

University of Crete
Computer Science Department

*The Frame Problem in Web Service
Specifications*

George Baryannis

Master's Thesis

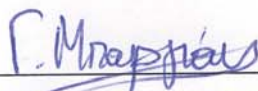
Heraklion, July 2009

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΡΗΤΗΣ
ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΕΠΙΣΤΗΜΗΣ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟ ΠΡΟΒΛΗΜΑ ΠΛΑΙΣΙΟΥ ΣΤΙΣ ΠΡΟΔΙΑΓΡΑΦΕΣ
ΗΛΕΚΤΡΟΝΙΚΩΝ ΥΠΗΡΕΣΙΩΝ

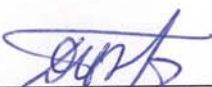
Εργασία που υποβλήθηκε από τον
Μπαργιάννη Γεώργιο
ως μερική εκπλήρωση των απαιτήσεων για την απόκτηση
ΜΕΤΑΠΤΥΧΙΑΚΟΥ ΔΙΠΛΩΜΑΤΟΣ ΕΙΔΙΚΕΥΣΗΣ

Συγγραφέας:

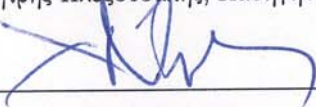


Γεώργιος Μπαργιάννης, Τμήμα Επιστήμης Υπολογιστών

Εισηγητική Επιτροπή:



Δημήτρης Πλεξουσάκης, Καθηγητής, Επόπτης



Χρήστος Νικολάου, Καθηγητής, Μέλος



Γιάννης Τζίτζικας, Επίκ. Καθηγητής, Μέλος

Δεκτή:



Πάνος Τραχανιάς, Καθηγητής,
Πρόεδρος Επιτροπής Μεταπτυχιακών Σπουδών

Ηράκλειο, Ιούλιος 2009

The Frame Problem in Web Service Specifications

George Baryannis

Master's Thesis

Computer Science Department, University of Crete

Abstract

Service-Oriented Architecture has emerged in recent years as a prominent design style that enables an IT infrastructure to allow different applications to exchange data and participate in business processes, regardless of the underlying complexity of the applications, such as the exact implementation or the operating systems or the programming languages used to develop them. SOAs can be implemented using Web Services, software systems that are designed to support interoperable machine-to-machine interaction over a network. In order to effectively find and invoke a Web Service, its provider must provide a complete specification for it. Devising such complete Web Service specifications comes with many issues that need to be solved.

This thesis explores the frame problem and its effects in devising Web Service specifications. The frame problem encompasses the issues raised when trying to concisely state in a specification that nothing changes except when explicitly mentioned otherwise. The argument that Web Services are in fact affected by the frame problem is supported by a multi-faceted motivating example that covers both atomic and composite service specifications.

A solution approach for the frame problem in Web Service specifications is proposed, based on knowledge gained from related research in procedure specifications and an algorithm for the automatic application of the solution to preexisting Web Service specifications. The solution schema and the algorithm are adapted and integrated in the OWL-S Semantic Web service framework. Moreover, an implementation of the algorithm is offered, which takes existing OWL-S Web Service specifications as

input and modifies them accordingly, so that they are ridden of all issues related to the frame problem.

Supervisor: Dimitris Plexousakis

Professor

Computer Science Department

University of Crete

Το Πρόβλημα Πλαισίου στις Προδιαγραφές Ηλεκτρονικών Υπηρεσιών

Γεώργιος Μπαργιάννης

Μεταπτυχιακή Εργασία

Τμήμα Επιστήμης Υπολογιστών, Πανεπιστήμιο Κρήτης

Περίληψη

Τα τελευταία χρόνια, η Υπηρεσιοστρεφής Αρχιτεκτονική έχει αναδειχθεί σε εξέχοντα τρόπο σχεδίασης που καθιστά τις υποδομές Πληροφοριακής Τεχνολογίας ικανές να παρέχουν σε εφαρμογές τη δυνατότητα ανταλλαγής δεδομένων και συμμετοχής σε επιχειρησιακές διεργασίες, ανεξάρτητα από θέματα που σχετίζονται με την υποκείμενη πολυπλοκότητα των εφαρμογών, όπως την επακριβή υλοποίηση ή τα λειτουργικά συστήματα και τις γλώσσες προγραμματισμού που χρησιμοποιήθηκαν για την ανάπτυξή τους. Η Υπηρεσιοστρεφής Αρχιτεκτονική μπορεί να υλοποιηθεί χρησιμοποιώντας Ηλεκτρονικές Υπηρεσίες, συστήματα λογισμικού που είναι σχεδιασμένα για να παρέχουν διαλειτουργικότητα και διαδραστικότητα μεταξύ μηχανών διαμέσου ενός δικτύου. Για την αποτελεσματική εύρεση και κλήση μιας Ηλεκτρονικής Υπηρεσίας, ο παροχέας της πρέπει να παρέχει ένα πλήρες σύνολο προδιαγραφών γι'αυτή. Η σχεδίαση τέτοιων πλήρων προδιαγραφών Ηλεκτρονικών Υπηρεσιών συνοδεύεται από πολλά ζητήματα τα οποία απαιτούν επίλυση.

Η παρούσα εργασία εξερευνά το πρόβλημα πλαισίου και τις επιπτώσεις του στη σχεδίαση προδιαγραφών Ηλεκτρονικών Υπηρεσιών. Το πρόβλημα πλαισίου περιλαμβάνει όλα τα ζητήματα που προκύπτουν όταν γίνεται προσπάθεια σε προδιαγραφές να δηλωθεί με συνοπτικό τρόπο ότι τίποτα δεν αλλάζει εκτός εάν δηλώνεται ρητά το αντίθετο. Το επιχείρημα ότι οι Ηλεκτρονικές Υπηρεσίες επηρεάζονται στην πραγματικότητα από το πρόβλημα πλαισίου υποστηρίζεται σ'αυτή την εργασία από ένα πολύπλευρο παράδειγμα που καλύπτει προδιαγραφές τόσο απλών όσο και σύνθετων Ηλεκτρονικών Υπηρεσιών.

Προτείνεται μια μέθοδος επίλυσης του προβλήματος πλαισίου στις προδιαγραφές Ηλεκτρονικών Υπηρεσιών η οποία βασίζεται σε σχετική έρευνα στον τομέα των προδιαγραφών διαδικασιών, καθώς και ένας αλγόριθμος για την αυτόματη εφαρμογή της μεθόδου επίλυσης σε προϋπάρχουσες προδιαγραφές Ηλεκτρονικών Υπηρεσιών. Η μέθοδος επίλυσης και ο αλγόριθμος προσαρμόζονται και ενσωματώνονται στο πλαίσιο περιγραφής Σημασιολογικών Ηλεκτρονικών Υπηρεσιών OWL-S. Παρέχεται επίσης μια υλοποίηση του αλγορίθμου, η οποία παίρνει υπάρχουσες προδιαγραφές Ηλεκτρονικών Υπηρεσιών σε OWL-S ως είσοδο και τις τροποποιεί ανάλογα, ώστε να απαλλαγούν από όλα τα ζητήματα που σχετίζονται με το πρόβλημα πλαισίου.

Επόπτης Καθηγητής: Δημήτρης Πλεξουσάκης

Καθηγητής

Τμήμα Επιστήμης Υπολογιστών

Πανεπιστήμιο Κρήτης

*Στους γονείς μου,
Βασίλη και Μαριάννα*

Ευχαριστίες

Θα ήθελα να ευχαριστήσω θερμά τον επόπτη μου, Καθηγητή κ. Δημήτρη Πλευξουσάκη, για την εμπιστοσύνη που μου έδειξε στο ξεκίνημα των μεταπτυχιακών μου σπουδών και την πολύτιμη βοήθεια και καθοδήγησή του καθ'όλη τη διάρκεια τους.

Επίσης, ευχαριστώ πολύ τον Καθηγητή κ. Χρήστο Νικολάου, μέλος της εξεταστικής επιτροπής της μεταπτυχιακής μου εργασίας, τόσο για το χρόνο που αφιέρωσε όσο και για τη συνεργασία που είχαμε στα πλαίσια των μεταπτυχιακών μου σπουδών.

Ακόμα, θα ήθελα να ευχαριστήσω τον Επίκ. Καθηγητή κ. Γιάννη Τζιτζικα, μέλος της εξεταστικής επιτροπής της μεταπτυχιακής μου εργασίας, για τις πολύτιμες παρατηρήσεις και συμβουλές του.

Ένα μεγάλο ευχαριστώ αξίζει σε όλους τους φίλους μου, που απέκτησα ερχόμενος στην πόλη του Ηρακλείου, για την αμέριστη συμπαράσταση και τις πολλές ευχάριστες στιγμές που περάσαμε αυτά τα χρόνια.

Επιπλέον ευχαριστώ πάρα πολύ και τον αδελφικό μου φίλο, Γιώργο Μαδεμλή, για τη συνεχή και ανιδιοτελή υποστήριξη και συμπαράσταση που μου παρέχει όλα αυτά τα χρόνια.

Το μεγαλύτερο ευχαριστώ όμως ανήκει δικαιοματικά στους γονείς μου, Βασίλη και Μαριάννα και στον αδελφό μου Αλέξανδρο για την αδιάκοπη και πολύτιμη στήριξη και βοήθεια τους ώστε να ολοκληρώσω με επιτυχία τη μεταπτυχιακή μου εργασία. Σας ευχαριστώ πολύ!

Contents

List of Tables	xvii
List of Figures	xviii
1 Introduction	1
1.1 Motivation	1
1.2 Contributions	3
1.3 Outline	5
2 Background Theory: From Services to the Frame Problem	7
2.1 Service-Oriented Architecture	7
2.1.1 Principles of Service-Oriented Architecture	8
2.2 Web Services	10
2.2.1 Web Services Technology Stack	13
2.3 The Semantic Web	16
2.3.1 Knowledge Representation	17
2.3.2 Ontologies and Ontology Languages	18
2.3.3 SOA and Web Services in the Semantic Web	20
2.4 Semantic Web Services Description	21
2.4.1 Semantic Annotations for WSDL (SAWSDL)	22
2.4.2 OWL-S: Semantic Markup for Web Services	24
2.4.2.1 Service Profile	25

2.4.2.2	Service Model	26
2.4.2.3	Service Grounding	28
2.4.3	Web Service Modeling Ontology (WSMO)	29
2.4.3.1	WSMO Ontologies and Web Services	30
2.4.3.2	WSMO Goals and Mediators	31
2.4.4	Semantic Web Services Framework (SWSF)	32
2.4.4.1	Semantic Web Services Language (SWSL)	32
2.4.4.2	Semantic Web Services Ontology (SWSO)	33
2.4.5	Comparison	35
2.5	Automated Web Service Composition	37
2.5.1	Workflow-related Approaches	38
2.5.2	AI Planning Approaches	40
2.6	The Frame Problem	42
3	Related Work	45
3.1	Solutions to the Frame Problem	45
3.1.1	Fluent Occlusion	46
3.1.2	Predicate Completion	47
3.1.3	Situation Calculus / Successor State Axioms	48
3.1.4	Fluent Calculus	50
3.1.5	Event Calculus	51
3.1.6	Explanation Closure Axioms	52
3.2	The Frame Problem in the Web Service domain	54
3.2.1	Situation Calculus	54
3.2.2	Event and Fluent Calculus	56
3.2.3	Linear Logic Deductive Planning	57
3.2.4	WSMO Working Group	58

4	The Frame Problem in Web Service Specifications	61
4.1	A Motivating Example	61
4.1.1	Atomic Web Service Specifications	63
4.1.1.1	Money Withdrawal Service	63
4.1.1.2	Wish List Update Service	65
4.1.1.3	Recommendations Update Service	66
4.1.2	Writing Frame Axioms	67
4.1.2.1	The necessity of Frame Axioms	67
4.1.2.2	Frame Axioms in more complex specifications	69
4.1.3	Service Composition and the Frame Problem	71
4.2	Addressing the Frame Problem	75
4.2.1	Expressing Change Axioms	77
4.2.2	Change Axioms in Web Service Languages	80
5	Automatic Production of Change Axioms	85
5.1	Atomic Service Specifications	86
5.2	Composite Service Specifications	87
5.3	OWL-S Service Descriptions	89
5.3.1	SWRL-FOL	89
5.3.2	An Example OWL-S Process Model	91
5.4	Implementation	96
5.4.1	Correctness	103
5.4.2	Detecting Inconsistencies	104
5.5	Evaluation	106
5.5.1	Execution Time	108
6	Conclusions and Future Work	113
6.1	Conclusions	113
6.2	Future Work	115

Bibliography	117
A CMU OWL-S API	127
A.1 Overview	127
A.1.1 Parsing and Writing OWL-S files	127
A.1.2 Building and Storing OWL-S Objects	128
A.2 Modifications to the CMU OWL-S API	130

List of Tables

2.1	Comparison of Semantic Web Service Frameworks	36
4.1	Pre/Postconditions for the Money Withdrawal Service	64
4.2	Pre/Postconditions for the Wish List Update Service	66
4.3	Pre/Postconditions for the Recommendations Update Service	66
4.4	Wish List Update Service Specification w/ Frame Axioms	68
4.5	Recommendations Update Service Specification w/ Frame Axioms	68
4.6	Money Withdrawal Service Specification w/ Frame Axioms	70
4.7	Composite Update Service w/ Frame Axioms	74
4.8	Money Withdrawal Service Specification w/ Change Axioms	78
4.9	Wish List Update Service Specification w/ Change Axioms	78
4.10	Recommendations Update Service Specification w/ Change Axioms	79
4.11	Composite Update Service Specification w/ Change Axioms	79
4.12	Composite Update Service Change Axioms In WSML	81
4.13	Composite Update Service Change Axioms In SWSL-FOL	81
4.14	Composite Update Service Change Axioms In SWSL-FOL	82
5.1	Algorithm for Atomic Web Service Specifications	86
5.2	Algorithm for Composite Web Service Specifications	88
5.3	SWRL-FOL formulas and variables	90
5.4	Algorithm for OWL-S Service Models	110
5.5	Evaluation of the Implementation	112

List of Figures

2.1	A representation of concepts defined by a WSDL document	12
2.2	The Web Services Architecture Stack	14
2.3	SAWSDL Annotations	23
2.4	Top level of the OWL-S ontology	25
2.5	The four main WSMO components	30
4.1	Example Scenario	62
4.2	Example Scenario: Order Execution	63
4.3	Order Execution: UML Activity Diagram	72
4.4	Order Execution: Composite Service	83
5.1	Flow Chart of the final algorithm	111

Chapter 1

Introduction

1.1 Motivation

Web services and service-oriented architecture (SOA) in general have emerged in recent years as a major technology for deploying automated interactions between distributed and heterogeneous applications [61] and have motivated a great deal of research on many topics such as service foundations, composition, management and monitoring as well as service design and development [62]. An inherent issue in all these topics, and one that directly or indirectly affects the research on them, is how to devise a formal specification of a Web service.

The need for formal specifications in the field of Web services is consistent with their respective need in software process specifications [15]. Formal specifications allow for a precise description of what a Web service is supposed to do and under what circumstances. In essence, complete specifications must offer us the knowledge of what conditions must be held before the service execution, what outputs will be produced given a certain set of inputs, what conditions must be held after a successful service execution and how the world is affected by that execution. Formal specifications also support the formal proof of certain properties, such as the fact that the execution of a service maintains the global state invariants [4], i.e. conditions that

CHAPTER 1. INTRODUCTION

must be true both before and after the service execution. Moreover, they allow for formal calculation of other properties such as the degree of similarity between services and the ability to substitute one with another, or the composability degree between sets of services, that checks whether a composition schema is possible or not, to avoid runtime failures [55].

A complete formal Web service specification should allow us to thoroughly know and, in most cases, predict the service behavior under any circumstance, effectively answering in the best possible way a simple question: "What does this service do?". This answer should not be limited to a set of inputs and outputs or an interface description, as provided by current standards such as WSDL [21] and should not have to deal with the service's inner workings, e.g. its source code, which a service consumer most probably has no access to and no clear understanding of.

Preparing formal specifications, however, comes with a great deal of issues that need to be solved. One particular family of problems that emerge in formal specifications expressed in the form of preconditions and postconditions is referred to in the field of Artificial Intelligence as the frame problem [52]. The frame problem stems from the fact that including clauses that state only what is changed when preparing formal specifications is inadequate since it may lead to inconsistencies and compromise the capacity of formally proving certain properties of specifications. Instead, one should also include clauses, called frame axioms, that explicitly state that apart from the changes declared in the rest of the specification, nothing else changes. Stating these frame axioms concisely is a rather complex task. Solving the frame problem essentially means finding a way to state frame axioms without resulting in extremely lengthy, complex, possibly inconsistent, obscure specifications and at the same time retaining the ability of proving formal properties of the specifications.

While the frame problem has been thoroughly described and addressed in AI literature, it has not been examined with regard to its existence in Web service specifications and the effects it has in service-oriented architecture in general. The pre-

condition and postcondition notation is universally used in functional descriptions of Web services, e.g. in the Service Profile class of OWL-S [51], the Web Service capability subcomponent of WSMO [66] or the Service Descriptors of SWSO [9]. Thus, one should expect that issues arising in formal specifications using the precondition and postcondition notation, an example of which is the frame problem, will also be encountered in formal Web service descriptions.

It should be apparent that ignoring the existence of the frame problem while preparing a formal Web service description will negate the benefits of formal specifications described in the beginning of this section. As a result, formal proofs of properties of Web service descriptions will be severely compromised, as the absence or ineffective statement of frame axioms will eventually lead to errors. The effects of the frame problem become further apparent when one attempts to compose Web services and produce a formal description of the resulting composite service. The conjunction of formal descriptions of the atomic services that participate in a composition will lead to inconsistencies in many cases, as all atomic service descriptions come with separate sets of frame axioms that may be contradicting. These remarks apply to any other service-related procedure that deals, at any point, with the precondition and postcondition clauses included in the service description, either by simply stating them, raising the issue of how to produce a concise description that takes into account the frame problem, or by using such a description as a basis to prove that some properties hold or not.

1.2 Contributions

In this thesis, we first set out to support the argument that the frame problem in Web Service specifications is indeed existent and can cause serious issues in Web Service frameworks and applications. A series of examples is presented, which hopefully makes the case that ignoring the existence of the frame problem and not trying to

address it is a major impediment in the evolution and advancement of Web Services and service-oriented architectures in general.

We then propose a solution schema to address the frame problem in Web Service specifications, in a similar way that it has been addressed in procedure specifications, which share many features with our case. The solution essentially rewrites frame axioms into a new set of clauses called explanation closure axioms or change axioms which are more concise and can be easily adapted in case we want to combine two or more specifications together.

This thesis also includes an algorithm that automatically produces change axioms, given service specifications as input. The algorithm deals with both atomic service specifications (simple services, executed in one step) as well as composite service specifications, where several atomic services are combined together using a defined composition schema (such as sequential, parallel, or iterational composition). The algorithm is kept simple enough to be applicable all kinds of different service specifications in any existing service description framework.

Finally, the algorithm is adapted and specialized in order to be used for OWL-S Service descriptions. The adaptation takes into account all special characteristics of an OWL-S specification and tries to render the algorithm more suitable for the case of OWL-S, removing or modifying some features, while adding others. This work also included an implementation of the final algorithm in Java. The result is an application which takes an OWL-S service description as input and produces a modified OWL-S service description as output, which keeps the same knowledge as the input but adds change axioms in any case they are needed. This application can be used by any framework that is based on OWL-S, in order to transform any service descriptions that are invoked, composed or produced by the framework, essentially freeing the framework from the issues caused by the frame problem.

1.3 Outline

The rest of this document is organized as follows. Chapter 2 presents a thorough examination of the background theory that is helpful to be understood before delving into the specifics of the actual work of the thesis. The background theory includes discussions on service-oriented computing, service-oriented architectures, Web Services, the Semantic Web, automated Web Service composition as well as the most popular Semantic Web Services frameworks.

Chapter 3 offers a concise description of work related to this thesis. Two main areas of research are covered. The first one deals with the different solutions that have been proposed for the frame problem in the field of Artificial Intelligence, since the problem was first proposed by McCarthy and Hayes in 1969. The second involves research in the field of Web Services that has, on some level, touched upon the matter of the frame problem, its existence in Web Services and Web Service specifications and the repercussions it may cause.

Chapter 4 first deals with stating preconditions and postconditions for composite Web services and then illustrates the issues raised due to the frame problem, through a series of further examples. In the second part of the chapter, the solution that we propose for the frame problem in Web Service specifications is explained in full detail.

Chapter 5 examines the issue of automatically producing the change axioms that are the heart of the solution to the frame problem presented in the previous chapter. An algorithm is presented that aims to automatically produce change axioms for an atomic Web Service specification. This algorithm is then extended to support composite Web Service specifications. The second part of this chapter deals with adapting the algorithm in order to support OWL-S service descriptions. An implementation of the algorithm is briefly presented and evaluated at the end of the chapter.

Chapter 6 summarizes the work of this thesis, offering some concluding remarks and gives pointers as to how this work can be further expanded and evolved.

Chapter 2

Background Theory: From Services to the Frame Problem

In this chapter, we will attempt to present all theories that from the background of this thesis. We will begin with the emergence of Service-Oriented Computing and Service-Oriented Architecture. Then, we will examine the technologies of Web Services and the Semantic Web and see how they fit in the Service-Oriented Computing paradigm. Afterwards, we will explore the different research efforts for describing Web Services in the Semantic Web and for automatically composing Web services in general. Finally, we will close our background exploration with the definition and implications of the frame problem.

2.1 Service-Oriented Architecture

In the last few years, Computer Science has observed a paradigm shift in the way software becomes available to end-users. The emergent paradigm is based on abstracting away from software and considering all resources as services that are available on-demand, forming what is called *Service-Oriented Computing (SOC)* [62]. SOC relies on services in order to facilitate the development of interoperable, dynamic, inex-

CHAPTER 2. BACKGROUND THEORY: FROM SERVICES TO THE FRAME PROBLEM

pensive and widely distributed applications. Services are entities that enable access to one or more capabilities, using a prescribed interface and in conformance to constraints and policies [27]. The interface as well as the constraints and policies are part of each service's description. Services are platform-independent and become available via a certain network, such as the Internet or the Universal Mobile Telecommunications System (UMTS). An important and characteristic feature of services is the fact that they are loosely-coupled, allowing ad hoc and dynamic binding in contrast to a dynamically linked library, an assembly or the traditional linker that binds together functions to form an executable.

The service deployment model can be applied to any application component or any preexisting code so that they can be transformed into an on-demand, network-available service. The general model of providers licensing applications to customers for a contractually-bound use as a service has been known as *Software as a Service (SaaS)* [11]. This practice is not limited to services, but can be applied to many other functions including communication, infrastructure and platforms, thereby leading to the concept of Everything as a Service which is the core of cloud computing [71].

Amidst this multitude of available services, the need for a set of design guidelines and principles in SOC endeavors is apparent. This need is satisfied through the *Service-Oriented Architecture (SOA)*. SOA is a style of design that guides all aspects of creating and using services throughout their lifecycle [58]. SOA enables an IT infrastructure to allow different applications to exchange data and participate in business processes, regardless of the underlying complexity of the applications, such as the exact implementation or the operating systems or the programming languages used to develop them.

2.1.1 Principles of Service-Oriented Architecture

The fundamental design principle of SOA is undoubtedly service-orientation. The paradigm of service-orientation is, in turn, defined by several other design principles

which have been analyzed in [24] and will be briefly outlined here.

Service loose coupling is a primary SOA principle. The goal is to decouple the service contract between the service provider and the service consumer from the service implementation, the service technology (which may be proprietary), the underlying composition that may constitute the service and so on. The service consumer needs to be independent from all these details so that flexibility and reusability of the services are maximized. Related to loose coupling is the next SOA concept, service abstraction. **Service abstraction** emphasizes the need to hide as much of the underlying details of a service as possible. The consumer needs to know only what is part of the service description available in the contract with the provider. An example of service abstraction is that service consumer don't need to know whether the service they are invoking actually encapsulates an entire service composition rather than being an atomic service. They only need to know that the service they are invoking offers the functionalities agreed upon with the provider.

Another SOA design principle is **service autonomy**. Applying this principle means that service designers should assess the amount of shared resources a service may need to rely on. Services should have a high control over their underlying runtime execution environment. Less control essentially means compromised predictability of the service performance and behavior. Service autonomy is reduced when services participate in compositions, thereby making it crucial to ensure that the building blocks of a composition are as autonomous as possible. Like service autonomy, **service statelessness** also focuses on ensuring the flexibility of a service, by deferring the management of state information when necessary, thus minimizing resource consumption. Storing excessive amounts of state information can compromise the availability and scalability of a service.

The reason services exist is to be consumed by service requesters and offer the functionality they provide. Discovering them and employing their capabilities should be facilitated by the services themselves. **Service discoverability** is a design prin-

ciple that captures exactly that notion. Services should contain meta-information in their descriptions that can be easily discovered and interpreted by anyone who searches for them. What should be contain in this meta-information is debatable and certainly not fixed. A prominent example is semantic information which, along with the generic umbrella of the Semantic Web will be covered in Section 2.3.

Services contain logic that is only partially known to the end user, which allows for viewing services as reusable resources. **Service reusability** as a principle focuses on dividing logic to services with the intention of promoting reuse. In this sense, service providers try to identify any potentially reusable logic and expose it as a separate service. More complex logic and more advanced functionality can be achieved by applying a paramount SOA design principle, **service composability**. Services are viewed not only as single function providers but as effective composition participants, regardless of the size and complexity of the composition. The ability to effectively compose services is a critical requirement for achieving some of the fundamental goals of service-oriented computing. The ultimate result of SOA is a collection of services with functionalities that can be composed in various schemas and stages in order to help satisfy required capabilities that can't be covered otherwise. Service composition will be revisited in Section 2.5.

2.2 Web Services

A SOA can be implemented using several different technologies such as SOAP [16], REST [30], CORBA [60] and Web Services. Web services are considered the most promising service-oriented technology. Before we examine Web services in the context of SOA, let's begin by examining their official definition by the World Wide Web Consortium (W3C).

A Web Service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface de-

CHAPTER 2. BACKGROUND THEORY: FROM SERVICES TO THE FRAME PROBLEM

scribed in a machine-processable format (specifically WSDL). Other systems interact with the Web Service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards. [13]

This definition gives a concise presentation of the advantageous characteristics of Web Services, which we will briefly examine here.

