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

Roberti Pierluigi (ITC-IRST)
Marco Pistore (University of Trento)

with contributions from:
Walter Binder (EPFL), Ion Constantinescu (EPFL)
Axel Polleres (UIBK), Holger Lausen (UIBK),
Paolo Traverso (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 a possible theoretical approach for combining service discovery, functional-level service composition, and process-level service composition. We analyse the challanges in composition of these functionalities (integration of Web service discovery and composition) and propose a theoretical way to solve them.
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	August 10, 2005
State	draft
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
0.1	06.06.05	Pierluigi Roberti	Document Creation
0.15	08.06.05	Pierluigi Roberti	Section 1.2 added
0.20	10.06.05	Pierluigi Roberti	Chapter 6 and first content Section 1.1 added
0.25	12.06.05	Pierluigi Roberti	Section 4.1 added
0.30	13.06.05	Pierluigi Roberti	Chapter 3 and Section 2.1 added
0.35	15.06.05	Walter Binder	Section 4.1 and Section 5.2 added
0.45	22.06.05	Pierluigi Roberti	Section 4.4 and Section 4.5 added
0.50	02.08.05	Marco Pistore	Changed the structure, revised sections 3 and 4
1.00	10.08.05	Marco Pistore	Revision after reviews' feedback

Executive Summary

This document addresses the problem of providing an end-to-end approach to automatically discover, select, contract 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 has to extend the discovery problem, in case a single service that can fulfill the requester goal cannot be found, by selecting a combination of existing services that can fulfill it, based on their functional descriptions (i.e. capabilities).

The process-level composition covers a later phase of the overall composition task and permits to build an actually executable web service, also called orchestration. 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. The goal is to obtain the executable code that implements the composition.

This document is a first step towards the integration of discovery and composition, which focuses on setting the requirements for such an integration. 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 also defines a reference architecture for integrating discovery and composition. The definition of the theoretical framework underlying the integrated discovery and composition approach, as well as the investigation of the techniques necessary to support this approach, is the objective of an extended version of this deliverable, due by month 24 of the project.

Contents

Chapter 1

Introduction

This document addresses the problem of providing an integrated approach to automatically discover, select, contract 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 that is built on top of these components, and that exploits them in a combined, iterative approach, to build a composed, executable web service that is able to match the user's request.

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 services 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]. One of the key open problems for semantic web services is to combine discovery and 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 *integrating 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, process-level composition needs to deal with descriptions of web services in terms of complex, composite 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].

As a consequence, 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.

Given the set of selected web services, and given the composition goal, the “process-level” composition phase 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 flights seats according to a specified goal.

We can identify some critical issues related to both the different parts and the complete integrated approach. For the service discovery it is important to enable a mechanization of this service: the reason is to allow automatically locating and contracting available services to perform a given business activity. For an integrated approach we need a flexible integration of the discovery service in order to have a dynamic selection of web service available to cover the different (discovery) goal. If service discovery is not able to find a service that matches the user requirements, it may be still possible to compose (integrate) several services to provide the required functionality. In our open environment, we invoke service discovery recursively. The programmer of a composed service has to know which basic services are available, and because of the openness of the environment, the set of available services changes continuously.

For automated service composition, the service composition engine must have an up-to-date view of the available services. As the number of published services may be extremely large (assuming the wide-spread acceptance and adoption of semantic web services), the service composition engine may not be able to maintain a copy of all published service descriptions. Hence, the service composition engine has to dynamically interact with service directories to get back services when needed.

Once a functional service composition has been computed, we are interested in the concrete interactions between the different services. Typically, a service consists not only of a single function, so it may be necessary to invoke several times in a specific order the functions (methods) of a particular Web service. The work in the area of automatic discovery and composition is being applied to Web services in order to keep the intervention of the human user to the minimum. Semantic mark-up can be exploited to automate the tasks of discovering services, executing them, composing them and enabling seamless interoperation between them, thus enabling intelligent Web services.

1.1 Overview of this document

This document is a first step towards the integration of discovery and composition.

In particular, it provides an example describing a scenario where discovery and composition need to be integrated in order to match a customer's request. It also defines a reference architecture for integrating discovery and composition.

In Chapter 2 we briefly recall the basic concepts of web service discovery, functional level composition, and process level composition.

In Chapter 3 we introduce a use case that will be used to define the requirements and direct the definition of the integrated discovery and composition approach.

In Chapter 4 we discuss an architecture for integrating discovery and composition of Semantic Web Services.

