D2.4.6 – A Theoretical Integration of Web Service Discovery and Composition

Marco Pistore (University of Trento)
Paolo Traverso (ITC-IRST)

with contributions from:
Walter Binder (EPFL), Ion Constantinescu (EPFL),
Holger Lausen (UIBK), Axel Polleres (UIBK),
Pierluigi Roberti (ITC-IRST), Michal Zaremba (NUIG)

Abstract.
EU-IST Network of Excellence (NoE) IST-2004-507482 KWEB
Deliverable D2.4.6 (WP2.4)

In this document we propose an approach for combining service discovery, functional-level service composition, and process-level service composition. In particular, we propose a theoretical setting and a software architecture to solve this problem.
Keyword list: semantic Web service discovery, functional level composition of Web service, process level composition of Web service, integration of Web service discovery and composition

Document Identifier	KWEB/2005/D2.4.6A/v1.0
Project	KWEB EU-IST-2004-507482
Version	v1.0
Date	February 8, 2006
State	final
Distribution	public

Knowledge Web Consortium

This document is part of a research project funded by the IST Programme of the Commission of the European Communities as project number IST-2004-507482.

University of Innsbruck (UIBK) - Coordinator
Institute of Computer Science
Technikerstrasse 13
A-6020 Innsbruck
Austria
Contact person: Dieter Fensel
E-mail address: dieter.fensel@uibk.ac.at

École Polytechnique Fédérale de Lausanne (EPFL)
Computer Science Department
Swiss Federal Institute of Technology
IN (Ecublens), CH-1015 Lausanne
Switzerland
Contact person: Boi Faltings
E-mail address: boi.faltings@epfl.ch

France Telecom (FT)
4 Rue du Clos Courtel
35512 Cesson Sévigné
France. PO Box 91226
Contact person : Alain Leger
E-mail address: alain.leger@rd.francetelecom.com

Freie Universität Berlin (FU Berlin)
Takustrasse 9
14195 Berlin
Germany
Contact person: Robert Tolksdorf
E-mail address: tolk@inf.fu-berlin.de

Free University of Bozen-Bolzano (FUB)
Piazza Domenicani 3
39100 Bolzano
Italy
Contact person: Enrico Franconi
E-mail address: franconi@inf.unibz.it

Institut National de Recherche en Informatique et en Automatique (INRIA)
ZIRST - 655 avenue de l'Europe -
Montbonnot Saint Martin
38334 Saint-Ismier
France
Contact person: Jérôme Euzenat
E-mail address: Jerome.Euzenat@inrialpes.fr

Centre for Research and Technology Hellas / Informatics and Telematics Institute (ITI-CERTH)
1st km Thermi - Panorama road
57001 Thermi-Thessaloniki
Greece. Po Box 361
Contact person: Michael G. Strintzis
E-mail address: strintzi@iti.gr

Learning Lab Lower Saxony (L3S)
Expo Plaza 1
30539 Hannover
Germany
Contact person: Wolfgang Nejdl
E-mail address: nejdl@learninglab.de

National University of Ireland Galway (NUIG)
National University of Ireland
Science and Technology Building
University Road
Galway
Ireland
Contact person: Christoph Bussler
E-mail address: chris.bussler@deri.ie

The Open University (OU)
Knowledge Media Institute
The Open University
Milton Keynes, MK7 6AA
United Kingdom
Contact person: Enrico Motta
E-mail address: e.motta@open.ac.uk

Universidad Politécnica de Madrid (UPM)
Campus de Montegancedo sn
28660 Boadilla del Monte
Spain
Contact person: Asunción Gómez Pérez
E-mail address: asun@fi.upm.es

University of Karlsruhe (UKARL)
Institut für Angewandte Informatik und Formale
Beschreibungsverfahren - AIFB
Universität Karlsruhe
D-76128 Karlsruhe
Germany
Contact person: Rudi Studer
E-mail address: studer@aifb.uni-karlsruhe.de

University of Liverpool (UniLiv)
Chadwick Building, Peach Street
L697ZF Liverpool
United Kingdom
Contact person: Michael Wooldridge
E-mail address: M.J.Wooldridge@csc.liv.ac.uk

University of Manchester (UoM)
Room 2.32. Kilburn Building, Department of Computer
Science, University of Manchester, Oxford Road
Manchester, M13 9PL
United Kingdom
Contact person: Carole Goble
E-mail address: carole@cs.man.ac.uk

University of Sheffield (USFD)
Regent Court, 211 Portobello street
S14DP Sheffield
United Kingdom
Contact person: Hamish Cunningham
E-mail address: hamish@dcs.shef.ac.uk

University of Trento (UniTn)
Via Sommarive 14
38050 Trento
Italy
Contact person: Fausto Giunchiglia
E-mail address: fausto@dit.unitn.it

Vrije Universiteit Amsterdam (VUA)
De Boelelaan 1081a
1081HV. Amsterdam
The Netherlands
Contact person: Frank van Harmelen
E-mail address: Frank.van.Harmelen@cs.vu.nl

Vrije Universiteit Brussel (VUB)
Pleinlaan 2, Building G10
1050 Brussels
Belgium
Contact person: Robert Meersman
E-mail address: robert.meersman@vub.ac.be

Work package participants

The following partners have taken an active part in the work leading to the elaboration of this document, even if they might not have directly contributed to writing parts of this document:

École Polytechnique Fédérale de Lausanne
Freie Universität Berlin
National University of Ireland Galway
Universität of Innsbruck
University of Manchester
University of Trento

Changes

Version	Date	Author	Changes
0.90	24.12.05	Marco Pistore	First complete version of the deliverable
1.00	06.02.06	Marco Pistore	Revised version

Executive Summary

This document addresses the problem of providing an end-to-end approach to automatically discover and compose available semantic web services in order to fulfill a given user request specified as a composition goal. This requires the integration of different functionalities, namely web service discovery, and two levels of service composition: (i) functional level composition and (ii) process level composition. Each of these functionalities has been already investigated and applied in isolation to semantic web services (see Deliverable 2.4.2), but their integrated usage is still an open problem.

The problem of automatic discovery of services can be seen as the problem of locating a service that can fulfill some requester objectives. Functional-level composition extends the discovery problem, in case it is not possible to find a single service that can fulfill the composition goal. In functional-level composition, a set of existing services is selected which, combined in a suitable way, can jointly fulfill the composition goal. The process-level composition covers a later phase of the overall composition task and permits to build an executable web service that implements the composition. In this phase we assume that the set of Web services necessary for defining the composition has already been found, and that we have to work out the details of how to interact with them.

This document proposes a theoretical setting and a software architecture to solve the problem of integrating discovery and composition. More precisely, it provides an example describing a scenario where discovery and composition need to be integrated in order to match a customer's request. It investigates existing languages and the underlying theoretical models necessary for defining the key elements of the problems (existing services, composition goal, and executable composite service). Finally, it proposes a software architecture for integrating discovery and composition.

Contents

1 Introduction **1**
1.1 Overview of this document . . . 2

2 Background **4**
2.1 Service Discovery . . . 4
2.2 Functional level composition . . . 5
2.3 Process level composition . . . 6

3 Languages for Integrating Discovery and Composition **8**
3.1 The Virtual Travel Agency Scenario . . . 8
3.2 Modeling the Web Services . . . 11
3.2.1 The WSMO approach . . . 11
3.2.2 The BPEL4WS approach . . . 23
3.3 Modeling the Composition Goal . . . 27
3.4 Modeling the Composite Service . . . 30

4 Theoretical Models **32**
4.1 Models for the Domain Ontology . . . 32
4.2 Models for the Web Services . . . 32
4.3 Models for the Composition Goal . . . 35
4.4 Models for the Composite Service . . . 37

5 An Architecture for Integrating Discovery and Composition **38**
5.1 Overview of the Architecture . . . 38
5.2 Definition of the Interfaces . . . 40
5.3 Core Functionalities . . . 43

6 Conclusions **46**
6.1 Extending the Architecture: Reputation . . . 46

Chapter 1

Introduction

This document addresses the problem of providing an integrated approach to automatically discover, select, and compose available semantic web services into a new, executable web service that matches a given user request specified as a composition goal. The generation of the composed, executable process requires the integration and harmonization of different existing functionalities available for semantic web services, such as discovery, functional level composition and process level composition. Our goal is to propose a theoretical solution and a software architecture that are built on top of these components, and that exploits them in a combined, iterative approach in order to build a composed, executable service.

The automated composition of web services is one of the most promising ideas and — at the same time — one of the main challenges for the taking off of service oriented applications: services that are composed automatically can perform new functionalities by interacting with services that are published on the web, thus significantly reducing the time and effort needed to develop new web based and service oriented applications. It has been widely recognized that one of the key elements for the automated composition of web services is semantics: unambiguous descriptions of web service capabilities and web service processes (e.g., in languages such as OWL-S [Coa03] or WSMO [WSM05]), which provide the ability to reason about web services, and to automate web services tasks, like web service discovery and composition, see, e.g., [MSZ01].

Most of the work on the composition of semantic web services has focused so far on the problem of composition at the *functional level*, i.e., composition by matching preconditions and effects of services described as atomic components, which, given some inputs, return some outputs [PSK02, CFB04]. Functional level composition is usually combined with service discovery techniques, that are exploited to find suitable service instances to be composed [CFB04]. One of the key open problems for semantic web services is to extend functional level composition in order to automatically generate composed web services that can be directly executed to invoke component services to achieve some composite goal. This is a key step in reducing effort, time and errors due to manual composition at the programming level.

The problem of such an *end-to-end integration of discovery and composition* is far from trivial. We need to take into account the fact that, in real cases, component services are not atomic, and it cannot in general be executed in a single request-response step. In general, each component service may be specified as an interaction protocol, where different "atomic" invocations and responses are combined into complex execution patterns. While the details on the exact protocol required to interact with an existing service are not important in discovery upfront, they become essential when we aim at generating composed web services that are executable. For this reason, the composition of executable services needs to deal with descriptions of web services in terms of complex processes, that consist of arbitrary combinations of atomic interactions, in the style, e.g., of OWL-S process models [Coa03] or based on an abstract machine model such as in WSMO interfaces [WSM05].

The approach for an end-to-end integration of discovery and composition proposed in this paper works in two steps. In the first step, that is, at *discovery* time and during the *functional level* composition, it is necessary to identify a set of web services that, interacting with each other, may be able to match the composition request. The focus is on required inputs and provided outputs of the services in order to generate the outputs needed by the user. For instance, it is at this level we discover that a "hotel booking" service and a "flight booking" service are necessary to satisfy a vacation request from a user. In the second step, given the set of selected web services, and given the composition goal, the *process-level* composition is responsible of generating automatically an executable composed web service. For instance, given the process models of two available web services for "hotel booking" and for "flight booking", we aim at generating an executable composed service, say "virtual travel agency". By interacting with the "hotel booking" and "flight booking" services, the composed service books hotel rooms and flight seats according to a specified goal.

This document discusses a reference theoretical setting for solving the problem of integrating discovery and composition, as discussed above. Moreover, it describes a software architecture that re-uses existing approaches for discovery, functional-level composition, and process-level composition. The next step will be the development of a prototype tool based on the approach described in this deliverable.

1.1 Overview of this document

The structure of this deliverable is as follows. In Chapter 2, we briefly recall the basic concepts of web service discovery, functional level composition, and process level composition. Chapter 3 introduces the languages used for describing the different elements that participate to a discovery and composition task. More precisely, we will discuss two languages for defining the component services (WSMO and BPEL4WS), a language for defining composition requirements, and a language for defining the composite service (BPEL4WS). In this chapter, a use case will be used to illustrate the different languages as well as the composition task. In Chapter 4, we discuss the theoretical setting that we exploit for implementing the integrated discovery and composition task. In particular, we define suitable structures for modeling the elements of the composition introduced in Chapter 3. Chapter 5 proposes an architecture for integrating discovery and composition of Semantic Web Services. It describes the main software components on this architecture and the interfaces among them. The goal is to define an architecture that re-uses as much as possible existing approaches and algorithms, in order to allow for a quick implementation of a prototype tool. Finally, Chapter 6 summarizes our results and plans for future work, and describes how the proposed architecture can be extended to take into account aspects related to reputation in the composition task.

Chapter 2

Background

In this section we briefly recall the notions of service discovery, functional level composition, and process level composition. More details can be found in Deliverable 2.4.2 [Lar04a].

2.1 Service Discovery

Web Service Discovery is the process of finding and selecting a suitable web service that can be invoked to match a user's request. Discovery is a complex process that, in the general case, consists of different steps.

