Web Service discovery

It would be interesting to find out what other use cases for Web Service discovery the Inheritance Relationships (IR) can offer, together with the ones already introduced in <Section 3: Use cases>. The IR information could for example be used, to produce different clusters of the connected services.

Protégé plug-in

In order to explore the possibilities a tool can provide for inheritance in OWL-S, the prototype is developed as standalone software. Protégé is quite popular in the OWL community and provides already a plug-in (i.e. tab)[52] for modeling OWL-S services. Therefore, it might help improving the popularity of the approach of this thesis to build a new Protégé plug-in for OWL-S Inheritance Relationships. This plug-in could directly communicate with the already existing OWL-S plug-in and would therefore be better accessible for future web service developers.

Chapter summary

This chapter discusses the current meaning and possible future work for specific issues. Key questions are: to what extend multiple-inheritance can be useful for OWL-S services; when a reinterpretation of an Inheritance Relationship (IR) is meaningful and what the consequences are of such a reinterpretation.

Additionally, I present my personal opinion related to the domain of semantic web in general in order to position the improvements for OWL-S (provided by this thesis) in the big picture of semantic web and semantic web services.

[52] See http://protege.stanford.edu/conference/2005/submissions/abstracts/accepted-abstract-elenius.pdf (Accessed on October 9, 2007)

12 Acknowledgements

I am grateful to Dr. Abraham Bernstein (Professor at the University of Zurich) for allowing this thesis to be written at the National University of Singapore. I am also grateful to Dr. Jin Song DONG (Associate Professor at the National University of Singapore) for welcoming me at the National University of Singapore and supporting me in writing this thesis.

Furthermore, I would like to thank Dr. Yuan Fang LI very much for supporting me with his advice during the whole development of this thesis.

Additionally, I would like to thank Zheng LU, Ingo Oppermann and Yuzhang FENG for their advice on programming issues. Finally, I would like to thank Olivier Lambercy for reviewing this thesis.

13 References

Laurel J. Brinton et al. (2001). Historical linguistics 1999 : selected papers from the 14th International Conference on Historical Linguistics, Vancouver, 9-13 August 1999. Amsterdam ; Philadelphia: John Benjamins Publishing Company.

Akkiraju, Rama; Farrell, Joel; Miller, John; Nagarajan, Meenakshi; Schmidt, Marc-Thomas; Sheth, Amit; Verma, Kunal. (2005, November 7). *Web Service Semantics: WSDL-S*. Retrieved July 18, 2007, from W3C Member Submission 7 November 2005: http://www.w3.org/Submission/WSDL-S/

Battle, Steve; Bernstein, Abraham; Boley, Harold; Grosof, Benjamin; Gruninger, Michael; Hull, Richard; Kifer, Michael; Martin, David; McIlraith, Sheila; McGuinness , Deborah; Su, Jianwen; Tabet, Said. (2005, September 9). *SWSF: Semantic Web Services Framework*. Retrieved June 4, 2007, from W3C Member Submission 9 September 2005: http://www.w3.org/Submission/SWSF/

Battle, Steve; Bernstein, Abraham; Boley, Harold; Grosof, Benjamin; Gruninger, Michael; Hull, Richard; Kifer, Michael; Martin, David; McIlraith, Sheila; McGuinness , Deborah; Su, Jianwen; Tabet, Said. (2005, September 9). *SWSL: Semantic Web Services Language*. Retrieved June 5, 2007, from W3C Member Submission 9 September 2005: http://www.w3.org/Submission/SWSF-SWSL/

Bernstein, A. (2005). *So what is a (Diploma) Thesis? A few thoughts for first-timers.* Zurich, Switzerland: University of Zurich.

Bernstein, Abraham ; Grosof, Benjamin. (2003). *Beyond Monotonic Inheritance: Towards Semantic Web Process Ontologies.* University of Zurich, Department of Informatics. Zurich, Switzerland: University of Zurich.

Christensen, Erik; Curbera, Francisco; Meredith, Greg; Weerawarana, Sanjiva. (2001, March 15). *Web Services Description Language (WSDL) 1.1*. Retrieved July 18, 2007, from W3C Note 15 March 2001: http://www.w3.org/TR/wsdl

De Bruijn, Jos; Bussler, Christoph; Domingue, John; Fensel, Dieter; Hepp, Martin; Keller, Uwe; Kifer, Michael; König-Ries, Birgitta; Kopecky, Jacek; Lara, Rubén; Lausen, Holger; Oren, Eyal; Polleres, Axel; Roman, Dumitru; Scicluna , James; Stollberg , Mic. (2005, June 3). *Web Service Modeling Ontology (WSMO)*. Retrieved July 18, 2007, from W3C Member Submission 3 June 2005: http://www.w3.org/Submission/WSMO

De Roever, W.-P., & Engelhardt, K. (1999). *Data Refinement, Model-Oriented Proof Methods and their Comparison.* New York: Cambridge University Press.

DL Implementation Group. (2006, March 10). *DL Implementation Group (DIG)*. Retrieved September 21, 2007, from The new DIG interface standard (DIG 2.0): http://dl.kr.org/dig/

Genesereth, M. R. (1999). *KIF: Knowledge Interchange Format (Draft)*. Retrieved June 5, 2007, from draft proposed American National Standard (dpANS): http://logic.stanford.edu/kif/

Gudgin, Martin; Hadley, Marc; Mendelsohn, Noah; Moreau, Jean-Jacques; Nielsen, Henrik Frystyk; Karmarkar, Anish; Lafon, Yves. (2007, April 27). *SOAP Version 1.2*. Retrieved July 19, 2007, from W3C Recommendation 27 April 2007: http://www.w3.org/TR/soap12-part1/

Harold, E. R. (1997). Polymorphism and Inheritance. In E. R. Harold, *Java developer's resource : a tutorial and on-line supplement* (pp. Chapter 7, 169-189). Upper Saddle River, N.J.: Prentice Hall.