Finally, our results and plans for future work are summarized in Chapter 5. Future work will include the definition of the theoretical framework underlying an integrated discovery and composition and the implementation of an integrated web service discovery and composition algorithm.

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*.

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 $W_1, \ldots, W_n$,, and a list of constrains between them, the problem of building a process level composition consist of finding a program that interacts with these Web services in a suitable way, in order to achieve a given composition requirement (goal of composition) and to follow the constraints. We call the process of combine the different service selected *process-level service composition*. 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 an 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, that consist of arbitrary (conditional and iterative) combinations of atomic interactions, and there 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. For this point of view the Web service is defined as an activity flow or as an interaction protocol.

In Deliverable 2.4.2 [Lar04a] we distinguished three steps to achieve process level composition:

1. **Processing the Component Web Services**: This step consists of acquiring the process-level descriptions of the existing component Web services, and analyzing them for what concerns the interaction protocols that they implement.

2. **Synthesis of the Composition**: During this step, the process implementing the composition of the Web services is automatically generated staring from the outcome of step 1 and from the composition requirement.

3. **Deployment and Execution of the Composed Service**: In this step, the process generated in step 2 is translated into executable code and deployed on a web service application engine.

Chapter 3

Example scenario: Virtual Travel Agencies

In this section, we present an example that will be used 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 in 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.

To describe the example 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 part of WSML. Different solutions may get adopted from the WSMO group when these concepts will be considered for inclusion in WSML. Moreover, in the examples we freely overrules WSML syntactic constraints and requirements, if this was useful for readability purposes.

3.1 The Virtual Travel Agency Scenario

The Virtual Travel Agency (VTA) is an e-Tourism service provider which offers a 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 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 in 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.

Participating actors.

- Customer: the end-user that requests a trip booking 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 his 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 a 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 his 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 service providers and compose them in order to generate an executable code composing these services according to the goal.

3.2 Discovery/Composition Goal

In this section we describe one of the inputs of the integrated discovery and composition procedure, namely the goal specifying 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 goal patterns) that are associated to the different trip requests that the customer can submit.

The goal is based on an ontology, which defines the basic concepts in the travel domain (trips, accommodations, clients...) This ontology is described in the following, exploiting the WSML syntax.

```
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 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 Compatible(ofType _date, ofType _date)
```

Listing 1: Basic Travel Ontology

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 2: Composition Goal

In brief, 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 the 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** 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 be defined the composition goal using a WSML-like syntax, the structure and contents of the goal departs substantially from goals considered 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.

3.3 Web Services

In this subsection we describe in detail the different web services implementing the tourism service providers. 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.

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

```
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 3: Ontology for Flight

We now report the capability-level WSML definition of the Flight Booking Service.

```
webService fl#BookFlight
    capability
        sharedVariables {?date, ?start, ?dest, ?client}
        precondition definedBy
            exists {?req}
            (?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
            _#offer [trv#date  hasValue ?date,
                     trv#start  hasValue ?start,
                     trv#destination  hasValue ?dest,
                     trv#flightNumber hasValue _#fn,
                     fl#client  hasValue ?client]  memberOf fl#FlightOffer.

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

Listing 4: Flight Booking Service: Capability Level

Notice the different roles of precondition, assumption, postcondition and effect. The precondition is used to express constrains 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 results of the execution to the web services such as the fact that the flight has actually been booked.

We now proceed with describing the choreography interface of the Flight Booking Service. For describing this interface, we use the syntax of WSMO Choreography [SPR$^+$05] with a slight extension, based on control state ASMs [BS03], a particular form of ASMs which does not restrict the model by any means but adds names control states by means of a simple designated controlled function *ctl_state*.

A control state ASM is an ASM with one particular designated controlled function *ctl_state* (which has a range a finite number of Integers {1,...,n} or a finite enumeration of state-descriptors) and each transition rule having the form:

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

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.

```
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 choose {?fn} with
               exists {?s,?f} (?f[trv#date hasValue ?date,
                                  trv#start hasValue ?start,
                                  trv#destination hasValue ?dest,
                                  fl#flightNumber hasValue ?fn] memberOf fl#FlightAvailable)
               do
```

```
                add(_#offer[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
              endDo
            else
              add(_#notAvailable[trv#date hasValue ?date,
                             trv#start hasValue ?start,
                             trv#destination hasValue ?dest] memberOf fl#FlightNotAvailable)
              ctl_state := fl#noAvail
            endIf
       endDo
     endIf

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

     if (ctl_state = fl#offerMade) then
       if _#offer memberOf {fl#FlightCancel,fl#FlightOffer}
         ctl_state := fl#cancelled
     endIf
```