Goal Discovery. Starting from a user desire (expressed using natural language or any other means), goal discovery will locate the pre-defined goal that fits the requester's desire from the set of pre-defined goals, resulting on a selected pre-defined goal. Such pre-defined goal is an abstraction of the requester's desire into a generic and reusable goal.

Goal Refinement. The selected pre-defined goal is refined, based on the given requester desire, in order to actually reflect such desire. This step will result in a formalized requester goal.

Service Discovery. Available services that can, according to their abstract capabilities, potentially fulfill the requester goal are discovered. As the abstract capability is not guaranteed to be correct, we cannot assure at this level that the service will actually fulfill the requester goal.

Service Contracting. The services discovered based on their abstract capabilities have an associated contracting capability. This contracting capability will be used in service contracting to determine if the selected service can actually fulfill the requester goal, establishing a contract agreement. If this is the case, the result will be a contracted service.

In the scope of this deliverable, we will focus on *Service Discovery*. In this step, we can assume that existing web services are defined by the following elements:

- its inputs I, i.e., the values they receive when invoked;
- its outputs O, i.e., the values they return to the invoker in case of successful invocation;
- the pre-conditions P, i.e., the assumptions that need to hold (on the inputs and on the status of the world) in order to guarantee a successful execution of the service;
- the effects E, i.e., the changes that occur in case of successful execution.

Also a discovery goal can be defined by a similar tuple of elements: in this case, inputs and pre-conditions express the values and the conditions that are known to the invoker, while the outputs and effects are the expected results of the service invocation.

In this context, if elements $S = (I, O, P, E)$ define a service and elements $g = (I_g, O_g, P_g, E_g)$ define a discovery goal, we say that S matches g (according to the plug-in matching: see [Lar04a] for more details) if the following conditions hold:

- $I \sqsubseteq I_g$ and $P \sqsubseteq P_g$
  (i.e., the inputs requested by the services can be obtained from those known by the invoker, and similarly for the pre-conditions);
- $O \sqsupseteq O_g$ and $E \sqsupseteq O_g$
  (i.e., the outputs requested by the invoker can be obtained from those returned by the service, and similarly for the effects).

2.2 Functional level composition

If there is no single service that is able to fulfil a given goal (no complete match), it may still be possible to select a set of partially matching services that can be composed in the form of a workflow in order to fulfil the goal. We call the process of goal decomposition and service selection *functional-level service composition*.

Functional-level service composition addresses the problem of selecting a set of services that, combined in a suitable way, are able to match a given goal. Each existing service is defined in terms of an atomic interaction, i.e., in terms of its input and output parameters, and possibly also in terms of its preconditions and effects. Functional-level service composition exploits the information that is provided e.g. in an OWL-S service profile or in a WSMO service capability model.

The goal defines the overall functionality that the composed service has to implement, again in terms of its inputs, outputs, preconditions, and effects.

The approach to functional-level service composition proposed in Deliverable 2.4.2 [Lar04a] is based on forward chaining. Informally, the idea of forward chaining is to iteratively select a possible service S and apply it to a set of input parameters provided by a goal G (i.e., all inputs required by S have to be available). If applying S does not solve the problem (i.e., still not all the outputs required by the goal G are available) then a new goal G' can be computed from G and from the outputs generated by S and the whole process is iterated.

In order for a service S to be applicable to the inputs available from a goal G, all of the inputs required by the service S need to correspond to some compatible parameter in the inputs provided by the goal G. This means that the "role" of the goal parameter has to be the same as, or more specific than, that of the service parameter, and also the range of values that the goal parameter can take has to be more specific than that accepted by the service S.

Upon successful functional-level service composition, the selected services are arranged in a workflow that respects the data-dependencies between the services (i.e., constraints on the order in which the services may be executed).

2.3 Process level composition

Given a set of existing Web services $W_1, \ldots, W_n$, the problem of building a process level composition consists of finding a program that interacts with these Web services in a suitable way, in order to achieve a given composition requirement (or composition goal). We call *process-level service composition* the process of generating this program. Let us consider for instance the case of the Virtual Travel Agency, and let us assume that a set of Tourism service providers has been identified for solving a customer request. These services can consists, for instance, of a Flight Booking service (or a Train Journey Booking service) and a Hotel Booking service that are adequate for the specific request of the customer, e.g., the specific destination (the selection of such Web services can be the result of a functional level composition). The goal of process-level composition is to obtain the executable code that invokes these Web services, in order to obtain an offer for the customer's request.

In the definition of the executable code implementing the composition, we need to take into account the fact that, in real cases, booking a hotel is not an atomic step, but requires instead a sequence of operations, including authentication, submission of a specific request, negotiation of an offer, acceptance (or refusal) of the offer, and booking the room. That is, Web services $W_1, \ldots, W_n$ are usually composite, i.e., the interaction with them does not consist of a single request-response step, but they require to follow a complex protocol in order to achieve the required result. Moreover, the steps defining the complex interaction are not necessarily defining a sequence. Indeed, these steps may have conditional, or non-nominal outcomes (e.g., authentication can fail; there may be no offer available from an existing service...) that affect the following steps (no request can be submitted if the authentication fails; if there is no offer available, an order cannot be submitted...). It may also be the case that the same operation can be repeated iteratively, e.g., in order to refine a request or to negotiate the conditions of the offer.

The details on the exact sequence of operations required to interact with an existing service are not essential in discovery. Taking these details into account becomes unavoidable when the executable code implementing the composition has to be generated. For this reason, in process level composition the existing Web services need to be described in terms of complex, composite processes. These processes consist of arbitrary (conditional and iterative) combinations of atomic interactions, and these atomic interactions may have conditional outcomes. (Process level service composition exploits the information that is provided e.g. in an OWL-S service profile or in a WSMO service capability model). As a consequence, also the generated executable code has to be a complex program, since it has to take into account all possible contingencies occurring in the interaction with the Web services.

Automated composition starts from a set of web services, and from a composition requirement, and generates an executable web service which implements the composed service. The synthesis of a composite web service is not limited to atomic component Web services. The output of this component is to define an interaction protocol with the selected services, so that an executable implementation of the composition is obtained. From this point of view the Web service is defined as an activity flow or as an interaction protocol.

Chapter 3

Languages for Integrating Discovery and Composition

In this chapter, we introduce some reference languages used for describing the different elements that participate to a discovery and composition task. To this purpose, we will use an example that will also allow us to define the requirements and direct the definition of the integrated discovery and composition approach. The use case is in the context of e-Tourism services, and consists of the composition of existing transport and accommodation services in order to provide a Virtual Travel Agency service to the end user. We refer to Deliverable 2.4.1 [Lar04b] for further information on this use case.

In order to model semantic aspects of the participating services in the Virtual Travel agency, we will present domain ontologies in WSML. Furthermore, we will relate these semantic descriptions of the domain ontologies to behavioral descriptions of the services. We will show two possible approaches for this: On the one hand, using "classical" Web service technologies, where we express the behavior using BPEL4WS, on the other hand we will show how to describe these behavioral aspects natively in an abstract state machine based model based on WSMO. For the latter, we exploit a syntax based on that of the WSML [WSM05] language. Notice however that several constructs and concepts used in this chapter are not yet part of WSML's official syntax. Different solutions may get adopted from the WSMO group when these concepts will be considered for inclusion in WSML. Moreover, in the examples we sometimes slightly deviate from WSML syntactic constraints and requirements, if this is useful for readability purposes.

3.1 The Virtual Travel Agency Scenario

The Virtual Travel Agency (VTA) is an e-Tourism service provider which offers travel booking services to the end user by using and interacting with other, more basic e-Tourism service providers. The functionality of the VTA is that of a traditional travel agency: getting a request from a customer, dealing with different e-Tourism providers to put together an appropriate offer covering the customer request, arranging all the booking (and payment) with the different providers, and transparently offering the final trip arrangement to the customer. In the context of this deliverable, we assume that the available e-Tourism providers should be located dynamically by the VTA, with no need for prior agreements, and that the business process of the VTA should be composed dynamically based on the request received and the available providers. In the following, we describe the use case in more detail.

Goal/Context. The customer wants to make a trip to a given location (e.g., Paris) for a given period of time (e.g., staying there from August 10 to August 15). The customer sends his request to the VTA, which has to build a package including a travel to/from Paris and an accommodation for all the nights spent in Paris. Clearly, the hotel has to be booked according to the flight (i.e., if the flight arrives on August 9, then the hotel has to be booked from August 9).

The VTA should take care of locating the necessary tourism service providers (e.g., suitable flight providers for the trip, hotels in Paris...) and contact them. Finally, a suitable offer will be returned to the customer and upon acknowledgement either both the accommodation and the travel shall be booked or none, which requires a weak form of transactionality for the composed service.

Participating actors.

- Customer: the end-user that sends a trip booking request to the VTA.
- Tourism service providers: commercial companies that provide specific tourism services.
- VTA: the intermediary between the Customer and the tourism service providers. It provides tourism packages to customers by aggregating the separate services of different tourism service providers.

Scenario/Steps.

1. The user constructs a trip request, including all her requirements and preferences.
2. The user submits the request to the VTA.
3. The VTA receives the request and interprets it.
4. The VTA selects a set of tourism service providers in order to satisfy the received trip request.

5. The VTA generates the executable code necessary to interact with the selected tourism service providers.

6. The VTA executes the generated code, interacting with the selected tourism service providers in order to collect all the information from the tourism service providers, aggregate them and prepare a trip offer.

7. If the interaction with the tourism service providers is successful, the VTA delivers the aggregated offers to the customer. Otherwise, other combinations of tourism service providers are selected (step 4).

8. The customer receives an offer for her trip (or a failure message reporting that no offer is possible).

In this deliverable we focus on steps 4 and 5 of the scenario described above. That is, given a goal encoding the request of the user, we show how it is possible to select a set of suitable tourism services and compose them in order to generate an executable code composing these services according to the goal.

We assume that the terminology used in the travel domain is based on an ontology, which defines the basic domain concepts (trips, accommodations, clients...) This ontology is described in the following, using WSML syntax. For details on ontology modelling in WSML, we refer the reader to [LKMR+05]. The ontology defines the concepts of Client, Location, Trip, and Accommodation, along with their attributes. For instance, a Trip is defined by an unique identifier (this can be the flight, or the train number), by a date, and by start and destination locations. The concepts of a trip being available is formalized as concept TripAvailable, which is defined as a sub-concept of Trip. Similarly for the concepts of a trip being booked, but in this case the client who booked the trip becomes an attribute of TripBooked. Similar concepts are introduces also for the accommodations.

```
wsmlVariant _"http://www.wsmo.org/wsml/wsml−syntax/wsml−flight"

namespace {trv _"http://www.example.org/Travel"}

 [...]

ontology trv#simpleTravelOntology

/* Client doing the travel */
concept trv#Client
  trv#name ofType _string
  trv#gender ofType _string

/* Destination of the travel */
concept trv#Location
  trv#name ofType _string

/* Trips */
concept trv#Trip
     trv#id  ofType (1 1)  _string
     trv#date  ofType _date
     trv#start   ofType trv#Location
     trv#destination  ofType trv#Location
```

```
/* Accommodation */
concept trv#Accommodation
     trv#id  ofType (1 1)  _string
     trv#date  ofType _date
     trv#location  ofType trv#Location

/* Trips/accommodations being available */
concept trv#TripAvailable subConceptOf trv#Trip

concept trv#AccommodationAvailable subConceptOf trv#Accommodation

/* Trips/accommodations being booked */
concept trv#TripBooked subConceptOf trv#Trip
     trv#pax ofType (1 *)  trv#Client

concept trv#AccommodationBooked subConceptOf trv#Accommodation
     trv#pax ofType (1 *)  trv#Client

/* The first date is the user requested date, the second one is
   the trip date, the relation tells us if the two dates are
   compatible (the trip date should contain the user requested date,
   but some additional days can be added before and/or after the
   requested dates, e.g., due to constraints in the flights ) */

relation trv#Compatible(ofType _date, ofType _date)
```

Listing 1: Basic Travel Ontology

3.2 Modeling the Web Services

In the following subsections we describe in detail the behavioral aspects of the different web services implementing some e-service providers, in order to enable composition of such services. More precisely, we assume that there are three such web services available: a simple Flight Booking Service, a Train Booking Service and a Hotel Booking Service which we will describe in two possible ways. First, we will describe how such services could be modeled in an ontology-based machine model which describes the stateful interface of a Web Service, in terms of an ontology enabled abstract state machine. Next, we describe the same model using one of the commonly used standards in Web Service technologies, namely BPEL4WS.

In the subsequent chapter 4 we will show that both these models allow for reductions to finite state machines on top of which efficient process level composition methods can be applied.