The defining characteristic of a Web Service is the use of the Internet and the World Wide Web as the communication medium for services to interact with each other and with service consumers. By using WWW, Web Services exploit the existing URI infrastructure in order to be located by anyone having access to the Web. The URI scheme gives a name to each Web Service which uniquely identifies it and allows one to use all existing operations on URIs in order to access it.

Web Services make extensive use of the XML language [17]. From the definition of the messages exchanged between services to the service description, everything is based on XML, which is quite advantageous. XML is a simple language, both machine- and human-readable, with an intuitive hierarchical structure. It is also self-describing, as each XML data structure contains both a description of its structure and the content itself. Web Services communicate by exchanging XML messages encoded using SOAP. SOAP defines a general pattern for Web Service messages, some message exchange patterns, how XML information can be encoded in such messages and how these messages can be transported over the Internet using existing protocols such as HTTP or SMTP.

XML is also used as the definition language for the de facto language used for describing Web Services, the Web Services Description Language (WSDL) [21]. A WSDL description of a service consists of a set of endpoints or ports. A port associates a network address with a port type, an abstract collection of supported operations, similar to an interface. Operations represent the functionality offered by the service. Each operation defines a simple exchange of messages, which, in their turn, define

CHAPTER 2. BACKGROUND THEORY: FROM SERVICES TO THE FRAME PROBLEM

the data format of a single request or response to the service. Each message consists of parts which is analogous to the way a program function consists of parameters. Message parts can be of any simple or complex XML datatype. As illustrated in Figure 2.1, a WSDL document can be divided into an abstract and concrete part, with the abstract part containing the types, messages, port types and operations definitions and the concrete part containing the ports and the bindings between ports and port types.

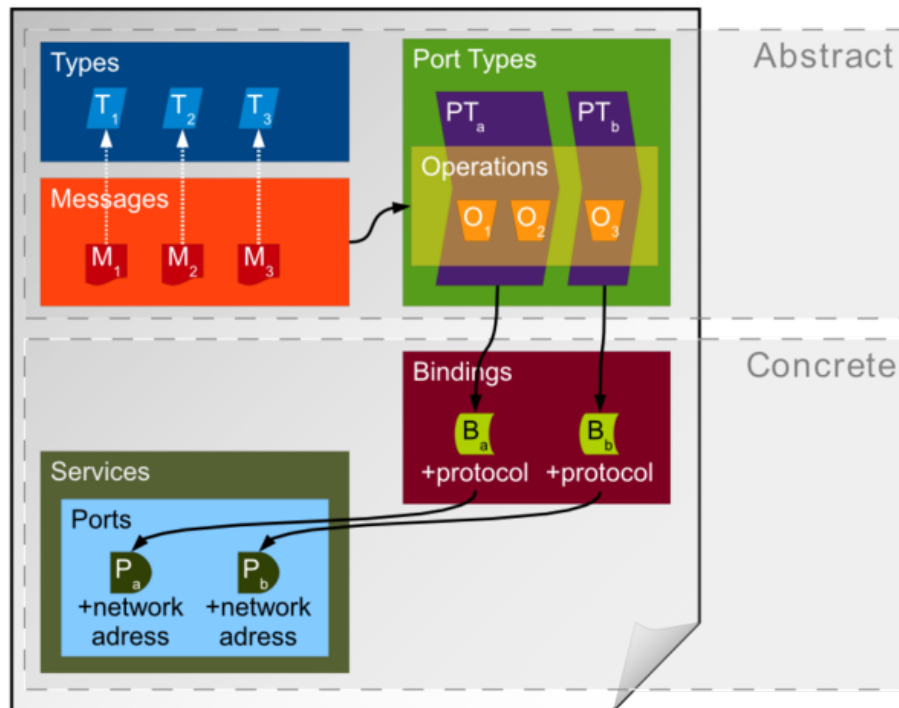


Figure 2.1: A representation of concepts defined by a WSDL document

2.2.1 Web Services Technology Stack

Web Services have generated a great deal of research and have been the focus of a lot of standardization efforts. To organize the technologies related to Web Services and the corresponding research areas, W3C has tasked the W3C Web Service Architecture Working Group to develop and maintain a model that describes how Web Services are generally structured. This model is called the Web Services Architecture Stack (WSA Stack) and is shown in Figure 2.2

The basis for all the layers in the WSA Stack is the **Communications** layer. It deals with the transport mechanisms used to exchange messages over the Internet, much like the Application Layer in the classical network architecture. Some examples are given, such as HTTP, SMTP, FTP and others but no transport mechanism is defined as the de facto one since this is a choice determined by the specific application at hand. On top of the Communications layer lies the core layer of the WSA Stack, the **Messages** layer. This layer provides encapsulation for network messages so that they are represented in a way that is interpretable by machines and humans alike. As mentioned above, SOAP is used for packaging and exchanging messages for Web Services and this is explicitly illustrated in the model. Along with SOAP, the model also considers SOAP extensions for reliable message transport, support for transactions and so on.

The third layer, counting from below, is the **Descriptions** layer and includes technologies for formally describing Web Services. A Web service description is indispensable since the service consumer needs it to access and use the service. WSDL is the de facto language for Web Services description, however the description it provides, as it should be apparent from the brief outline of the language offered previously, is only syntactical, containing basically the interface definition of the service. A semantic description is required for a machine to have the same level of understanding of the service's capabilities as a human would. We will defer this discussion to the next chapter, when we discuss the Semantic Web and the ground-breaking features

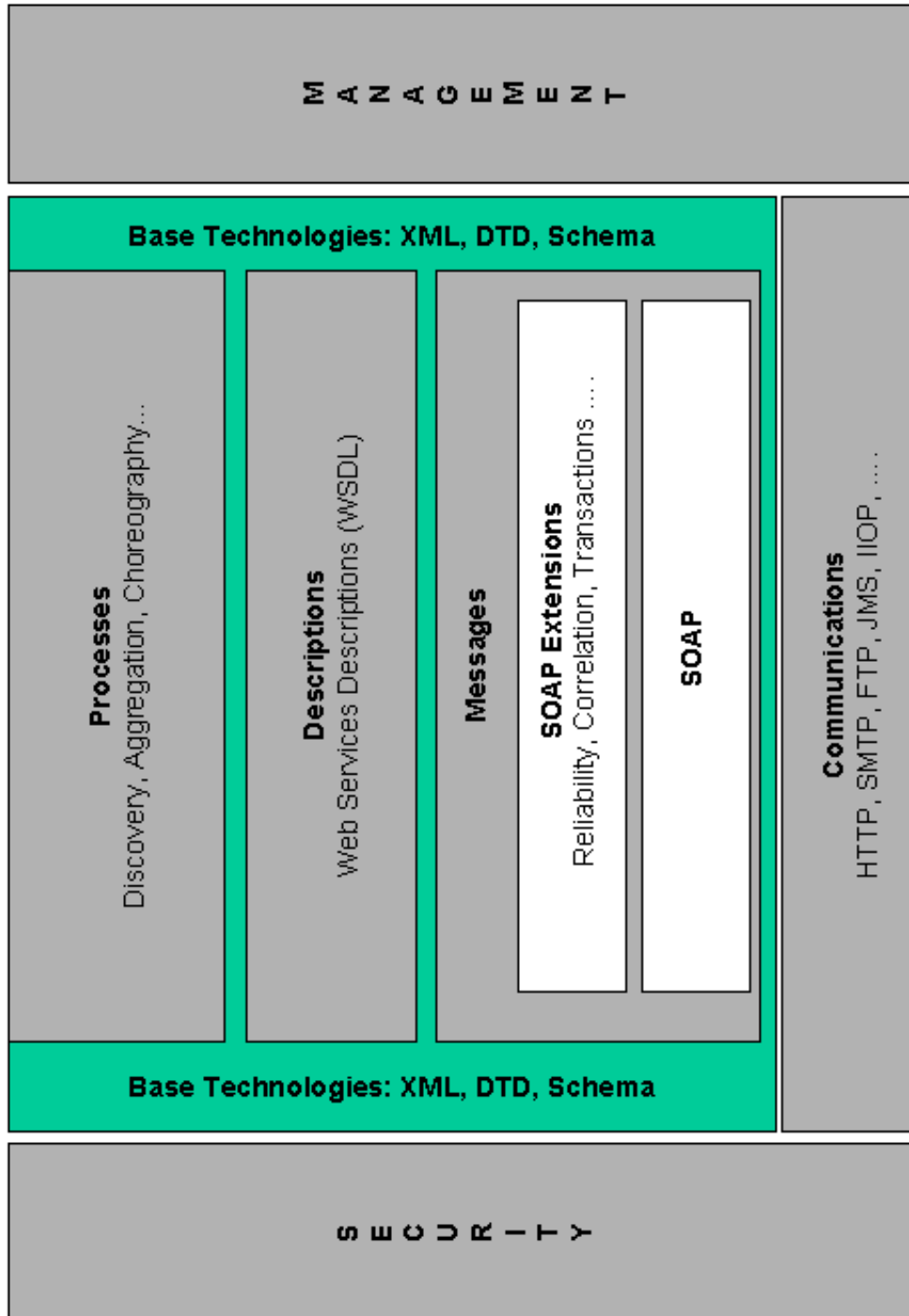


Figure 2.2: The Web Services Architecture Stack

it promises.

The top layer, named the **Processes** layer, includes all processes that are related to Web Services apart from the service description process which was the focus of the previous layer. The most important processes are Web Service Discovery and the various forms of composing or aggregating Web Services. Web Service Discovery deals with the methods used for locating a Web Service based on an explicit representation of the user's needs. This process relies on the existence of reliable service registries, such as Universal Description, Discovery and Integration (UDDI) [6], which however has somewhat failed to fulfill its role as a universal service registry.

As far as processes for composing Web Services are concerned, there are several models, such as Orchestration, Choreography or Coordination. Service Orchestration organizes services in a process flow focusing on implementing a composite service based on a service request. Service Choreography focuses on the interaction protocol between several partner services, defining the public processes of the partners and how they communicate together to achieve a certain goal. Service Coordination involves temporarily grouping a set of service instances which interact according to a coordination protocol and under the supervision of a special partner, the coordinator, who decides on the outcome of the protocol at the end of the activity. It is important to note that the WSA stack explicitly displays the use of XML, DTD and XML Schema in the Messages, Descriptions, and Processes layers.

In addition to the layers described already, there are some aspects that transcend a single layer, relating to all layers from Communications to Processes. These are displayed as vertical blocks in the WSA stack and involve Security and Management. The **Security** block deals with issues such as message encryption, message signing, malevolent service providers and so on. **Management**, on the other hand, has to do with the availability and reliability of a Web Service, viewing Web Services through the domain of business application requirements. Research in this area involves Quality of Service (QoS) aspects and service contracts such as Service Level Agreements

(SLAs).

The WSA Stack model shows the many different aspects that come under the umbrella of Web Services. Research in all of these areas is active worldwide, which is one of the reasons why Web Services hold much promise in realizing the vision of SOA. However, keeping the focus of Web Services on integration, automatization and machine to machine interaction requires tools that the traditional Internet and World Wide Web do not support. In a perfect SOA world, an application component in need for a specific service explicitly describes this need in a formal, concise and machine-understandable way and tasks a directory service with satisfying this need. The directory service will try to match the description of the necessary service with the descriptions of the services it is aware of. A list of matches is compiled and a candidate service is selected and is automatically bound to the requesting application component. This scenario is not realizable because the service descriptions offered by WSDL are inadequate and incapable of handling such cases. The Semantic Web technologies have the potential to make this scenario a reality.

2.3 The Semantic Web

The Semantic Web is an initiative that attempts to revolutionize the Web as we know it, by representing Web content in a form that is more easily machine-processable and by using intelligent techniques to take advantage of these representations [5]. In this new Web, the meaning of information plays a far more important role, helping search engines understand more clearly what the user requests and hence enabling them to return more accurate results than before. The Semantic Web is propagated by W3C and has generated a great deal of research by the industry and the academic community. One of the most important aspects of Semantic Web research is how to represent knowledge in a formal, machine-understandable and machine-processable manner. We will begin with an overview on basic knowledge representation aspects

and then we will attempt to view SOA and Web Services through the prism of the Semantic Web.

2.3.1 Knowledge Representation

Representing knowledge essentially means using symbols to represent real-world artifacts in a computational model and then performing reasoning by manipulating these symbols in order to answer questions and infer new knowledge. Current Semantic Web technologies offer three different forms of representing knowledge: Semantic Networks, rules and logic.

Semantic Networks use graphs as a form of knowledge representation, with nodes representing concepts and arcs representing relations between these concepts. Graphs are a powerful tool and are a more suitable form for computation than natural language. In Semantic Networks, one can use special relations to denote that a concept is a generalization or a specialization of another concept, that two concepts have a part-whole relationship or an individual-class relationship. Semantic Networks are more suitable for representing taxonomic structures rather than concrete individuals.

Another form of knowledge representation is to use rules and draw conclusions from them. **Rules** have a head and a body and hold the semantics that if the body part is true, then the head part is also true. In rule-based reasoning, a set of facts (rules with an empty body) is used as a knowledge base and then rules are successively applied in order to infer new knowledge. Complex sets of rules can be used to efficiently derive implicit facts from explicitly given ones. Rule languages are part of many Semantic Web frameworks, as we will observe in Section 2.4.

However, both Semantic Networks and rules rely on **logic** in order to be precisely formalized. The most prevalent logic representation formalism is first-order predicate calculus. First-order logic uses constructs such as predicates, functions, variables and logical connectives in order to construct logical formulas that describe the domain of interest. A set of logical formulas is called a theory. First-order logic allows for

statements that are always true (tautologies). Based on a theory one can derive logical consequences from it. Rules can be formalized using first-order logic or logic programming formalisms. Semantic Networks, on the other hand, can be formalized using description logics which contain concepts, roles (similar to roles) and constructors (such as intersection, union and negation). Description logics [7] are widely used in the design of ontologies and ontology languages, which we will examine next.

2.3.2 Ontologies and Ontology Languages

A definition of an ontology in the context of the Semantic Web community and Computer Science in general is offered by Tim Gruber in [35].

An ontology is an explicit specification of a conceptualization, which encompasses the objects, concepts, and other entities that are presumed to exist in some area of interest and the relationships that hold them.

Ontologies provide the Semantic Web with the tools to formulate and express semantic annotations to existing content. Knowledge representation languages such as the ones presented previously are used in order to formally express an ontology and ensure that it is machine-processable and machine-interpretable. Ontologies aim to conceptualize one particular domain at a time. It is impossible and rather naive to expect the definition of an ontology that covers a multitude of unrelated domains as this requires a common consensus between contrasting communities. Moreover, even within a particular domain, the narrower the scope of the ontology, the more flexible and useful the ontology will be.

An ontology is comprised of concepts, relations and instances. A concept represents each ontological category that is relevant to the domain of interest, much like in Semantic Networks or description logics and like unary predicates in first-order logic. A relation provides a semantic link from a concept or an instance to another concept or instance. An instance represents a named and identifiable concrete object in the

CHAPTER 2. BACKGROUND THEORY: FROM SERVICES TO THE FRAME PROBLEM

domain of interest, similar to the constants in logic. Using concepts, relations and instances one can express statements. An ontology is a set of these statements. An ontology can be graphically visualized primarily for better understanding by humans, interpreted as a logical theory by a reasoner or encoded in an ontology language for storage or transfer.

Ontologies differ according to their subject of conceptualization. Based on this categorization, there are four types of ontologies [37]. Top-level ontologies or upper ontologies describe very abstract and general concepts that may be shared across domains and as such, are rarely directly used in applications but other ontologies use them as basis. Domain ontologies capture the knowledge within a specific domain of interest. Task ontologies capture the knowledge about a particular task. Domain and task ontologies are more specific than upper ontologies and are usually kept independent of each other (domain ontologies independent of task ontologies and vice-versa). Finally, application ontologies have the narrowest scope and use both domain and task ontologies to describe a certain task execution in a particular domain context.

Ontology languages make ontologies available to information systems by encoding them in a concise and portable manner. The most prominent ontology languages in the Semantic Web are RDF [45] and OWL [10]. RDF serves as a language to represent knowledge as meta data about resources. An RDF ontology consists of RDF triples, which are sentences of the form of subject-predicate-object. The subject is the resource to be described, the predicate is a specific property of this resource and the object serves as a value of this property for this resource. URIs are used for naming entities. An RDF document is visualized as an RDF graph which is similar to a Semantic Network. RDF can be serialized as an XML document which is very useful for machine processing. While RDF is used to relate resources using properties, the RDF Vocabulary Description, known as RDF Schema (RDFS) [18] introduces resources classes and class hierarchies which facilitates the formulation of

RDF vocabularies.

The Web Ontology Language, or OWL, is the first attempt at dealing with the tradeoff between expressivity of an ontology language and scalability of reasoning, resulting in three different languages, OWL-Lite, OWL-DL and OWL-Full. OWL-Full is compatible with RDF and RDFS and is the most expressive variant, while OWL-Lite and OWL-DL are based on description logics, with OWL-DL being the most supported one. An OWL document contains classes, properties and individuals, similar to concepts, relations and instances in RDF, respectively. Classes can be defined based on existing ones using special relations such as intersection, complement, subclass, and equivalence. Classes and individuals are strictly separated, in contrast to concepts and instances in RDF. OWL offers three different syntaxes: the abstract syntax for human consumption, the RDF/XML syntax for compatibility with RDF and RDFS and a more compact XML syntax independent from RDF.

2.3.3 SOA and Web Services in the Semantic Web

Having briefly presented the Semantic Web, we go back to the discussion of SOA and Web Services in order to see how Semantic Web technologies can be exploited in their context. SOA involves searching for services based on functional and non-functional requirements and interoperating with those selected after the search. However, services will not be able to interact automatically, especially when scaling up to millions of services without automatization of the processes we mentioned in the WSA Stack, from discovery, negotiation and adaptation, to composition, invocation and monitoring. Thus, machine-processable and machine-interpretable semantics are critical in the realization of the SOA vision. Bringing semantics to SOAs leads to Semantically Enabled Service-Oriented Architectures (SESAs) as described in [19]. Semantically enabled SOAs place semantics at the core of service-orientation and provides rich semantic specifications to the building blocks of the architectures.

Bringing semantics into the world of SOA means that Web Services will not be

equally powerful in semantically enabled SOAs as in regular ones. The obvious thing to do is enhance Web Services with semantics, giving rise to Semantic Web Services (SWSs) [69]. SWSs use are described using semantic annotations based on ontologies, allowing for richer descriptions that are not limited to interface and port information. Semantically rich service descriptions allow for more effective service registries, easier service discovery and more powerful service composition. However, deciding which service is suitable given a user request becomes an even more demanding task with SWSs, since complex reasoning such as dynamic planning may be necessary. Moreover, traditional service description languages such as WSDL are unsuitable for describing SWSs. New service description languages based on ontologies and ontology languages are necessary and that will be our focus in the next section.

2.4 Semantic Web Services Description

In order to achieve the goals of SESAs, SWSs need to be described with a computational, machine-interpretable representation, using techniques based on knowledge representation, as was briefly presented in the previous section. The Semantic Web has led to the development of a set of formal languages and techniques for describing knowledge in a way which permits reasoning with it. When attempting to describe a SWS, one should choose one of these formal languages and then employ an ontology in order to define what specific concepts and relations are to be part of the description, in addition to their semantics. The formal language as well as the ontology selected (or created, if there is no suitable one) should allow for the concise and complete specification of the actions the service consists of, the results of these actions and the conditions under which the service produces these results.

There are four major research initiatives in the area of SWSs description. These are, in the order that we will examine them, the Semantic Annotations for WSDL and XML Schema (SAWSDL) [28], OWL-S [51], the Web Service Modeling Ontology

(WSMO) [66] and the Semantic Web Services Framework (SWSF) [9]. While the latter three are fully-fledged frameworks, SAWSDL is directly associated with WSDL, providing extensions to semantically annotate existing WSDL descriptions without defining the semantic information. We will begin our exploration with SAWSDL since it is closest to the existing world of Web Services.

2.4.1 Semantic Annotations for WSDL (SAWSDL)

SAWSDL is based on and shares the same design principles of previous work done by IBM and the LSDIS Lab of the University of Georgia and published as a W3C Member Submission with the title Web Service Semantics (WSDL-S) [2]. SAWSDL defines a way to semantically annotate WSDL interfaces and operations as well as XML Schema types, linking them to concepts in an ontology or a mapping document. The annotation mechanism is independent of ontology or mapping languages. The annotations are meant to help during Web service discovery, invocation and composition.

In Figure 2.3, the various elements of a WSDL description are shown and the ones that can be annotated in SAWSDL are marked clearly. As we can see, there are two basic semantic annotation constructs, through the extension attributes `modelReference` and `schemaMapping`. The **modelReference** attribute allows multiple annotations to be associated with a given WSDL or XML Schema component via a set of URIs. These URIs may identify concepts expressed in different semantic representation languages. When a component is annotated with a `modelReference` that includes multiple URIs, each of them applies to the component, but no logical relationship between them is defined by this specification. SAWSDL does not define any particular way to obtain the documents linked by these URIs, it is simply recommended that the URIs resolve to a semantic concept definition document, which could be placed directly within the WSDL document, provided that it is expressed using XML. `modelReference` attributes can be used in WSDL interfaces, operations

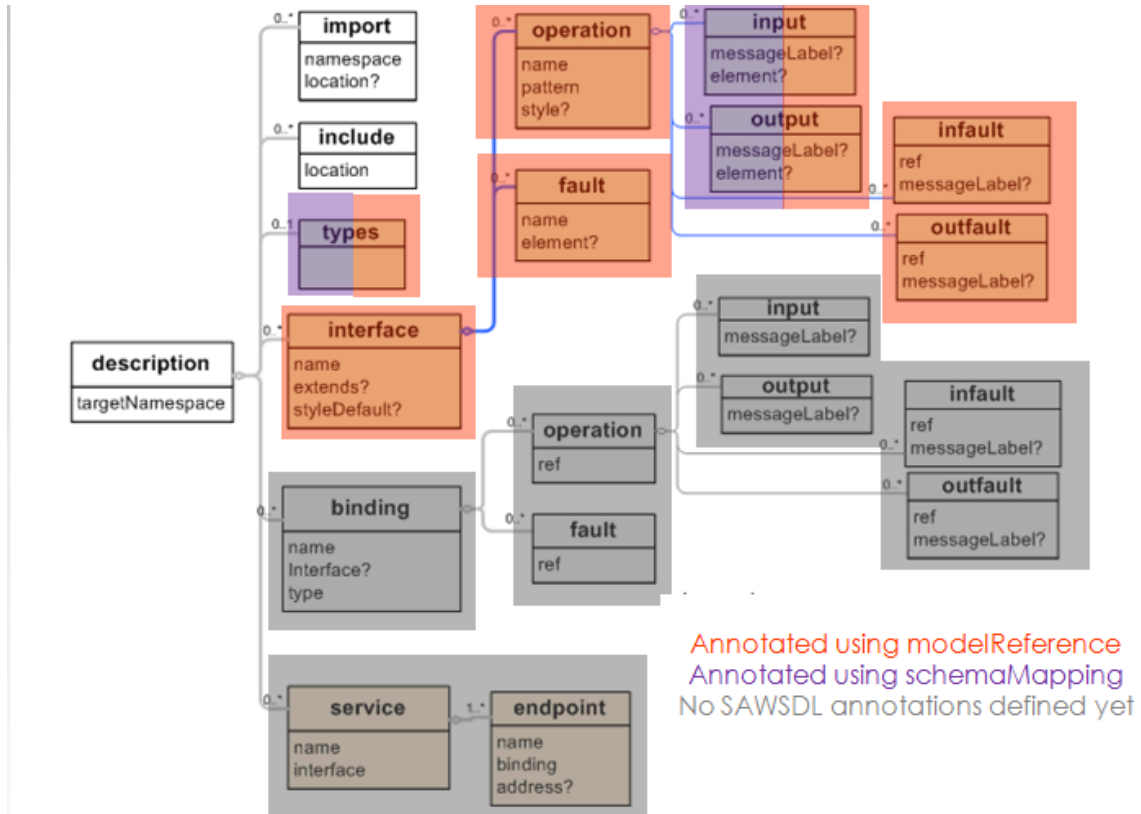


Figure 2.3: SAWSDL Annotations

and faults, as well as XML Schema elements, types and attributes.

As far as the second annotation mechanism is concerned, two attributes are provided: **liftingSchemaMapping** and **loweringSchemaMapping**. These are used to address post-discovery issues in using a Web service since there may still be mismatches between the semantic model and the structure of the inputs and outputs. Schema mapping relates the instance data defined by an XML Schema document with some semantic data defined by a semantic model (an ontology). Such mappings are useful in general when the structure of the instance data does not correspond trivially to the organization of the semantic data. The mappings are used when mediation code is generated to support invocation of a Web service. Schema mappings may be added to global type definitions as well as to global element declarations. It

is possible to specify either lowering or lifting information as well as both together on the same schema element. While `liftingSchemaMapping` defines how an XML instance document is transformed to data that conforms to some semantic model, `loweringSchemaMapping` goes the opposite direction, defining how data in a semantic model is transformed to XML instance data.

The SAWSDL extension can be applied to both WSDL 1.1 and WSDL 2.0 documents, although the latter case is more seamless than the first. Following from the ability of mapping WSDL 2.0 to RDF, SAWSDL-annotated WSDL documents can also be mapped to RDF. SAWSDL keeps the semantic model outside WSDL, making the approach independent from any ontology language. However, without describing how the use of annotations in different languages relate to one another, it is rather difficult, if not impossible to formally define requests, queries or matching between service requests and service descriptions. The efforts described in the rest of this section claim to do just that, among others.

2.4.2 OWL-S: Semantic Markup for Web Services

OWL-S is the result of a collaborative effort by researchers at several universities and organizations, including University of Toronto, Yale University, University of Maryland at College Park and Carnegie Mellon University, as well as SRI International and Nokia. The researcher's goal is to establish a framework within which these Web service descriptions are made and shared, employing a standard ontology, consisting of a set of basic classes and properties for declaring and describing services. The ontology structuring mechanisms of OWL provided them with an appropriate, Web-compatible representation language framework within which to realize their goal.

