D2.4.2 Semantics for Web Service Discovery and Composition

Rubén Lara (Universität Innsbruck)

with contributions from:
Walter Binder (EPFL), Ion Constantinescu (EPFL), Dieter Fensel (UIBK), Uwe Keller (UIBK), Jeff Pan (VUM), Marco Pistore (UniTn), Axel Polleres (UIBK), Ioan Toma (UIBK),

Paolo Traverso (UniTn), Michal Zaremba (NUIG)

Abstract.
EU-IST Network of Excellence (NoE) IST-2004-507482 KWEB
Deliverable D2.4.2 (WP2.4)

The description of Web services with formal, explicit semantics promises to bring a new level of automation to current Web services. In this context, the automatic location and composition of existing services to fulfill a given user request without the need for neither prior agreements nor costly custom code plays a central role. In this document, we analyze the challenges in automatic Web service discovery and composition and propose ways to solve them.
Keyword list: Web services, semantic Web services, semantic Web service discovery, semantic Web service composition

Document Identifier	KWEB/2005/D2.4.2/v1.1
Project	KWEB EU-IST-2004-507482
Version	v1.1
Date	January 29, 2005
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
0.05	02.08.04	Rubén Lara	creation
0.1	24.08.04	Rubén Lara	Sections 1.1 and 2.1 added
0.3	07.10.04	Rubén Lara	Structure for discovery updated, first content on discovery, extended introduction and added use case
0.35	10.10.04	Rubén Lara	Chapters 1 and 2 reviewed, and structure of chapter 2 updated
0.36	15.10.04	Marco Pistore	Chapter 3 split in two parts for function-level and process-level composition
0.5	29.11.04	Rubén Lara	Pre-final versions of Chapters 1 and 2
0.7	10.12.04	Walter Binder	Section 3.1 added
0.75	12.12.04	Rubén Lara	Section 3.1 reviewed
0.85	14.12.04	Marco Pistore	Section 3.2 and Chapter 4 added
0.9	15.12.04	Rubén Lara	Section 3.2 and Chapter 4 reviewed. Chapter 5 added.
1.0	17.12.04	Rubén Lara	Content fully integrated.
1.05	13.01.05	Rubén Lara	Revision following comments from quality assessor
1.1	19.01.05	Rubén Lara	Revision following comments from quality controller

Executive Summary

This document addresses the problems of automatic discovery and composition of (Web) services using semantic annotations. The problem of automatic discovery of services can be seen as the problem of locating a service that can fulfill some requester objectives. Composition can be seen at two levels: functional-level composition and process-level composition. Functional-level composition naturally extends the discovery problem selecting, in case a single service that can fulfill the requester goal cannot be found, a combination of existing services that can fulfill it. Process-level composition covers a later phase of the overall composition task. Here 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.

In this document we introduce the discovery and composition problems and ways to solve them. A conceptual model for discovery, involving the description of requester goals based on pre-defined goals, the discovery of potential candidate services, and the contracting of the discovered services is provided. Based on this model, we analyze to what extent current proposals for service discovery cover our needs. Theoretical foundations for the discovery and contracting problem are provided and their relation to currently available reasoning engines and languages is discussed.

Composition is addressed at the two levels introduced above, namely: functional-level composition and process-level composition. Planning-based techniques for solving both problems are presented and their efficiency assessed.

Finally, how discovery and the two types of composition can be integrated is discussed. This discussion identifies possible integration paths that will be further investigated in the future, as part of the Knowledge Web project.

Contents

Chapter 1

Introduction

1.1 Semantic Web Services

The Web is a tremendous success story. Starting as an in-house solution for exchanging scientific information it has become, in slightly more than a decade, a world wide used media for information dissemination and access. In many respects, it has become the major means for publishing and accessing information. Its scalability and the comfort and speed in disseminating information has no precedent. However, it is solely a Web for humans. Computers cannot ”understand” the provided information and in return do not provide any support in processing this information. Two complementary trends are about to transform the Web, from being for humans only, into a Web that interweaves computers to provide support for human interactions at a much higher level than is available with current Web technology.

- The semantic Web is about adding machine-processable semantics to data. The computer can ”understand” the information and therefore process it on behalf of the human user (cf. [Fen03]).
- Web services try to employ the Web as a global infrastructure for distributed computation, for integrating various applications, and for the automation of business processes (cf. [ACKM03]). The Web will not only be the place where human readable information is published but the place where global computing is realized.