3.2.1 The WSMO approach

The behavioral description of a Web Service in WSMO is divided in a functional level description, i.e. the high-level *capability* and an interface description modelling the *choreography interface* of the respective service, i.e. how to interact with the service in a stateful way in order to achieve the desired functionality.

We now introduce the concepts defining the messages which the separate services use to interact with. They are defined by means of ontologies described in WSML. We report the ontology for the Flight service — the ontologies for Train and Hotel are very similar.

```
wsmlVariant _"http://www.wsmo.org/wsml/wsml−syntax/wsml−flight"

namespace {trv _"http://www.example.org/Travel",
           fl  _" http :// www.example.org/BookFlight"}

 [...]

ontology fl#simpleFlightOntology
importsOntology trv#simpleTravelOntology

 concept fl#Flight  subConceptOf trv#Trip
    fl#flightNumber  ofType (1 1) _string

 axiom definedBy
    ?x[fl#flightNumber  hasValue ?fn] memberOf fl#Flight implies ?x[trv#id hasValue ?fn] memberOf trv#Trip.

 concept fl#FlightAvailable  subConceptOf {trv#TripAvailable, fl#Flight}
    fl#seatNumber ofType _string

 concept fl#FlightBooked subConceptOf {trv#TripBooked, fl#Flight}
    fl#seatNumber ofType _string

 //  The following  concepts define the messages received/sent by the Flight service

 concept fl#FlightRequest subConceptOf trv#Trip
     fl#client   ofType trv#Client

 concept fl#FlightNotAvailable subConceptOf trv#Trip

 concept fl#FlightOffer  subConceptOf {trv#Flight,fl#FlightRequest}

 concept fl#FlightConfirm subConceptOf fl#FlightOffer

 concept fl#FlightCancel subConceptOf fl#FlightOffer
```

Listing 2: Ontology for Flight

The ontology for Flight is composed of two parts. In the first part, the basic concepts introduced for generic trips in Listing 1 are refined to the specific case of flights. The unique identifier of the trip is also defined as the flight number. In the second part, additional concepts are introduced that will be used later on as messages in the interactions of the Flight service with its invoker.

WSMO capability descriptions

In WSMO, the capability of a Web Service is described by some non-functional properties and by means of conditions that need to hold before the service can be executed and by the result that is achieved by service execution. Leaving out non-functional aspects, which are out of the scope of this deliverable, WSMO divides these conditions and results into four categories: *preconditions* for defining conditions on the information space (that is the information used for computation) that have to hold before execution, i.e. on the input required by the service; *assumptions* as conditions on the world (denoting aspects not related to computation) that have to hold before execution; *postconditions* define conditions on the information space after execution, i.e. the output of the service, and *effects* as conditions on the world that hold after service execution. These constitutive elements of capability descriptions are expressed as axioms over ontologies. A set of *shared variables* can be declared which are implicitly all-quantified and whose scope is the whole Web service capability. Informally, the logical interpretation of a Web service capability is that for any values taken by the shared variables, the precondition and the assumption imply the postcondition and the effect.

As an example we model here the capability of the Flight Booking Service – again Train and Hotel can be modeled analogously.

```
namespace {trv _"http://www.example.org/Travel",
           fl  _" http :// www.example.org/BookFlight"}

webService fl#BookFlight

 capability fl#BookFlightCap
        sharedVariables {?req, ?date, ?start, ?dest, ?client}
        precondition definedBy
            ?req[
                 trv#date hasValue ?date,
                 trv#start hasValue ?start,
                 trv#destination hasValue ?dest,
                 fl#client hasValue ?client]
             memberOf fl#FlightRequest.

        assumption definedBy
             exists {?flight} ( ?flight [ trv#start hasValue ?start,
                                trv#destination hasValue ?dest,
                                trv#date hasValue ?date] memberOf fl#FlightAvailable).

        postcondition definedBy
              fl#offer (?req)[trv#date hasValue ?date,
                          trv#start hasValue ?start,
                          trv#destination hasValue ?dest,
                          trv#flightNumber hasValue fl#fn(?req),
                          fl#client hasValue ?client] memberOf fl#FlightOffer.

        effect definedBy
             fl#booking(?req)[ trv#date hasValue ?date,
                          trv#start hasValue ?start,
                          trv#destination hasValue ?dest,
                          fl#flightNumber hasValue fl#fn(?req),
                          trv#pax hasValue ?client] memberOf fl#FlightBooked.
```

Listing 3: Flight Booking Service: Capability Level

Notice the different roles of precondition, assumption, postcondition and effect. The precondition is used to express constraints on the inputs the requester should be able to provide to the service. The assumption expresses constraints for a successful execution of the service that the requester cannot control (the fact that a flight is actually available). The postcondition expresses new information made available to the requester after a successful service execution (an offer with relevant information for the requester such as the flight number). The effect, finally, expresses the real world effects of the execution of the web services such as the fact that there exists an actual booking for the requested flight.

The functional dependency between the input (request) and the parameters of the output/effects are modeled by function symbols in this example, e.g. the function symbol booking/1 is dependent on the identifier of the original request ?req). The above capability says that for any request with given date, start and end location, and client a flight offer will be given as output and a booking will be made, under the assumption that a respective flight is available.

We remark that, unfortunately the terms used to denote the four elements used for defining web service capabilities in WSMO are different from the "standard" ones discussed in Section 2.1. Preconditions, in particular denote different elements. WSMO preconditions correspond to inputs, WSMO assumptions to preconditions, WSMO postconditions to outputs, and WSMO effects to effects.

WSMO choreography interface descriptions

At the capability level, no fine grained description of complex interaction patterns with the service is possible. The capability only gives a coarse-grained description of the advertised functionality of the service, leaving out the interaction steps which have to be taken in order to achieve this functionality or the possibility of failure, i.e. the capability level description only covers "successful" interaction sequences in terms of what the service wants to advertise.

However, in order to figure out how to actually execute a possibly complex service, one needs a more fine grained description. This description of the stateful interaction process with the service can be expressed for instance in a workflow-like language such as BPEL4WS. WSMO takes a different approach, where such interactions are described in terms of abstract state machines [BS03].

In the following we briefly recap the Choreography description model of WSMO. For further details of the model used here, we refer to the current draft of the WSMO working group [FSPR05]. The syntax used in this draft (and also in this deliverable) might still be subject to slight changes, but is stable enough to illustrate the used model. We will indicate whenever we deviate from the proposed syntax in [FSPR05] and explain respective extensions which we deem necessary in detail.

Basic Abstract State Machines Abstract State Machines (ASMs for short), formerly known as Evolving Algebras [BS03], provide a means to describe systems in a precise manner using a semantically well founded mathematical notation. The core principles are the definition of ground models and the design of systems by refinements. Ground models define the requirements and operations of the system expressed in mathematical form. Refinements allows the expression of the classical divide and conquer methodology for system design in a precise notation which can be used for abstraction, validation and verification of the system at a given stage in the development process.

Abstract State Machines are divided into two main categories, namely, Basic ASMs and Multi-Agent ASMs. The former express the behavior of a system within the environment. Multi-Agent ASMs allow to express the behavior of the system in terms of multiple entities that are collaborating to achieve a functionality. For the description of a single party behavior, we are merely interested in basic ASMs.

A basic ASM is defined in terms of a state signature plus a finite set of transition rules which are executed in parallel. It may involve non-deterministic behavior. Finite state machines can be viewed as a special case of such basic ASMs.

The state signature for classic ASMs usually simply consists of a set of static and dynamic functions, which can be locally updated by the so called update rules. Dynamic functions are classified in four other categories, namely, *controlled*, *monitored* (or *in*), *interaction* (or *shared*) and *out*. Controlled functions are directly updatable by the rules of the machine M only. Thus, they can neither be read nor updated by the environment. Monitored functions can only be updated by the environment and read by machine M and hence constitute the externally controlled part of the state. Shared functions can be read and updated by both the environment and the rules of the machine M. Out functions can be updated but not read by M, but can be read by the environment. Furthermore, ASMs define the so-called derived functions. These are functions neither updatable by the machine or the environment but which are defined in terms of other static and dynamic (and derived) functions.

The most basic rules are Updates which take the form of assignments (also called function updates) as follows:

$$f(t_1, \dots, t_n) := v$$

where f is an n-ary function (specified by the signature) and the terms $t_1, \dots, t_n$ denote the *location* where the dynamic f is to be updated to the new value v, determining the function value of $f(t_1, ..., t_n)$ in the next state. The location-value pairs $((t_1, \dots, t_n), v)$ are called basic updates and represent a basic unit of state change in the ASM.

More complex transition rules are defined recursively, as follows. (Note that for the sake of clarity, we slightly deviate here from the original syntax used in [BS03].) First, transition rules can be guarded by a Condition as follows:

if $Condition$ then $Rules$ endIf

Here, the $Condition$ is an arbitrary closed first order formula (or any other logic underlying the signature, i.e. the concept of ASMs is decoupled from the underlying logical formalism. Such a guarded transition rule has the semantics that the $Rules$ in its scope are executed in parallel, whenever the $Condition$ holds in the current state.

Next, basic ASMs allow some form of universally quantified parallelism by transition rules of the form

```
forAll $Variable$ with $Condition$ do $Rules$ endForall
```

Here, the $Variable$ is a variable occurring freely in $Condition$, with the meaning that the $Rules[Variable/Value]$ are executed in parallel for all possible bindings of the $Variable$ to a concrete $Value$ such that the $Condition[Variable/Value]$ holds in the current state. Here, $Condition[Variable/Value]$ (and $Rules[Variable/Value]$, resp.) stand for the condition (or rule, resp.) where each occurrence of Variable is replaced by Value.

Similarly, basic ASMs allow for non-deterministic choice by transition rules of the form

```
choose $Variable$ with $Condition$ do $Rules$ endChoose
```

Here, as opposed to the forAll rule, one possible binding of the $Variable$ such that the $Condition$ holds is picked non-deterministically by the machine and the $Rules$ are executed in parallel only for this particular binding.

A single ASM execution step is summarized as follows:

1. Unfold the rules, according to the current state and conditions holding in that state, to a set of basic updates.

2. Execute simultaneously all the updates.

3. If the updates are consistent (i.e. no two different updates update the same location with different values, which means that there must not be a pair of updates $(loc, v), (loc, v') with v \neq v'$), then the result of execution yields the next state.

4. All locations which are not affected by updates, keep their values.

These steps are repeated until no condition of any rule evaluates to true, i.e. the unfolding yields an empty update set. In case of inconsistent updates, the machine run is invalid.

Readability and structure of general ASMs can be improved by introducing so called control states as syntactic sugar. Such control states allow to view ASMs as a straightforward extension of finite state machines and thus have desirable properties like high-level graphical representation and modularization of the machine. A Control State ASM is an ASM with one particular controlled function $ctlState$ (which has as its range a finite number of integers or a finite enumeration of state-descriptors) and each transition rule having the form:

```
if $ctlState = i$ then
    if $cond_1$ then
        $rule_1$
        $ctlState := j_1$
    endIf
    ...
    if $cond_n$ then
        $rule_n$
        $ctlState := j_n$
    endIf
  endIf
```

where $i, j_1, \ldots, j_n$ are control state descriptors and $rule_1, \ldots, rule_n$ are rule sets.

Basically, control state ASMs are FSMs enriched by synchronous parallelism and data manipulation (and thus possibly infinite again). Note that control state ASMs are no expressive restriction of general ASMs but make it easier to define control flow.

WSMO Choreography Model The Choreography part of a service interface describes the behavior of the service from a client point of view; this definition is in accordance to the following definition given by W3C Glossary[1]: *Web Services Choreography concerns the interactions of services with their users. Any user of a Web service, automated or otherwise, is a client of that service. These users may, in turn, may be other Web Services, applications or human beings.*

To this end, based on the basic ASM model, a choreography interface in WSML is described as an ASM which operates on the concepts and relations of a WSML ontology to describe the state signature. In analogy to function classification in ASMs, concepts and relations in WSML choreography interfaces are classified as *in, out, controlled, shared, static*.

The state for the given signature of a WSMO choreography is defined by all legal WSMO identifiers, concepts, relations and axioms. The elements that can change and that are used to express different states of a choreography, are instances of concepts and relations which are used in a similar way to locations in ASMs. These changes are expressed in terms of creation of new instances or changes of attribute values.

In order to link to traditional Web Service interface descriptions such as WSDL, WSMO allows to bind concepts of classes *in*, *out*, and *shared* to be bound to the inputs and outputs of operations in a WSDL description of the respective service.