Listing 5: Flight Booking Service: Intrerface Level

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

```
webService htl#BookHotel
    capability
        sharedVariables {?date, ?loc, ?client}
        precondition definedBy
            exists {?req}
            (?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
            _#offer [trv#date hasValue ?date,
                     trv#location hasValue ?loc,
                     htl#client hasValue ?client] memberOf htl#HotelOffer.
        effect definedBy
            _#booking[trv#date hasValue ?date,
                      trv#location hasValue ?loc,
                      htl#hotelName hasValue _#hn,
                      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
      if choose {?name} with
         exists {?h} (?h[trv#date hasValue ?date,
                        trv#hotelName hasValue ?name,
                        trv#location hasValue ?loc] memberOf trv#HotelAvailable)
         do
           add(_#offer[trv#date hasValue ?date,
                      trv#hotelName hasValue ?name,
                      trv#location hasValue ?loc,
                      htl#client hasValue ?client] memberOf htl#HotelOffer)
         ctl_state := htl#offerMade
         endDo
      else
         add(_#notAvailable[trv#date hasValue ?date,
                      trv#location hasValue ?loc] memberOf htl#HotelNotAvailable)
         ctl_state := htl#noAvail
      endIf
    endDo
  endIf

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

  if (ctl_state = htl#offerMade) then
    if _#offer memberOf {htl#HotelCancel,htl#HotelOffer}
      ctl_state := htl#cancelled
  endIf
```

Listing 6: Hotel Booking Service

3.4 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. Since the description of orchestration interfaces in WSML is still under development, we use pseudo-syntax here to illustrate the example.

The composed service should request a flight; if no flight is available then the service fails. Otherwise, an hotel is requested; if no hotel is available, then the flight is cancelled and the service fails. Otherwise, both flight and hotel are confirmed and the service ends with success.

```
namespace trv _"http://www.example.org/Travel"
          fl  _"http :// www.example.org/BookFlight"
          htl _"http :// www.example.org/BookHotel"
          vta _"http :// www.example.org/VTA"

webService vta#VTA

interface vta#VTAInterface
  orchestration
    stateSignature
      importsOntology {htl#simpleHotelOntology,fl#simpleFlightOntology}

    inputVariables
        ?start  ofType trv#Location
        ?dest   ofType trv#Location
        ?date   ofType _date
        ?client ofType trv#Client

    outputVariables
        ?flight ofType fl#Flight
        ?hotel  ofType htl#Hotel

    ctl_state{vta#start ,vta#flightRequested,vta#hotelRequested,vta#noFlight,vta#noHotel,vta#booked}

    transitionRules

    // Request a flight
    if (ctl_state = vta#start) then
        add(_#fReq[trv#date hasValue ?date,
                   trv#start hasValue ?start,
                   trv#destination hasValue ?dest,
                   fl#client hasValue ?client] memberOf fl#FlightRequest)
        ctl_state = vta#flightRequested
    endIf

    // Flight offer received: request and hotel
    if (ctl_state = vta#flightRequested and
          exists {?fo,?fn} (?fo[trv#date hasValue ?date,
                               trv#start hasValue ?start,
                               trv#destination hasValue ?dest,
                               fl#flightNumber havValue ?fn,
```

```
                            fl#client   hasValue ?client] memberOf fl#FlightOffer)) then
          ?flight  = ?fo
          add(_#fReq[trv#date hasValue ?date,
                     trv#location  hasValue ?dest,
                     htl#client   hasValue ?client] memberOf htl#HotelRequest)
          ctl_state  = vta#hotelRequested

// No flight available: terminate with failure
if (ctl_state  = vta#flightRequested and
       exists {?fna} ?fna[trv#date hasValue ?date,
                          trv#start  hasValue ?start,
                          trv#destination  hasValue ?dest,
                          fl#client   hasValue ?client] memberOf fl#FlightNotAvailable)) then
          ctl_state  = vta#noFlight
endIf

// Hotel offer received: confirm both flight and hotel and terminare sucessfully
if (ctl_state  = vta#hotelRequested and
       exists {?ho,?hn} (?ho[trv#date hasValue ?date,
                          trv#location  hasValue ?dest,
                          htl#hotelName hasValue ?hn,
                          htl#client   hasValue ?client] memberOf htl#HotelOffer)) then
          ?hotel = ?ho
          if exists {?fn} ( ?flight [fl#flightNumber havValue ?fn]) then
              add(_#fAck[trv#date hasValue ?date,
                         trv#start  hasValue ?start,
                         trv#destination  hasValue ?dest,
                         fl#flightNumber  hasValue ?fn,
                         htl#client   hasValue ?client] memberOf fl#FlightConfirm)
              add(_#hAck[trv#date hasValue ?date,
                         trv#location  hasValue ?dest,
                         htl#hotelName hasValue ?hn,
                         htl#client   hasValue ?client] memberOf htl#HotelRequest)
          endIf
          ctl_state  = vta#booked
endIf

// No hotel available: cancel the flight and terminare with failure
if (ctl_state  = vta#hotelRequested and
              exists {?hna} (?hna[trv#date hasValue ?date,
                                  trv#location  hasValue ?dest,
                                  htl#client   hasValue ?client] memberOf htl#HotelNotAvailable)) then
          if exists {?fn} ( ?flight [fl#flightNumber havValue ?fn]) then
              add(_#fNack[trv#date hasValue ?date,
                          trv#start  hasValue ?start,
                          trv#destination  hasValue ?dest,
                          fl#flightNumber  hasValue ?fn,
                          htl#client   hasValue ?client] memberOf fl#FlightConfirm)
          endIf
          ctl_state  = vta#noHotel
endIf
```