In Figure 2.4, the structuring of OWL-S in subontologies is shown. This structuring of the ontology aims to provide three essential types of knowledge about a service. The service profile subontology is used to describe what the service provides for the prospective clients. This information is used to advertise the service, construct

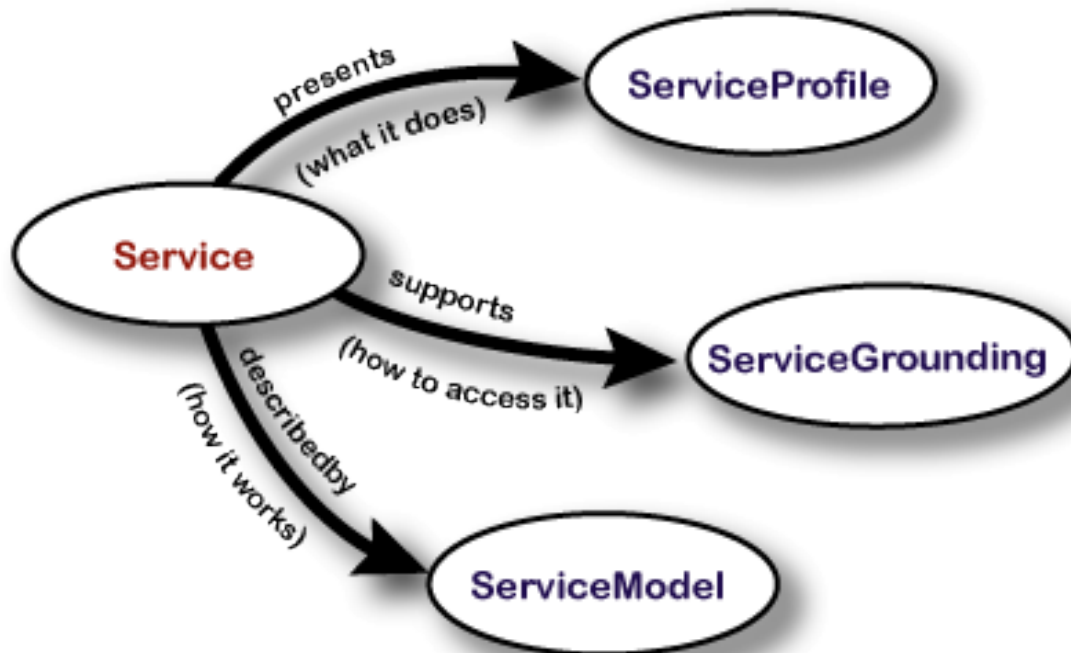


Figure 2.4: Top level of the OWL-S ontology

service requests and perform matchmaking. The service process model describes how a service works in order to enable invocation, enactment, composition, monitoring and recovery. Finally, the service grounding specifies how to access the service by providing the needed details about transport protocols.

2.4.2.1 Service Profile

Service Profile allows service providers to advertise the services they offer, in such a way that the service requesters can easily find what they are looking for. Service providers can create their own specialized subclasses of Service Profile to suit their needs. The specification has a built-in subclass, Profile, which is just an example of that. The Profile subclass describes three dimensions of a web service: the service provider, the service functionality and a set of service characteristics. The first dimension deals with the entity that provides the service and contains contact information

to anyone that may be associated with the service.

The second dimension of the Profile subclass is essentially the functional description of the web service. It contains the inputs that are necessary for the service to be executed, the preconditions that must be met to ensure a valid execution, the information that is generated as the output of the service as well as any effects to the state of the world that result from the execution of the service. This set of information is referred to as IOPEs (Inputs, Outputs, Preconditions and Effects). In some cases, it is interesting to couple an output and an effect as they are directly related and as a result the functional description is also referred to as IOPRs (with R denoting results).

Apart from these two dimensions, service providers is free to include any feature that they see fit to include in a service description. For instance, a Profile may contain information about the category of the service using an existing classification system such as the United Nations Standard Products and Services Code (UNSPSC). Also, a very important feature that should be part of a service description is the Quality of Service (QoS). QoS is a major factor in service discovery and selection, as searching only based on the functional description will yield services that may advertise the required operations, but may also be unreliable, slow or even malicious. OWL-S version 1.2 provides no properties relating to these features, leaving their specification entirely at the hands of service providers.

2.4.2.2 Service Model

The Service Model subontology in OWL-S describes the service functionality and specifies the ways a client may interact with the service in order to achieve its functionality. This is done by expressing the data transformation with the inputs and the outputs and the state transformation with the preconditions and effects. Although the Profile and the Process Model play different roles during the Web Service life-cycle they are two different representations of the same service, so it is natural to

CHAPTER 2. BACKGROUND THEORY: FROM SERVICES TO THE FRAME PROBLEM

expect that the IOPEs of one are reflected in the IOPEs of the other. There are no constraints between Profiles and Process Models descriptions, so they may be inconsistent without affecting the validity of the OWL expression. The Profile of a service provides a concise description of the service to a registry, but once the service has been selected the Profile is useless and the client will use the Service Model to control the interaction with the service. A Profile is usually a subset of a Service Model, containing only the information that is necessary in order to advertise the Web Service.

Similarly to the Service Profile class, Service Model also has a built-in subclass, Process. Process contains all the IOPEs of a particular service and views a Web Service similarly to a business process or a workflow. It is important to understand that a process is not a program to be executed, but a specification of the ways a client may interact with a service. A Process is further divided into three subclasses, atomic, simple and composite processes. An atomic process is a description of a service that expects one (possibly complex) message and returns one (possibly complex) message in response. It is the basic unit of implementation, is directly invocable by passing the appropriate parameters and is executed in a single step. A simple process provides, through an abstraction mechanism, specialized views of atomic processes or simplified representations of composite processes for purposes of planning and reasoning.

Composite processes are decomposable into other (non-composite or composite) processes; their decomposition can be specified by using control constructs such as Sequence, Split-Join or If-Then-Else, representing sequential, parallel and conditional composition schemas respectively. which are discussed below. A composite process is not a behavior a service will do, but a behavior (or set of behaviors) the client can perform by sending and receiving a series of messages. One crucial feature of a composite process is its specification of how its inputs are accepted by particular subprocesses, and how its various outputs are produced by particular subprocesses.

In many parts of a Service Model, conditions may need to be declared (precondi-

tions, or conditional effects, for instance). OWL-S ontologies are designed to accept conditions expressed in any language, but Semantic Web Rule Language (SWRL) [39] is the one favored. SWRL extends OWL by providing support for Horn-like rules. These rules are of the form of an implication between an antecedent (body) and consequent (head) with the meaning that whenever the conditions specified in the antecedent hold, then the conditions specified in the consequent must also hold. An XML syntax based on RuleML [1] and the OWL XML Presentation Syntax as well as an RDF concrete syntax based on the OWL RDF/XML syntax are provided.

2.4.2.3 Service Grounding

While the Service Profile and Service Model describe what a service provide and its inner design and how it works, the grounding of a service specifies the details of how to access the service. Many different kinds of information are involved: protocol and message formats, serialization, transport, and addressing. The role of grounding is mainly to bridge the gap between semantic description of web services and the existing service description models which are mainly syntactic, thus realizing process inputs and outputs as messages that are sent and received.

Following the trend set by the two other subontologies, Service Grounding has a built-in subclass, Grounding, which offers a grounding of OWL-S to the most prominent syntactic service description language, WSDL. An OWL-S/WSDL grounding is based upon a series of correspondences between the two languages:

1. OWL-S atomic processes correspond to WSDL operations
2. OWL-S inputs and outputs correspond to WSDL input and output messages
3. OWL-S input and output types correspond to WSDL abstract types

The example of a grounding mechanism that involves OWL-S and WSDL that is presented in the OWL-S specification, explicitly states that both languages are required for the full specification of that mechanism. OWL-S alone or WSDL by

itself cannot form a complete grounding specification. Both languages however lack something. On the one hand, WSDL is unable to express the semantics of an OWL class as it is not a semantic language and lacks many required features. On the other hand, OWL-S has no means, as currently defined, to express the binding information that WSDL captures. As a result, both languages are indispensable in a grounding declaration and this enforces the notion of an OWL-S/WSDL grounding that uses OWL classes as the abstract types of message parts declared in WSDL, and then relies on WSDL binding constructs to specify the formatting of the messages.

2.4.3 Web Service Modeling Ontology (WSMO)

WSMO is a conceptual model for describing various aspects related to Semantic Web services, produced by the ESSI WSMO working group which consists mainly of members of DERI and The Open University. The objective of WSMO and its accompanying efforts is to solve the application integration problem for Web Services by defining a coherent technology for Semantic Web Services. As illustrated in Figure 2.5, four main components are defined in WSMO. Ontologies provide the formal semantics to the information used by all other components. Goals specify objectives that a client might have when consulting a Web service. Web services represent the functional and behavioral aspects which must be semantically described in order to allow semi-automated use. Finally, Mediators provide interoperability facilities among the other components.

To formalize WSMO, Web Service Modeling Language (WSML) was also developed by the WSMO working group. The formal grounding of the language is based on a number of logical formalisms that we touched upon in a previous Section, namely, description logics, first-order logic and logic programming. There are different WSML variants which use some or all of these formalisms. WSML-Core is based on logic programming, WSML-DL is a full-fledged description logic language, WSML-Rule allows the use of functional symbols and WSML-Full unifies WSML-DL and WSML-Rule,

under a common syntactic umbrella, inspired by first-order logic. Besides providing its own language for modeling ontologies, WSML allows the import and use of RDF Schema and OWL ontologies for Web service description.

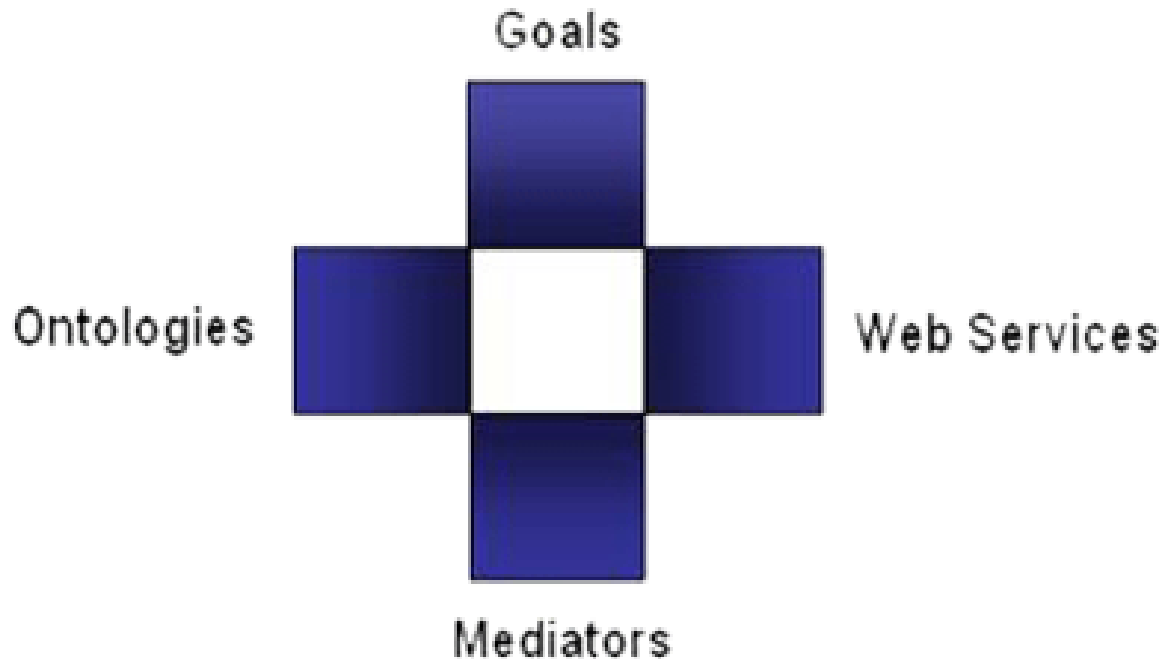


Figure 2.5: The four main WSMO components

2.4.3.1 WSMO Ontologies and Web Services

Ontologies provide domain specific terminologies for describing the other elements. The basic blocks of an ontology are concepts, relations, functions, instances, and axioms. Concepts, relations and instances play similar roles to the ones we described earlier in the general definition of an ontology. Axioms are specified as logic expressions and help to formalize domain specific knowledge. Functions are specialized relations that have a unary range. When declaring a function one must include an axiom that states the functional dependency. Non-functional properties can also be part of a WSMO ontology, as can other ontologies which can be imported either or

directly, or through the intervention of a mediator if there are inconsistencies between concepts, relations or functions.

Web Services description in WSMO is realized using two different viewpoints: the interface and the capabilities. WSMO capabilities are similar to the IOPEs used in WSMO. A capability should contain preconditions and assumptions that must be true before the execution of the service, postconditions that are true if the service has executed successfully, and effects that illustrate how the service execution changes the world. As in ontologies, non-functional properties can also be declared, as well as imported ontologies and used mediators.

A WSMO interface describes how the functionality of the Web Service can be achieved. A twofold view of the operational competence of the Web Service is offered: orchestration and choreography, Orchestration offers a description of how the overall functionality of the Web service is achieved by means of cooperation of different Web service providers, while choreography is essentially the description of the communication pattern that allows to one to consume the functionality of the Web service. WSMO allows more than one interface to be defined, so multiple choreographies can be defined for a Web Service, hence allowing for multiple modes of interaction. Choreographies and orchestrations are based conceptually on Abstract State Machines, defining a set of states where each one is described by an algebra, and guarded transitions to express state changes by means of updates.

2.4.3.2 WSMO Goals and Mediators

WSMO Goals define exactly what a service requester demands from a Web Service to offer. The requester (whether it is a human or an agent) defines the goals and the corresponding system tries to find services or combinations of services that can realize them. In order to produce a complete goal description, we need to describe the interface and the capabilities of a Web Service that, if it existed, it would completely satisfy our request. In this way, requests and services are strongly decoupled, since

requests are based directly on goal descriptions.

Finally, Mediators connect heterogeneous components of a WSMO description when one encounters structural, semantic or conceptual incompatibilities. Mismatches can arise at the data or process level. Mediators are extremely useful in distributed and open environments such as the Internet, where vastly different objects must interoperate. Four types of mediators are defined. Ontology mediators help import elements of one ontology to another, goal mediators define at what level a goal is equivalent to another, Web Service mediators assist in the interaction of different Web Services and Web Service-to-goal mediators can answer whether a Web Service satisfies partly or completely a goal, or whether a Web Service demands the satisfaction of a goal.

2.4.4 Semantic Web Services Framework (SWSF)

SWSF is another effort to realize the Semantic Web Service vision and is influenced by both OWL-S and WSMO. The Semantic Web Service Initiative is behind SWSF, with partners that also contributed in OWL-S, such as SRI International and University of Toronto and other important partners such as Hewlett Packard, Massachusetts Institute of Technology, Bell Labs Research and Stanford University. SWSF comprises of two major components, the Semantic Web Services Ontology (SWSO) and the Semantic Web Services Language (SWSL).

2.4.4.1 Semantic Web Services Language (SWSL)

SWSL is used to specify formal characterizations of Web service concepts and descriptions of individual services. It includes two sub-languages. SWSL-FOL is based on first-order logic and is used primarily to express the formal characterization of Web service concepts. SWSL-Rules is based on logic-programming and rule-based reasoning and is used to support the use of the service ontology in reasoning and execution environments. SWSL has been designed to address the needs of Semantic

Web Services, but is also a general-purpose language.

SWSL-Rules is designed to provide support for a variety of tasks that range from service profile specification to service discovery, contracting, policy specification, and so on. The language is layered to make it easier to learn and to simplify the use of its various parts for specialized tasks that do not require the full expressive power of SWSL-Rules. The core layer of SWSL-Rules contains only Horn rules and the rest of the layers build on top of that, allowing such extensions as disjunctions, negation, quantifiers and implication in the rule body and conjunction and implication in the rule head as well as common object-oriented features inspired by F-logic.

SWSL-FOL is the first-order logic subset of SWSL. The SWSL language includes all the connectives used in first-order logic and, therefore, syntactically first-order logic is a subset of SWSL. SWSL-Rules and SWSL-FOL share significant portions of their syntax. However, not every first-order formula in SWSL-FOL is a rule and the rules in SWSL-Rules are not first-order formulas. SWSL-FOL is also layered, with extensions adding equality, object-oriented and higher order features. Both SWSL-FOL and SWRL-Rules can be serialized in XML using the RuleML syntax. RuleML-style serialization of SWSL enables interoperation with other XML applications for rules and provides an encoding for transporting SWSL-Rules via the SOAP infrastructure of Web Services.

2.4.4.2 Semantic Web Services Ontology (SWSO)

SWSO presents a conceptual model by which Web services can be described, and a formal characterization of that model in first-order logic, using SWSL-FOL. This axiomatization of SWSO in first-order logic is called FLOWS (First-Order Logic Ontology for Web Services). In addition, the axioms from FLOWS have been translated into the SWSL-Rules language, resulting in an ontology which relies on logic-programming semantics and is called ROWS (Rules Ontology for Web Services). SWSO is partly inspired by the principles of situation calculus [52], modeling the changing portions

of the real world abstractly, using fluents.

Similarly to OWL-S, SWSO is divided into three major components: Service Descriptors, Process Model and Grounding. **Service Descriptors** are the equivalent to OWL-S Service Profile and WSMO non-functional properties and contain basic identifying information for the service, such as the service name, author, contact information and so on. The set of properties can be expanded to include other properties, domain-specific or not.

The SWSO **Process Model** provides an abstract representation for Web Services, their impact on the world and the transmission of messages between them. It is based on the Process Specification Language [36], a formally axiomatized ontology that was originally developed to enable sharing of descriptions of manufacturing processes. The fundamental building block of an SWSO Process Model is the atomic process, which is directly invocable, has no subprocesses and can be executed in a single step. An atomic process is associated with a set of IOPEs, which are expressed using fluents. However, inputs and outputs are treated as special kinds of preconditions and effects. An input is a knowledge precondition while an output is a knowledge effect.

Service composition is modeled by complex activities using a set of control constraints (similar to the OWL-S control constructs), ordering constraints (specifying just the order of execution) occurrence constraints (linking constraints with a specific activity occurrence) and state constraints (associating triggered activities with states). IOPEs are defined for atomic processes only. The IOPEs from complex activities are inferred by the atomic processes that constitute the activity. As far as messages are concerned, SWSO differentiates itself from OWL-S and WSMO, defining explicit objects for both messages and channels, the latter giving some structure to how messages are transmitted between services.

Finally, SWSO **Grounding** allows abstract SWSO service descriptions to be linked with one or more concrete realizations, such as WSDL. The central role of an SWSO/WSDL grounding is to provide an execution environment with the informa-

tion it needs at runtime to send and receive messages associated with WSDL service descriptions. The following series of correspondences between SWSO and WSDL are defined:

1. SWSO complex activities correspond to WSDL operations
2. WSDL message patterns corresponds to the patterns of messages contained within the control structure of complex activities
3. SWSO messages are serialized as WSDL messages, using XML schema for message types

2.4.5 Comparison

Before completing the talk on Semantic Web Services description, we offer a summary and comparison of the languages presented. We omit SAWSDL from the comparison, because it is not a fully fledged semantic framework as the three others are, and as such, it can't be compared on the same basis.

From the languages presented in this section, OWL-S is the most mature one and the most commonly used in service discovery and composition research approaches. OWL-S, however, doesn't come without its drawbacks, as it is pointed out in [8], which prevent it from being used in practical real-world scenarios. For example, its process model is neither an orchestration model nor a choreography model. It is essentially a behavioral description of the service. Moreover, OWL-S views Web Service execution as a collection of remote procedure calls, which applies only to synchronous communication, leaving asynchronous communication out of the picture.

WSMO also has been used in many research approaches, even though it hasn't been around as long as OWL-S. The process model of WSMO offers both an orchestration and a choreography view. However, the orchestration view is rather primitive, and the WSMO Choreography model contains transition rules that only represent local constraints. In addition, WSMO Choreography needs to be formalized in order

CHAPTER 2. BACKGROUND THEORY: FROM SERVICES TO THE FRAME PROBLEM

to allow reasoning and verification about it.

As far as SWSF is concerned, it is a very ambitious approach and has the most elaborate formalization of the three semantic framework efforts and relies on tried and proven foundations for both the included ontology and the language. However, most of its features are present in either OWL-S or WSMO, without necessarily keeping the best of both efforts. In Table 2.1 a comparison between OWL-S, WSMO and SWSF is provided, based on the following criteria:

- The support for Semantic Web Services discovery
- The process modeling support (i.e. orchestration, choreography, behavioral interface)
- The formalisms to which the behavioral semantics are mapped
- The support of synchronous and/or asynchronous inter-process communication
- The existence of research approaches on composition using the languages

Semantic Web Service Frameworks	SWS Discovery Support	Process Model	Behavioral Semantics	Inter-Process Communication	Existing Composition Approaches
OWL-S	Yes	Limited Orchestration	Situation Calculus, Petri Nets	Synchronous	Yes
WSMO	Yes	Choreography	ASMs	Both	No
SWSO	Yes	Both	Same as OWL-S	Synchronous	No

Table 2.1: Comparison of Semantic Web Service Frameworks

2.5 Automated Web Service Composition

Approaches to the Web service composition problem have been exceptionally diverse and offer different interpretations of what should be addressed in a composition approach. Original approaches focused on static or design-time composition, where the component services are chosen, linked and compiled before runtime and the composition schema does not change throughout the execution. Static composition is effective only as long as there are no significant changes in the participating services, which is rarely the case. A less restrictive and more realistic approach would be to adapt the composition schema to reflect unpredictable changes and effects at runtime. Such dynamic or runtime composition approaches should be more effective in real-world service environments where everything is fluid and flexible, new services become available while others disappear and many unpredictable events can occur at runtime.

Another separating characteristic between approaches is whether the composition schema is created manually or automatically, in other words, the level of user intervention in the process of service composition. Automated composition aims to create a composition schema based on a set of user-defined goals and a set of atomic services by applying a series of composition rules. In our survey, we will focus on efforts that include a degree of automation in the composition process while also presenting manual composition models that have been widely used.

The process of creating a composition schema in order to satisfy a series of goals set by a requester is a really complex and multifaceted problem, since one has to deal with many different issues at once. First of all, it involves searching in an ever-growing global service repository in order to find matching services that may contribute to the complete satisfaction of the user's requirements. Assuming that these services have been found, one has to successfully combine them, resolving any conflicts and inconsistencies between them, since they most certainly will be created by different people using different implementation languages and systems. Since inconsistencies

may occur at runtime, it may be necessary to predict such events so as to ensure that the system will run correctly. Finally, even after having overcome these issues, we have to be able to adapt to the dynamic characteristics of service-based systems, with services going offline, new services becoming online, and existing services changing their characteristics.

Attempting to overcome all these problems manually, will most certainly lead to a composition schema that is not fault-proof while the whole procedure will consume a lot of time and resources and cannot be considered scalable. Therefore, it is apparent that a certain degree of automation needs to be introduced to the composition procedure. Approaches that involve automation in the creation of the composition schema as well as during its execution constitute a major family known as automated service composition.

In this section, we will attempt to briefly summarize the main SOC efforts to provide an automated Web Service composition framework. This summary will present the two major categories of approaches, the workflow-related approaches, which is the earliest chronologically and the AI planning approaches. Throughout the section, we will offer pointers as to how existing approaches could be evolved to support Semantic Web Services, using the frameworks and languages presented in Section 2.4.

2.5.1 Workflow-related Approaches

Workflow organization and management have been a major research topic for more than twenty years. As a result, there has been a lot of effort on how to represent a sequence of actions. Drawing mainly from the fact that a composite service is conceptually similar to a workflow, it is possible to exploit the accumulated knowledge in the workflow community in order to facilitate Web service composition. Workflow-related techniques may either provide a static composition scheme, meaning that only selection and binding of atomic Web services is performed automatically or a dynamic one, where the process model creation is also automatic.

Composition frameworks based on workflow techniques were one of the initial solutions proposed for automatic Web service composition. In [20], the authors present EFlow, a framework for specifying and managing service compositions. In EFlow, a service composition is represented by a graph containing three different kinds of nodes. Service nodes represent that a service is invoked, either an atomic or a composite one. Decision nodes are used to describe the execution flow, while event nodes represent the exchange of events between services. The graph is constructed manually, thus EFlow is considered a semi-automatic composition platform. However, the binding of concrete services to the abstract service nodes is done automatically and can be modified dynamically. Service nodes contain code that enables a search to be performed each time the flow leads to a particular node. As a result, the composition schema adapts dynamically to any unexpected changes in the availability of services.

[3] and [49] deal with abstract workflows. [3] argues that business process flow specifications should be abstractly defined and only bound to concrete services at runtime. The authors present a BPEL engine that is able to take an abstract workflow specification and dynamically bind Web services in the flow taking into consideration any relations and dependencies between services as well as domain constraints. [49] advances one step further by allowing for the automatic generation of an abstract workflow based on a high-level goal. An integrated Reasoning Service aims to create a combination of available services that provide the required functionality to satisfy the goal, based on inputs, outputs, preconditions and effects. Then, as in all previously mentioned frameworks, the abstract workflow tasks are matched with concrete deployed services. If no service is found for a task, the Reasoning Service along with a Matchmaking Service attempt to compose existing services in order to create a service that is match for that particular task. The matching process results to an executable graph for the composition.

BPEL has become the de facto orchestration model for workflow-related composition approaches. The fact that classic BPEL flows only enable static Web Service

composition has led to research efforts such as the ones presented in the previous paragraph in order to allow for some type of dynamic binding. As far as semantics are concerned, there has been a recent effort from the University of Stuttgart to link BPEL to the WSMO semantic framework. BPEL For Semantic Web Services (BPEL4SWS), as it is called, uses BPEL^{light} [59] as its base, which is, in its essence a WSDL-less BPEL, i.e. leaving out the interface description while keeping the BPEL process model. Semantic Web Service descriptions are then attached to BPEL^{light} processes so that semantic-based frameworks can be used to discover and select Semantic Web Services that implement the functionality required by a BPEL activity.

2.5.2 AI Planning Approaches

AI planning techniques involve generating a plan containing the series of actions required to reach the goal state set by the service requester, beginning from an initial state. All approaches in this family rely on one of the many planning techniques that the AI community has proposed and incorporates it in the process model creation phase of the composition framework.