Nevertheless, the current Web service technology, based on SOAP [W3C03], WSDL [CCMW01] and UDDI [BCE+02], only addresses the syntactical aspects of a Web service and, therefore, only provide a set of rigid services that cannot adapt to a changing environment without human intervention. The human programmer has to be kept in the loop and scalability as well as economy of Web services are limited [FB02]. The vision of semantic Web services is to describe the various aspects of a Web service using explicit, machine-understandable semantics, enabling the automatic location, combination

and use of Web services. The work in the area of semantic Web is being applied to Web services in order to keep the intervention of the human user to the minimum. Semantic markup can be exploited to automate the tasks of discovering services, executing them, composing them and enabling seamless interoperation between them [Coa04], thus enabling intelligent Web services.

The description of Web services in a machine-understandable fashion 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, both within and across enterprise boundaries, is the mechanization of service discovery. Automatically locating and contracting available services to perform a given business activity can considerably reduce the cost of integration and can enable a much more flexible integration, where providers are dynamically selected based on what they provide and possibly other non-functional properties such as trust, security, etc.

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. This process is called service composition. We distinguish between manual and automated service composition.

In the case of manual service composition, the user explicitly specifies how certain basic services have to interact in order to provide the desired functionality. The specification of the composed service may be represented as a workflow. In the case of automated service composition, a service composition engine automatically generates a representation of the composed service based on user requirements. There are also hybrid approaches involving manual and automated service composition. In such a setting, the composition engine may serve as an interactive programming tool to help and guide the programmer definining complex services. The programmer may specify some constraints (a partial workflow) and the composition engine fills in services in order to complete the workflow in a consistent way.

In an open environment, service composition always involves service discovery. 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 has to 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 discover relevant services when needed.

Automated service composition may be regarded at different levels of abstraction. From a high-level point of view, we may focus on the required inputs and provided outputs of services and compose services in order to generate the outputs needed by the user.

We call this kind of high-level composition *functional service composition*, which is addressed in Section 3.1. There we show how standard planning techniques can be adapted for functional service composition. Moreover, we discuss the integration of functional service composition with dynamic service discovery.

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 (method), but a given service API has to be used according to some service protocol e.g. it may be necessary to invoke several service functions (methods) in a specific order. Hence, a fine-grained, *process-level composition* is needed in order to generate an executable composed service. In Section 3.1 we present an approach for process-level composition based on model checking techniques.

The potential benefits of semantic Web services have led to the establishment of an important research area, both in academia and industry. Several initiatives have appeared for semantically annotating Web services, providing different descriptions of Web services and their related aspects that, in turn, bring a different kind of support for discovery and composition. In the following sections, we present the major initiatives in the area.

1.1.1 OWL-S

OWL-S [Coa04] is a collaborative effort by BBN Technologies, Carnegie Mellon University, Nokia, Stanford University, SRI International and Yale University to define an ontology[1] for semantic markup of Web services. OWL-S, currently at version 1.1[2], is intended to enable automation of Web service discovery, invocation, composition, interoperation and execution monitoring by providing appropriate semantic descriptions of services. The purpose of OWL-S is to define a set of basic classes and properties for declaring and describing services i.e. an ontology in OWL [DS04] for describing Web services that enable users and software agents to automatically discover, invoke, compose and monitor Web resources offering services, under specified constraints.

Support for discovery

The upper OWL-S ontology consists of three core elements, namely: *service profile*, *service model*, and *service grounding*. From these, the *service profile* provides the information needed for an agent to discover a service [Coa04].

The OWL-S profile unifies the vision of both the requester and the provider. It can be used to describe what the requester expects from the service execution and the actual service functionality from a provider point and view. Therefore, the requester is expected

[1] Ontologies are defined in [Gru93b] as *formal, explicit specifications of shared conceptualizations*. We will use this definition throughout this document.

[2] For a complete account of all DAML-S and OWL-S versions, we refer the reader to http://www.daml.org/services/owl-s/.

to define a service profile which describes the service he is looking for, while the provider will describe a Web service by providing one or more[3] profiles giving information about what the service actually does.