As opposed to basic ASMs, the most basic form of rules are not assignments, but we deal with basic operations on instance data in ontologies, such as adding, removing and updating instances to the signature ontology. To this end, we define atomic update functions to add, delete, and update instances, which allow us do add and remove instances to/from concepts and relations and add and remove attribute values for particular instances. In WSMO Choreography, these basic updates are defined as a set of fact modifiers which are of three different types.

1. add($fact$)
2. delete($fact$)

[1]`http://www.w3.org/TR/ws-gloss/`

3. update($fact(old => new)$), or simply
4. update($fact$)

A $fact$ can be either a membership fact (x memberOf y, or r(t_1,...,t_n) for an n-ary relation r), respectively), an attribute fact (x[a hasValue y] for concepts or a combination of concept membership and attribute value facts in the form of a WSML molecule abbreviating conjunctions of membership and attribute facts,see [LKMR$^+$05] for details. More complex transition rules are defined recursively, analogous to classical ASMs by if-then, forAll-do and choose-do rules. Note that the current definition in the latest version of [FSPR05] allows only add and delete but we add the update primitive for convenience of modelling.

An update rule of form 3 can take one of the following forms:

- update(r(t_1,...,t_{i-1},$t_{i_{old}} => t_{i_{new}}$, t_{i+1},...,t_n)) which is a shortcut for the set of rules
  delete(r(t_1,...,t_{i-1},$t_{i_{old}}$, t_{i+1},...,t_n))
  add(r(t_1,...,t_{i-1},$t_{i_{new}}$, t_{i+1},...,t_n))
- update(x memberOf $y_{old} => y_{new}$) which is a shortcut for the set of rules
  delete(x memberOf y_{old})
  add(x memberOf y_{new})
- update(x[a hasValue $y_{old} => y_{new}$]) which is a shortcut for the set of rules
  delete(x[a hasValue y_{old}])
  add(x[a hasValue y_{new}])

An update rule of form 4 can take one of the following forms:

- update(r(t_1,...,t_n)) which is a shortcut for the rules
  forall {?x1, ... ,?xn} with ?x1 $\neq t_1$ or ... or ?xn $\neq t_n$ do delete(r(?x1,...,?xn)) endForall
  add(r(t_1,...,t_n))
- update(x memberOf y) which is a shortcut for the rules
  forall ?y1 with ?y1 $\neq y$ do delete(x memberOf ?y1) endForall
  add(x memberOf y)
- update(x[a hasValue y]) which is a shortcut for the rules
  forall ?y1 with ?y1 $\neq y$ do delete(x[a hasValue ?y1]) endForall
  add(x[a hasValue y])

Compared with basic ASMs, in WSMO choreography the following restrictions apply to Conditions and Variables

- A (WSML Full) Condition is a restricted form of WSML logical expressions where all free variables which are not bound be enclosing choose or forAll constructs are interpreted as being existentially quantified
- A WSML Core Condition is a WSML Full logical expression which consists only of molecules built up from memberOf and hasValue atoms and the logical connectives and and or where all unbound variables are existentially quantified (i.e. a condition is a conjunctive query)

For further convenience of modeling, we introduce the following shortcut to model common unconditional nondeterminism which is often used in finite state machines and particularly useful for interface descriptions:

$$rule_1|rule_2|\ldots|rule_n$$

is short for

```
choose ?x with ?x = 1 or ?x = 2 or ?x = n do
    if ?x= 1 then $rule_1$ endIf
    if ?x= 2 then $rule_1$ endIf
    ⋮
    if ?x= $n$ then $rule_1$ endIf
endChoose
```

The semantics of WSMO ASMs is defined analogously to basic ASMs.

The *state* S for the given signature of a WSMO choreography is defined by all legal WSMO identifiers, concepts, relations and axioms. The elements that can change and that are used to express different states of a choreography, are instances of concepts and relations which are used in a similar way to locations in ASMs. These changes are expressed in terms of creation of new instances or changes of attribute values.

A run of a Choreography interface is defined analogously to the runs of basic ASMs in [BS03], i.e. runs are defined as sequences of possible single execution steps:

Possible execution steps are defined by

$$S' = S \setminus \{fact|\mathsf{delete}(fact) \in U\} \cup \{fact|\mathsf{add}(fact) \in U\}$$

where S is the current state, U is a consistent update set, and S' is the resulting state of applying U in S.

We recall that the choreography interface is defined by a set of transition rules R: Let O denote the imported signature ontology(ies) and S denote the current state. We define update sets for (R, S) inductively:

- $U(\mathsf{add}(fact), S) = \mathsf{add}(fact)$
- $U(\mathsf{delete}(fact), S) = \mathsf{delete}(fact)$

- $U(R,S) = \bigcup_{r \in R} U(r,S)$
- $U(\text{if } Cond \text{ then } R, S) = \begin{cases} U(R,S) \text{ if } O \cup S \models Cond \\ \emptyset \text{ otherwise} \end{cases}$
- $U(\text{forall } ?Var \text{ with } Cond \text{ do } R \text{ endForall}, S) =$
  $\{U_{R\theta,S} | \theta \text{ such that } \theta = \{?Var/id\} \text{ where } id \text{ is an identifier such that } O \cup S \models Cond\theta\}$
- $U(\text{choose } ?Var \text{ with } Cond \text{ do } R \text{ endChoose}, S) = \begin{cases} U(R\theta, S) \text{ where } \theta = \{?Var/id\} \text{ with } id \text{ being a} \\ \text{non-deterministically chosen identifier such that } O \cup S \models Cond \\ \emptyset \text{ if } O \cup Scup\{exists?Var(Cond)\} \text{ is unsatisfiable.} \end{cases}$

An update set U is consistent if it does not contain any two elements add($fact$) and delete($fact$) and the resulting state

$$S' = S \setminus \{fact|\text{delete}(fact) \in U\} \cup \{fact|\text{add}(fact) \in U\}$$

is consistent with the signature ontology, i.e. $S' \cup O$ is satisfiable.

We now proceed with describing the choreography interface of the Flight Booking Service. For describing this interface, we use the syntax defined above with another slight extension to [FSPR05] towards control state ASMs, using the new keyword ctl_state to enumerate the possible control state descriptors.

```
interface fl#BookFlightInterface
  choreography
    stateSignature
      importsOntology fl#simpleFlightOntology

    in
      fl#FlightRequest withGrounding _"http ://... ",
      fl#FlightConfirm withGrounding _"http ://... ",
      fl#FlightCancel withGrounding _"http ://... "
    out
      fl#FlightNotAvailable withGrounding _"http ://... ",
      fl#FlightOffer withGrounding _"http ://... "
    shared
      fl#Flight ,
      fl#FlightAvailable ,
      fl#FlightBooked

    ctl_state {fl#start ,fl#offerMade,fl#noAvail ,fl#confirmed,fl#cancelled}

    transitionRules
      if (ctl_state = fl#start ) then
        forall {?req,?date,?start,?dest,?client} with
          ?req[trv#date hasValue ?date,
               trv#start hasValue ?start,
               trv#destination hasValue ?dest,
               fl#client hasValue ?client] memberOf fl#FlightRequest
        do
          if exists {?f} (?f[trv#date hasValue ?date,
                             trv#start hasValue ?start,
                             trv#destination hasValue ?dest] memberOf
                        fl#FlightAvailable ) then
              choose {?fn} with
                exists {?f} (?f[trv#date hasValue ?date,
                               trv#start hasValue ?start,
                               trv#destination hasValue ?dest,
                               fl#flightNumber hasValue ?fn] memberOf
                             fl#FlightAvailable )
              do
                add( fl#offer (?req)[trv#date hasValue ?date,
                                    trv#start hasValue ?start,
                                    trv#destination hasValue ?dest,
                                    fl#flightNumber hasValue ?fn,
                                    fl#client hasValue ?client] memberOf
                     fl#FlightOffer )
                ctl_state := fl#offerMade
              endChoose
            else
              add(fl#notAvailable(?req)[trv#date hasValue ?date,
                                  trv#start hasValue ?start,
                                  trv#destination hasValue ?dest] memberOf
                  fl#FlightNotAvailable )
              ctl_state := fl#noAvail
            endIf
      endForall
    endIf

    if ( ctl_state = fl#offerMade) then
      forall {?offer, ?client} with ( ?offer [ fl#client hasValue ?client]
                                   memberOf {fl#FlightConfirm,fl#FlightOffer}) do
          add(?offer[trv#pax hasValue ?client] memberOf trv#FlightBooked)
          ctl_state := fl#confirmed
        endForall
    endIf

    if ( ctl_state = fl#offerMade) then
      forall {?offer} with ( ?offer memberOf {fl#FlightCancel,fl#FlightOffer}) do
        ctl_state := fl#cancelled
      endForall
    endIf
```

Listing 4: Flight Booking Service: Interface Level

The branching on condition

```
if exists {?f} (?f[trv#date hasValue ?date,
                  trv#start hasValue ?start,
                  trv#destination hasValue ?dest] memberOf
                  fl#FlightAvailable) then
  [. . . ]
else
  [. . . ]
endIf
```

in control state fl#start in the listing above is fully non-deterministic for the service consumer, since it depends on a shared concept of available flights which is only actually known to the service provider. Thus, we could equally reformulate this part of the choreography interface description using the non-deterministic choice operator defined above:

```
if ( ctl_state = fl#start ) then
  forall {?req,?date,?start,?dest,?client} with
         ?req[trv#date hasValue ?date,
          trv#start hasValue ?start,
          trv#destination hasValue ?dest,
          fl#client hasValue ?client] memberOf fl#FlightRequest
  do
    add( fl#offer (?req)[trv#date hasValue ?date,
                       trv#start hasValue ?start,
                       trv#destination hasValue ?dest,
                        fl#flightNumber  hasValue fl#fn(?req),
                         fl#client   hasValue ?client] memberOf fl#FlightOffer)
    ctl_state  :=  fl#offerMade
    |
    add(fl#notAvailable(?req)[trv#date  hasValue ?date,
                             trv#start   hasValue ?start,
                             trv#destination  hasValue ?dest]
       memberOf fl#FlightNotAvailable)
    ctl_state  :=  fl#noAvail
enfIf
```

Listing 5: Hotel Booking Service - unconditionally non-deterministic version

This shorter description contains less semantic information. Moreover, it does not allow to exploit the *assumption* stated in the capability in listing 3, i.e., that we can expect that flights are normally available. Further elaboration of the connection between capability and interface description is on our agenda.

We omit the description of the capability and of the interface level for the Train Booking Service, which we assume to be similar to those of the Flight. We report instead the description of the Hotel Booking Service

```
namespace {trv _"http://www.example.org/Travel",
           htl  _"http :// www.example.org/BookHotel"}

webService htl#BookHotel

capability htl#BookHotelCap
   sharedVariables {?req,?date, ?loc, ?client}
   precondition definedBy
     ?req[trv#date  hasValue ?date,
          trv#location  hasValue ?loc,
          htl#client   hasValue ?client]
     memberOf htl#HotelRequest.

   assumption definedBy
     exists {?hotel} (?hotel[trv#date  hasValue ?date,
                             trv#location  hasValue ?loc]
                    memberOf htl#HotelAvailable).

   postcondition definedBy
     htl#offer (?req)[trv#date  hasValue ?date,
                      trv#location  hasValue ?loc,
                      htl#client   hasValue ?client]
     memberOf htl#HotelOffer.

   effect  definedBy
     htl#booking(?req)[trv#date  hasValue ?date,
                       trv#location  hasValue ?loc,
                       htl#hotelName hasValue htl#hn(?req),
                       trv#pax hasValue ?client]
     memberOf trv#HotelBooked.

   interface htl#BookHotelInterface

     choreography
       stateSignature
         importsOntology htl#simpleHotelOntology

         in
           htl#HotelRequest withGrounding _"http ://... " ,
           htl#HotelConfirm withGrounding _"http ://... ",
           htl#HotelCancel withGrounding _"http ://... "
         out
           htl#HotelNotAvailable withGrounding _"http ://... ",
           htl#HotelOffer withGrounding _"http ://... "
         shared
           htl#Hotel,
           htl#HotelAvailable,
           htl#HotelBooked

       ctl_state {htl#start,htl#offerMade,htl#noAvail,htl#confirmed,htl#cancelled}

       transitionRules
         if (ctl_state = htl#start) then
           forall {?req,?date,?loc,?client} with
             ?req[trv#date hasValue ?date,
                  trv#location hasValue ?loc,
                  htl#client hasValue ?client] memberOf htl#HotelRequest
            do
              add(htl#offer(?req)[trv#date hasValue ?date,
                                  trv#hotelName hasValue ?name,
                                  trv#location hasValue ?loc,
                                  htl#client hasValue ?client] memberOf htl#HotelOffer)
              ctl_state := htl#offerMade
              |
              add(htl#notAvailable(?req)[trv#date hasValue ?date,
                                 trv#location hasValue ?loc] memberOf htl#HotelNotAvailable)
              ctl_state := htl#noAvail
            endForall
         endIf

       if (ctl_state = htl#offerMade) then
         forall {?client,?offer} with (?offer[ htl#client hasValue ?client] memberOf {htl#HotelConfirm,
              htl#HotelOffer})
          do
            add(?offer[trv#pax hasValue ?client] memberOf trv#HotelBooked)
            ctl_state := htl#confirmed
          endForall
       endIf

       if (ctl_state = htl#offerMade) then
         forall {?offer} with ?offer memberOf {htl#HotelCancel,htl#HotelOffer} do
          ctl_state := htl#cancelled
         endForall
       endIf
```

Listing 6: Hotel Booking Service

3.2.2 The BPEL4WS approach

BPEL4WS [ACD+03] provides an operational description of the (stateful) behavior of web services on top of the service interfaces defined in their WSDL specifications. A BPEL4WS description of a service identifies the partners of a service, its internal variables, and the operations that are triggered upon the invocation of the service by some of the partners. Operations include assigning variables, invoking other services and receiving responses, forking parallel threads of execution, and nondeterministically picking one amongst different courses of actions. Standard imperative constructs such as if-then-else, case choices, and loops, are also supported.