A large number of AI planning techniques involves expressing a domain theory in classical logic. A set of rules is defined and plans are derived based on these rules. In [4], the authors extend composition rules to include new constraints known as invariants, in order to capture the knowledge that a property must not change from a state to another. These constraints are also adapted to the existing service description architecture of OWL-S. The resulting set of composition rules is used in a backward planning approach enriching it with reasoning about service compositionality. The backward planner creates the composition plan by decomposing the given set of goals and trying to find services that satisfy a sub-goal while indulging the composition rules for that particular composition. A successful plan is returned upon reaching a point where all goals are satisfied.

CHAPTER 2. BACKGROUND THEORY: FROM SERVICES TO THE FRAME PROBLEM

Apart from rule-based planning, another AI planning in the field of logic is planning with situation calculus. The situation calculus represents the world and its change as a sequence of situations, where each situation is a term that represents a state and is obtained by executing an action. McIlraith and Son [54] adapt and extend Golog, a logic programming language built on top of the situation calculus in order to allow for customizable user constraints, nondeterministic choices and other necessary modifications to better satisfy the requirements of service composition.

Despite the advantages of the aforementioned pure logic-based approaches, such as the ability to prove certain properties of domain theories and precise semantics, the AI planning community developed and uses several different formalisms to express planning domains. Planning Domain Definition Language (PDDL) [32] is widely recognized as a standardized input for state-of-the-art planners and is based on earlier languages such as STRIPS and ADL. In [64], WSDL descriptions along with service-annotation elements are transformed to PDDL documents. To deal with the problem of partial observability of state that is often encountered in service compositions where lack of information is common, the framework allows for replanning when additional knowledge has been acquired.

Hierarchical Task Network (HTN) planning [25] [26] provides hierarchical abstraction to deal with the complexity of real-world planning domains, such as Web services. HTN planning allows for decomposition of tasks into subtasks. Klusch et al. [42] present OWLS-Xplan, a framework that combines graph-based with HTN planning and uses PDDL descriptions. An OWLS2PDDL converter performs the matching between OWL-S services and PDDL actions. PDDXML, an XML dialect of PDDL is also developed, to facilitate SOAP communication of PDDL descriptions. As far as the planning component is concerned, Xplan is a hybrid planner that combines a Relaxed Graphplan heuristic [38], with an HTN component that aims to decompose any complex task that is encountered during the graph generation. The framework also supports replanning in order to adapt to dynamically changing environments.

2.6 The Frame Problem

As we witnessed throughout this chapter Service-Oriented Computing and Service-Oriented Architectures are closely linked with semantics and we briefly explored some efforts to take semantics into account when developing Web Service frameworks. However, introducing semantics to service specifications doesn't come without issues. A common feature in all Semantic Web Service frameworks is the use of IOPEs in service descriptions. However, using the precondition/postcondition notation doesn't come without some issues, one of which is the frame problem.

The frame problem was originally defined in Artificial Intelligence by McCarthy and Hayes in [52]. The authors were concerned with the fact that when describing a series of actions in a dynamic domain in logic, one has to account for what each action changes and doesn't change. This was immediately considered as a problem that needed to be dealt with, especially for a large number of actions. For instance, if we handle theories containing n actions and m statements that describe each action, then we might have to write down mn conditions that explicitly declare what these actions do not change. The authors do propose a solution which, however, was proven shortly afterwards to not always be correct.

The frame problem is essentially the problem of stating succinctly in a logic theory that nothing else changes, except when it is explicitly mentioned otherwise. The extra conditions that we need to use in order to achieve this purpose are called *frame axioms*. The process of defining in a formal and complete way the frame axioms is definitely a demanding, difficult and error-prone one and makes theories lengthier and harder to handle.

The frame problem applies to the case of formal descriptions, such as procedure specifications [15] or, inevitably, Web Service specifications. Formal specification languages are characterized by criteria such as expressiveness, notational suitability and the capacity to support formal treatment. Notational suitability is the degree to which a specification language can provide simple, concise, understandable, modular

CHAPTER 2. BACKGROUND THEORY: FROM SERVICES TO THE FRAME PROBLEM

and easily modifiable specifications, while formal treatment involves formally proving properties of the specification. The frame problem can compromise most of these abilities and render a formal specification language inadequate and unsuitable for the very purpose that it was created.

[15] succeeds in presenting the reasons why procedure specifications are affected by the frame problem and the same reasons apply to the case of Web Service specifications. Apart from the fact that stating frame axioms results in lengthy and complex specifications and renders consistency proofs and any kind of formal treatment of the specification prohibitively expensive, there are three other characteristics of procedure specifications which worsen the frame problem. Using conditional cases in specifications is one of them. The problem with conditionals is that we have to state ahead of time the list of things that might change in the procedure, and this forces us to state explicitly for any branch of the conditional the things that do not change.

Moreover, conjoining specifications which already contain frame axioms is an even more problematic task. Since we most probably will conjoin specifications that don't take into account the existence of one another, then it is possible that some of the frame axioms of one specification contradict the frame axioms of the second specification. Conjoining these contradicting specifications will unfortunately result in an inconsistent specification, which is definitely something we want to avoid. Finally, related to that issue, is the handling of inheritance, since it involves conjoining the specification of the superclass to the additional assertions introduced by the subclass.

As we argued in the introductory chapter, the issues raised by the frame problem and described here affect Web Service specifications in general and especially Semantic Web Service specifications. The next chapter will briefly summarize the solutions that have been proposed for the frame problem in Artificial Intelligence and then we will examine to what extent these solutions have been successfully applied in the case of Web Services.

CHAPTER 2. BACKGROUND THEORY: FROM SERVICES TO THE FRAME
PROBLEM

Chapter 3

Related Work

In this chapter, we will explore the various solution schemas that have been proposed for the frame problem in the field of Artificial Intelligence in general, presenting them in a rough chronological order. We will leave the solution that was the basis of this thesis last. In the second part of this chapter, we will examine the frame problem from the viewpoint of the Service-Oriented Computing research community, analyzing any efforts that have considered the implications of the frame problem rather than choosing to ignore them.

3.1 Solutions to the Frame Problem

To illustrate the solution, we will present a very simple scenario and apply each solution to it. Suppose that we have a credit card, which can be valid or invalid, and a payment processed through that credit card, which can be completed or not. We can use two predicates $\text{valid}(t)$ and $\text{completed}(t)$ which indicate whether the credit card is valid or not and the payment is completed or not at time t . Suppose that, at time 0, the credit card is valid and the payment has not been completed yet. At time 1, an action takes place that completes the payment. The following formulas express this knowledge:

$valid(0)$
 $\neg completed(0)$
 $true \rightarrow completed(1)$ ¹

We don't deal with preconditions or specifying when an action is executed for this presentation in order to keep the example as simple as possible. The frame problem occurs in this case because, we don't know whether the completion of the payment has any effect to the validity of the credit card. Both $valid(0)$, $\neg valid(1)$ and $valid(0)$, $valid(1)$ can be true. However, if we add the frame axiom $valid(0) \equiv valid(1)$, then we ensure that only the second pair of conditions is true. However, we need to add such a frame axiom for every pair of action and condition, which is easy for one action as in the case of our simple example, but definitely not easy in real-world scenarios.

3.1.1 Fluent Occlusion

Three years after the frame problem was defined by McCarthy and Hayes, a solution was published by Erik Sandewell [23]. Sandewell's approach relies on the fact that not only the values of conditions are represented but also whether they are affected by the last executed action. This knowledge is formulated using a new set of conditions, called *occlusions*. The definition of occlusions states that a condition is occluded if an action has just been executed and as a result of that execution, the condition has been made true or false.

Returning to the example, we can use two new predicates $occlude_valid(t)$ and $occlude_completed(t)$ to denote occlusion. Occlusion predicates are true only when an action affecting the corresponding condition is executed. A predicate can change

¹We use this notation, instead of stating simply that $completed$ is true at time 1, in order to differentiate a fact from an effect. The first two formulas state facts, while the third is an effect in our domain, even if it will always hold.

value only if the corresponding occlusion predicate is true at the next time point.

$$\neg \text{valid}(0)$$
$$\neg \text{completed}(0)$$
$$\text{true} \rightarrow \text{completed}(1) \wedge \text{occlude_valid}(1)$$
$$\forall t. (\neg \text{occlude_valid}(t)) \rightarrow (\text{valid}(t-1) \equiv \text{valid}(t))$$
$$\forall t. (\neg \text{occlude_completed}(t)) \rightarrow (\text{completed}(t-1) \equiv \text{completed}(t))$$

The occlusion approach is successful if occlusion predicates are true only when they are made true as a result of an action. The techniques of circumscription or predicate completion (as described in the following subsection) must be used to ensure that fact. It should be noted that Sandewell later defined a formal framework for the specification of dynamic domains, which includes occlusion. This framework, PMON [68], provides tools for assessing the correctness of logics of action and change.

3.1.2 Predicate Completion

Occlusion predicates denote permission to change, not change itself. Since we want to ensure that such predicates are only made true as a result of an action, we can use predicates that explicitly denote change. A predicate changes if and only if the corresponding change predicate is true. An action results in a change if and only if it makes true a condition that was previously false or vice versa. The practice of formally expressing the assumption that all we know about a predicate is the only information that exists (or in logic terms, the sufficient conditions are also necessary) has been known as predicate completion [22]. For our running example, the change of predicates is denoted with the addition of extra formulas, as follows:

$$\neg \text{valid}(0)$$
$$\neg \text{completed}(0)$$

$$\neg \text{valid}(0) \wedge \text{true} \rightarrow \text{change_valid}(0)$$
$$\forall t. \text{change_valid}(t) \equiv (\text{valid}(t) \neq \text{valid}(t+1))$$
$$\forall t. \text{change_completed}(t) \equiv (\text{completed}(t) \neq \text{completed}(t+1))$$

Each of the latter three formulas that were added serves a specific purpose. The first ensures that completing the payment changes the value of the completed predicate. The second formula states that the change predicate for valid is true at time t if and only if the predicate valid has changed value from t to $t+1$. The last formula states the same thing for the predicate completed.

3.1.3 Situation Calculus / Successor State Axioms

The situation calculus [47] is a logic formalism for representing and reasoning about dynamic domains. It was first introduced by McCarthy and Hayes in the same article where the frame problem was introduced, laying the foundations for the situation calculus being used as a solution to it. In 1991, Reiter published an updated version [65] of the situation calculus, which included a solution to the frame problem through the introduction of successor state axioms. Before we present these axioms, let's briefly examine the main features of situation calculus.

The situation calculus uses first-order logic formulas to represent changing scenarios based on the notion of actions, situations and fluents. *Actions* in the situation calculus can be quantified and there are special predicates (labeled Poss) which indicate that an action is executable in a particular situation. *Situations* represent a history of action occurrences. The situation before any actions have been performed is called the initial situation and is denoted as S_0 . The dynamic world is modeled as progressing through a series of situations as a result of various actions being performed. To denote that a situation s is reached when action a is performed, the function $do(a, s)$ is used. The result of this function is a situation. Finally, fluents are statements whose truth value may change. *Fluents* are denoted by special predicates

which take a situation as their final argument. Apart from true/false values, fluents can also take a situation-dependent value.

A situation calculus domain is formalized by a series of axioms. The first set of axioms contains the foundational axioms of the situation calculus. Other axioms include the *action preconditions*, which are formulas that contain Poss predicates and explicitly define when these predicates are true (which conditions must be held for an action to be possible in a situation). *Effect axioms* on the other hand, specify how fluents are affected by the execution of an action. Effects may be conditional, depending on the current state.

To solve the frame problem, Reiter introduced a new set of axioms, called *successor state axioms*, which are based on the fact that the value of a condition after the execution of an action equals to true if and only if one of the following applies: the action makes the condition true, or the condition was previously true and the action does not make it false. Successor state axioms have the following form:

$$Poss(a, s) \rightarrow [F(\vec{x}, do(a, s)) \leftrightarrow \gamma_F^+(\vec{x}, a, s) \vee (F(\vec{x}, s) \wedge \neg\gamma_F^-(\vec{x}, a, s))]$$

The formula γ_F^+ describes the conditions under which action a in situation s makes the fluent F become true in the successor situation $do(a,s)$. Likewise, γ_F^- describes the conditions under which performing action a in situation s makes fluent F false in the successor situation. Given that, the successor state axiom general form can be read as follows: given that it is possible to perform action a in situation s , the fluent F would be true in the resulting situation $do(a,s)$ if and only if performing a in s would make it true, or it is true in situation s and performing a in s would not make it false.

In our running example, if we use *validate_card(t)* and *invalidate_card(t)* as action predicates to denote the credit card has been rendered valid or invalid, respectively, then we can write the following:

$\neg valid(0)$

$\neg completed(0)$

$validate_card(0)$

$\forall t.valid(t + 1) \equiv validate_card(t) \vee (valid(t) \wedge \neg invalidate_card(t))$

The difference in this solution, compared to the others, is that we don't declare a (frame) axiom for every action, but for every condition. Thus, we base the changes on any aspect of the world on the conditions that are part of our knowledge.

3.1.4 Fluent Calculus

The fluent calculus [70] is a variant of the situation calculus. The main difference is that situations in fluent calculus are considered representations of states. The explicit purpose of the fluent calculus, as stated by its author is to solve both the representational and inferential aspects of the frame problem. This essentially means that the fluent calculus not only describes what actions leave unchanged but also how to infer these non-effects.

Another discerning characteristic of the fluent calculus is that states are not represented by predicates, but by terms. A term in first-order logic is a representation of an object or a composition of objects, while a predicate is a representation of a condition that can be evaluated to true or false over a given set of terms. Converting predicates into terms in first-order logic is called reification. The solution of the frame problem in the fluent calculus is based on reification.

To see an example of terms in fluent calculus, let's consider the state, in our example, in which the credit card is valid and the payment is completed. This is represented by the term $valid \circ completed$. This term represents a possible state, not a state that we are currently in, as a term cannot be evaluated to true or false. We need a predicate, such as $state(valid \circ completed, 1)$, in order to specify that the term

represents the current state at time 1.

Having presented the state formulation in the fluent calculus, we can describe the actual solution it offers to the frame problem. The solution is based on specifying the effects of actions by stating how a term (which represents a state) changes when the action is executed. No other change, except what is explicitly stated, is caused by the action. For our running example, we need to specify the state changes caused by the actions of validating the card. These are given by the following formulas:

$$state(s \circ valid, 1) \equiv state(s, 0)$$

$$state(s, 1) \equiv state(s \circ valid, 0)$$

The first formula states the action of validating the credit card at time 0, making the condition true. The second formula states the opposite action, of invalidating the card at time 0, which makes the condition false.

3.1.5 Event Calculus

The event calculus, first introduced by Kowalski and Sergot [43] and extended by Miller and Shanahan [56] with a classical logic axiomatization, is another logic formulation to represent and reason about actions and their effects. In the vocabulary of the event calculus, action occurrences are called events, while actions in general are event types. The basic idea of the event calculus is to state that fluents (predicates whose value changes depending on time) are true at particular points in time if they have been initiated by an event at some earlier point and are not terminated by another event in the meantime. On the other hand, fluents are false if they have been previously terminated and not initiated in the meantime.

To represent fundamental notions such as initiation and termination, a set of special predicates is declared. $Happens(a, t)$ indicates that action a happens at time t , $HoldsAt(f, t)$ means that fluent f is true at time t , while $Initiates(a, f, t)$ and

$Terminates(a, f, t)$ state that if action a happens at time t , it will initiate (or terminate) the fluent f . Also, $Clipped(t_1, f, t_2)$ and $Declipped(t_1, f, t_2)$ respectively state that fluent f is terminated or initiated during the time period from t_1 to t_2 .

While Initiates, Happens and Terminates have domain-dependent axiomatizations, the rest are axiomatized independent of the domain. With regard to the frame problem, the interest is placed on the HoldsAt axiomatization. The event calculus follows the lead of the situation calculus and successor state axioms and states that HoldsAt is true or false based on the following formulas:

$$\begin{aligned} HoldsAt(f, t) &\leftarrow [Happens(a, t_1) \wedge Initiates(a, f, t_1) \wedge (t_1 < t) \wedge \neg Clipped(t_1, f, t)] \\ \neg HoldsAt(f, t) &\leftarrow [Happens(a, t_1) \wedge Terminates(a, f, t_1) \wedge (t_1 < t) \wedge \neg Declipped(t_1, f, t)] \end{aligned}$$

The first formula states that fluent f is true at time t if an action a has taken place in the past, and this action makes the fluent true as an effect, while in the mean time the fluent has not been made false. The second formula states that fluent f is false at time t if an action a has taken place in the past, and this action makes the fluent false as an effect, while in the mean time the fluent has not been made true. Hence, the solution of the frame problem in the event calculus borrows the same principles from the solution in the situation calculus.

3.1.6 Explanation Closure Axioms

In [15], the authors give examples of situations where formal specifications of procedures in the standard precondition/postcondition notation suffer from the effects of the frame problem. They argue that these effects are not ones to be overlooked, as they become more severe when dealing with real-world specifications that contain conditionals, conjunctions and inheritance. Previous efforts in the software specification community commonly embedded the frame axioms in the specification language semantics, gravely compromising a formal specification language with respect to its

capacity to support the formal treatment of specifications due to computational issues, in addition to not giving the freedom to the specifier to state frame axioms in the way he finds most suitable.

Their solution is partially inspired by the work of Reiter in the situation calculus. Their approach adds a second component to the specification process: in addition to specifying postconditions, one should also look for a set of assertions that explain the circumstances under which each predicate or function might change value from one program state to the next. They propose two sets of axioms. The first, effect axioms state what happens when a procedure is executed and are of the following form ($pre_p(x)$ and $post_p(x)$ symbolize the preconditions and postconditions of the procedure $p(x)$):

$$\forall x.[Occur(p(x)) \Rightarrow pre_p(x) \wedge post_p(x)]$$

The frame axioms are replaced by a new set of axioms, explanation closure axioms or change axioms, which view the effects of a procedure from a different perspective. For every predicate R , the specifier includes change axioms to describe under what circumstances R might change truth value. The change axioms have the following form:

$$\begin{aligned} \forall \alpha \forall y [R(y) \wedge \neg R'(y) \wedge Occur(\alpha) \Rightarrow \\ \exists z_1 (\alpha = p_1(z_1)) \vee \\ \exists z_2 (\alpha = p_2(z_2))] \end{aligned}$$

This essentially means that predicate R changes truth value only if one of the procedures p_1, p_2 has executed successfully. Explanation closure axioms offer a state-oriented rather than a procedure-oriented perspective to the frame problem. The specifier doesn't need to assert what each procedure does not change, declaring instead

what procedures could have effected a change to a particular element of the program state. The authors argue that this practice produces short, effective and relatively modular specifications for software, and deals effectively with the problems mentioned before that are not properly addressed by other approaches.

One important advantage is that these axioms are being phrased entirely within Predicate Logic, thus making it unnecessary to build special theorem provers to reason with specifications. This advantage is of particular importance to our work, which is based on the explanation closure axioms solution. We will examine the solution in more detail, focusing on how it can be adapted for the case of Web Service specifications in Chapter 3.

3.2 The Frame Problem in the Web Service domain

While there is a great deal of related literature on the frame problem in the field Artificial Intelligence in general, the frame problem with regard to its effects in Web service specifications has only been addressed in few publications. Most of the publications utilize one of the solutions that we examined in the previous section, while some use formalisms that circumvent the frame problem. We will start with works based on the three different calculi we have mentioned (situation, fluent and event calculi), then we will explore the solution of Linear Logic deductive planning and we will close with the work done by the WSMO working group on or related to the frame problem.

3.2.1 Situation Calculus

Situation calculus has been used as a logic formalism in a variety of research efforts in the Web Service domain. [57] provides a complete framework for simulation, verification and automatic composition of Web Services using situation calculus as its

basis. The main contribution of the work with regard to the frame problem is that the authors provide a translation from OWL-S to the situation calculus by defining a subset of the semantics of OWL-S in situation calculus terms. Atomic processes are translated to actions, preconditions and inputs are written as action precondition formulas and, most importantly, effects and outputs are transformed into successor state axioms. This allows the authors to address the frame problem representationally. Then, Petri Nets are used as a composition model and their graph structure and computational semantics capture the solution to the frame problem, achieving the completion assumption that nothing changes, except when explicitly stated.

Successor state axioms are also used in Golog [48] and its extension ConGolog [33], two high-level programming languages that are based on the situation calculus. Golog programs contain an action theory in situation calculus which is used when searching how to achieve a particular situation which is set as goal. ConGolog is an attempt to address the limitations of Golog in concurrent processes support. Both Golog and ConGolog are used in some works on Web service composition. For example, McIlraith et al. [54] adapted and extended Golog to support generic, self-sufficient programs, user-defined constraints and nondeterministic choices and made similar modifications to a ConGolog interpreter in order to support automatic Web Service composition. The Golog procedures that are created through this framework are based on situation calculus notions, including the successor state axioms.

[46] proposes a means of specifying causal links and causal laws into Web Service composition by integrating Description Logics reasoning and the situation calculus. Causal links define relations between semantic descriptions of interoperating services according to the level of matching between them (from exact match, to subsumption, intersection and so on). Causal laws are conditions that specify what causal links are sufficient in a composition scenario. Based on situation calculus, the authors define causal laws as successor state axioms, using once again, the solution proposed by Reiter for the frame problem.

Finally, [50] presents an approach to reasoning about Web services in a temporal action theory. Web services are described by specifying their interaction protocols in an action theory based on a dynamic, linear-time, temporal logic. Programs are expressed as regular expressions, actions are defined by means of action and precondition laws, social facts are specified by means of commitments whose dynamics are ruled by causal laws, and temporal properties are expressed by means of temporal formulas. epistemic modalities are used in order to distinguish what is known from what is socially unknown. With regard to the frame problem, the authors introduce a completion construction on the epistemic domain description, which defines suitable successor state axioms in the same style and semantics as Reiter's solution.

All efforts that are based on the solution proposed by Reiter, have the disadvantage that they use logic formalisms that are not supported by any current Semantic Web Service frameworks, in direct contrast to classical first-order predicate logic which is universally supported. This means that employing situation calculus as logic formalism in order to solve the frame problem will also mean adapting Semantic Web Service frameworks so that they are based on situation calculus. In addition, as stated in [44], Golog may not be suitable for Web services in some cases, even though it allows for a way to state frame axioms. This is due to the fact that it is not possible to present and reason about multiple copies of literals in world states.

3.2.2 Event and Fluent Calculus

Apart from situation calculus, its variants have also been used, although limitedly, in some Web Service-related research efforts. uses the event calculus for the automated preparation and execution of Web Service compositions. As with many AI planning techniques that we presented in Section 2.5.2, this work views the composition problem as an AI planning problem and tries to solve it using and Abductive Theorem Prover which tries to achieve the goal list by proving one by one the elements which constitute the goal. The authors also present an OWL-S to event calculus translation

scheme, where atomic processes are mapped to events and all common composition schemas are axiomatized using event calculus predicates. This includes the HoldsAt predicates, which, as we mentioned is a formulation that offers a solution to the frame problem.

Fluent calculus is used in [67] in conjunction with semantic formalisms in order to realize automated Web Service composition. Viewing the Web Service composition process once again as an AI planning problem, they use a domain ontology to semantically describe Web Services, then translate it into a fluent calculus knowledge base which is then used from the planner to select services in order to satisfy the goals set. A prototype is also presented, which implements an application scenario from the social event planning domain.

The same critique offered for approaches based on the situation calculus also applies for the case of event and fluent calculi. These logic formalisms are not explicitly supported by any current Semantic Web Service frameworks. However, it should be noted that SWSO is founded upon situation calculus principles, hence the support of any of these calculi by that particular framework could be a realizable task.

3.2.3 Linear Logic Deductive Planning

Apart from solutions that use logic formalisms to express frame axioms in a concise way, there are some solutions that claim to use techniques that solve the frame problem by circumventing it, without the need to include frame axioms. One of these techniques is Linear Deductive Planning [34]. The authors claim that it provides a natural way for representing causal relations between actions and resources, through the notion of a frame, similar to a state vector. A number of fluents are attached to the frame and the effect of an action is described by telling which fluents are changed, presuming that all others remain unchanged.

In his PhD thesis, Küngas [44], formalizes partial deduction in linear logic and uses it as a formal foundation for an automated Web Service composition framework.

Partial deduction derives a more specific (and more efficient) logic program while preserving the meaning of the original one. Composite services are represented in linear logic as follows:

$$(\Gamma_c, \Gamma_v); \Delta_c \vdash ((I \otimes P) \multimap (O \otimes E) \oplus F) \otimes \Delta_r$$

where both Γ_c and Γ_v are sets of extra-logical axioms representing respectively available value-added Web services and core Web services. Δ_c is a multiplicative conjunction of non-functional constraints. Δ_r is a multiplicative conjunction of nonfunctional results. While I represents a set of input parameters of the service, O represents output parameters returned by the service. P and E are respectively multiplicative conjunctions of preconditions and effects, while F is an additive disjunction representing possible exceptions. The formula is read as follows: given a set of available Web services and non-functional attributes, try to find a combination of services that computes O from I as well as changes the world state from P to E . If the execution of the required Web service fails, an exception in F is thrown.

This work is an example of a complete Web Service framework with strong logic foundations, using formal methods to support a multi-agent system capable of searching and composing Web Services. The fact that it is based on a logic formalism that overcomes the frame problem is an added bonus. However, it is hardly a straightforward task to coordinate this framework with existing Semantic Web frameworks such as OWL-S, WSMO or SWSO, allowing it to accept service descriptions that are semantically described in these frameworks.

3.2.4 WSMO Working Group

From the three Semantic Web Service frameworks that we presented in Section 2.3, WSMO is the only one that explicitly mentions the frame problem in its description. The WSMO Working Group relates the Choreography element of the WSMO Service

Interface with the frame problem and also provides a deliverable on the functional description of Web Services which deals with complete and incomplete descriptions and the frame problem.

The Choreography element of the WSMO Service Interface uses a state-based mechanism inspired from the Abstract State Machines (ASM) [14] formalism. An ASM is defined as a set of transition rules of the form If (Condition) then Updates. The updates are of the form $f(t_1, \dots, t_n) := t$. The execution of these rules is understood as updating. These updates correspond only to the given state and only by applying the indicated function f for the indicated parameters and only for them, leaving everything else unchanged. Hence, this formalism essentially avoids the frame problem. However, skipping the frame problem only in a part of a Web Service description is far from being a complete solution.