Listing 7: VTA Composite Service

Chapter 4

Integration Discovery and Composition: An Architecture

In this section we discuss an architecture for integrating discovery and composition of Semantic Web Services. The description is very high-level since its formal definition and refinement (along with the investigation of the underlying theory and the implementation of the algorithm) will be the focus of the work in the forthcoming months.

4.1 Basic Architecture

The architecture is depicted in Figure 4.1. 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).

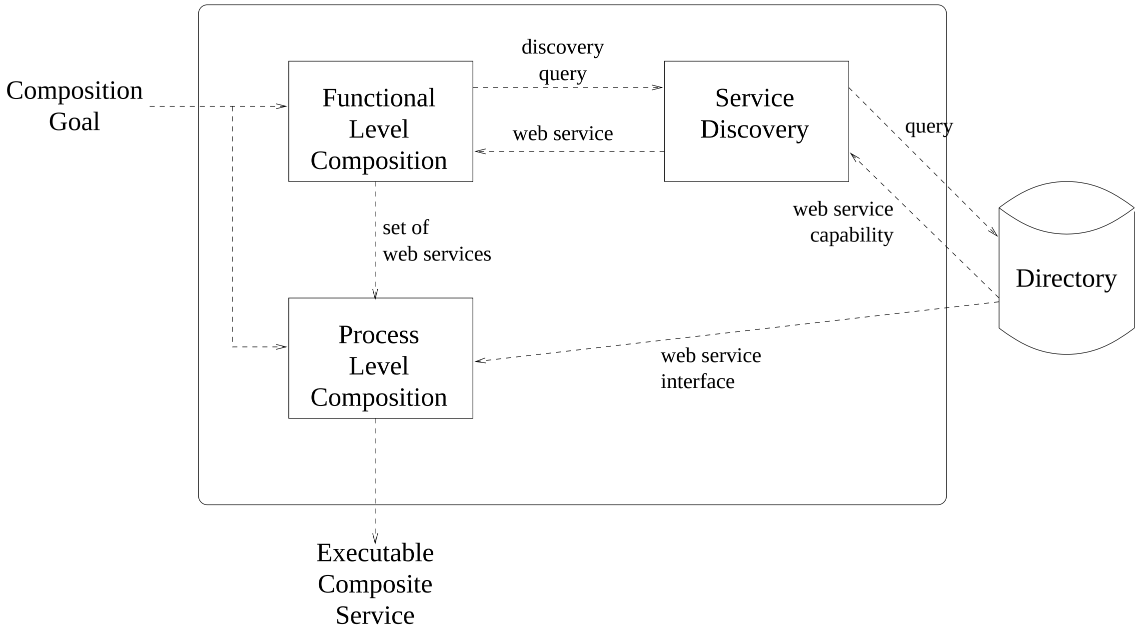

Figure 4.1: Schema Integration Discovery and Composition.

“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.

“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 4.1 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. As 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 precess.