In Listing 7 we report the WSDL specification of the Flight Booking service[2] The file starts with a description of the structure messages relevant for the interactions with the service. Then, the WSDL specification defines the invocation and reply operations provided by the service. Operations are collected in port types, that are associated to different communication channels of the flight booking service with its partners. In our example, we define two port types, namely `Flight_PT` for the incoming requests and messages and `Flight_CallbackPT` for the outgoing messages from the flight to the invoking service. Finally, the WSDL specification defines bi-directional links between the flight booking service and its partners. In our case, there is only one of such links, between the flight booking service and the customer invoking it.

Listing 8 reports the BPEL4WS specification that, building on top of the WSDL specification, describes the protocol that one has to follow to interact with the flight booking service. The BPEL4WS specification starts declaring the partners involved in the interaction and the links and roles among them (the WSDL partnerLinkType declarations are exploited here): in our case, the only partner is the client invoking the service. Then, a set of variables is defined. These variables are exploited in the interactions, as containers for incoming and outgoing messages, as well as for the internal computations of the service.

The main part of the BPEL4WS specification consists in the specification of the sequence (or flow) of activities that define the interactions with the service. This description is given from the point of view of the flight booking service. The execution the service starts with the reception of a request message from a client. The flight booking service decides internally whether there are available flights (`switch` statement named `checkAvailability`). We remark that the BPEL4WS code specifies that this internal decision is taken by the flight service, but it does not expose "how" this decision is taken. Indeed, the latter information is not needed to the client for interacting with the flight service.

In case no flight is available, then the flight booking service answers to the client with a `flightNotAvailable` message and terminates. Otherwise, an offer is defined and sent to the client. We remark that the definition of the offer, performed in the `assign` activity, is `opaque`, i.e., the details on how the offer is defined are not exposed. Similarly to the verification of available flights, the details on how an offer is defined are internal to the implementation of the service, and should not be disclosed to the client.

A BPEL4WS specification describes in a very detailed way the interactions that need to be carried out with a web service in order to exploit it. However, this is still not sufficient to allow for the purpose of automatically composing such web service with other services. Indeed, it is necessary to describe also the "semantic" aspects of such interactions. We do this by extending the BPEL4WS specification with "semantic annotations" (see also [PTBM05a]). It is necessary first of all to associate a "semantic" meaning to the input and output operations defined in the WSDL file: this is done in the annotated WSDL

[2]For the sake of readability, we have simplified the WSDL specification and of the following BPEL4WS code, omitting some of the most technical parts.

```
<definitions name="Flight">

    <message name="flightRequestMsg">
       <!-- Flight booking request -->
       <part name="date" type="xsd:string"/>
       <part name="start" type="xsd:string"/>
       <part name="destination" type="xsd:string"/>
       <part name="client" type="xsd:string"/>
    </message>
    <message name="flightOfferMsg">
       <!-- Offer from the flight booking service -->
       <part name="flightNo" type="xsd:integer"/>
    </message>
    <message name="flightNotAvailableMsg">
       <!-- Answer message from the flight booking service if no
            flight is available -->
    </message>
    <message name="flightConfirmMsg">
       <!-- Flight booking confirmation from the requester -->
    </message>
    <message name="flightCancelMsg">
       <!-- Flight booking cancellation, if the requester is not
            interested in the offer received from the booking service -->
    </message>

    <portType name="Flight_PT">
        <operation name="flightRequest">
            <input  message="tns:flightRequestMsg"/>
        </operation>
        <operation name="flightConfirm">
            <input  message="tns:flightConfirmMsg" />
        </operation>
        <operation name="flightCancel">
            <input  message="tns:flightCancelMsg" />
        </operation>
    </portType>

    <portType name="Flight_CallbackPT">
        <operation name="flightOffer">
            <input message="tns:flightOfferMsg" />
        </operation>
        <operation name="flightNotAvailable">
            <input  message="tns:flightNotAvailableMsg" />
        </operation>
    </portType>

    <plnk:partnerLinkType name="FlightRequest_PLT">
        <plnk:role name="FlightRequest_Server">
            <plnk:portType name="tns:Flight_PT"/>
        </plnk:role>
        <plnk:role name="FlightRequest_Client">
            <plnk:portType name="tns:Flight_CallbackPT"/>
        </plnk:role>
    </plnk:partnerLinkType>

</definitions>
```

Listing 7: Definition of the *Flight* service.

```
<process name="Flight">
   <partnerLinks>
      <partnerLink name="client" partnerLinkType="FlightRequest_PLT"
         myRole="FlightRequest_Server" partnerRole="FlightRequest_Client"/>
   </partnerLinks>

   <variables>
      <variable name="req" messageType="flightRequest"/>
      <variable name="offer" messageType="flightOffer"/>
   </variables>

   <sequence name="main">
      <receive operation="flightRequest" variable="req" partnerLink="client"/>
      <switch name="checkAvailability">
         <case name="isNotAvailable">
            <invoke operation="flightNotAvailable" partnerLink="client"/>
         </case>
         <otherwise name="isAvailable">
            <assign name="prepareOffer">
               <copy><from opaque="yes"/>
                  <to variable="offer" part="fl"/></copy>
            </assign>
            <invoke operation="flightOffer" inputVariable="offer" partnerLink="client"/>
            <pick name="waitAcknowledge">
               <onMessage operation="flightConfirm" partnerLink="client"/>
               <onMessage operation="flightCancel" partnerLink="client"/>
            </pick>
         </otherwise>
      </switch>
   </sequence>
</process>
```

Listing 8: The BPEL process of the *Flight* service.

specification reported in Listing 9, that is based on the WSDL-S approach [SVMR05]. One can notice that, while both parts `start` and `client` of a request message have the same type (they are strings), the semantic annotations define their role in the definition of a trip (i.e., start of a trip and the client of the trip).

```
<definitions name="Flight">

    <message name="flightRequestMsg">
       <!-- Flight booking request -->
       <part name="date" type="xsd:string" wssem:modelReference="trv#Trip[date]"/>
       <part name="start" type="xsd:string" wssem:modelReference="trv#Trip[start]"/>
       <part name="destination" type="xsd:string"
                                wssem:modelReference="trv#Trip[destination]"/>
       <part name="client" type="xsd:string" wssem:modelReference="trv#Client"/>
    </message>
    <message name="flightOfferMsg">
       <!-- Offer from the flight booking service -->
       <part name="flightNo" type="xsd:integer" wssem:modelReference="trv#Trip[id]"/>
    </message>
    <message name="flightNotAvailableMsg">
       <!-- Answer message from the flight booking service if no
            flight is available -->
    </message>
    <message name="flightConfirmMsg">
       <!-- Flight booking confirmation from the requester -->
    </message>
    <message name="flightCancelMsg">
       <!-- Flight booking cancellation, if the requester is not
            interested in the offer received from the booking service -->
    </message>

    ....

</definitions>
```

Listing 9: The annotated WSDL definition of the *Flight* service.

A second usage of semantic annotations is to define the outcome of an interaction with a web service. In our example it is clear that a flight has been booked only if the flight is available, the flight service sends an offer, and the client acknowledge the acceptance of the offer. To express this in the BPEL4WS specification, we need to differentiate the possible terminal states of the interactions with the flight booking service. In Listing 10 this is obtained through the semantic annotations associated to the "dummy" `empty` activity which marks the successful terminal state of the interaction. Notice the usage of the `add` operator, already described in the previous subsection for the WSMO approach.

3.3 Modeling the Composition Goal

In this section we describe one of the inputs of the integrated discovery and composition procedure, namely specifying the goal and formalizing the request of the Customer. This goal is defined by the VTA, according to the request of the customer. We do not discuss here in detail how this goal is obtained. A possibility is that the VTA has a set of goals (or

```
<process name="Flight">
   <partnerLinks>
      <partnerLink name="client" partnerLinkType="FlightRequest_PLT"
         myRole="FlightRequest_Server" partnerRole="FlightRequest_Client"/>
   </partnerLinks>

   <variables>
      <variable name="req" messageType="flightRequest"/>
      <variable name="offer" messageType="flightOffer"/>
   </variables>

   <sequence name="main">
      <receive operation="flightRequest" variable="req" partnerLink="client"/>
      <switch name="checkAvailability">
         <case name="isNotAvailable">
            <invoke operation="flightNotAvailable" partnerLink="client"/>
         </case>
         <otherwise name="isAvailable">
            <assign name="prepareOffer">
               <copy><from opaque="yes"/>
                  <to variable="offer" part="fl"/></copy>
            </assign>
            <invoke operation="flightOffer" inputVariable="offer" partnerLink="client"/>
            <pick name="waitAcknowledge">
               <onMessage operation="flightConfirm" partnerLink="client">
                  <empty wssem:effect="forall ?tr with (?tr memberOf trv#TripBooked
                     and ?tr[trv#date hasValue '/req/date',
                        trv#start hasValue '/req/start',
                        trv#destination hasValue '/req/destination',
                        trv#id hasValue '/offer/flightNo',
                        trv#client hasValue '/req/client']) add(?tr)"/>
               </onMessage>
               <onMessage operation="flightCancel" partnerLink="client"/>
            </pick>
         </otherwise>
      </switch>
   </sequence>
</process>
```

Listing 10: The BPEL process of the *Flight* service.

goal patterns) that are associated to the different trip requests that the customer can submit. The goal formalizing the request of the customer is defined on top of this ontology, as follows.

```
compositionGoal BookTrip

      sharedVariables {?date, ?start, ?dest, ?client}

      precondition definedBy
            ?date memberOf _date and
            ?start memberOf trv#Location and
            ?dest memberOf trv#Location and
            ?client memberOf trv#Client

      assumption definedBy
            exists {?t, ?a, ?d} {
                  ?d memberOf _date and trv#Compatible(?date,?d) and
                  ?t [ trv#start hasValue ?start,
                     trv#destination hasValue ?dest,
                     trv#date hasValue ?d] memberOf trv#TripAvailable and
                  ?a [ trv#location hasValue ?dest,
                     trv#date hasValue ?d] memberOf trv#AccommodationAvailable
      effect definedBy
            exists {?t, ?a, ?d} {
                  ?d memberOf _date and trv#Compatible(?date,?d) and
                  ?t [ trv#start hasValue ?start,
                     trv#destination hasValue ?dest,
                     trv#date hasValue ?d,
                     trv#client hasValue ?client] memberOf trv#TripBooked and
                  ?a [ trv#location hasValue ?dest,
                     trv#date hasValue ?d,
                     trv#client hasValue ?client] memberOf trv#AccommodationBooked

      recovery definedBy
            (neg exists ?t [ trv#client hasValue ?client] memberOf trv#TripBooked) and
            (neg exists ?a [ trv#client hasValue ?client] memberOf trv#AccommodationBooked)
```

Listing 11: Composition Goal

The **sharedVariables** correspond to the inputs of the customer and the **precondition** defines conditions on these variables (e.g., their types). The **assumption** defines the condition under which the composition is supposed to complete successfully, returning an offer to the customer — in our example, if there are suitable trip and accommodation available. The **effect** statement defines what is supposed to happen if the execution of the composition is successful — in our case, suitable trip and accommodation have to be booked. Finally, the **recovery**[3] statement defines what is supposed to happen if the execution of the composition is *not* successful — no trips or accommodations have to be booked. According to this recovery statement, if a trip has already been booked, but a failure occurs when booking the accommodation (e.g., since there are no rooms available), then the trip has to be cancelled — otherwise the first clause of the recovery statement would be violated.