In [41], the WSMO Working Group offers a thorough survey on the functional description of Web Services. When examining existing logic formalisms, the authors mention that a lot of work has been done on solving the frame problem in the situation calculus and argue that Transaction Logic avoids the frame problem because it uses restrictive update formalisms such as delete and insert, without elaborating further. On applying patterns from software specifications to Web Service descriptions, the authors state that it is necessary to extend assertion languages to support the description of Web Services in terms of possible state transitions in order to deal with the dynamic aspects of the world.

Based on these principles, the authors define a functional description language that encompasses characteristics from both existing functional Web Service descriptions and state-based formalisms. Moving on, they propose some extensions to that language, one of which deals with complete and incomplete descriptions and the frame problem. They use an explicit complete/incomplete notation for postconditions. A postcondition flagged as complete means that no other positive statements (formulas that do not use negation as the top-level operator) besides the postcondition itself

CHAPTER 3. RELATED WORK

are demanded to hold. A postcondition flagged as incomplete means that no frame axioms are expressed, so one should be careful when handling these postconditions. This is also an example of avoiding the frame problem rather than attempting to solve it.

Chapter 4

The Frame Problem in Web Service Specifications

Having covered both the background theory behind this work and any research efforts closely related to it, we now move on to the core of this thesis, which entails our efforts to identify the existence of the frame problem in Web Service specifications and address the problem in a way that is most suited to the Web Service domain. In this chapter, a motivating example will be presented which attempts to illustrate the issues raised by the frame problem when devising Web Service specifications. On the second half of this chapter, we will present our solution schema in detail.

4.1 A Motivating Example

For our example, we will draw inspiration from the domain of online shopping. Let's consider the following simple scenario:

A client enters the web site of an online shop and searches for items that may interest him. The online shop offers two additional functionalities to assist him with his search. A wish list contains all the items he has previously expressed explicit interest for purchasing, while recommenda-

tions are offered based on his previous purchases and other user-related information. If an item interests him, he puts it in his (virtual) shopping cart. When he is finished, he proceeds to checkout, where he reviews his credit card and delivery address information and confirms his order. Once his order is confirmed, the system executes the order.

A general view of the tasks described in this scenario can be seen in Figure 4.1.

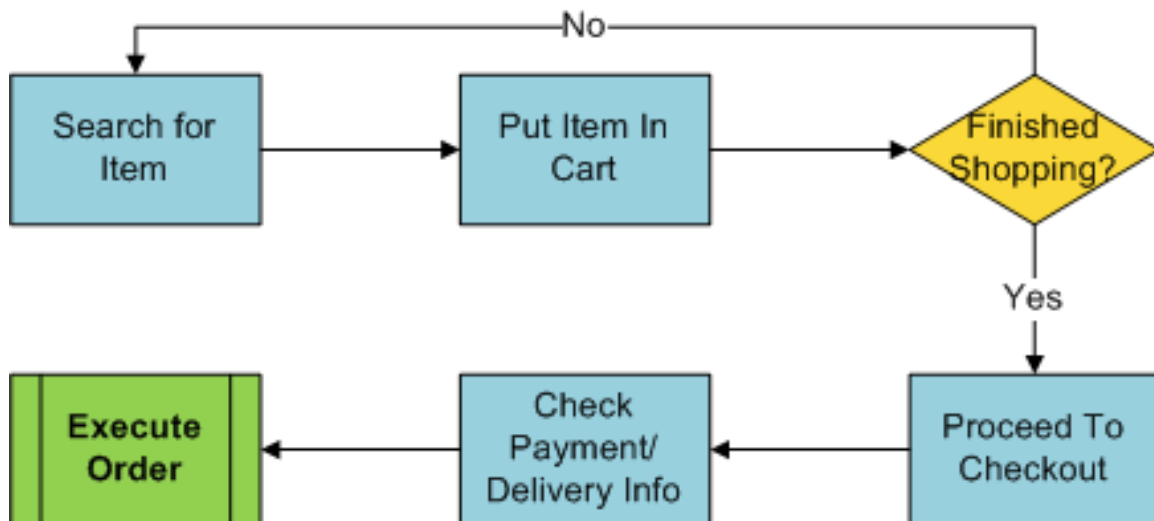


Figure 4.1: Example Scenario

The execution of the order requires from the system to do the following actions:

- The total cost of the items purchased is withdrawn from the client's credit card account
- The system initiates the actions necessary for the items to be packaged and sent to the client
- For each item purchased, the item is removed from the client's wish list (if it is contained)

- For each item purchased, the most closely associated one based on various criteria is selected and put into a list containing the recommendations for the client

These steps are shown in Figure 4.2. Let's assume that we want to use Web Services to perform these steps, with the exception of the second one that involves initiating packaging and delivery of the package. We will attempt to devise specifications for these services, in terms of their preconditions and postconditions.

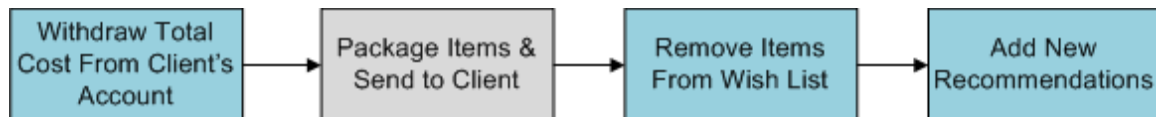


Figure 4.2: Example Scenario: Order Execution

4.1.1 Atomic Web Service Specifications

4.1.1.1 Money Withdrawal Service

Let's begin with the Web Service that will deal with the first step of the order execution. A service that is tasked to withdraw an amount of money from a bank account associated with a credit card can go through with the withdrawal if that amount of money is available. Thus, a precondition that should hold before the service begins execution is that the balance of the client's account should at least equal to the amount of money about to be withdrawn. Moving on, let's say that the bank our service works with imposes a limit for the amount of money an account holder can withdraw in a day. If the limit has been reached, the account is banned for the rest of the day. Based on that, we should add another precondition that makes sure that the account is not banned when the service is executed. The Web Service should then check, after withdrawing the money, if the daily limit has been reached. If it has, the account must be banned. If the total amount withdrawn for the day is

nearing the daily limit, a warning should be issued to the cardholder, otherwise it shouldn't. These can be expressed with postconditions that state how the account status is modified, depending on the total money withdrawn and the daily limit.

Stating these conditions in natural language is fairly easy. We need, however, to state them in a machine-understandable way. For the purposes of this chapter and to make sure that our claims are independent of any specification language, we will use the neutral notation of first-order predicate logic. A possible predicate logic encoding of the preconditions and postconditions of the money withdrawal service, as stated in the previous paragraph is shown in Table 4.1.

<p>Preconditions</p>
<p>$Valid(creditCard, account) \wedge$ $balance(account) \geq A$</p>
<p>Postconditions</p>
<p>$balance'(account) = balance(account) - A \wedge$ $withdrawalTotal'(account) \geq dailyLimit(account)$ $\Rightarrow \neg Valid'(creditCard, account)$</p>
<p>$withdrawalTotal'(account) < dailyLimit(account) \wedge$ $dailyLimit(account) - withdrawalTotal'(account) \leq W$ $\Rightarrow Warn'(creditCard, account) \wedge Valid'(creditCard, account)$</p>
<p>$withdrawalTotal'(account) < dailyLimit(account) \wedge$ $dailyLimit(account) - withdrawalTotal'(account) > W$ $\Rightarrow \neg Warn'(creditCard, account) \wedge Valid'(creditCard, account)$</p>

Table 4.1: Pre/Postconditions for the Money Withdrawal Service

The unprimed/primed notation is used to refer to predicates and functions calcu-

lated immediately before and after the execution of the service, respectively. We use the variables A to denote the amount of money that will be withdrawn and W to denote the maximum difference between the total amount withdrawn and the daily limit before a warning is issued. Variables *creditCard* and *account* are self-explanatory. As far as predicates¹ are concerned, $Valid(x,y)$ is true when the credit card x associated with the account y is valid (the daily limit has not been reached) while $Warn(x,y)$ is true when the credit card x associated with the account y is flagged as warned for the day because the daily withdrawal limit is nearing. The functions that we use are²: $balance(x)$ which returns the balance of account x , $withdrawalTotal(x)$ which returns the total amount of money withdrawn from account x during the current day, and $dailyLimit(x)$ which returns the daily withdrawal limit of account x .

4.1.1.2 Wish List Update Service

The second service that we will attempt to describe is the one that updates the client's wish list. This service will be executed once for each item that is purchased by the client. A precondition for its execution should be to ensure that the item is indeed part of the wish list, or else executing it is only a waste of time. In addition, we need to make sure that the order in which the item belongs has been flagged as completed, meaning that the money has been withdrawn and the package has been sent. When the service is executed successfully, we should expect that the item is no longer included in the wish list. Encoded in predicate logic, the preconditions and postconditions of the wish list update service are shown in Table 4.2.

Two predicates are introduced in this specification. $Completed(x)$ is true when the order x is flagged as completed. $Included(x,y)$ is true when the wish list x contains the item y .

¹Predicate symbols used in formulas in the example will begin with an upper case letter and function symbols with lower case ones.

²For the sake of readability, we assume that all functions used in first-order logic statements in this section are always defined.

Preconditions
<i>Completed(order) ∧</i>
<i>Included(clientWishList, item)</i>
Postconditions
<i>¬Included'(clientWishList, item)</i>

Table 4.2: Pre/Postconditions for the Wish List Update Service

4.1.1.3 Recommendations Update Service

The recommendations update service is similar to the wish list update service. This service will, too, be executed once for each item that is purchased by the client, in order to find the most closely associated item to the one purchased and include it in the personalized recommendations for the client. We assume that items that are recommended to a client are kept in a list similar to the wish list. A precondition for its execution should be to ensure that the selected item is not already recommended to the user, in other words not already part of the recommendations list. As in the wish list update service, we also need to check that the order in which the item belongs has been flagged as completed. When the service is executed successfully, the selected item must be part of the recommendations list. These can be expressed in predicate logic as shown in Table 4.3.

Preconditions
<i>Completed(order) ∧</i>
<i>¬Included(clientRecoms, associatedItem)</i>
Postconditions
<i>Included'(clientRecoms, associatedItem)</i>

Table 4.3: Pre/Postconditions for the Recommendations Update Service

We use the same predicates that were introduced in the specification of the wish

list update service. This is very important, as we will see later in this chapter, as it causes the effects of the frame problem to aggravate.

4.1.2 Writing Frame Axioms

After having expressed the preconditions and postconditions for the three Web Services of the example, we pose a simple yet extremely important question: Are these specifications complete, are we covered for every possible case with these conditions? The answer is negative, since we need to specify (using frame axioms) if the changes that are explicitly mentioned in the conditions are the only ones that take place. We argue that writing the frame axioms for this example and in general, albeit necessary, is not straightforward and problem-free and will attempt to prove our argument in the rest of this section.

4.1.2.1 The necessity of Frame Axioms

Suppose that we want to use the wish list update service and we want to make sure that after completing execution, no other lists except the one for which we executed the service will be affected. If we want to prove that this fact is true or false, the only way to do it is based on the postconditions of the service. However, no proof can be made relying solely on the conditions in Table 4.2. We need to add frame axioms that explicitly state that nothing else changes, except what is mentioned in the existing postconditions. We need to add a single frame axiom for each predicate or function that is part of our knowledge. What constitutes our knowledge may vary from one case to another. If we see the wish list update service as the sole service that we are aware of, then we only need to add frame axioms for the predicates *Included* and *Completed*. These axioms are shown in Table 4.4 in bold, to distinguish them from the preexisting postconditions.

The frame axioms that we added allow us to prove that no lists that shouldn't be changed are affected by the execution of the service and the orders that are completed

Preconditions
$Completed(order) \wedge$
$Included(clientWishList, item)$
Postconditions
$\neg included'(clientWishList, item) \wedge$
$\forall x, y [x \neq clientWishList \vee y \neq item \Rightarrow Included(x, y) \equiv Included'(x, y)] \wedge$
$\forall x, y [Completed(x, y) \equiv Completed'(x, y)]$

Table 4.4: Wish List Update Service Specification w/ Frame Axioms

remain flagged as completed. If we consider the wish list service in relation to the other two services, then we may need to add frame axioms for predicates and functions that are contained in the specifications of the other two services. For example, if we need to prove that executing the wish list service doesn't change anything related to credit card accounts, then we may need to include frame axioms for predicates and functions such as *balance*, *Valid* and *Warn* in our specification.

Dealing with the frame axioms for the recommendations update service is similar, since both services include the same predicates in their specifications. Table 4.5 shows the complete specification for that service, including the frame axioms.

Preconditions
$Completed(order) \wedge$
$\neg Included(clientRecoms, associatedItem)$
Postconditions
$Included'(clientRecoms, associatedItem) \wedge$
$\forall x, y [x \neq clientRecoms \vee y \neq associatedItem \Rightarrow Included(x, y) \equiv Included'(x, y)] \wedge$
$\forall x, y [Completed(x, y) \equiv Completed'(x, y)]$

Table 4.5: Recommendations Update Service Specification w/ Frame Axioms

Frame axioms are absolutely necessary for a specification to be considered complete and useful when attempting to formally prove properties of the specification. However, in some cases one can explicitly state that a specification, in its current form, is declared complete, considered true and known only what is already stated. This approach, followed for example in [41] avoids the frame problem, by employing a concept similar to negation as failure, since anything that doesn't derive from the specification explicitly declared as being complete is considered false. In this case, frame axioms are not necessary, since the frame problem is circumvented by presuming that an incomplete description as complete. However, this may cause problems when using the specification in conjunction with other specifications that don't employ this kind of presumption. Specifications arbitrarily flagged as complete will only remain consistent as long as the same practice is applied to all other related specifications.

4.1.2.2 Frame Axioms in more complex specifications

Expressing frame axioms in the previous two cases is relatively straightforward. However, Web Service specifications are usually more complex than containing just one postcondition. The money withdrawal service is a good example of that. As we described above, the postconditions of that service deal with 3 different cases, depending on how close the total money withdrawn is to the daily withdrawal limit. Each conditional case has different effects than the others, hence a separate set of frame axioms is needed for each one of them. These are shown in Table 4.6.

We have included a set of frame axioms for each branch of the conditional, where we state that the predicate that changes is the only one that does and only for the specific values of arguments that are stated. In addition to those, we include two frame axioms that state that `withdrawalTotal` changes only for the account at hand, while `dailyLimit` doesn't change for any value of its argument. Also, we have two frame axioms that state that `Valid` doesn't change at all, while `balance` changes only for the client's account.

Preconditions

$$\begin{aligned} &Valid(creditCard, account) \wedge \\ &balance(account) \geq A \end{aligned}$$

Postconditions

$$\begin{aligned} &balance'(account) = balance(account) - A \\ &\forall x[x \neq account \Rightarrow balance(x) = balance'(x)] \\ &withdrawalTotal'(account) \geq dailyLimit(account) \Rightarrow \neg Valid'(creditCard, account) \wedge \\ &\quad \forall x, y[x \neq creditCard \vee y \neq account \Rightarrow Valid(x, y) \equiv Valid'(x, y)] \\ &\quad \forall x, y[Warn(x, y) \equiv Warn'(x, y)] \\ \\ &withdrawalTotal'(account) < dailyLimit(account) \wedge \\ &dailyLimit(account) - withdrawalTotal'(account) \leq W \Rightarrow Warn'(creditCard, account) \\ &\quad \forall x, y[x \neq creditCard \vee y \neq account \Rightarrow Warn(x, y) \equiv Warn'(x, y)] \\ &\quad \forall x, y[Valid(x, y) \equiv Valid'(x, y)] \\ &withdrawalTotal'(account) < dailyLimit(account) \wedge \\ &dailyLimit(account) - withdrawalTotal'(account) > W \Rightarrow \neg Warn'(creditCard, account) \\ &\quad \forall x, y[x \neq creditCard \vee y \neq account \Rightarrow Warn(x, y) \equiv Warn'(x, y)] \\ &\quad \forall x, y[Valid(x, y) \equiv Valid'(x, y)] \end{aligned}$$

Table 4.6: Money Withdrawal Service Specification w/ Frame Axioms

This example aimed to show that when Web Service specifications become more complex, e.g. when they include conditionals with multiple branches, then the frame problem becomes more difficult to solve, since stating frame axioms becomes a complicated task with many different parameters that need to be taken into consideration. Moreover, the resulting specification, while being complete, is also rather lengthy and computing formal proofs based on it is a more difficult and error-prone task. In the next and final part of the motivating example, we will deal with the case of composite services, where stating frame axioms becomes even more complicated and also tends

to lead to inconsistent specifications.

4.1.3 Service Composition and the Frame Problem

The need for Web service composition is a natural consequence of the fact that a given task may not be executed completely by a single service alone. An important issue when composing services, as we mentioned in 2.5 is to produce a correct specification for the composite service, based on the specifications of each service that takes part in the composition schema. The composite service specification depends mainly on the services the composite service is constructed from and the interaction between these services. In [4], a set of composition rules is defined for many different composition schemas, including sequential, parallel, iterational and conditional composition among others. These sets of rules describe the conditions under which a composition schema is possible and valid by determining correlations between the constraints in the individual Web service specifications, essentially creating the specification for the composite Web service. We will use these rules in the design of our composition schema.

Previously, we argued that the frame problem exists and is a cause of certain problems when preparing specifications for individual Web services. In order to prepare specifications for a composite Web service, it is necessary to work based on the specifications of the services that are the building blocks of the composition and combine them in a certain way that makes sense with regard to the composition schema. Thus, it should be expected that the frame problem will once again stand in the way.

The three services that we created offer functionalities that are necessary for the completion of the order execution process, as it is shown in Figure 4.2. Based on this figure, we can understand that the money withdrawal service has to be executed first, followed by a repeated execution of a parallel composition of the wish list update and recommendation update services for each one of the purchased items. This service composition schema along with the composition rules that apply to it is shown in

Figure 4.4 at the end of this chapter. A UML Activity Diagram for the composite service is shown in Figure 4.3.

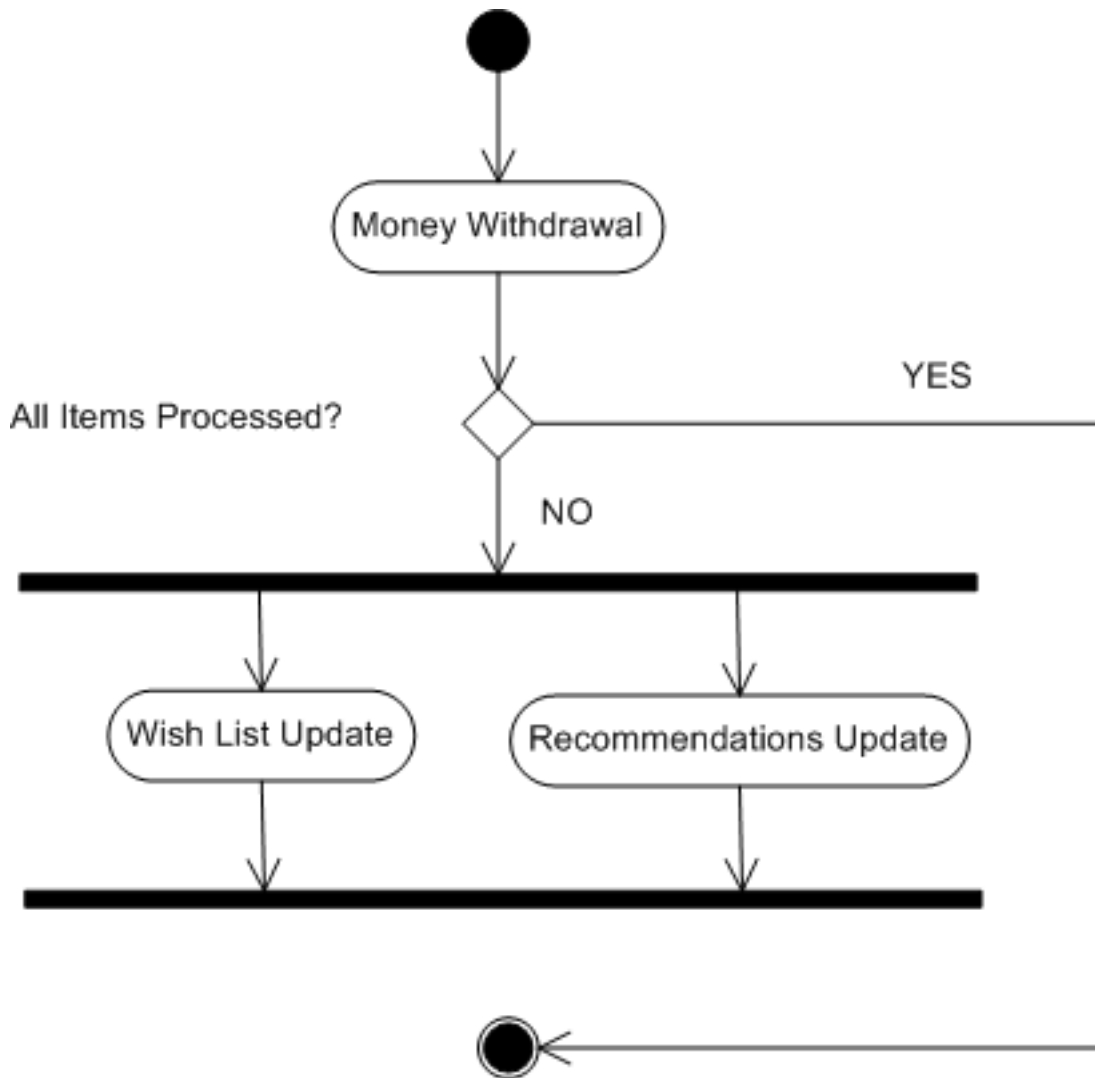


Figure 4.3: Order Execution: UML Activity Diagram

As we can see in the figure, starting from the lowest level we have the parallel composition of the wish list and recommendations update services. This block is iterated N times, where N is the number of purchased items. This block can also be viewed as a black box service, having as preconditions the conjunction of the pre-

CHAPTER 4. THE FRAME PROBLEM IN WEB SERVICE SPECIFICATIONS

conditions of the services it is comprised of (symbolized as $PRE_1 \wedge PRE_2$) and as postconditions the conjunction of the postconditions of the services (symbolized as $POST_1 \wedge POST_2$). The money withdrawal service is executed sequentially with the iterations of the parallel composition, preceding them. The complete order execution service, if viewed as a black box as well, has as preconditions the preconditions of the money withdrawal service (symbolized as PRE_3) and as postconditions the conjunction of the postconditions of the last iteration of the parallel composition (which can be symbolized as $POST_1^N \wedge POST_2^N$)

As far as composition rules are concerned, in order to be able to execute the iterational composition, we must ensure in each step that the rule $POST_1^i \wedge POST_2^i \equiv PRE_1^{i+1} \wedge PRE_2^{i+1}$ evaluates to true. A similar rule must be satisfied for the sequential composition of the money withdrawal service and the iterational block. The rule is written as $POST_3 \equiv PRE_1^1 \wedge PRE_2^1$.

In order to make sure that the composition schema is valid, we need to start from the lowest building blocks and try to create a specification for them. The lowest building block of our example is the parallel composition of the wish list update and recommendations update services. As we mentioned above, in order to create the specification for that service, we need to perform a conjunction of the specifications of the two component services. The resulting specification is shown in Table 4.7.

However, as we can see, the resulting specification is inconsistent, since one of the postconditions states that the predicate *Included* remains unchanged for any pair of parameters except for *clientWishList* and *item*, while another states that the predicate *Included* remains unchanged for any pair of parameters except for *clientRecoms* and *associatedItem*. The conflicting postconditions are shown in *italic*. The inconsistency is a direct result of including frame axioms in our specification since we are forced to explicitly state what does not change in each separate specification, which will eventually lead to inconsistencies when we attempt to conjoin Web services that use the same predicates in their specifications.

Preconditions

$$\begin{aligned}
 & \text{Completed}(\text{order}) \wedge \\
 & \text{Included}(\text{clientWishList}, \text{item}) \\
 & \neg \text{Included}(\text{clientRecoms}, \text{associatedItem})
 \end{aligned}$$

Postconditions

$$\begin{aligned}
 & \neg \text{Included}'(\text{clientWishList}, \text{item}) \wedge \\
 & \text{Included}'(\text{clientRecoms}, \text{associatedItem}) \wedge \\
 & \forall x, y [x \neq \text{clientWishList} \vee y \neq \text{item} \Rightarrow \text{Included}(x, y) \equiv \text{Included}'(x, y)] \wedge \\
 & \forall x, y [x \neq \text{clientRecoms} \vee y \neq \text{associatedItem} \Rightarrow \text{Included}(x, y) \equiv \text{Included}'(x, y)] \wedge \\
 & \forall x, y [\text{Completed}(x, y) \equiv \text{Completed}'(x, y)]
 \end{aligned}$$

Table 4.7: Composite Update Service w/ Frame Axioms

If we attempt to continue implementing the composition schema, having as basis this inconsistent specification, then all other specifications will be inconsistent. Even if we view the complete composite service that executes an order as a black box, considering only the preconditions of the service executed first and the postconditions of the service executed last, we will once again have an inconsistent specification, since the service executed last is a parallel composition of the wish list update and recommendations update services.

It should be explicitly stated that the issues caused in this particular example due to the frame problem may not be attributed to some special characteristics of the example or due to a naive approach in the composition of the two services. The inclusion of frame axioms in atomic service specifications makes them more complex and lengthy and if they are combined using conjunctions to form a composite service specification, it is most likely that one of these conjunctions will lead to inconsistencies.

In general, the frame problem is bound to appear and possibly cause inconsistencies in composite service specifications due to two main reasons. First of all,

conjunctions are heavily used in almost all composition schemas, whether it is sequential, parallel, iterational, or conditional composition and so on, as it has been examined in [4]. It is also worth noting that attempting to introduce inheritance in service specifications will also lead to the frame problem, as mentioned in [29]. Second, it is highly possible that services being composed together will deal with the same knowledge, hence having specifications containing some common predicates, associated however with contradicting frame axioms. The more frame axioms are included, the more possible it is to conjunct statements that are not consistent with each other.

The exclusive use of predicate logic to express preconditions and postconditions in the motivating example may lead to the rapid conclusion that predicate logic is the reason behind the inadequate representation of frame axioms and the effects it has. However, other languages that are used to express preconditions and postconditions for Web service specifications such as SWRL, DRS [53] or KIF [31] don't have any special constructs in order to represent the condition that nothing else changes except what is explicitly stated, so there is no support for stating frame axioms in a concise and short way. In the next section, we will explore a solution approach to deal with the frame problem in Web service specifications.