Another issue left open by the current architecture, which has to be addressed in the refinement of the theoretical framework that will be undertaken in the next months, 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 in taking into account non-functional requirements such as QoS, costs, security... in the selection and composition of the services. In the next subsection we discuss how such an extension can be defined for a specific class on non-functional requirements, related to service reputation.

4.2 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 [Rep04]. 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.

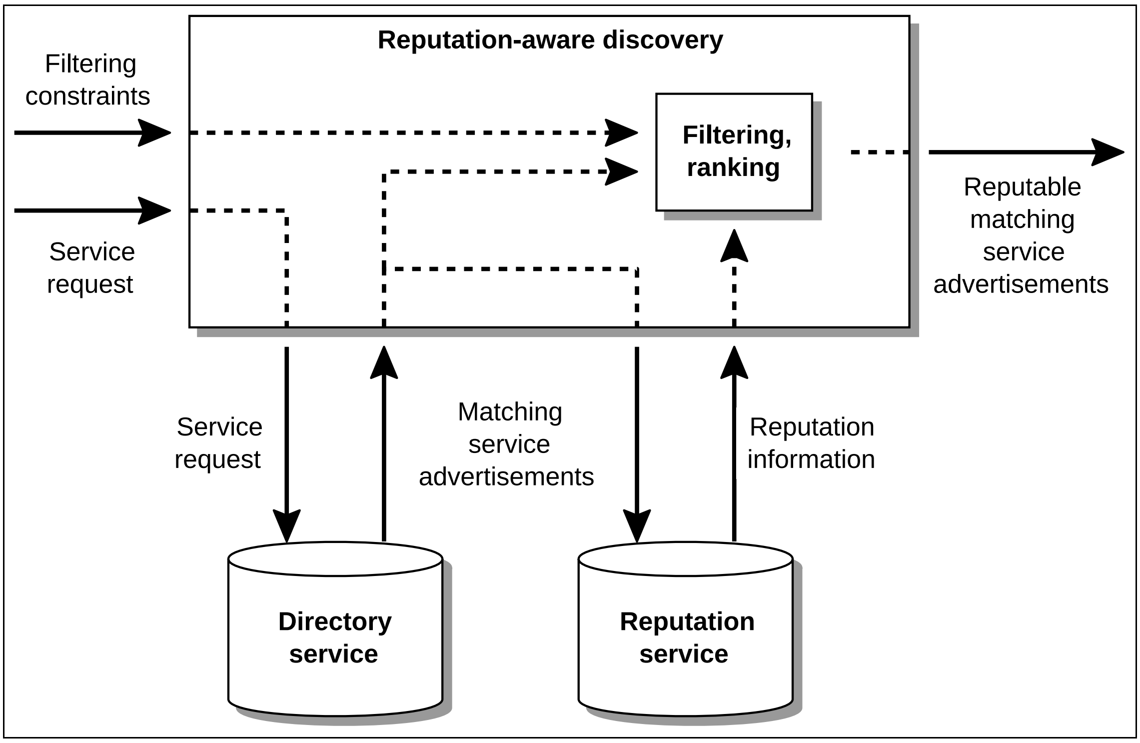

Figure 4.2: A wrapper for service discovery filters and ranks matching service advertisements according to their reputation.

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 4.2 illustrates our approach.

This approach has the advantage that it does not require any changes to the integrated discovery and composition architecture in Figure 4.1. 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.

Chapter 5

Conclusions

The automatic creation of an executable web service starting to 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 use to build an executable web service. The schema proposed in the Chapter 4 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 match 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 an example of integrated discovery and composition and on the definition of a reference architecture. Future work will focus on the definition of the theoretical framework underlying an integrated discovery and composition. This will require to refine the definitions of the languages and underlying models used for defining composition goals, web service choreographies, and executable composite services. These theoretical investigations are planned to be completed within month 24. A second line in our future work is the implementation of an integrated web service discovery and composition algorithm. The implementation is planned to be completed within month 30.

Bibliography

[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.

[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.

[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.

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

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

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

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

[Rep04] Reputation mechanism. In *Knowledge Web Deliverable D2.4.1*, 2004.

[SPR+05] James Scicluna, Axel Polleres, Dumitru Roman, Cristina Feier, and Dieter Fensel. Ontology-based choreography and orchestration of WSMO services. Deliverable d14v0.2, WSMO, 2005. Available from http://www.wsmo.org/TR/d14/v0.2/20050702/.

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