We would like to remark that, while the composition goal is defined using a WSML-like syntax, the structure and contents of the goal departs substantially from goals consid-

[3]Recovery goals denote another extension of the syntax defined in [FSPR05]. This feature will proof to be very useful in the context of process-level composition, as described in the next section.

ered in WSMO, which define much less constrained requirements on the services to be discovered (e.g., they do not require the specification of preconditions and assumptions, and do not contain recovery statements). The form of goals adopted here is necessary to represent all requirements necessary for an end-to-end discovery, functional-level and process-level composition of web services.

3.4 Modeling the Composite Service

Starting from the goal and the descriptions of the existing services defined previously, the integrated discovery and composition algorithm is supposed to select a set of tourism service providers (e.g., the Flight Booking Service and the Hotel Booking Service) and to combine them into executable code that the VTA can execute in order to interact with these services and find an offer for the customer's request.

We will now specify a hand-woven composition in terms of an executable specification of an orchestration interface of the composed service. The generation of such an executable orchestration from the given Flight and Hotel Services is the overall target of process level composition. Although we could also use a semantic specification in terms of WSMO Orchestration here, we limit ourselves to describe the composed service directly in terms of executable BPEL4WS, since the syntax for WSMO Orchestration is still under development.

This BPEL4WS specification declares first of all two links with the two partners of the interaction, namely the flight booking service and the hotel booking service. It then defines some variables. One can see that three kinds of variables can be defined: input variables (corresponding to the user's parameters specified by the composition goal), output variables (defining the outputs of the composite service), and additional message variables (used for the interactions with the component services). We remark that the input variables are annotated with references to the precondition variables appearing in the goal. We require that each BPEL4WS input variable is annotated with a semantic reference, that defines the origin of the variable. Output variables can be annotated similarly, if the goal contains postcondition variables. In this case, we require that a BPEL4WS output variable exists for each variable defined in the goal postcondition, so that the value of the goal variable is defined at the end of the execution. In our case, we do not have goal postconditions, so the output variables are not annotated.

```
<process name="VTA">
   <partnerLinks>
      <partnerLink name="Flight" partnerLinkType="FlightRequest_PLT"
         myRole="FlightRequest_Client" partnerRole="FlightRequest_Server"/>
      <partnerLink name="Hotel" partnerLinkType="HotelRequest_PLT"
         myRole="HotelRequest_Client" partnerRole="HotelRequest_Server"/>
   </partnerLinks>
   <variables>
      <!-- INPUT variables -->
      <variable name="start" type="xsd:string" wssem:goalReference="goal#?start"/>
      <variable name="destination" type="xsd:string" wssem:goalReference="goal#?dest"/>
      <variable name="date" type="xsd:string" wssem:goalReference="goal#?date"/>
      <variable name="client" type="xsd:string" wssem:goalReference="goal#?client"/>

      <!-- OUTPUT variables -->
      <variable name="flightNo" type="xsd:integer"/>
      <variable name="hotel" type="xsd:string"/>

      <!-- MESSAGE variables -->
      <variable name="flightReq" messageType="flightRequestMsg"/>
      <variable name="flightOff" messageType="flightOfferMsg"/>
      <variable name="hotelReq" messageType="hotelRequestMsg"/>
      <variable name="hotelOff" messageType="hotelOfferMsg"/>
   </variables>
   <sequence>
      <assign>
         <copy><from variable="start"/><to variable="flightReq" part="start"/></copy>
         <copy><from variable="destination"/><to variable="flightReq" part="destination"/></copy>
         <copy><from variable="date"/><to variable="flightReq" part="date"/></copy>
         <copy><from variable="client"/><to variable="flightReq" part="client"/></copy>
      </assign>
      <invoke operation="flightRequest" variable="flightReq" partnerLink="Flight"/>
      <pick>
         <onMessage operation="flightNotAvailable" partnerLink="Flight">
            <!-- Recovery termination -->
         </onMessage>
         <onMessage operation="flightOffer" variable="flightOff"partnerLink="Flight">
            <assign>
               <copy><from variable="start"/><to variable="hotelReq" part="start"/></copy>
               <copy><from variable="destination"/><to variable="hotelReq" part="destination"/></copy>
               <copy><from variable="date"/><to variable="hotelReq" part="date"/></copy>
               <copy><from variable="client"/><to variable="hotelReq" part="client"/></copy>
            </assign>
            <invoke operation="hotelRequest" variable="hotelReq" partnerLink="Hotel"/>
            <pick>
               <onMessage operation="hotelNotAvailable" partnerLink="Hotel">
                  <invoke operation="flightCancel" partnerLink="Flight"/>
                  <!-- Recovery termination -->
               </onMessage>
               <onMessage operation="hotelOffer" variable="hotelOff"partnerLink="Hotel">
                  <invoke operation="flightConfirm" partnerLink="Flight"/>
                  <invoke operation="hotelConfirm" partnerLink="Hotel"/>
                  <assign>
                     <copy><from variable="flightOff" part="flightNo"/><to variable="flightNo"/></copy>
                     <copy><from variable="hotelOff" part="hotel"/><to variable="hotel"/></copy>
                  </assign>
                  <!-- Successful termination -->
               </onMessage>
            </pick>
         </onMessage>
      </pick>
   </sequence>
</process>
```

Listing 12: The BPEL process of the composed VTA service.

Chapter 4

Theoretical Models

In this chapter we provide a theoretical framework for the integrated discovery and composition of Web services. More precisely, we will provide a formal characterization of the different elements corresponding to the inputs and the outputs of a discovery and composition problem.

4.1 Models for the Domain Ontology

The first element that defines the input for a discovery and composition problem is a reference domain ontology (see Section 3.1). This ontology defines the basic concepts that characterize the domain at hand, and is used as a basis both for the definition of the composition requirement and for the component web services.

Our goal is to keep the domain ontology as simple as possible. For this reason, we use a very elementary ontological language, which is based on the $\mathcal{ALN}$ description logic and on a generalized acyclic TBox [BN03]. In this ontological language, (an excerpt of) the domain ontology of Listing 1 is modeled as follows: One can see that our formal model for the domain ontology consists of the definition of a set of concepts, of the super-concepts they are derived from, and of their attributes. Some of the features in the WSML domain ontology, such as the fact that there is a one-to-one association between trips and trip identifiers, cannot be represented in our ontology model, and are hence omitted from Figure 4.1.

4.2 Models for the Web Services

Web services need to be modeled at two levels of abstraction, namely, at the functional (or capability) level, and at the process (or interface) level.

$$
\begin{array}{rcl}
\mathsf{\_date} & \sqsubseteq & \top \\
\mathsf{\_string} & \sqsubseteq & \top \\
\mathsf{trv\#Client} & \sqsubseteq & \forall \mathsf{name.\_string} \sqcap \forall \mathsf{gender.\_string} \\
\mathsf{trv\#Location} & \sqsubseteq & \forall \mathsf{name.\_string} \\
\mathsf{trv\#Trip} & \sqsubseteq & \forall \mathsf{id.\_string} \sqcap \forall \mathsf{date.\_date} \sqcap \\
 & & \forall \mathsf{start.trv\#Location} \sqcap \forall \mathsf{destination.trv\#Location} \\
\mathsf{trv\#TripAvailable} & \sqsubseteq & \mathsf{trv\#Trip} \\
\mathsf{trv\#TripBooked} & \sqsubseteq & \mathsf{trv\#Trip} \sqcap \forall \mathsf{pax.trv\#Client}
\end{array}
$$

Figure 4.1: The $\mathcal{ALN}$ model for the domain ontology.

Functional level model

At the functional level, a web service W is described by:

- an ontology that extends the domain ontology with the specific concepts of the web service at hand (see, e.g., the Flight ontology of Listing 2);
- a set of inputs, outputs, preconditions, and effects, as described in Section 2.1.

Formally:

Definition 1 *The functional-level description of a web service W for domain D is a tuple $(\Omega_W, I_W, O_W, P_W, E_W)$, where:*

- *Ω_W is an extension of the domain ontology Ω_D;*
- *I_W is a set of assertions of the form $i : C$, stating that v is an (input) individual that belongs to concept C;*
- *O_W is a set of assertions of the form $o : C$, stating that v is an (output) individual that belongs to concept C;*
- *P_W is a condition on Ω_W and I_W that specifies the preconditions of the web service invocation;*
- *E_W is a condition on Ω_W, I_W, and O_W that specifies the effects of the web service invocation.*

For instance, in the case of the flight service, the functional-level description is defined as follows:

- the ontology is:

$$\begin{array}{rcl}
\mathsf{fl\#Flight} & \sqsubseteq & \mathsf{trv\#Trip} \sqcap \forall \mathsf{flightNumber.\_string} \\
\mathsf{fl\#FlightAvailable} & \sqsubseteq & \mathsf{trv\#TripAvailable} \sqcap \mathsf{fl\#Flight} \sqcap \forall \mathsf{seatNumber.\_string} \\
\mathsf{fl\#FlightBooked} & \sqsubseteq & \mathsf{trv\#TripBooked} \sqcap \mathsf{fl\#Flight} \sqcap \forall \mathsf{seatNumber.\_string} \\
\ldots & \ldots & \ldots
\end{array}$$

- The inputs are: ?date : _date, ?start: trv#Location, ?dest: trv#Location, ?client : trv#Clients.
- The output is: ?flight : fl#Flight.
- The precondition corresponds to the assumption that appears in Listing 3.
- The effect corresponds to the effect that appears in Listing 3.

Process level model

At the process level, we model web services as *annotated state transition systems*, i.e., as state transition systems that are marked with semantic annotations. State transition systems model the behavior of the service, while ontological semantic annotations describe the meaning of data the service deals with.

A state transition system models a dynamic system that can be in one of a set of possible *states* (some of which are marked as *initial states*) and can evolve to new states as a result of performing some *actions*. We distinguish actions in *input actions*, *output actions*, and τ. *Input actions* represent the reception of messages, *output actions* represent messages sent to external services, and τ is a special action, called *internal action*, that represents internal evolutions that are not visible to external services. In other words, τ represents the fact that the state of the system can evolve without producing any output, and without consuming any inputs. A *transition relation* describes how the state can evolve on the basis of inputs, outputs, or of the internal action τ.

Definition 2 (State transition system)
A state transition system Σ *is a tuple* $\langle \mathcal{S}, \mathcal{S}^0, \mathcal{I}, \mathcal{O}, \mathcal{R} \rangle$ *where:*

- $\mathcal{S}$ *is the finite set of states;*
- $\mathcal{S}^0 \subseteq \mathcal{S}$ *is the set of initial states;*
- $\mathcal{I}$ *is the finite set of input actions;*
- $\mathcal{O}$ *is the finite set of output actions;*
- $\mathcal{R} \subseteq \mathcal{S} \times (\mathcal{I} \cup \mathcal{O} \cup \{\tau\}) \times \mathcal{S}$ *is the transition relation.*

In an annotated state transition system, we associate to each state a set of *concept assertions* and *role assertions*. This configures a state as the assertional component of a knowledge representation system. The assertions are given in description logic, and the ontology plays the role of the terminological component. Therefore, *concept assertions* are formulas of the form $a : C$ (or $C(a)$) and state that a given individual a belongs to the interpretation of the concept C. *Role assertions* are formulas of the form $a.R = b$ (or $R(a, b)$) and state that a given individual b is a value of the role R for a.

Definition 3 (Annotated state transition system)
An annotated state transition system *is a tuple* $\langle \Sigma, \Omega, \Lambda \rangle$ *where:*

- Σ *is a state transition system,*
- Ω *is an ontology,*
- $\Lambda : \mathcal{S} \rightarrow 2^{\mathcal{A}_\Omega}$ *is the annotation function, where* $\mathcal{A}_\Omega$ *is the set of all the concept assertions and role assertions defined over* Ω*.*

When mapping a web service to an annotated state transition system, component Σ traces the evolution of the "status" of the service interface. More precisely, the states of the state transition system correspond to the activities of the BPEL4WS specification, or to the value of variable **ctl_state** in the case of WSMO. The ontology Ω is that defined in the functional level description of the web service. Finally, the annotation function associates to each state the facts that hold when the execution of the service reaches that state. These facts are obtained by the semantic annotations enriching the WSDL and BPEL4WS specifications, or by the facts that define the state of the WSMO specification according to the rules discussed in Section 3.2.1.