4.2 Addressing the Frame Problem

Through the example presented in the previous section, it is apparent that attempting to completely state all frame axioms when devising Web service specifications is problematic, especially when conjoining such specifications to create composite services. The frame axioms, as expressed in the example, offer a procedure-oriented perspective to the frame problem, explicitly asserting what predicates each procedure does not change in addition to those it changes. In [15], the authors identified this fact as the source of the frame problem and aimed to replace the procedure-oriented

perspective with a state-oriented one, which we will explore in this section.

Instead of declaring what predicates don't change in each Web service specification, we can reverse our viewpoint and declare, for each element of the service specifications we are creating, which services may result in changing them. Thus, we don't aim to write a set of frame axioms for each individual Web service specification, but we create assertions that explain the circumstances under which each predicate or function might be modified from one state to another. These assertions, called explanation closure axioms or change axioms in [15], provide a state-oriented perspective to specifications.

Before applying the solution to the motivating example, we will try to justify our decision to select the explanation closure axioms solution from the variety of solutions that have been proposed for the frame problem. First of all, this solution was created, having procedure specifications in mind. Procedure specifications share a lot of common features with Web Service specifications. In fact, a Web Service execution can be viewed as the execution of a procedure which is available over the network at a specified endpoint, the source code of which is unknown to us. In both cases, we have inputs that must be supplied and outputs that are expected, preconditions that must be satisfied prior to execution and postconditions that must hold for an execution to be considered successful. Of course, Web Services are not merely a set of Remote Procedure Calls (RPC), as we have definitely proven throughout Chapter 2. However, when it comes to specifications, there is a common ground and that can be exploited in our case.

Another characteristic of the explanation closure axioms solution that contributed greatly in our decision to apply this solution instead of another, is the fact that the solution schema is expressed in first-order predicate logic. This is very important if we want our solution to be applicable in a Semantic Web Service framework like the ones presented in Section 2.4. Predicate logic has straightforward semantics that are widely known and this is really helpful in our case. As we will witness in the

rest of this section, due to the fact that the solution schema is initially expressed in first-order predicate logic allows us to express it in a relatively easy way in WSMO, SWSO and OWL-S service descriptions.

4.2.1 Expressing Change Axioms

To be able to express the change axioms, a simple extension to the first-order predicate logic is proposed, that adds a special predicate symbol, named *Occur* and a special variable symbol named α . The semantics for these two additions are simple. Variable α is used to refer to services taking part in the specification. $Occur(\alpha)$ is a predicate of arity 1 that is true if and only if the service denoted by the variable α has executed successfully. It is possible to negate *Occur* using \neg in order to express the opposite semantics. There is no apparent use for quantifiers in the case of *Occur*.

Let's revisit the specification of the money withdrawal service in 4.6. Recall that we had a conditional specification with 3 different cases depending on whether the client account would be flagged as warned or not, or invalidated for the day. The frame axioms can be replaced by change axioms using the *Occur* predicate as shown in 4.8.

The change axioms that were added state that under no circumstances do all predicates and functions included in the specifications change except for the following cases:

- Function *balance* changes only if the Money Withdrawal service has executed, for the account x and cost A
- Predicate *Valid* changes only if the Money Withdrawal service has executed and the total money withdrawn is greater than the daily limit
- Predicate *Warn* changes only if the Money Withdrawal service has executed and the difference between the daily limit and the total money withdrawn is less than or equal to W .

Preconditions

$$\text{Valid}(\text{creditCard}, \text{account}) \wedge \\ \text{balance}(\text{account}) \geq A$$

Postconditions

$$\text{balance}'(\text{account}) = \text{balance}(\text{account}) - A \\ \forall \alpha, \forall x[(\text{balance}(x) \neq \text{balance}'(x)) \wedge \text{Occur}(\alpha) \Rightarrow \alpha = \text{MoneyWithdrawal}(x, A)] \\ \forall \alpha, \forall x, y[\text{Valid}(x, y) \neq \text{Valid}'(x, y) \wedge \text{Occur}(\alpha) \Rightarrow \\ \alpha = \text{MoneyWithdrawal}(x, A) \wedge \text{withdrawalTotal}'(x, y) \geq \text{dailyLimit}(x)] \\ \forall \alpha, \forall x, y[\text{Warn}(x, y) \neq \text{Warn}'(x, y) \wedge \text{Occur}(\alpha) \Rightarrow \\ \alpha = \text{MoneyWithdrawal}(x, A) \wedge \text{dailyLimit}(x) - \text{withdrawalTotal}'(x, y) \leq W]$$

Table 4.8: Money Withdrawal Service Specification w/ Change Axioms

The resulting specification is much more concise and clear. The specifications for the other two services are similarly modified, as shown in Tables 4.9 and 4.10.

Preconditions

$$\text{Completed}(\text{order}) \wedge \\ \text{Included}(\text{clientWishList}, \text{item})$$

Postconditions

$$\forall \alpha, \forall x, y[\text{Completed}(x, y) \neq \text{Completed}'(x, y) \wedge \text{Occur}(\alpha) \Rightarrow \text{false}] \\ \forall \alpha, \forall x, y[\text{Included}(x, y) \neq \text{Included}'(x, y) \wedge \text{Occur}(\alpha) \Rightarrow \alpha = \text{WishListUpdate}(x, y)]$$

Table 4.9: Wish List Update Service Specification w/ Change Axioms

The change axioms that we added for both services express that under no circumstances does the predicate *Completed* change, while the predicate *Included* changes only if the corresponding service (and no other) has executed successfully. No other changes whatsoever agree with the specifications. In addition, we should note that by including change axioms, some postconditions become redundant and can be removed,

Preconditions

$$\begin{aligned}
 & \text{Completed}(\text{order}) \wedge \\
 & \neg \text{Included}(\text{clientRecoms}, \text{item})
 \end{aligned}$$

Postconditions

$$\begin{aligned}
 & \forall \alpha, \forall x, y [\text{Completed}(x, y) \not\equiv \text{Completed}'(x, y) \wedge \text{Occur}(\alpha) \Rightarrow \text{false}] \\
 & \forall \alpha, \forall x, y [\text{Included}(x, y) \not\equiv \text{Included}'(x, y) \wedge \text{Occur}(\alpha) \Rightarrow \alpha = \text{RecomsUpdate}(x, y)]
 \end{aligned}$$

Table 4.10: Recommendations Update Service Specification w/ Change Axioms

since the knowledge expressed by each precondition is captured by the equivalent change axiom.

The most important consequence of replacing frame axioms with change axioms appears in the case of service composition. Composing the service specifications of the wish list update and recommendations update services now will not lead to inconsistencies due to the nature of change axioms and the resulting change axioms can easily be produced using disjunction to express that some predicates or functions can be modified by more than one services. The new specification for the composite service is shown in Table 4.11.

Preconditions

$$\begin{aligned}
 & \text{Completed}(\text{order}) \wedge \\
 & \text{Included}(\text{clientWishList}, \text{item}) \\
 & \neg \text{Included}(\text{clientRecoms}, \text{item})
 \end{aligned}$$

Postconditions

$$\begin{aligned}
 & \forall \alpha, \forall x, y [\text{Completed}(x, y) \not\equiv \text{Completed}'(x, y) \wedge \text{Occur}(\alpha) \Rightarrow \text{false}] \\
 & \forall \alpha, \forall x, y [\text{Included}(x, y) \not\equiv \text{Included}'(x, y) \wedge \text{Occur}(\alpha) \Rightarrow \\
 & \quad \alpha = \text{WishListUpdate}(x, y) \vee \alpha = \text{RecomsUpdate}(x, y)]
 \end{aligned}$$

Table 4.11: Composite Update Service Specification w/ Change Axioms

The new specification for the composite service essentially states that for the service to begin execution the known preconditions must be met and any change to the predicate included signifies that a successful execution of either wish list update service or recommendations update service has taken place. This statement includes the postconditions of the two individual services, since the successful execution of them is tied to a change to the value of the predicate included.

The aforementioned solution to the frame problem not only deals with the inconsistencies caused by the conventional statement of frame axioms, but also leads to more concise specifications. However, it should be noted that whenever a new service is added to our closed system, all change axioms that are affected must be modified to include the new knowledge. This may sound overwhelming, but as we have glimpsed in the example, the addition of a new service involves adding a disjunction to the change axioms of the predicates it changes. This is certainly more straightforward and cleaner than the previous practice of adding a postcondition to each service specification for every new predicate encountered.

4.2.2 Change Axioms in Web Service Languages

The solution proposed in the previous subsection is, at its basis, a reformulation of existing first-order logic specifications to ones that use the special predicate *Occur*. Most semantic Web service specification frameworks support languages that include first-order logic notations. This allows us to examine how one can express change axioms in the most important Semantic Web service ontologies and frameworks. We will begin with WSMO and the language it contains, WSML, then we will move on to SWSO and SWSL. We will finish this chapter with OWL-S and an extension of SWRL to fully support first-order logic, SWRL-FOL [63].

When describing services in the WSMO framework, the language used for expressing rules is WSML. In WSML, *Occur* can be expressed as a concept carrying the semantics of a service execution. The variable α can then be expressed as a WSML

variable. For example, the change axioms included in the composite service specification that was presented before, can be written as a set of WSML rules as shown in Table 4.12.

$$\begin{aligned}
 & \textit{forall } ?a, ?o1, ?i1 \\
 & (\textit{completed}(?o1, ?i1) \textit{ and neg } (\textit{completedpr}(?o1, ?i1) \textit{ and Occur}(?a) \textit{ implies false}) \\
 & \textit{forall } ?a, ?x1, ?i1 \\
 & (\textit{included}(?x1, ?i1) \textit{ and neg } (\textit{includedpr}(?x1, ?i1) \textit{ and Occur}(?a) \textit{ implies} \\
 & (?a = \textit{updWishList} \textit{ or } ?a = \textit{updRecList}))
 \end{aligned}$$

Table 4.12: Composite Update Service Change Axioms In WSML

SWSO includes SWSL as its de-facto rule language. SWSL, as mentioned in 2, contains two sub-languages: SWSL-FOL and SWSL-Rules. SWSL-FOL, a full first-order logic language, is what we need in order to express the change axioms. These are shown in 4.13.

$$\begin{aligned}
 & \textit{forall } ?a, ?o1, ?i1 \\
 & (\textit{false} \textit{ :- completed}(?o1, ?i1) \textit{ and neg } (\textit{completedpr}(?o1, ?i1) \textit{ and Occur}(?a)) \\
 & \textit{forall } ?a, ?x1, ?i1 ((?a = \textit{updWishList} \textit{ or } ?a = \textit{updRecList}) \textit{ :-} \\
 & \textit{included}(?x1, ?i1) \textit{ and neg } (\textit{includedpr}(?x1, ?i1) \textit{ and Occur}(?a))
 \end{aligned}$$

Table 4.13: Composite Update Service Change Axioms In SWSL-FOL

OWL-S is the last Semantic Web framework to examine, because it will be the focus of the next chapter. In OWL-S, logic formulas and rules may be expressed using SWRL. For first-order logic formulas, an extension to SWRL, called SWRL-FOL [63] has been proposed. In SWRL-FOL, Occur can be expressed as a unary predicate which has the meaning that its argument belongs to a certain OWL class. The variable α can then be expressed as an individual variable. The change axioms

CHAPTER 4. THE FRAME PROBLEM IN WEB SERVICE SPECIFICATIONS

included in the composite service specification that was presented in this section, can be written as shown in Table 4.14.

$\textit{forall} \text{ (I-variable(a) I-variable(o1) I-variable(i1) \textit{implies}$ $\text{ (Antecedent (completed(I-variable(o1) I-variable(i1))$ $\text{and neg (completedpr((I-variable(o1) I-variable(i1)) and Occur(I-variable(a))$ Consequent())
$\textit{forall} \text{ (I-variable(a) I-variable(x1) I-variable(i1) \textit{implies}$ $\text{ (Antecedent (included(I-variable(x1) I-variable(i1))$ $\text{and neg (includedpr((I-variable(x1) I-variable(i1)) and Occur(I-variable(a))$ $\text{Consequent(I-variable(a) = updWishList or I-variable(a) = updRecList)}$

Table 4.14: Composite Update Service Change Axioms In SWSL-FOL

It should be noted that these are not considered complete Web service descriptions in any of the frameworks mentioned (OWL-S, WSMO or SWSO). They are simply examples of how change axioms can be expressed, without dealing with the underlying ontologies and any other parts of the Web service description.

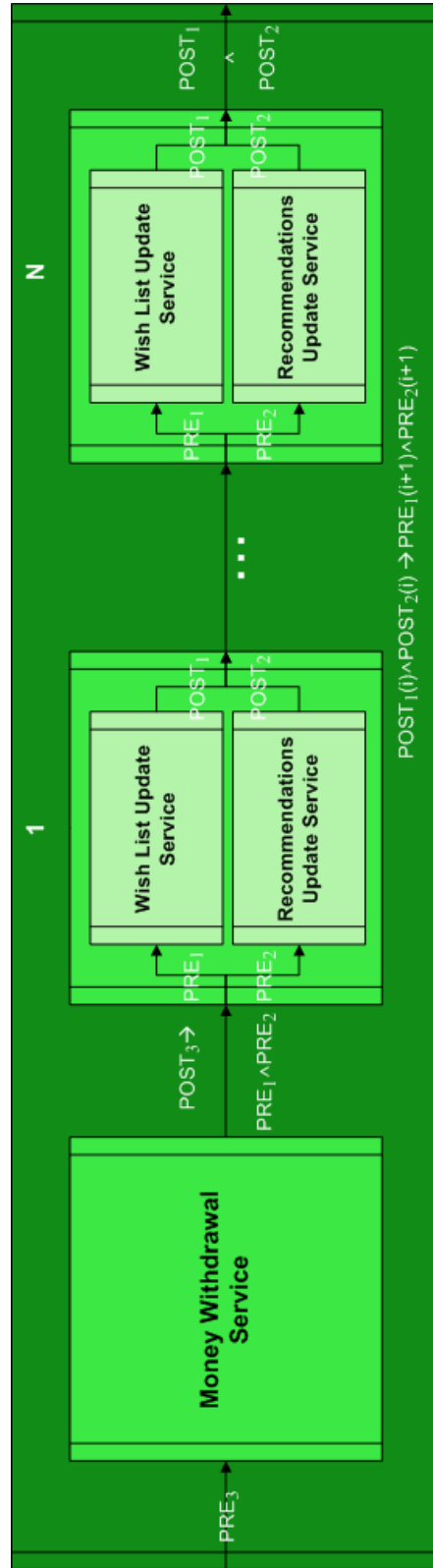


Figure 4.4: Order Execution: Composite Service

CHAPTER 4. THE FRAME PROBLEM IN WEB SERVICE SPECIFICATIONS

Chapter 5

Automatic Production of Change

Axioms

Having defined our proposed solution for the frame problem and shown how it can be expressed in various Semantic Web service rule languages, we turn our focus on sketching an algorithm for automatically producing change axioms, given a service description using the precondition/postcondition notation. A complete set of change axioms should contain one axiom for every predicate or function contained in all the preconditions and postconditions stated in the description. Thus, the main goal for an algorithm that automatically produces change axioms is to find every distinct predicate and function included in the specification. We will gradually build the algorithm, starting from atomic service specifications, moving on to composite service specifications and finally adapting the algorithm to work with OWL-S service descriptions. For simplicity, we will only explicitly mention predicates in the description of the algorithm, but the same steps apply for functions as well.

5.1 Atomic Service Specifications

In atomic Web Services, there is only one set of preconditions and one set of postconditions that we need to examine. We need to check the postconditions first and compile a list of all predicates that are used in the conditions. For each predicate, we first need to check if a corresponding change axiom already exists. If a change axiom exists, we don't need to do anything, since there is already a clause that states if the corresponding predicate or function is changed or not by the execution of the service.

If a change axiom doesn't exist, we need to add one that reflects the change in the value of the predicate. If the predicate remains unchanged, the right-hand part of the clause should be false, otherwise it should be of the form $\alpha = \langle \text{service-name} \rangle$. A possible encoding of these steps in an algorithm written in pseudo-code is shown in Table 5.1.

```
For each predicate {  
  If(corresponding frame axiom exists)  
    do nothing  
  else{  
    If(predicate remains unchanged)  
      add change axiom with false as consequent  
    else  
      add change axiom with  $\alpha = \langle \text{service-name} \rangle$  as consequent  
  }  
}
```

Table 5.1: Algorithm for Atomic Web Service Specifications

Determining if the predicate remains unchanged depends on whether or not it is contained in the preconditions. If it is contained we need to check whether negation appears in both cases (precondition and postcondition) or not. We will elaborate on

that when we adapt the algorithm for OWL-S descriptions.

5.2 Composite Service Specifications

In the case of composite service specifications, the main difference is that we need to check each participating service specification separately to create change axioms for the predicates it contains. Hence, we have multiple sets of preconditions and postconditions to examine. Moreover, if a change axiom already exists for a predicate, it doesn't necessarily mean that we don't need to modify it, since it may have been added in a previous step, when we were dealing with a different service. If the new service affects the predicate value, this needs to be reflected in the change axiom.

Taking these points into consideration, we need to do a separate search for predicates in each participating service specification, and for each predicate we find, we should check if the corresponding change axiom has already been added. If it has, we then check if the predicate remains unchanged after the execution of the service we are examining. If it remains unchanged, we don't need to modify at all the existing change axiom. On the other hand, if the service execution changes the value of the predicate, we need to add a simple statement of the form $\alpha = \langle \text{service-name} \rangle$ to the consequent of the clause. Disjunction should be used to separate the existing part of the clause from the new addition.

If no change axiom exists for the current predicate, we need to add one, in the same spirit that we made the additions in the previous case. Hence, we add a frame axiom with a false consequent, if the predicate remains unchanged by the execution of the particular service we are examining, otherwise the consequent should be of the form $a = \langle \text{service-name} \rangle$. Based on these, the algorithm is modified as shown in Table 5.2, in order to support composite service specifications.

Before dealing with the adaptation of the algorithm to the case of OWL-S service descriptions, let's briefly examine the complexity of the algorithm. The complexity

```
For each participating service {  
  For each predicate {  
    If(corresponding frame axiom exists){  
      If(predicate remains unchanged)  
        do nothing  
      else  
        add  $\alpha = \langle \text{service-name} \rangle$  using disjunction in the consequent part  
    }  
  }  
  else{  
    If(predicate remains unchanged)  
      add change axiom with false as consequent  
    else  
      add change axiom with  $\alpha = \langle \text{service-name} \rangle$  as consequent  
  }  
}
```

Table 5.2: Algorithm for Composite Web Service Specifications

of the composite services version of the algorithm is directly dependent on the number of the participating services and on the number of predicates contained in their specifications. Also, the complexity depends on how expensive it is to find whether a change axiom already exists for a selected predicate or not. If we consider a composition with m participating services, with each service having n distinct predicates (and functions), then we can assume that checking if a change axiom for a particular predicate has been added will cost no more than $O(\log n)$. Given this assumption, the complexity of the algorithm for composite service description can be equal to $O(mn \log n)$. Thus, the size of m and n are very important and determine the speed at which the algorithm will be able to provide results. We expect that these num-

bers will not be large as most compositions in average tend to contain a reasonable amount of participating services and, at the same time, few service specifications are very complex, so as to contain a large number of distinct predicates.

5.3 OWL-S Service Descriptions

In order to adapt the algorithm for automatic production of change axioms to a specific Semantic Web Service framework, we need to focus on the special features of this framework. First of all, we need to decide which parts of an OWL-S service description (i.e. service profile, service model and service grounding) we need to handle in order to produce change axioms. The OWL-S description that we will use needs to contain all preconditions and postconditions for all services, as well as the interconnection between the services and the composition schema, if we deal with a composite service. The obvious and only choice is the OWL-S service model. Hence, our algorithm will take as input an OWL-S service model, and will have to examine all processes contained in that model.

5.3.1 SWRL-FOL

Since our algorithm will directly handle preconditions and postconditions that are expressed in SWRL-FOL (similar to the frame axioms that we expressed at the end of the previous chapter), we need to familiarize ourselves to a greater extent with the language. SWRL-FOL is an extension to SWRL, which means it inherits all features included in SWRL and described in Chapter 2. A SWRL ontology in contains a sequence of axioms and facts, including SWRL rules. SWRL-FOL extends this in order to include assertions which are written as first-order formulas. First-order formulas can contain conjunction, disjunction, negation and other constructs as shown in Table 5.3. Atoms are exactly the same as in SWRL.

Variables can be either individual variables, having an OWL class as type, or data

foformula ::= atom
'and(' foformula ')'
'or(' foformula ')'
'neg(' foformula ')'
'implies(' foformula foformula ')'
'equivalent(' foformula foformula ')'
'forall(' variable variable foformula ')'
'exists(' variable variable foformula ')'
atom ::= description '(' i-object ')'
dataRange '(' d-object ')'
individualvaluedPropertyID '(' i-object i-object ')'
datavaluedPropertyID '(' i-object d-object ')'
sameAs '(' i-object i-object ')'
differentFrom '(' i-object i-object ')'
builtIn '(' builtinID d-object ')'
variable ::= 'I-variable(' URIreference description ')'
'D-variable(' URIreference dataRange ')'

Table 5.3: SWRL-FOL formulas and variables

variables belonging to a specified datatype. Using these constructs one can practically express any first-order logic formula in SWRL-FOL. SWRL-FOL has, like SWRL and OWL, an XML concrete syntax and an XML/RDF syntax. We will be using the XML concrete syntax, which is based on the RuleML-FOL XML syntax [12]. To express predicates and functions, individual property atoms will be used.

SWRL-FOL may be an effort to extend SWRL rules with first-order logic constructs, however it is far from complete. The main lacking feature is the support of

functions. Functions do not fit into the RDF or OWL paradigm and thus do not immediately fit into SWRL-FOL. Although the author in [63] sketches a way to support functions in SWRL-FOL it still is very vague and no concrete syntax is offered. We will thus have to omit functions from the algorithm while adapting it for OWL-S descriptions.

5.3.2 An Example OWL-S Process Model

We now return to the motivating example of the previous Chapter (see Section 4.1), in order to base our adaptation of the algorithm to a concrete OWL-S Service Model. Recall that the example contained three atomic services, the money withdrawal service, the wish list update service and the recommendations update service, which constitute a larger composite service that executes an order in an online shop. Because of the lack of support for functions in OWL-S/SWRL-FOL descriptions, we will have to focus on a subset of the example, since the money withdrawal service uses functions in both its preconditions and postconditions. We will focus instead on the parallel composition of the wish list update service and the recommendations update service. This simplification does not affect the essential goal of the algorithm which is to produce change axioms given a service description.

Below, a partial OWL-S process model for the parallel composition of the two services is shown. Only the parts that are relevant to the algorithm are shown, namely, the definition of the atomic processes, the preconditions and the results (which contain the postconditions).

```
<process:AtomicProcess rdf:ID="WishListProc">
  <process:hasResult rdf:resource="#Result1"/>
  <process:hasPrecondition rdf:resource="#Precondition1"/>
  <process:hasOutput rdf:resource="#reportOut"/>
  <process:hasInput rdf:resource="#wlOrderIn"/>
  <process:hasInput rdf:resource="#wlItemIn"/>
```

```
<process:hasInput rdf:resource="#wlListIn"/>
</process:AtomicProcess>
<process:AtomicProcess rdf:ID="RecListProc">
  <process:hasResult rdf:resource="#Result2"/>
  <process:hasPrecondition rdf:resource="#Precondition2"/>
  <process:hasOutput rdf:resource="#reportOut"/>
  <process:hasInput rdf:resource="#rlOrderIn"/>
  <process:hasInput rdf:resource="#rlItemIn"/>
  <process:hasInput rdf:resource="#rlListIn"/>
</process:AtomicProcess>

<expr2:SWRLFOL-Condition rdf:ID="Precondition1">
  <expr:expressionBody rdf:parseType="Literal">
    <swrlf:Assertion owl:name="Pre1">
      <swrlf:And rdf:parseType="Collection">
        <swrlx:individualPropertyAtom swrlx:property="completed">
          <ruleml:Var>o1</ruleml:Var>
          <ruleml:Var>i1</ruleml:Var>
        </swrlx:individualPropertyAtom>
        <swrlx:individualPropertyAtom swrlx:property="included">
          <ruleml:Var>wl1</ruleml:Var>
          <ruleml:Var>i1</ruleml:Var>
        </swrlx:individualPropertyAtom>
      </swrlf:And>
    </swrlf:Assertion>
  </expr:expressionBody>
</expr2:SWRLFOL-Condition>
<expr2:SWRLFOL-Condition rdf:ID="Precondition2">
```

```
<expr:expressionBody rdf:parseType="Literal">
  <swrlf:Assertion owlx:name="Pre2">
<swrlf:And rdf:parseType="Collection">
<swrlx:individualPropertyAtom swrlx:property="completed">
  <ruleml:Var>o1</ruleml:Var>
  <ruleml:Var>i1</ruleml:Var>
</swrlx:individualPropertyAtom>
  <swrlf:Neg>
<swrlx:individualPropertyAtom swrlx:property="included">
  <ruleml:Var>r1</ruleml:Var>
  <ruleml:Var>i1</ruleml:Var>
</swrlx:individualPropertyAtom>
</swrlf:Neg>
</swrlf:And>
</swrlf:Assertion>
  </expr:expressionBody>
</expr2:SWRLFOL-Condition>

<process:Result rdf:ID="Result1">
  <process:inCondition>
    <expr2:SWRLFOL-Condition rdf:ID="Postcondition1">
      <expr:expressionBody rdf:parseType="Literal">
        <swrlf:Assertion owlx:name="Post1">
          <swrlf:Neg>
            <swrlx:individualPropertyAtom swrlx:property="included">
              <ruleml:Var>w1</ruleml:Var>
              <ruleml:Var>i1</ruleml:Var>
            </swrlx:individualPropertyAtom>
```

```
</swrlf:Neg>
    </swrlf:Assertion>
    </expr:expressionBody>
</expr2:SWRLFOL-Condition>
</process:inCondition>
<process:withOutput rdf:resource="\#OutputBinding1"/>
</process:Result>
<process:Result rdf:ID="Result2">
    <process:inCondition>
        <expr2:SWRLFOL-Condition rdf:ID="Postcondition2">
            <expr:expressionBody rdf:parseType="Literal">
                <swrlf:Assertion owl:name="Post2">
                    <swrlx:individualPropertyAtom swrlx:property="included">
                        <ruleml:Var>rl1</ruleml:Var>
                        <ruleml:Var>i1</ruleml:Var>
                    </swrlx:individualPropertyAtom>
                </swrlf:Assertion>
            </expr:expressionBody>
        </expr2:SWRLFOL-Condition>
    </process:inCondition>
    <process:withOutput rdf:resource="\#OutputBinding1"/>
</process:Result>
```

In an OWL-S Service Model, service specifications are declared on an atomic service level. This means that preconditions and postconditions are declared for each atomic service separately and the specification for the composite service is implicitly derived from the atomic service specifications without, however, any explicit declaration of their preconditions and postconditions. Thus, the change axioms that will be produced by the algorithm will be declared for each atomic service separately.