Horrocks, Ian; Patel-Schneider, Peter F.; Boley, Harold; Tabet, Said; Grosof, Benjamin; Dean, Mike. (2004, May 21). *SWRL: A Semantic Web Rule Language*. Retrieved June 5, 2007, from W3C Member Submission 21 May 2004: http://www.w3.org/Submission/SWRL/

Malone, T. W.; Crowston, K. G.; Lee, J.; Pentland, B.; Dellarocas, C.; Wyner, G.; Quimby, J.; Osborn, C. S.; Bernstein, A.; Herman, G.; Klein, M.; O'Donnell, E. (1999, March). Tools for inventing organizations: Toward a handbook of organizational processes. *Management Science* , 425-443.

Martin, David; Ankolekar, Anupriya; Burstein, Mark; Denker, Grit; Elenius, Daniel; Hobbs, Jerry; Kagal, Lalana; Lassila, Ora; McDermott, Drew; McGuinness, Deborah; McIlraith, Sheila; Paolucci, Massimo; Parsia, Bijan; Payne, Terry; Sabou, Marta; et al. (n.d.). *OWL-S 1.1 Release: Examples*. Retrieved July 2003, 2007, from The DARPA Agent Markup Language Homepage: http://www.daml.org/services/owl-s/1.1/examples.html

Martin, David; Burstein, Mark ; Hobbs, Jerry ; Lassila, Ora ; McDermott, Drew ; McIlraith, Sheila ; Narayanan, Srini ; Paolucci, Massimo ; Parsia, Bijan ; Payne, Terry ; Sirin, Evren ; Srinivasan, Naveen ; Sycara, Katia ;. (2004, November 22). *OWL-S: Semantic Markup for Web Services*. Retrieved June 1, 2007, from OWL-S: Semantic Markup for Web Services: http://www.w3.org/Submission/OWL-S/

Matthew W. Guah; Wendy L. Currie. (2006). Web services. In M. W. Guah, & W. L. Currie, *Internet strategy: the road to web services solutions* (pp. Chapter II, 8-17). Hershey, PA: IRM Press.

McDermott, D. (2004, January 12). *DRS: A Set of Conventions for Representing.* Retrieved June 5, 2007, from The DARPA Agent Markup Language Homepage: http://www.daml.org/services/owl-s/1.0/DRSguide.pdf

MSID. (2007, February 15). *Process Specification Language (PSL)*. Retrieved July 18, 2007, from Manufacturing Systems Integration Division: http://www.mel.nist.gov/psl/

OASIS Open. (2006). *UDDI*. Retrieved July 19, 2007, from Advancing Web Services Discovery Standard: http://www.uddi.org/

Turi, D. (2004, April 21). *DIG Interface*. Retrieved September 21, 2007, from DIG Interface: http://dig.sourceforge.net/

W3C: World Wide Web Consortium. (2007, April 24). *Web Ontology Language*. Retrieved July 19, 2007, from W3C: http://www.w3.org/2004/OWL/

Welty, C., & Patel-Schneider, P. (1993). *The Principles of Knowledge Representation and Reasoning*. Retrieved September 2007, 26, from The Principles of Knowledge Representation and Reasoning: http://www.kr.org

Wikipedia. (2007, June 19). *Inheritance (computer science)*. Retrieved July 12, 2007, from Wikipedia, the free encyclopedia: http://en.wikipedia.org/wiki/Inheritance_(computer_science)

14 Glossary

A

- **Applications of Inheritance Relationships (IR):** There are three proposed applications of IRs: namely *Web Service Customization*, *Extension* and *Manipulation* which can be used complementary.

D

- **Default inheritance.** Default inheritance means that every element that gets inherited by default can get modified, i.e. overridden or removed.

I

- **Inheritance Relationship (IR).** An Inheritance Relationship (IR) is a relationship between two OWL-S Web Services within which one service can generally inherit the OWL-S service model and the complete service grounding from another service.

S

- **SubService.** A SubService is a Web Service which inherits from another Web Service (SuperService) by using the proposed Inheritance Relationship (IR).
- **SuperService.** A SuperService is a Web Service which gets inherited from by another Web Service (SubService) by using the proposed Inheritance Relationship (IR).

U

- **URI.** A URI is a Uniform Resource Identifier which allows one to identify a resource across the internet.

W

- **Web Service.** A Web Service is a software system that supports the machine-to-machine interaction over a network (Web services, 2006). In this document this term is equivalent to the term *service*.

15 Attachments

15.1 Online version

- **Inheritance profile:** The ontology source code of the proposed inheritance profile extension for OWL-S is online at:

http://www.fo-ss.ch/simon/DiplomaThesis/InheritanceProfile/InheritanceProfile.owl

- **Prototype:** The prototype is online at:

http://www.fo-ss.ch/simon/DiplomaThesis/IR_prototype/

- **Use cases:** The service ontologies of the use cases created with the prototype are online at:

http://www.fo-ss.ch/simon/DiplomaThesis/Ontologies/

15.2 Abstract

The abstract is available as an additional document both in English and German.

15.3 CD-ROM

The CD-ROM contains all documents created within this thesis:

- The thesis
- Abstract of the thesis
- Abstract of the thesis in German
- Inheritance profile
- Source code of the prototype
- Ontologies of the use case services (created within the prototype)