4.3 Models for the Composition Goal

The model of the composition goal has to take into account that the goal needs to be used both for the functional level composition and for the process level composition. For this reason, a goal will define both a set on inputs/outputs/preconditions/effects, necessary for the functional level composition and a formula in a specific goal language that is suitable for process level composition. The reference ontology for defining all these elements is the domain ontology discussed in Section 4.1.

We now give a formal definition of goal conditions on domain ontology Ω: these will be useful for defining both preconditions and effects and the process-level goal language.

Definition 4 (Goal condition)
Let $a : C$ *be a concept assertion and* $a.R = b$ *be a role assertion defined w.r.t. ontology* Ω*. Then a* goal condition *is defined as follows:*

$$p \;=\; a : C \mid a.R = b \mid p \;\textsf{OR}\; p \mid p \;\&\; p \mid \textsf{NOT}\; p$$

We are now ready to define the functional-level part of a composition requirement.

Definition 5 (Composition requirement)
Let Ω be a domain ontology. A functional level composition requirement is a tuple $F = (I, O, P, E)$, where:

- I *is a set of* input assertions $i : C$ *on* Ω*;*
- O *is a set of* output assertions $o : C$ *on* Ω*;*
- P *is a* goal condition *on* Ω*,* I*,* O *that specifies the precondition of the composition;*
- E *is a* goal condition *on* Ω*,* I*,* O *that specifies the effect of the composition.*

In the case of the VTA scenario, all the components discussed previously can easily be extracted from the composition goal presented in Listing 7. We remark that the recovery condition appearing in that listing is not exploited here. Indeed, this condition is used only at the process level, as we will describe below.

The functional-level composition requirement is sufficient for selecting a set of web services, as done in discovery and functional-level composition. However, when one moves to process-level composition, goals need to be extended to express complex requirements that are not limited to reachability conditions (like get to a state where both the flight and the hotel are reserved). Most often, goals need to express recovery conditions and preferences (as in the case of the goal in Listing 7), or temporal conditions (e.g., do not reserve the hotel until you have reserved the flight). These kinds of goals, combining preferences and temporal conditions, can be expressed in the EaGLe language [DLPT02], which can be used to express conditions on the whole behavior of a service, conditions of different strengths, and preferences among different subgoals.

Definition 6 *An EaGLe* composition goal $g \in \mathcal{G}$ *over goal conditions* $p \in \mathcal{P}rop$ *are defined as follows:*

$$\begin{aligned} g := p \mid g \textit{ And } g \mid g \textit{ Then } g \mid g \textit{ Fail } g \mid \textit{Repeat } g \mid \\ \textit{DoReach } p \mid \textit{TryReach } p \mid \textit{DoMaint } p \mid \textit{TryMaint } p \end{aligned}$$

Goal **DoReach** p specifies that condition p has to be eventually reached in a strong way, for all possible non-deterministic evolutions of the state transition system. Similarly, goal **DoMaint** q specifies that property q should be maintained true despite non-determinism. Goals **TryReach** p and **TryMaint** q are weaker versions of these goals, where the plan is required to do "everything that is possible" to achieve condition p or maintain condition q, but failure is accepted if unavoidable. Construct g_1 **Fail** g_2 is used to model preferences among goals and recovery from failure. More precisely, goal g_1 is considered first. Only if the achievement or maintenance of this goal fails, then goal g_2 is used as a recovery or second-choice goal. Consider for instance goal **TryReach** c **Fail DoReach** d. The sub-goal **TryReach** c requires to find a plan that tries to reach condition c. During the execution of the plan, a state may be reached from which it is not possible to reach c. When such a state is reached, goal **TryReach** c fails and the recovery goal **DoReach** d is considered. Goal g_1 **Then** g_2 requires to achieve goal g_1 first, and then to move to goal g_2. Goal **Repeat** g specifies that sub-goal g should be achieved cyclically, until it fails. Finally, goal g_1 **And** g_2 requires the achievement of both subgoals g_1 and g_2.

In the case of the VTA scenario, the process level composition goal consists of a main goal condition E and of a recovery condition R (see again Listing 7). In this case, the process-level composition goal is the EaGLe formula **TryReach** E **Fail DoReach** R, which expresses the requirement that the composite service should try (do its best) to achieve the main goal condition E, and, if this comes out to be impossible, guarantee to achieve at least the recovery condition R.

4.4 Models for the Composite Service

The last element for which we have to provide a model is the output of the integrated discovery and composition algorithm, namely the composite service. In the approach we propose, we simply model the composite services as a (non-annotated) state transition system (Definition 2).

Chapter 5

An Architecture for Integrating Discovery and Composition

In this chapter we discuss an architecture for integrating discovery and composition of Semantic Web Services. We start from an high-level overview of the architecture, then we focus on the description of the interfaces among the different components and on the definition of the core functionalities carried out by these components.

5.1 Overview of the Architecture

The architecture is depicted in Figure 5.1. It consists of three modules, corresponding to service discovery (SD), functional level composition (FLC) and process level composition (PLC). According to this architecture, the approach works in two phases. During the first phase, SD and FLC are interleaved in order to find a set of web services that, once composed, are able to satisfy the customer's goal. The second phase is entered once this set of web services has been found: PLC is called to generate the actual executable code implementing the composition. We will now give a more detailed description of the three blocks in the architecture.

"Service Discovery" Component.
Objective: Find a web service that matches a query.
Task: Mediate the accesses to the directory by caching existing results and matching new queries to already discovered services.
Input: Discovery query.
Uses: Web service capability descriptions in the directory.
Output: Web service that matches the query (or failure if no web service is found).

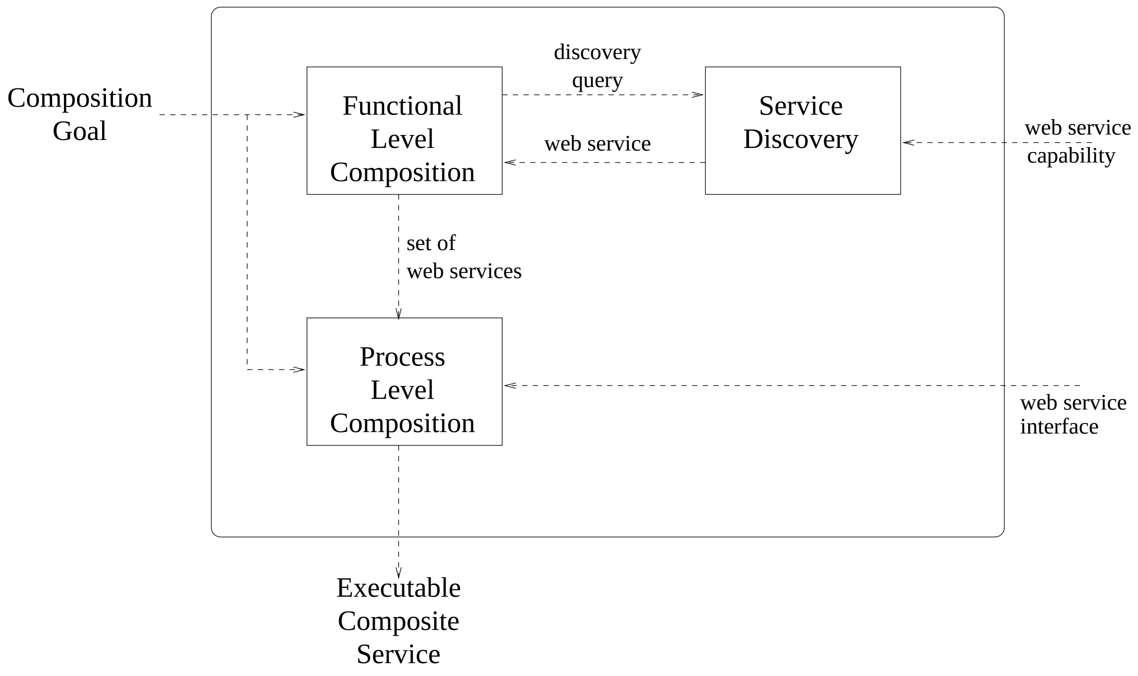

Figure 5.1: Schema Integration Discovery and Composition.

“Functional Level Composition” Component.
Objective: Find a set of Web service that match the composition goal.
Task: Progressively transform the composition goal into a set of web services matching it. Forward/backward chaining techniques like the one proposed in [CFB04] can be adopted here.
Input: Composition goal.
Output: Set of web service matching the goal (or subgoal).

“Process Level Composition” Component.
Objective: Build an executable composite web service.
Task: Given a set of web services matching a composition goal, generate the executable code that, once executed, interacts with the component services and achieves the goal. services.
Input: set of web services, composition goal.
Uses: Web service interface descriptions in the directory.
Output: Executable composite web services.

Figure 5.1 focuses of the data flow among the different components. The control flow is complex due to the necessity of managing failures in achieving the composition and backtracking of previous choices. For example, consider the SD component: it usually identifies several Web services able to match a given discovery query, however only one of them is returned to the FLC. In case the FLC component fails to find a suitable set of Web services, however, the control can be returned to the SD component and a different

Web service matching the goal can be considered. Similarly, if PLC is not able to generate the executable process satisfying the goal given a certain set of web services, the control can return to the FLC component, which can compute an alternative set of Web services solving the same FLC problem. An interesting issue is to define the relevant information to be sent back from PLC to FLC and from FLC to SD in order to direct the backtracking process.

An issue left open by the current architecture, which has to be addressed in the future work, is the definition of criteria for directing the SD and FLC components in the selection of the set of Web services, so that high-quality composite services are obtained. The problem here is that, while the quality of the composition can be evaluated only once the executable service has been composed by the PLC component, this quality reflects the choices done in FLC and SD.

Finally, an interesting extension of the proposed architecture consists of taking into account non-functional requirements such as QoS, costs, security... in the selection and composition of the services.

5.2 Definition of the Interfaces

In this section we define in a more precise way the interfaces among the different blocks described in Figure 5.1. We start by defining some basic data types, which we do not further describe in this document.

URI
Description: an unique identifies (e.g., the URL) for web services and associated BPEL4WS processes.

OntologyExpression
Description: a formula defining a condition in the $\mathcal{ALN}$ language.

EaGLeExpression
Description: a formula defining a condition in the EaGLe language.

We now define one of the key data types, namely that of ontology.

Ontology
Description: an ontology in the $\mathcal{ALN}$ language.
Methods:

- *getConcepts(): Set of Objects* – returns the sets of concepts of the ontology

- *getConceptDefinition(c: Object): OntologyExpression* – returns the definition of a concept

Different other data types are built on top of the Ontology data type. The simplest one is that of assertion.

Assertion
Description: an assertion of the form $i : C$
Attributes:

- *o: Ontology* – the underlying ontology
- *i: Object* – the individual i
- *c: Object* – the concept C for the individual; *c* belongs to *o.getConcepts()*

We are ready to define the data types for the discovery and for the composition requirements.

DiscoveryGoal
Description: the discovery requirement
Attributes:

- *DO: Ontology* – the domain ontology
- *I : Set of Assertion* – the input assertions on *DO*
- *O : Set of Assertion* – the output assertions on *DO*
- *P : OntologyExpression* – the goal precondition
- *E : OntologyExpression* – the goal effect

CompositionGoal
Description: the composition requirement
Attributes:

- *DO: Ontology* – the domain ontology
- *I : Set of Assertion* – the input assertions on *DO*
- *O : Set of Assertion* – the output assertions on *DO*
- *P : OntologyExpression* – the goal precondition
- *E : OntologyExpression* – the goal effect
- *G : EaGLeExpression* – the process level composition goal

We now define the data type describing a web service at the functional level.

WebService
Description: the definition of a Web service at the functional level
Attributes:

- *id: URI* – unique identifier of the Web service
- *DO: Ontology* – the domain ontology
- *SO: Ontology* – the service ontology
- *I : Set of Assertion* – the input assertions
- *O : Set of Assertion* – the output assertions
- *P : OntologyExpression* – the service precondition
- *E : OntologyExpression* – the service effect

The next date structures correspond to state transition systems and annotated state transition systems, respectively.

STS
Description: the definition of the state transition system associated to a (component or composite) service *Attributes*:

- *S: Set of Object* – the states of the STS
- ...

Notice that we did not describe a state transition system in detail, we only described explicitly one of its components, namely the set of states. We remark that, from a conceptual point of view, it consists of the components described in Definition 2, however efficient encodings of such structures, such as the ones exploited in [PTBM05b, PMBT05], are necessary to keep small the size of the objects and to allow for an efficient process level composition.