Based on this example, we can derive the basic features of the OWL-S version of the algorithm. First of all the algorithm should run for all atomic processes of the service model. For each one of these processes we need to read the postconditions one by one, in order to determine what change axioms we should add. For each postcondition, we should first create a list of all individual property atoms that are contained. For each individual property atom, our actions depend on whether this atom appears in any precondition or not. To that end, it would be helpful if we have already compiled a list of all the individual property atoms contained in the preconditions. If the atom (the predicate, that is) is not included in any precondition, then we have no way of determining whether the particular service makes any change to the particular predicate.

If an individual property atom is included in both a precondition and a postcondition, we can determine whether a change axiom should be added or not. We need to check if there is any negation in either the precondition or the postcondition or both. This is easily determined by exploiting the XML syntax of SWRL-FOL. If an atom is negated, then it should have a *Neg* element as parent. If both atoms or neither of them are negated, then the particular service does not change the predicate represented by the atom and this should be reflected in the change axiom we add. On the other hand, if only one of the atoms is negated, then the service changes the predicate value and we need to express that effect in the change axiom.

A change axiom will be added, regardless of whether the current service modifies the predicate value or not. The change axiom will have the same antecedent (body) for both cases, and a different consequent (head) depending on whether the service changes the predicate value. The consequent should be of the form $\alpha = \langle \text{service-name} \rangle$ if the service changes the predicate value, otherwise it should be false.

After checking the postconditions, we have to check the preconditions too, in case some predicates are included in preconditions but not in postconditions. While including a predicate in a postcondition but not in a precondition doesn't necessarily

mean that the predicate value is modified, the opposite, having a predicate in a precondition but not in a postconditions implies that a predicate remains unchanged, so we need to make this explicit using a change axiom with a false consequent.

Change axioms are added as extra postconditions in the service model description. Due to that fact, before we do any of the actions presented above for each postcondition, we need to check first if it is a change axiom that has been previously added. This is easily done by checking if an individual property atom with the name `Occur` exists in the condition. The name `Occur` is kept especially for the expression of change axioms, so it is an indication that a postcondition is a change axiom. Moreover, we can keep a list of the individual property atoms that we have already checked so as not to reexamine previously examined cases.

All the points that we mentioned here lead to the modified algorithm shown in pseudo-code in Table 5.4 at the end of this chapter. A flow chart describing the actions performed by the algorithm is shown in Figure 5.1, also at the end of this chapter.

5.4 Implementation

The algorithm for the automatic production of change axioms for the case of OWL-S Service Models has been implemented in Java. The initial concern during the implementation phase was how to handle OWL-S Service Model files. There have been few efforts to provide a complete API for OWL or OWL-S ontologies. We opted to use the OWL-S API created by the Intelligent Agents Laboratory of the Carnegie Mellon University [40] due to its Java implementation and complete and thorough handling of all aspects of an OWL-S ontology.

The CMU OWL-S API (see Appendix A for a detailed overview) offers mechanisms to read and write OWL-S ontology files, for all aspects of an OWL-S ontology (service profile, service model and service grounding). The parser reads OWL-S files and stores

the information contained in them as Jena [72] RDF models. These models can be manipulated via a multitude of functions, both OWL-generic, offered indirectly by the Jena framework, and OWL-S-specific, offered directly by the OWL-S API. We used the subset of the OWL-S API that is dedicated to parsing and writing OWL-S Service Model ontology files. Some minor modifications of the CMU OWL-S API were necessary and are detailed in Appendix A.

While the CMU OWL-S API was the most suitable choice for our work, it still lacks many important features that were absolutely necessary for the implementation of our algorithm. The most important deficiency is the fact that the provided API is solely dedicated on SWRL rules, since SWRL is the de-facto rule language supported by OWL-S. Hence, the API offers an SWRL parser only. Since we didn't have access to the source code of the SWRL parser in order to tweak it to support the extensions offered by SWRL-FOL, we had to implement a SWRL-FOL parser from scratch. Since the syntax of the SWRL-FOL rules in our example is concrete XML, we based our parser on a DOM parser. Each rule is stored as a string by the OWL-S API and then is fed to the parser which outputs a Document Object Model that can be easily navigated and manipulated for the needs of our implementation.

Another implementation issue that is worth mentioning is how we dealt with deciding whether a process changes the value of an individual property or not. For postconditions, we exploited the DOM model of each rule, in order to retrieve the parent node of the individual axiom at hand. If the parent node is a $\langle Neg \rangle$ element, then the atom is negated. The preconditions were treated similarly, however we needed to have access not only to the atom names contained in the preconditions so as to select the conditions containing the current atom that we were examining but also to the atom nodes themselves in order to be able to retrieve their parent nodes in the DOM models created by parsing the SWRL-FOL preconditions. To that end, what is mentioned in the algorithm as a precondition list is actually implemented as a linked list of pairs of individual property atom names and the DOM nodes where

the atoms are stored.

The creation of change axioms was implemented as a separate function that is called whenever a change axiom has to be added to the model. The `createChangeAxiom` function has the following parameters:

- `serviceName`: The name of the service that causes the change, or null if the change axiom has false as consequent
- `predicate`: The name of the predicate (or more accurately individual property atom) that is the focus of the change axiom
- `var1`: The first argument of the predicate
- `var2`: The second argument of the predicate

The $\alpha = \langle \text{service-name} \rangle$ part of the change axioms is expressed using the slotted RuleML syntax which is the only way that has been proposed to make up for the lack of support for functions in SWRL-FOL (the service-name in the change axiom can be viewed as a function that represents the execution of the service). The `createChangeAxiom` function provides a generic skeleton for the change axiom which is then specialized based on the input parameters. The final result is an RDF literal which is transformed to an SWRL rule implication and returned by the function. This implication is in the right format needed by the OWL-S API in order to create a new postcondition for the current process and add it to the model.

If we provide the OWL-S Service Model shown at the beginning of subsection 5.3.2 as input to the implementation that we described it will produce a new OWL-S Service Model as output, containing two new change axioms (one for the predicate completed and one for the predicate included) for each one of the two services. Here we show the two change axioms for the wish list update service.¹

¹To differentiate from a predicate evaluated before the execution of the service and one evaluated after the execution of the service, we add *_pr* to the name of the predicate to denote it as primed (i.e. evaluated after the execution).

```
<process:inCondition>
  <expr2:SWRLFOL-Condition>
    <expr:expressionBody rdf:parseType="Literal">
      <swrlf:Assertion xmlns:owlx="http://www.w3.org/2003/05/owl-xml#"
        xmlns:swrlf="http://www.daml.org/2004/11/fol/fol#"
        owlx:name="axiom1">
        <swrlf:Forall xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
          rdf:parseType="Literal">
            <ruleml:Var
              xmlns:ruleml="http://www.w3.org/2003/11/ruleml#">alpha</ruleml:Var>
            <ruleml:Var
              xmlns:ruleml="http://www.w3.org/2003/11/ruleml#">o1</ruleml:Var>
            <ruleml:Var
              xmlns:ruleml="http://www.w3.org/2003/11/ruleml#">i1</ruleml:Var>
            <swrl:Imp xmlns:swrl="http://www.w3.org/2003/11/swrl#">
              <swrl:body>
                <swrlf:And rdf:parseType="Collection">
                  <swrlx:individualPropertyAtom
                    xmlns:swrlx="http://www.w3.org/2003/11/swrlx#"
                    swrlx:property="completed">
                      <ruleml:Var
                        xmlns:ruleml="http://www.w3.org/2003/11/ruleml#">o1
                      </ruleml:Var>
                      <ruleml:Var
                        xmlns:ruleml="http://www.w3.org/2003/11/ruleml#">i1
                      </ruleml:Var>
                    </swrlx:individualPropertyAtom>
                  <swrlf:Neg>
```

```
<swrlx:individualPropertyAtom
  xmlns:swrlx="http://www.w3.org/2003/11/swrlx#"
  swrlx:property="completed_pr">
  <ruleml:Var
    xmlns:ruleml="http://www.w3.org/2003/11/ruleml#">o1
  </ruleml:Var>
  <ruleml:Var
    xmlns:ruleml="http://www.w3.org/2003/11/ruleml#">i1
  </ruleml:Var>
</swrlx:individualPropertyAtom>
</swrlf:Neg>
<swrlx:classAtom xmlns:swrlx="http://www.w3.org/2003/11/swrlx#">
  <owlx:Class owlx:name="Occur"></owlx:Class>
  <ruleml:Var
    xmlns:ruleml="http://www.w3.org/2003/11/ruleml#">alpha
  </ruleml:Var>
</swrlx:classAtom>
</swrlf:And>
</swrl:body>
<swrl:head>
</swrl:head>
</swrl:Imp>
</swrlf:Forall>
</swrlf:Assertion>
</expr:expressionBody>
<expr:expressionLanguage rdf:resource="file:///c:/Expression.owl#SWRL-FOL"/>
</expr2:SWRLFOL-Condition>
</process:inCondition>
```

```
<process:inCondition>
  <expr2:SWRLFOL-Condition>
    <expr:expressionBody rdf:parseType="Literal">
      <swrlf:Assertion xmlns:owlx="http://www.w3.org/2003/05/owl-xml#"
        xmlns:swrlf="http://www.daml.org/2004/11/fof/fof#"
        owlx:name="axiom2">
        <swrlf:Forall xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
          rdf:parseType="Literal">
            <ruleml:Var
              xmlns:ruleml="http://www.w3.org/2003/11/ruleml#">alpha
            </ruleml:Var>
            <ruleml:Var
              xmlns:ruleml="http://www.w3.org/2003/11/ruleml#">w1
            </ruleml:Var>
            <ruleml:Var
              xmlns:ruleml="http://www.w3.org/2003/11/ruleml#">i1
            </ruleml:Var>
            <swrl:Imp xmlns:swrl="http://www.w3.org/2003/11/swrl#">
              <swrl:body>
                <swrlf:And rdf:parseType="Collection">
                  <swrlx:individualPropertyAtom
                    xmlns:swrlx="http://www.w3.org/2003/11/swrlx#"
                    swrlx:property="included">
                    <ruleml:Var
                      xmlns:ruleml="http://www.w3.org/2003/11/ruleml#">w1
                    </ruleml:Var>
                    <ruleml:Var
```

```
        xmlns:rulxml="http://www.w3.org/2003/11/rulxml#">i1
    </rulxml:Var>
</swrlx:individualPropertyAtom>
<swrlf:Neg>
    <swrlx:individualPropertyAtom
        xmlns:swrlx="http://www.w3.org/2003/11/swrlx#"
        swrlx:property="included_pr">
        <rulxml:Var
            xmlns:rulxml="http://www.w3.org/2003/11/rulxml#">w1
        </rulxml:Var>
        <rulxml:Var
            xmlns:rulxml="http://www.w3.org/2003/11/rulxml#">i1
        </rulxml:Var>
    </swrlx:individualPropertyAtom>
</swrlf:Neg>
<swrlx:classAtom xmlns:swrlx="http://www.w3.org/2003/11/swrlx#">
    <owlx:Class owlx:name="Occur"></owlx:Class>
    <rulxml:Var
        xmlns:rulxml="http://www.w3.org/2003/11/rulxml#">alpha
    </rulxml:Var>
</swrlx:classAtom>
</swrlf:And>
</swrl:body>
<swrl:head>
    <swrlf:Equivalent>
        <rulxml:Var
            xmlns:rulxml="http://www.w3.org/2003/11/rulxml#">alpha
        </rulxml:Var>
```



```
<ruleml:Atom xmlns:ruleml="http://www.w3.org/2003/11/ruleml#">
  <Rel>file:///c:/TestProcess.owl#WishListProc</Rel>
</ruleml:Atom>
</swrlf:Equivalent>
</swrl:head>
</swrl:Imp>
</swrlf:Forall>
</swrlf:Assertion>
</expr:expressionBody>
<expr:expressionLanguage rdf:resource="file:///c:/Expression.owl#SWRL-FOL"/>
</expr2:SWRLFOL-Condition>
</process:inCondition>
```

5.4.1 Correctness

Having presented an algorithm for the automatic production of change axioms, we need to check the correctness of our approach, in order to ensure that the specification produced by the algorithm is indeed complete with regard to the frame problem. From the definition of the change axioms, we recall that this solution offers a state-oriented perspective to the frame axioms, demanding that we declare which services change each predicate or function. Thus, a complete specification should contain one change axiom for each distinct predicate and function contained in the initial set of preconditions and postconditions of the service.

Given that definition, we need to calculate the exact number of change axioms that should be added, given a specification with a certain number of predicates and functions. Some of these predicates and functions are contained in both preconditions and postconditions. Let's assume their amount equals to m . Additionally to them, there may also be some predicates and functions that exist solely in the preconditions, for which we also need to express change axioms. If their amount equals to n , then

an approach is correct if it results in producing $m+n$ distinct change axioms. The inverse is also true: if the resulting specification contains $m+n$ distinct change axioms, devised as we discussed, then the approach used to produce the specification is correct. Note that we can't add a change axiom for a predicate or function that is contained solely in a postcondition as we cannot assume its value before the execution of the service, if the exact value is not explicitly stated in a precondition.

Our approach can be considered correct by design. As we analyzed in subsection 5.3.2, our implementation first examines all postconditions and produces change axioms for all predicates² contained in both preconditions and postconditions. Thus, the first part of the implementation produces the first change axioms, equal to m , if m is the number of the distinct predicates. The second part of the implementation examines the preconditions list for any predicates that have not already been checked and are not part of any postcondition, hence producing extra change axioms, whose amount is equal to n , if n is the number of the predicates that are only part of the preconditions. Consequently, our approach will always produce the required amount, $m+n$, of change axioms.

5.4.2 Detecting Inconsistencies

As we have mentioned when explaining the issues raised by the frame problem in Chapter 4, some composite specifications may contain inconsistencies between the preconditions and postconditions declared in the separate specifications of the services that are composed together. Inconsistencies can be detected via the composition rules declared in [4]. In our implementation, we have included a function that offers a limited support for detecting inconsistencies between atomic processes that

²As mentioned earlier, due to the lack of explicit support for functions in SWRL-FOL, we omit functions from any discussion concerning the implementation. This doesn't compromise the generality of our approach, since treating functions is virtually identical to treating predicates when producing change axioms.

participate in an OWL-S Service Model.

In order to detect inconsistencies, one must find pairs of services in the process model of the composite model. In OWL-S, composite process models can be considered as trees whose nonterminal nodes are labeled with control constructs (denoting the type of composition e.g. sequence, parallel execution and so on), each of which has children specified using components. The leaves of the tree are invocations of other processes, indicated as instances of class `Perform`. Thus, we need to find pairs of these `Perform` instances. This is achieved by a recursive function which scans the composite process model beginning from its root. For each pair of `Perform` instances, checking for inconsistencies depends on the composition schema. In our implementation, we have dealt with the `Sequence` and `Split-Join` control constructs, but extending the work to support all control constructs is straightforward.

Inconsistencies in a `Sequence` control construct occur when a postcondition of the first service includes a predicate in the negated form while a precondition of the second service includes it in its normal form, and vice versa. We first scan the postconditions of the first service and create a list of the predicates contained (similarly to the list of predicates in preconditions in the previous part of the implementation) and then check this list against the predicates that are included in the preconditions of the second service.

As far as `Split-Join` control constructs are concerned, inconsistencies here are caused when the two services that are executed in parallel have either conflicting preconditions or postconditions. In other words, the first service may include a precondition where a predicate is in the negated form while at the same time the second service includes a precondition where the same predicate is in its normal form. The same can happen with the postconditions of the two service. We first compare the preconditions of the two services executed in parallel, then move on to their postconditions.

`Split-Join` parallel compositions, by definition, involve services that are expected

to begin execution at the same time, while execution of the composition is considered completed only when all participating services have completed execution. To that end, we checked for inconsistencies in both preconditions and postconditions. If one considers parallel compositions where services only expect to begin together, or only complete execution together, then inconsistencies need to be checked only at the precondition, or postcondition level, respectively.

5.5 Evaluation

We will attempt to evaluate our approach and implementation based on four distinct criteria. First we will deal with the level of applicability of the approach to existing Semantic Web Service frameworks. The second evaluation criterion involves the level of expressivity of the languages used in the approach. We will also briefly examine the reasoning demands of our approach compared to other approaches. Finally, we will evaluate our implementation based on its execution time.

It was stressed out early on in the description of our approach that one of our goals was to provide a solution approach that stays close to existing frameworks for Semantic Web Services description. Applicability of the approach to these frameworks was of utmost importance since they (especially OWL-S) are widely accepted as the only well-developed efforts for describing and supporting Semantic Web Services throughout their lifecycle. To that end, we attempted to apply our approach for the case of OWL-S Service Model descriptions containing SWRL-FOL rules, as shown in Section 5.3. After some necessary adaptations, we presented an implementation of the algorithm that works for OWL-S description, showing that it is in fact applicable to that particular Semantic Web framework. Moreover, in subsection 4.2.2, we also dealt with the cases of WSMO and SWSF by showing how change axioms can be expressed in the description languages used in these frameworks, namely WSML and SWSL. In comparison, existing solution schemas such as the ones presented in Section 3.2, use

calculi such as situation calculus and fluent calculus, which are not directly adaptable to current Semantic Web Service frameworks.

As far as the level of expressivity of the languages used in our approach, it is important to state beforehand that solving the frame problem usually requires the enhancement of existing description languages. This is true in our case as well, as we needed to extend first-order predicate logic with the special predicate *Occur* and special variable α in order to express change axioms. Moreover, when applying our approach to the case of OWL-S Service descriptions, we had to choose SWRL-FOL instead of SWRL because we needed the extensions it offers for expressing full first-order logic formulas. These choices led to our approach having a great level of expressivity, being able to support all kinds of change axioms. In addition, if we assume that SWRL-FOL will continue to be developed in order to support functions to a greater level, then it will be possible to express any kind of condition for a service, as long as it can be expressed in first-order logic. This is much more than what can be said for other approaches such as those that are based on situation calculus, such as [54], since they have limited expressivity, especially when it comes to dealing with intermediate states in a service execution.

However, there is always a tradeoff between expressivity and reasoning demands. Expanding a language with regard to its ability to express a broader range of knowledge means that more effort will be necessary for reasoning based on that knowledge. While reasoners for SWRL are available, SWRL-FOL hasn't matured enough and a reasoner has not been developed yet. This means that currently, it is impossible to evaluate whether our approach fares well with regard to reasoning demands, in comparison with other approaches. However, many efficient reasoners and theorem provers exist that are based on first-order logic and others offer rule support for SWRL and RuleML which are the languages on which SWRL-FOL is based. Hence, it seems possible that an effective reasoner for SWRL-FOL could be developed, however this is out of the scope of this thesis.

5.5.1 Execution Time

For the purpose of evaluating our implementation according to the execution time criterion, we can break it down into the following parts:

- Parsing of the input file and creation of the Jena RDF model for the OWL-S Service Model ontology
- For each atomic process, scanning of the preconditions and creation of the precondition predicates list
- For each atomic process, scanning of the postconditions and addition of the change axioms
- For each atomic process, scanning of the preconditions and addition of any extra change axioms
- For the overall composite process, scanning for inconsistencies
- Export of the modified model into an OWL-S Service Model ontology file

For our example, the complete execution of all the parts of the implementation takes roughly 22 seconds on average³. The parsing and writing of the OWL-S Service Model by the CMU OWL-S API take 16 seconds on average, which is a reasonable time given the multitude of elements created in an OWL-S Service Model ontology. This leaves us with roughly 6 seconds that are spent on the actual implementation of the algorithm for producing change axioms as well as the inconsistencies detection. This means that we need about 6 seconds to produce change axioms for the parallel composition of two services, each one of which has one precondition and one postcondition, involving two different predicates for each service specification.

³The times included in this section are the ones given by the NetBeans environment after a successful execution of the implementation of the algorithm.

In order to check the scalability of the implementation, we modified the services in the example in order to include 5 times more preconditions and postconditions, each one dealing with a different predicates. This means that each service has 5 preconditions and 5 postconditions, involving 10 different predicates. The total time for the modified example is roughly 35 seconds. The parsing and writing of the OWL-S Service Model by the CMU OWL-S API take 27 seconds this time, which means that the actual implementation of our algorithm as well as the inconsistencies detection takes roughly 8 seconds, meaning a 33% increase for a specification including 5 times more conditions than the original one.

The time-consuming aspects of the implementation include the initial scanning of the preconditions to create the precondition predicates list, the scanning of the postconditions to determine which change axioms to add and the second scanning of the preconditions to add any extra change axioms. These three actions involve traversing a list, while the second one involves traversing a second list (the precondition predicates list) for each iteration. All these actions are repeated for as many atomic process that are contained in the OWL-S Service model specification. Given the complexity of these actions, the reported times seem reasonable enough and, more importantly, seem to scale well.

The evaluation points described in this section are summarized in Table 5.5.

```

For each atomic process {
  For each postcondition {
    If(the postcondition is a change axiom)
      continue to next postcondition
    else{
      For each individual property atom {
        If(the atom has already been checked)
          continue to next atom
        else{
          If(atom exists in a precondition too){
            If(negation exists in both cases or neither)
              add change axiom with  $\alpha = \langle \text{service-name} \rangle$  as consequent
            else
              add change axiom with false as consequent
            flag atom as checked
          }
          else      continue to next atom
        }
      }
    }
  }
}

For each precondition {
  For each individual property atom {
    If(the atom has already been checked)
      continue to next atom
    else      add change axiom with false as consequent
  }
}

```

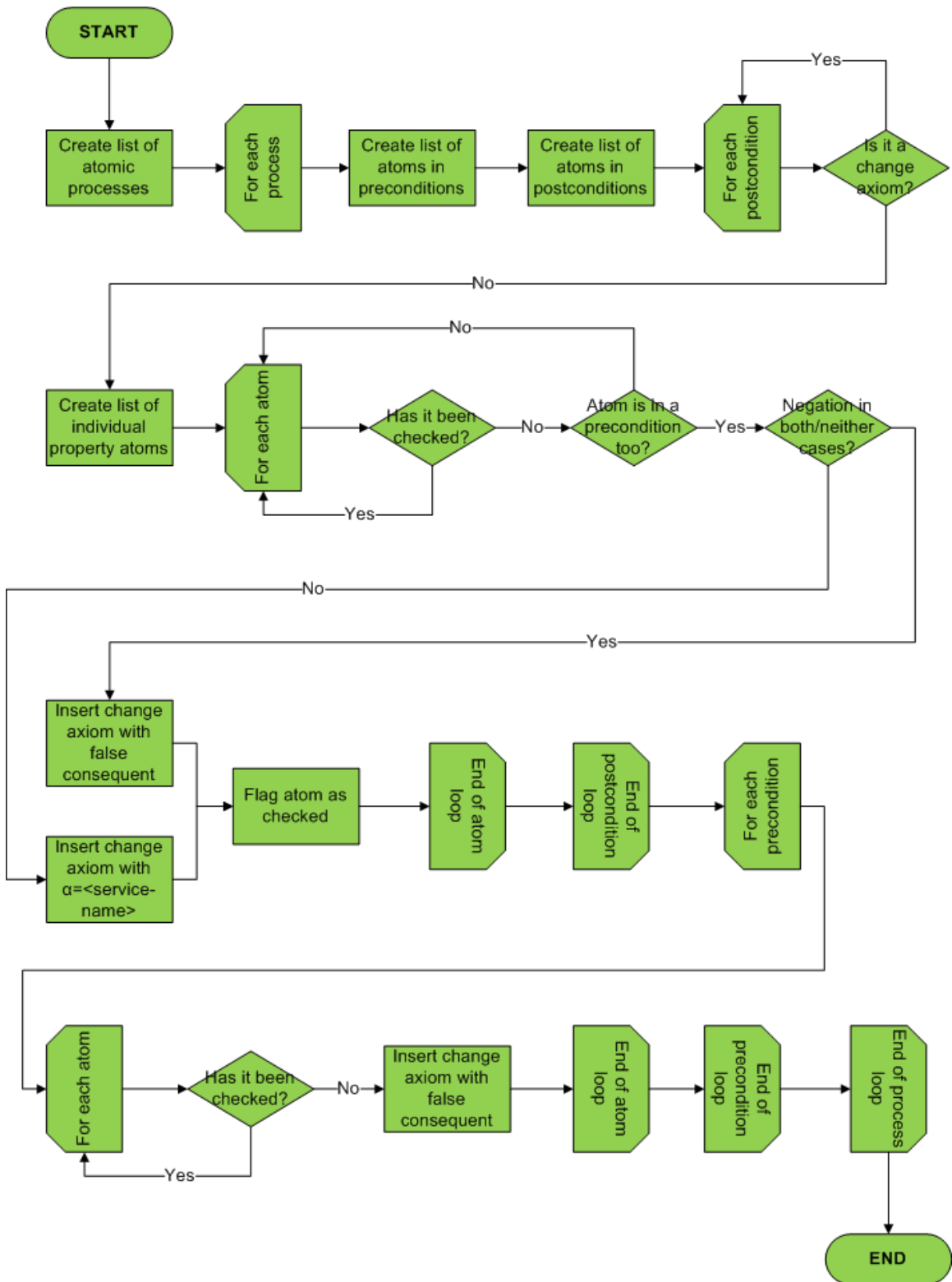



Figure 5.1: Flow Chart of the final algorithm

	No. of Services	Preconditions / Service	Postconditions / Service	Total No. of Predicates	Total Time (sec)	CMU OWL-S API Time (sec)	Total Time - OWL-S API Time (sec)
Initial Example	2	1	1	2	22	16	6
Scaled Example	2	5	5	10	35	27	8

Table 5.5: Evaluation of the Implementation

Chapter 6

Conclusions and Future Work

6.1 Conclusions

In this work, we explored the frame problem with regard to its existence in the field of Web Services. We argued that Web Service specifications, like any formal specification that uses the precondition and postcondition notation, are a distinct example of specifications that become problematic when trying to explicitly state that nothing changes, except when it is stated. A multi-faceted motivating example was presented, ranging from atomic Web service specifications to specifications of composite services. The examples illustrated the fact that including frame axioms makes the Web service specifications lengthier, more complex and more difficult to handle and use in order to formally prove properties. It was also shown that in the case of composite service specifications, the effects of the frame problem aggravate, as the inclusion of frame axioms may lead to inconsistencies when attempting to conjoin the resulting specifications.