ASTS
Description: the definition of the annotated state transition system associated to a component service *Attributes*:

- *id: URI* – unique identifier of the Web service interface
- *sts : STS* – the underlying STS
- *o : Ontology* – the service ontology
- *l : Map of Object to OntologyExpression* – the labeling function; the domain consists of the states in *sts.S*

5.3 Core Functionalities

We now define the core functionalities for our integrated discovery and composition architecture. First of all, we define some functionalities that are necessary to access to the description of a given service. In particular, the following two functions give access to the capability level and process level model of a Web service.

getFunctionalDescription(ws: URI): WebService
Description: gets the functional-level (or capability level) description of a Web service
Parameters:

- *ws: URI* – the unique identifier of the Web service

Returns:

- *WebService* – the capabilities of the Web service

getProcessDescription(ws: URI): ASTS
Description: gets the functional-level (or capability level) description of a Web service
Parameters:

- *ws: URI* – the unique identifier of the Web service

Returns:

- *ASTS* – the process level interface of the Web service

We remark that the previous two functions are responsible to generate the models that we exploit for the representation of Web services from the published description of these web services. Since the published description of the Web services can be in different languages (e.g., WSMO choreography or BPEL4WS for the process level description), these functions have different implementations, depending on the specific language (we will have, e.g., getProcessDescriptionBPEL and getProcessDescriptionWSMO).

The next function describes the core functionality of the discovery component.

search(g: DiscoveryGoal): Set of URI
Description: gets a set of services matching a given discovery goal
Parameters:

- *g: DiscoveryGoal* – the discovery goal

Returns:

- *Set of URI* – a set of Web services, each of which matches the discovery goal; each service is described by its unique identifier

Remark: The set of services need not be exhaustive, i.e., there might be services matching the goal that are not returned by the function. The exact details of which matching services are returned depends on the specific implementation of the discovery component.

The following definition corresponds to the functional level composition block.

FLC(g: CompositionGoal): Set of URI
Description: finds a set of services that jointly satisfy a composition goal
Parameters:

- *g: CompositionGoal* – the composition goal

Returns:

- *Set of URI* – a set of Web services that, if combined in a suitable way, satisfy the composition goal at the functional level; each service is described by its unique identifier

Finally, the function for process level composition.

PLC(stss: Set of ASTS, g: CompositionGoal): STS
Description: generates the process level composition of a set of annotated state transition systems *Parameters:*

- *stss: Set of ASTS* – the set of annotated state transition systems
- *g: CompositionGoal* – the composition goal

Returns:

- *STS* – the state transition system implementing the composite service

Remark: the set if URI returned by function FLC has to be converted into a set of STS before it can be passed to function PLC.

The last function we present here is responsible of deploying the composite service.

deploy(sts: STS)
Description: deploys the state transition system for the composite service *Parameters:*

- *sts: STS* – the state transition system implementing the composite service

Remark: the implementation of this function depends on the target language for the composite service. For this reason, we may have different implementations of this function, such as deployBPEL, deployJAVA, etc.

Chapter 6

Conclusions

The automatic creation of an executable web service starting from a user desire, is expected to have a great impact in areas of e-Commerce and Enterprise Application Integration, as it can enable dynamic and scalable cooperation between different systems and organizations.

An important step towards dynamic and scalable integration is understanding how discovery and composition could be used to build an executable web service. The schema proposed in Chapter 5 shows the logical relationship between the components of this integrated approach, and how it is possible to combine them to find a web service that matches a goal, if necessary to refine the goal and finally to compose the web services in a unique executable process.

In this document we have focused on describing suitable languages and theoretical models for such an integration of discovery and composition. Moreover, we have proposed an architecture that supports this integration. Future work will include the implementation of an integrated web service discovery and composition algorithm, based on the concepts and on the architecture discussed in this document. A prototype implementation is planned to be completed within month 30.

6.1 Extending the Architecture: Reputation

In this section we discuss an extension of the integrated discovery and composition architecture discussed previously, which takes into account reputation aspects in the selection of web services to be composed.

In an open environment where malicious parties may advertise false service capabilities the use of reputation services is a promising approach to mitigate such attacks. Misbehaving services receive a bad reputation (reported by disappointed clients) and will be avoided by other clients. Reputation mechanisms help to improve the global efficiency of the overall system because they reduce the incentive to cheat [Bir01]. Studies show that buyers seriously take into account the reputation of the seller when placing their bids in online auctions [HW01]. Moreover, it has been proven that in certain cases reputation mechanisms can be designed in such a way that it is in every party's interest to report correct reputation information (incentive compatible reputation services) [JF04]. Besides, reputation mechanisms can be implemented in a secure way [JF03].

A detailed description of existing reputation mechanisms and of their application to semantic web services is outside the scope of this document. The interested reader can find detailed information in Deliverable 2.4.9 [JFB05]. Here we outline a simple approach to integrate reputation mechanisms into the process of service selection.

We will provide a reputation web service that allows to query the reputation of other services and to submit reports. The reputation web services will have an extensible architecture, allowing to plugin and deploy different concrete reputation mechanisms for different example scenarios.

Integrating reputation mechanisms into the discovery process allows to filter out services that have a bad reputation and to rank matching services according to their reputation. Hence, we will provide a wrapper to the discovery component that first forwards a query to the standard discovery component and afterwards obtains the reputation of the discovered services by accessing the reputation web service. Based on the reputation of the discovered services, certain services may be removed from the result (if the reputation is below a given threshold) or the order of the discovered services in the result may be changed according to reputation (services with higher reputation come first). Figure 6.1 illustrates our approach.

This approach has the advantage that it does not require any changes to the integrated discovery and composition architecture in Figure 5.1. The reputation mechanisms are well encapsulated within the reputation web service. The discovery and composition components are fully functional without the reputation web service, which can be integrated by simply installing the aforementioned wrapper for the discovery component.

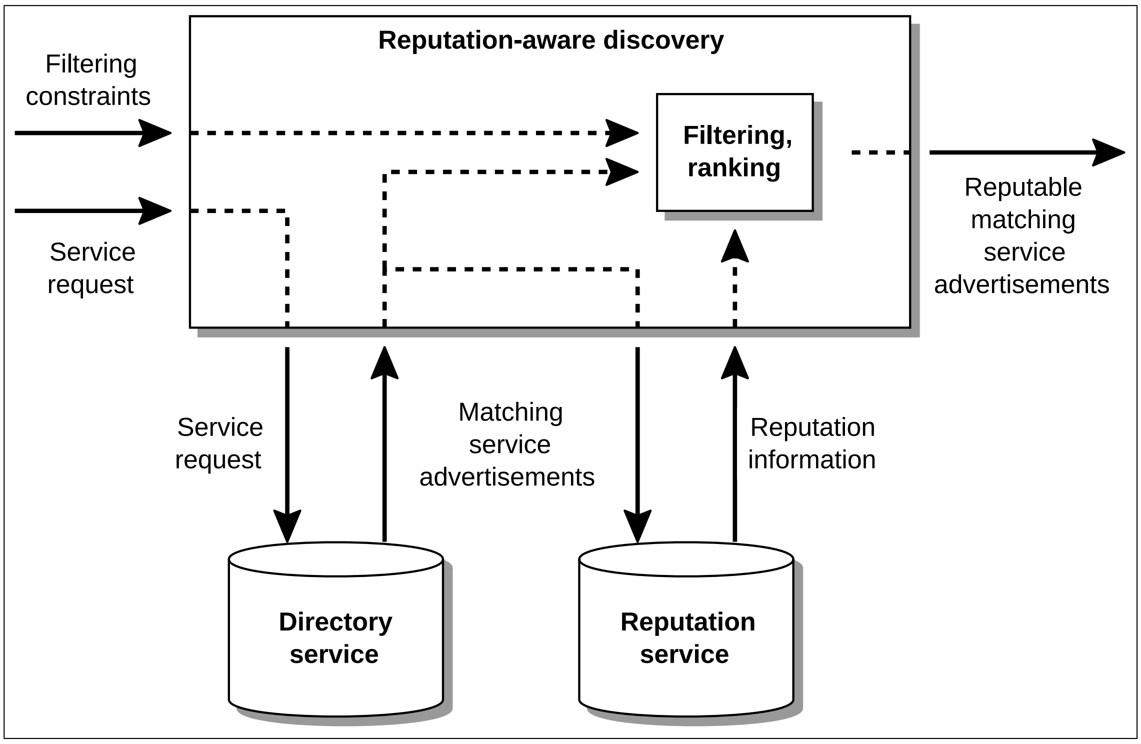

Figure 6.1: A wrapper for service discovery filters and ranks matching service advertisements according to their reputation.

Bibliography

[ACD$^+$03] T. Andrews, F. Curbera, H. Dolakia, J. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weeravarana. Business Process Execution Language for Web Services (version 1.1), 2003.

[Bir01] A. Birk. Learning to Trust. In R. Falcone, M. Singh, and Y.-H. Tan, editors, *Trust in Cyber-societies*, volume LNAI 2246, pages 133–144. Springer-Verlag, Berlin Heidelberg, 2001.

[BN03] F. Baader and W. Nutt. Basic Description Logics. In F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. Patel-Schneider, editors, *The Description Logic Handbook*, pages 43–95. Cambridge University Press, 2003.

[BS03] Egon Börger and Robert Stärk. *Abstract State Machines*. Springer, 2003.

[CFB04] I. Constantinescu, B. Faltings, and W. Binder. Typed Based Service Composition. In *Proc. WWW2004*, 2004.

[Coa03] The OWL Services Coalition. OWL-S: Semantic Markup for Web Services. In *Technical White paper (OWL-S version 1.0)*, 2003.

[DLPT02] U. Dal Lago, M. Pistore, and P. Traverso. Planning with a Language for Extended Goals. In *Proc. AAAI'02*, 2002.

[FSPR05] Dieter Fensel, James Scicluna, Axel Polleres, and Dumitru Roman. Ontology-based choreography and orchestration of WSMO services. Deliverable d14v0.2, WSMO, 2005. Available from http://www.wsmo.org/TR/d14/v0.2/20051008/.

[HW01] D.E. Houser and J. Wooders. Reputation in Internet Auctions: Theory and Evidence from eBay. University of Arizona Working Paper #00-01, 2001.

[JF03] R. Jurca and B. Faltings. An Incentive-Compatible Reputation Mechanism. In *Proceedings of the IEEE Conference on E-Commerce*, Newport Beach, CA, USA, 2003.

[JF04] R. Jurca and B. Faltings. “CONFESS”. An Incentive Compatible Reputation Mechanism for the Online Hotel Booking Industry. In *Proceedings of the IEEE Conference on E-Commerce*, San Diego, CA, USA, 2004.

[JFB05] Radu Jurca, Boi Faltings, and Walter Binder. Reputation mechanism. Knowledge Web Deliverable D2.4.9, 2005.

[Lar04a] Ruben Lara. Definition of semantics for web service discovery and composition. Knowledge Web Deliverable D2.4.2, 2004.

[Lar04b] Ruben Lara. Semantic requirements for web services description. Knowledge Web Deliverable D2.4.1, 2004.

[LKMR$^+$05] Holger Lausen, Uwe Keller, Francisco Martin-Recuerda, Jos de Bruijn, , and Michael Stollberg. A conceptual and formal framework for semantic web services. Knowledge Web Deliverable D2.4.5, 2005.

[MSZ01] S. McIlraith, S. Son, and H. Zeng. Semantic Web Services. *IEEE Intelligent Systems*, 16(2):46–53, 2001.

[PMBT05] M. Pistore, A. Marconi, P. Bertoli, and P. Traverso. Automated Composition of Web Services by Planning at the Knowledge Level. In *Proc. IJCAI’05*, 2005.

[PSK02] M. Paolucci, K. Sycara, and T. Kawamura. Delivering Semantic Web Services. In *Proc. WWW2003*, 2002.

[PTBM05a] M. Pistore, P. Traverso, P. Bertoli, and A. Marconi. An approach for the automated composition of BPEL processes. In *Proc. Workshop on WWW Service Composition with Semantic Web Service (WSCOMPS 2005)*, 2005.

[PTBM05b] M. Pistore, P. Traverso, P. Bertoli, and A. Marconi. Automated Synthesis of Composite BPEL4WS Web Services. In *Proc. ICWS’05*, 2005.

[SVMR05] A. Sheth, K. Verna, J. Miller, and P. Rajasekaran. Enhacing Web Service Descriptions using WSDL-S. In *EclipseCon*, 2005.

[WSM05] Web service modeling ontology (WSMO), June 2005. W3C member submission. Available at http://www.w3.org/Submission/WSMO/.