After having clearly depicted the issues raised because of the frame problem, we presented a solution approach that aims to offer a state-oriented rather than a procedure-oriented perspective to the problem of correctly stating framing axioms. The approach defines a type of axioms called explanation closure axioms, which need

to be stated for every predicate or function included in our knowledge. These axioms state precisely when each predicate or function may change, in other words, with the successful execution of which service procedure is a predicate or function modified. The explanation closure axioms, or change axioms, are updated whenever a new service becomes known that modifies existing or new predicates or functions. It was also shown how to express change axioms not only in simple first-order predicate logic but also in various rule languages contained in recent Semantic Web service efforts, such as SWRL-FOL, WSML and SWSL-FOL.

We then presented an algorithm that aims to automatically produced change axioms for an atomic Web Service specification. We described and explained how the algorithm is adapted to support composite service specifications and then further modified in order to be suitable for the specific case of OWL-S Service Model descriptions. An implementation of the final version of the algorithm was briefly presented and evaluated, showing that it can effectively and efficiently produce change axioms for OWL-S descriptions, with the limitations that these descriptions pose, such as lack of support for functions and implicit declaration of composite service specifications.

This thesis is a clear example of how current Semantic Web Service frameworks are still immature and lacking many features that are indispensable for the achievement of the goals set by service-oriented architectures and the Semantic Web vision. There is a great deal of open or partially solved issues that have to be addressed before these frameworks are able to provide and fully support the automation of many service-related procedures, from discovery, negotiation and adaptation, to composition, invocation and monitoring. Our work is hopefully a step in the right direction, as it attempts to provide more complete specifications than ever for Web Services. Offering a complete specification of a Web Service means that one can find it easier, can be absolutely sure that a Web Service matches his or her requirements and can predict, to a large extent, the service's behavior and the effects it has, as well as any inconsistencies that may appear when executing the service as part of a larger

composition schema.

Before discussing future work ideas, it should be stated with great emphasis that any future development in the research topics mentioned throughout this thesis depends on the existence of complete and powerful Web Service specification languages. Limited rule languages such as SWRL are undoubtedly inadequate and inefficient and even extensions such as SWRL-FOL, which has not seen any development since it was first proposed, are still lacking a great deal of important features such as support for functions and n-ary predicates. However, a language with a powerful syntax may not be easily and seamlessly integrated to Semantic frameworks using ontologies and striving to offer advanced and effective reasoning capabilities. There is always a tradeoff between expressivity and decidability that needs to be resolved.

6.2 Future Work

Some initial pointers for future work related to this thesis include the extension of the support for finding inconsistencies and conflicts between participating services in a composite service specification. More composition schemas apart from sequence and parallel execution can be examined and ways of handling inconsistencies once they are found can be explored.

The implementation of the algorithm can be further adapted to other Semantic Web Service frameworks such as WSMO and SWSF. Although these two efforts are not as mature and well-researched as OWL-S, it would be interesting to explore how the languages included with these frameworks, WSML and SWSL respectively can cope with the issue of expressing a complete service specification, including change axioms and how one can deal with their own limited syntax.

A major related issue is involves exploring whether the solution approach presented here can be evolved into a system that automatically creates Web service specifications that have all frame problem-related issues solved, given traditional

specifications as input and the possibility of including such a system into a general automatic composition framework that takes into account the frame problem when attempting to compose Web services.

Finally, it would be interesting to explore two other facets of the frame problem, which are considered as separate families of problems themselves, in the same way that we explored the frame problem. These are the ramification problem and the qualification problem. The ramification problem deals with indirect consequences of an action. It entails all issues related to representing what happens implicitly due to an action or to control secondary and tertiary effects within the same period. For the case of Web Services, the ramification problem may deal with effects that are caused by effects of a service execution, i.e. executing a service affects the world in a certain way, which then leads to secondary effects that derive from the primary ones.

On the opposite side, the qualification problem is concerned with the impossibility of listing all the preconditions required for a real-world action to have its intended effect. It can be argued that even if one considers a Web Service specification as complete, there might always be some conditions that have not been explicitly declared. Trying to declare more and more preconditions may then lead to a service that will never be able to be executed, since it would be less and less possible to simultaneously satisfy all these conditions and achieve the expected result of the service at the same time.

Bibliography

- [1] The RuleML Initiative. <http://ruleml.org>.
- [2] R. Akkiraju, J. Farrell, J. Miller, M. Nagarajan, M. Schmidt, A. Sheth, and K. Verma. Web Service Semantics - WSDL-S, A joint UGA-IBM Technical Note, version 1.0. Technical report, April 2005.
- [3] R. Akkiraju, K. Verma, R. Goodwin, P. Doshi, and J. Lee. Executing Abstract Web Process Flows. In *International Conference on Automated Planning and Scheduling (ICAPS)*, 2004.
- [4] V. Alevizou and D. Plexousakis. Enhanced specifications for web service composition. In *ECOWS*, pages 223–232. IEEE Computer Society, 2006.
- [5] G. Antoniou and F. van Harmelen. *A Semantic Web Primer, 2nd Edition*. The MIT Press, 2 edition, March 2008.
- [6] Ariba Inc., IBM Corp., and Microsoft Corp. Universal Description, Discovery, and Integration (UDDI). Technical white paper, uddi.org, September 2000.
- [7] F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, and P. F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, 2003.
- [8] S. Balzer, T. Liebig, and M. Wagner. Pitfalls of OWL-S – A Practical Semantic Web Use Case. In *ICSOC' 04: Proceedings of the 2nd International Conference*

BIBLIOGRAPHY

- on Service Oriented Computing*, pages 289–298, New York, NY, USA, November 2004.
- [9] S. Battle, A. Bernstein, H. Boley, B. Grosz, M. Gruninger, R. Hull, M. Kifer, D. Martin, S. McIlraith, D. McGuinness, J. Su, and S. Tabet. Semantic Web Services Framework (SWSF) Overview. World Wide Web Consortium, Member Submission SUBM-SWSF-20050909, September 2005.
- [10] S. Bechhofer, F. van Harmelen, J. Hendler, I. Horrocks, D. McGuinness, P. Patel-Schneijder, and L. A. Stein. OWL Web Ontology Language Reference. Recommendation, World Wide Web Consortium (W3C), February 2004.
- [11] K. Bennett, P. Layzell, D. Budgen, P. Brereton, L. Macaulay, and M. Munro. Service-based software: the future for flexible software. In *Seventh Asia-Pacific Software Engineering Conference (APSEC2000)*, pages 214–221, 2000.
- [12] H. Boley, M. Dean, B. Grosz, M. Sintek, B. Spencer, S. Tabet, and G. Wagner. FOL RuleML: First-Order Logic RuleML Language. World Wide Web Consortium, Member Submission SUBM-FOL-RuleML-20050125, January 2005.
- [13] D. Booth, H. Haas, F. McCabe, E. Newcomer, M. Champion, C. Ferris, and D. Orchard. Web Services Architecture. W3C Note NOTE-ws-arch-20040211, World Wide Web Consortium, February 2004.
- [14] E. Borger and R. F. Stark. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.
- [15] A. Borgida, J. Mylopoulos, and R. Reiter. On the Frame Problem in Procedure Specifications. *Software Engineering, IEEE Transactions on*, 21(10):785–798, 1995.

BIBLIOGRAPHY

- [16] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. F. Nielsen, S. Thatte, and D. Winer. Simple Object Access Protocol (SOAP) 1.1. W3c note, World Wide Web Consortium, May 2000.
- [17] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, F. Yergeau, and J. Cowan. Extensible Markup Language (XML) 1.1 (Second Edition). World Wide Web Consortium, Recommendation REC-xml11-20060816, August 2006.
- [18] D. Brickley and R. V. Guha. RDF Vocabulary Description Language 1.0: RDF Schema. W3c recommendation, W3C, February 2004. <http://www.w3.org/TR/2004/REC-rdf-schema-20040210/>.
- [19] M. Brodie, C. Bussler, J. de Bruijn, T. Fahringer, D. Fensel, M. Hepp, H. Lausen, D. Roman, T. Strang, H. Werthner, and M. Zaremba. Semantically Enabled Service-oriented Architectures - A Manifesto and a Paradigm Shift in Computer Science. Technical report, Digital Enterprise Research Institute (DERI), September 2006.
- [20] F. Casati, S. Ilnicki, L. jie Jin, V. Krishnamoorthy, and M.-C. Shan. Adaptive and Dynamic Service Composition in eFlow. In B. Wangler and L. Bergman, editors, *CAiSE*, volume 1789 of *Lecture Notes in Computer Science*, pages 13–31. Springer, 2000.
- [21] R. Chinnici, J.-J. Moreau, A. Ryman, and S. Weerawarana. Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language. World Wide Web Consortium, Recommendation REC-wsdl20-20070626, June 2007.
- [22] K. L. Clark. Negation as failure. pages 293–322, 1978.
- [23] Erik Sandewall. An approach to the Frame Problem and its Implementation. In *Machine Intelligence 7*, pages 195–204. Edinburgh University Press, 1972.
- [24] T. Erl. *SOA Principles of Service Design*. Prentice Hall PTR, 2007.

BIBLIOGRAPHY

- [25] K. Erol, J. A. Hendler, and D. S. Nau. Semantics for HTN planning. Technical Report CS-TR-3239, 1994.
- [26] K. Erol, J. A. Hendler, and D. S. Nau. UMCP: A Sound and Complete Procedure for Hierarchical Task-network Planning. In *AIPS*, pages 249–254, 1994.
- [27] J. A. Estefan, K. Laskey, F. G. McCabe, and D. Thornton. Reference Architecture for Service Oriented Architecture Version 1.0. Online PDF, 2008.
- [28] J. Farrell and H. Lausen. Semantic Annotations for WSDL and XML Schema. World Wide Web Consortium, Recommendation REC-sawsdl-20070828, August 2007.
- [29] S. Ferndrigger, A. Bernstein, J. Dong, Y. Feng, Y.-F. Li, and J. Hunter. Enhancing Semantic Web Services with Inheritance. pages 162–177. 2008.
- [30] R. T. Fielding and R. N. Taylor. Principled design of the modern Web architecture. *ACM Transactions on Internet Technologies*, 2(2):115–150, 2002.
- [31] M. R. Genesereth and R. Fikes. Knowledge Interchange Format: Version 3.0: Reference Manual. Stanford University, California, 1992.
- [32] M. Ghallab, A. Howe, C. Knoblock, D. McDermott, A. Ram, M. Veloso, D. Weld, and D. Wilkins. PDDL the planning domain definition language (1998). 1998.
- [33] G. D. Giacomo, Y. Lespe'rance, and H. J. Levesque. ConGolog, a concurrent programming language based on the situation calculus. *Artif. Intell.*, 121(1-2):109–169, 2000.
- [34] G. Gro'se, S. Ho"lldobler, and J. Schneeberger. Linear Deductive Planning. *J. Log. Comput.*, 6(2):233–262, 1996.
- [35] T. R. Gruber. A Translation Approach to Portable Ontology Specifications. *Knowledge Acquisition*, 5(2):199–220, 1993.

BIBLIOGRAPHY

- [36] M. Grüninger and C. Menzel. The Process Specification Language (PSL) Theory and Applications. *AI Magazine*, 24(3):63–74, 2003.
- [37] N. Guarino. Semantic Matching: Formal Ontological Distinctions for Information Organization, Extraction, and Integration. In M. Pazienza, editor, *Information Extraction: A Multidisciplinary Approach to an Emerging Information Technology*, pages 139–170. Springer, 1997.
- [38] J. Hoffmann. A Heuristic for Domain Independent Planning and its Use in an Enforced Hill-climbing Algorithm. In *PuK*, 2000.
- [39] I. Horrocks, P. F. Patel-Schneider, H. Boley, S. Tabet, B. Grosfand, and M. Dean. SWRL: A Semantic Web Rule Language Combining OWL and RuleML. W3C Member Submission, May 2004. Available at: <http://www.w3.org/Submission/SWRL>, last access on Dez 2008.
- [40] C. M. U. Intelligent Agents Laboratory, Computer Science Department. CMU OWL-S API Version 1.1.3. Web Site, 2008.
- [41] U. Keller and H. Lausen. Functional Description of Web Services. WSML Working Draft D28.1 v0.1, WSMO Working Group, January 2006.
- [42] M. Klusch, A. Gerber, and M. Schmidt. Semantic Web Service Composition Planning with OWLS-Xplan. In *1st International AAAI Fall Symposium on Agents and the Semantic Web*, pages 117–120. IEEE Computer Society, 2006.
- [43] R. A. Kowalski and M. J. Sergot. A Logic-based Calculus of Events. *New Generation Comput.*, 4(1):67–95, 1986.
- [44] P. Küngas. *Distributed Agent-Based Web Service Selection, Composition and Analysis through Partial Deduction*. PhD thesis, Norwegian University of Science and Technology, Trondheim, Norway, July 2006.

BIBLIOGRAPHY

- [45] O. Lassila and R. R. Swick. Resource Description Framework (RDF): Model and Syntax Specification. Recommendation, World Wide Web Consortium (W3C), February 1999.
- [46] F. Lécué, A. Léger, and A. Delteil. DL Reasoning and AI Planning for Web Service Composition. In *Web Intelligence*, pages 445–453. IEEE, 2008.
- [47] H. J. Levesque, F. Pirri, and R. Reiter. Foundations for the situation calculus. *Electron. Trans. Artif. Intell.*, 2:159–178, 1998.
- [48] H. J. Levesque, R. Reiter, Y. Lesperance, F. Lin, and R. B. Scherl. GOLOG: A Logic Programming Language for Dynamic Domains. *Journal of Logic Programming*, 31(1-3):59–83, 1997.
- [49] S. Majithia, D. W. Walker, and W. A. Gray. A Framework for Automated Service Composition in Service-Oriented Architectures. In C. Bussler, J. Davies, D. Fensel, and R. Studer, editors, *ESWS*, volume 3053 of *Lecture Notes in Computer Science*, pages 269–283. Springer, 2004.
- [50] A. Martelli and L. Giordano. Reasoning About Web Services in a Temporal Action Logic. In O. Stock and M. Schaerf, editors, *Reasoning, Action and Interaction in AI Theories and Systems*, volume 4155 of *Lecture Notes in Computer Science*, pages 229–246. Springer, 2006.
- [51] D. Martin, M. Burstein, J. Hobbs, O. Lassila, D. McDermott, S. McIlraith, S. Narayanan, M. Paolucci, B. Parsia, T. Payne, E. Sirin, N. Srinivasan, and K. Sycara. OWL-S: Semantic Markup for Web Services. Internet (<http://www.daml.org/services/owl-s/1.0/owl-s.pdf>), 2004.
- [52] J. McCarthy and P. Hayes. Some Philosophical Problems from the Standpoint of Artificial Intelligence. *Machine Intelligence*, 4:463–502, 1969.

BIBLIOGRAPHY

- [53] D. McDermott. DRS: A Set of Conventions for Representing Logical Languages in RDF. Online PDF, 2004.
- [54] S. A. McIlraith and T. C. Son. Adapting golog for composition of semantic web services. In *8th International Conference on Principles of Knowledge Representation and Reasoning*, Toulouse, France, April 2002.
- [55] B. Medjahed and A. Bouguettaya. A Multilevel Composability Model for Semantic Web Services. *IEEE Trans. Knowl. Data Eng.*, 17(7):954–968, 2005.
- [56] R. Miller and M. Shanahan. The Event Calculus in Classical Logic - Alternative Axiomatisations. *Electronic Transactions on Artificial Intelligence*, 3:77–105, 1999.
- [57] S. Narayanan and S. A. McIlraith. Simulation, Verification and Automated Composition of Web services. In *WWW*, pages 77–88, 2002.
- [58] E. Newcomer and G. Lomow. *Understanding SOA with Web Services*, chapter 1. Addison Wesley Professional, December 2004.
- [59] J. Nitzsche, T. van Lessen, D. Karastoyanova, and F. Leymann. BPEL^{light}. In *5th International Conference on Business Process Management*, 2007.
- [60] Object Management Group, Framingham, Massachusetts. *The Common Object Request Broker: Architecture and Specification — Version 2.6*, December 2001.
- [61] M. P. Papazoglou. *Web Services: Principles and Technology*. Pearson, Prentice Hall, 2008.
- [62] M. P. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann. Service-Oriented Computing: State of the Art and Research Challenges. *Computer*, 40(11):38–45, 2007.

BIBLIOGRAPHY

- [63] P. F. Patel-Schneider. A Proposal for a SWRL Extension to First-Order Logic. Proposal, DARPA DAML Program, November 2004.
- [64] J. Peer. A pddl based tool for automatic web service composition. In H. J. Ohlbach and S. Schaffert, editors, *PPSWR*, volume 3208 of *Lecture Notes in Computer Science*, pages 149–163. Springer, 2004.
- [65] Raymond Reiter. The Frame Problem in the Situation Calculus: A Simple Solution (Sometimes) and a Completeness Result for Goal Regression. In *Artificial Intelligence and the Mathematical Theory of Computation: Papers in Honor of John McCarthy*, pages 359–380. Academic Press, 1991.
- [66] D. Roman, U. Keller, H. Lausen, J. de Bruijn, R. Lara, M. Stollberg, A. Polleres, C. Feier, C. Bussler, and D. Fensel. Web Service Modeling Ontology. *Applied Ontology*, 1:77–106, 2005.
- [67] I. Salomie, V. R. Chifu, and I. Harsa. Towards automated web service composition with fluent calculus and domain ontologies. In *iiWAS '08: Proceedings of the 10th International Conference on Information Integration and Web-based Applications & Services*, pages 201–207, New York, NY, USA, 2008. ACM.
- [68] E. Sandewall. *Features and Fluents: The representaiton of knowledge about dynamical systems*. Oxford Science Publications, Oxford, UK, 1994.
- [69] R. Studer, S. Grimm, and A. Abecker, editors. *Semantic Web Services: Concepts, Technologies, and Applications*. Springer, Berlin, Heidelberg, 2007.
- [70] M. Thielscher. Introduction to the Fluent Calculus. *Electronic Transactions on Artificial Intelligence*, 2:179–192, 1998.
- [71] L. M. Vaquero, L. Rodero-Merino, J. Caceres, and M. Lindner. A break in the clouds: towards a cloud definition. *SIGCOMM Comput. Commun. Rev.*, 39(1):50–55, 2009.

BIBLIOGRAPHY

- [72] K. Wilkinson, C. Sayers, H. Kuno, and D. Reynolds. Efficient RDF Storage and Retrieval in Jena2. In *Proc. First International Workshop on Semantic Web and Databases*, 2003.

BIBLIOGRAPHY

Appendix A

CMU OWL-S API

A.1 Overview

The Carnegie Mellon University OWL-S API provides necessary functionalities to create and to manipulate OWL-S ontology files. It was initiated around 2004 and reached its first stable release in June 2005. The project is now part of the SemWebCentral portal and is monitored by Joseph Giampapa, Massimo Paolucci, Naveen Srinivasan, Roman Vaculin and the CMU Software Agents Laboratory in general. The current version is 1.1.3.

The API is written in Java and requires Java version 1.5 or higher. Apache Ant is also necessary in order to compile the project (version 1.6.5 or higher is recommended). The API is free software and can be redistributed and/or modified it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation.

A.1.1 Parsing and Writing OWL-S files

The CMU OWL-S API provides 4 different parsers, one for each OWL-S sub-ontology and one for the upper ontology. These are:

- OWLSProfileParser: used to parse OWL-S Profile files.
- OWLSProcessParser: used to parse OWL-S Service Model files. This parser was used in our implementation.
- OWLSGroundingParser: used to parse OWL-S Grounding files.
- OWLSServiceParser: used to parse OWL-S Service files.

These parsers call specialized functions for each different part of an input file and result in a Jena RDF model that represents the knowledge contained in the input file. These RDF models can, in turn, be exported in their previous form, as OWL-S ontology files using the appropriate writer. There are 4 different writers. These are:

- OWLSProfileWriter: used to write an OWL-S Profile.
- OWLSProcessWriter: used to write an OWL-S Process Model. This writer was used in our implementation.
- OWLSGroundingWriter: used to write an OWL-S Grounding.
- OWLSServiceWriter: used to write an OWL-S Service.

A.1.2 Building and Storing OWL-S Objects

The CMU OWL-S API contains builders which are used to create OWL-S objects. There are two different builders. *OWLS_Object_Builder* is used to create OWL-S objects while *OWLS_Store_Builder* is used to create stores or lists to store OWL-S Objects. These builders are instantiated as using their respective builder factory object, as shown below:

```
OWLS_Object_Builder builder = OWLS_Object_BuilderFactory.instance();  
OWLS_Store_Builder storeBuilder = OWLS_Store_BuilderFactory.instance();
```

When parsing OWL-S files, OWL-S builders are provided with Jena Individual class instances as input and generate corresponding OWL-S objects. For example, if one wants to generate an OWL-S Profile object from a Jena Individual instance representing a Profile, then the following code is necessary:

```
Individual profileInstance = ontModel.getIndividual("http://www.daml.org/services/owls1.1/BravoAir  
Profile profile = builder.createProfile(profileInstance);
```

OWL-S builders are also used in the opposite case, by OWL-S Writers to export ontologies in output files. For example, the following code snippet shows how to generate an Actor object.

```
Actor actor= builder.createActor("BravoAir-reservation");  
actor.setName("BravoAir Reservation department");  
actor.setAddress("Airstrip 2,Teetering Cliff Heights, Florida 12321,USA");  
actor.setTitle("Reservation Representative");  
actor.setPhone("412 268 8780");  
actor.setFax("412 268 5569");  
actor.setEmail("Bravo@Bravoair.com");  
actor.setWebURL("http://www.daml.org/services/daml-s/2001/05/BravoAir.html");
```

OWL-S Stores are used to store OWL-S objects. There are different stores for different OWL-S objects. For instance, to store the inputs of an atomic process we use the InputList store. The following code snippet shows how to create an input store and add inputs (input1 and input2 are Input objects) to it.

```
InputList inputList = storeBuilder.createInputList();  
inputList.addInput(input1);
```

```
inputList.addInput(input2);
```

CMU OWL-S API provides built-in error handling. Error handlers are used to manage the errors thrown during the parsing the OWL-S files. An Error Handler object should implement the `OWLSErrHandler` interface. A default implementation of Error Handler is provided, `DefaultOWLSErrHandler` which stores all the errors generated and can list them. Users can develop customized Error Handlers to suit their application needs.

A.2 Modifications to the CMU OWL-S API

During our implementation, we used the CMU OWL-S API to parse and write OWL-S Service Models. Some minor modifications were necessary so as to ensure a complete and seamless collaboration between the API and our implementation. The modifications are exclusively related to the support of SWRL-FOL conditions and expressions by the API. These are briefly presented in this section.

To support SWRL-FOL, we had to find all parts of the OWL-S API that dealt with SWRL and modify, extend or completely rewrite them so that they would seamlessly work whether the input was an SWRL expression or a SWRL-FOL one. The first modification actually is external to the API, it has to do with the `Expression.owl` subontology that is imported by all OWL-S ontology files. This ontology represents all possible types of OWL-S Expression and Condition elements. We modified that ontology and included specializations of the Expression and Condition classes for SWRL-FOL as well as a `LogicLanguage` element dedicated to SWRL-FOL. These additions are shown here:

```
<LogicLanguage rdf:ID="SWRLFOL">  
  <refURI rdf:datatype="&xsd;#anyURI">&swrlf;</refURI>
```

```
</LogicLanguage>
```

```
<owl:Class rdf:ID="SWRLFOL-Expression">
  <rdfs:subClassOf rdf:resource="http://www.daml.org/services/owl-s/1.1/
                                generic/Expression.owl#Expression"/>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="http://www.daml.org/services/owl-s/1.1/
                                generic/Expression.owl#expressionLanguage"/>
      <owl:hasValue rdf:resource="#SWRLFOL" />
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="http://www.daml.org/services/owl-s/1.1/
                                generic/Expression.owl#expressionBody"/>
      <owl:allValuesFrom rdf:resource="#&rdf;#XMLLiteral"/>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

```
<owl:Class rdf:ID="SWRLFOL-Condition">
  <rdfs:subClassOf rdf:resource="#SWRLFOL-Expression"/>
  <rdfs:subClassOf rdf:resource="http://www.daml.org/services/owl-s/1.1/
                                generic/Expression.owl#Condition"/>
</owl:Class>
```

Given these additions to the Expression.owl class, a series of closely-related mod-

ification were needed in several CMU OWL-S API files. First of all, in the *SemanticWebURI* class, we added a URI instance for SWRL-FOL, pointing to the online URI of the language. Then, in the *OWLSPProcessURI* class, we pointed the Expression and Condition URIs to the modified Expression.owl, instead of the original one.

Two other modifications were necessary for the initial support of SWRL-FOL conditions and expressions. In the *LogicLanguageImpl* class, we added an instance for SWRL-FOL, so that conditions and expressions can have their `expressionLanguage` property point to SWRL-FOL. Moreover, we added the RDF types `SWRLFOL_EXP` and `SWRLFOL_CON` in the *OWLSExpression* class. This is needed because this particular class parses all OWL-S expression and decides the class in which they belong. With this modification, SWRL-FOL conditions and expressions are correctly identified. We also commented out the SWRL parser invocation, since it doesn't support SWRL-FOL extensions.

Finally, some modifications were needed in the classes that deal with writing the OWL-S Service Models. In the *OWLSPProcessWriter* class, an addition of a parameter was necessary for the function that writes the model in an output file. We needed to directly import the namespace prefixes of the ontology, so they were added as an extra input parameter for the function. Also, in the *OWLSPProcessWriterDynamic* class, we had to completely rewrite the functions `writeCondition` and `writeExpression` so that SWRL-FOL expressions and conditions can be written correctly in the output file. We also explicitly import the modified Expression.owl that we mentioned before.