The profile is intended to capture three different types of information: what organization provides the service, what function the service computes, and a host of features that specify characteristics of the service [Coa04]. Notice that both the first and third kind of information can be regarded as non-functional properties of the Web service, while the second kind of information describes the function provided by the service. We will follow this categorization to give further details of the service profile. As explained before, the profile is also used to describe a user request, containing information about what is the organization that has to provide the service sought, what functionality the service has to fulfill, and what non-functional criteria are to be met.

Non-functional properties. The non-functional properties of the Web service contained in the service profile are: the service name, a textual description of the service, contact information of the service responsible, an external categorization of the service, and, finally, an expandable list of non-predefined properties.

Functionality description. The functional characterization of Web services is expressed in terms of the information transformation and the state change produced by the execution of the service. State change, modelled by preconditions and effects, refer to the change on the state of the world as a consequence of executing the service, and information transformation, modelled by inputs and outputs, refer to what information is required and what information is produced (generally depending on the information provided as input) by the service. Inputs, outputs, preconditions, and effects are normally referred to as IOPEs [Coa04].

The schema to describe IOPEs instances is defined in the service model, not in the service profile. Therefore, these instances are described in the service model and referenced from the service profile. OWL-S envisions that the IOPEs of the profile are a subset of those published by the model [Coa04].

OWL-S 1.1 is the first version of OWL-S making use of a concrete rule language to describe IOPEs, capturing the relation between inputs, outputs, preconditions and effects. The language used for this purpose is the Semantic Web Rule Language (SWRL)[4] [HPSB+04]. We provide a brief overview of the language below.

[3] OWL-S does not define any minimum cardinality constraint for the profile. Therefore, a Web service without any profile would still be a valid OWL-S description.

[4] However, the OWL-S ontology also allows to express conditions using DRS [McD04] or KIF [kif98]. However, DRS only provides a vocabulary for writing down arbitray formulas and does not prescribe their semantics, and the interfacing of KIF expressions with OWL concepts is unclear.

SWRL. SWRL is a proposal for combining OWL DL [DS04] and OWL Lite [DS04] with the Unary/Binary Datalog sublanguages of RuleML[5], enabling the combination of Horn-like rules with an OWL knowledge base. SWRL Rules have the form:

$$antecedent \rightarrow consequent$$

where both the $antecedent$ i.e. the rule body and the $consequent$ i.e. the rule head are a conjunction of atoms. Atoms can be of the form $C(x)$, $P(x, y)$, $Q(x, z)$, $sameAs(x, y)$ or $differentFrom(x, y)$, being C an OWL DL (not necessarily named) concept, P an OWL DL object property, Q an OWL DL data property, x and y are either variables or OWL instances, and z is either a variable or an OWL data value.

Variables are treated as universally quantified within the scope of the rule they appear in, and only safe rules are allowed[6]. Existential quantification can be captured by using the OWL *someValuesFrom* restriction.

Inputs and Outputs. OWL-S inputs and outputs are modelled as subclasses of *parameter*, which is in turn a subclass of SWRL variable with a property indicating the class or datatype the values of the parameter belong to [Coa04]. Local variables can be also used, and they are modelled in the ontology as subclasses of parameter. Inputs, outputs, and local variables have as scope the process where they appear.

The inputs and outputs defined in the service model are referenced from the profile via the *hasInput* and *hasOutput* properties, respectively. Local variables can be referenced via the *hasParameter* property.

Preconditions and effects. Preconditions are conditions on the state of the world that have to be true for successfully executing the service. They are modelled as conditions, a subclass of *expression*. Expressions in OWL-S specify the language in which the expression is described and the expression itself encoded as a (string or XML) literal. Effects describe conditions on the state of the world that are true after the service execution. They are modelled as *results*. A result has an *inCondition*, a *ResultVar*, an *OutputBinding*, and an *Effect*. The *inCondition* specifies the condition for the delivery of the result. The *OutputBinding* binds the declared output to the appropriate type or value depending on the *inCondition*. The *effects* describe the state of the world resulting from the execution of the service. The *ResultVars* play the role of local variables for describing results.

Conditions i.e. preconditions defined in the service model are referenced from the profile via the *hasPrecondition* property, and results via the *hasResult* property.

Figure 1.1 illustrates the description of the functional aspects of the profile. Please notice that, for simplicity reasons, the complete definition of some classes e.g. *binding* is omitted.

[5] see http://www.ruleml.org/ for details about the RuleML initiative.

[6] Variables that appear on the consequent must also appear on the antecedent.

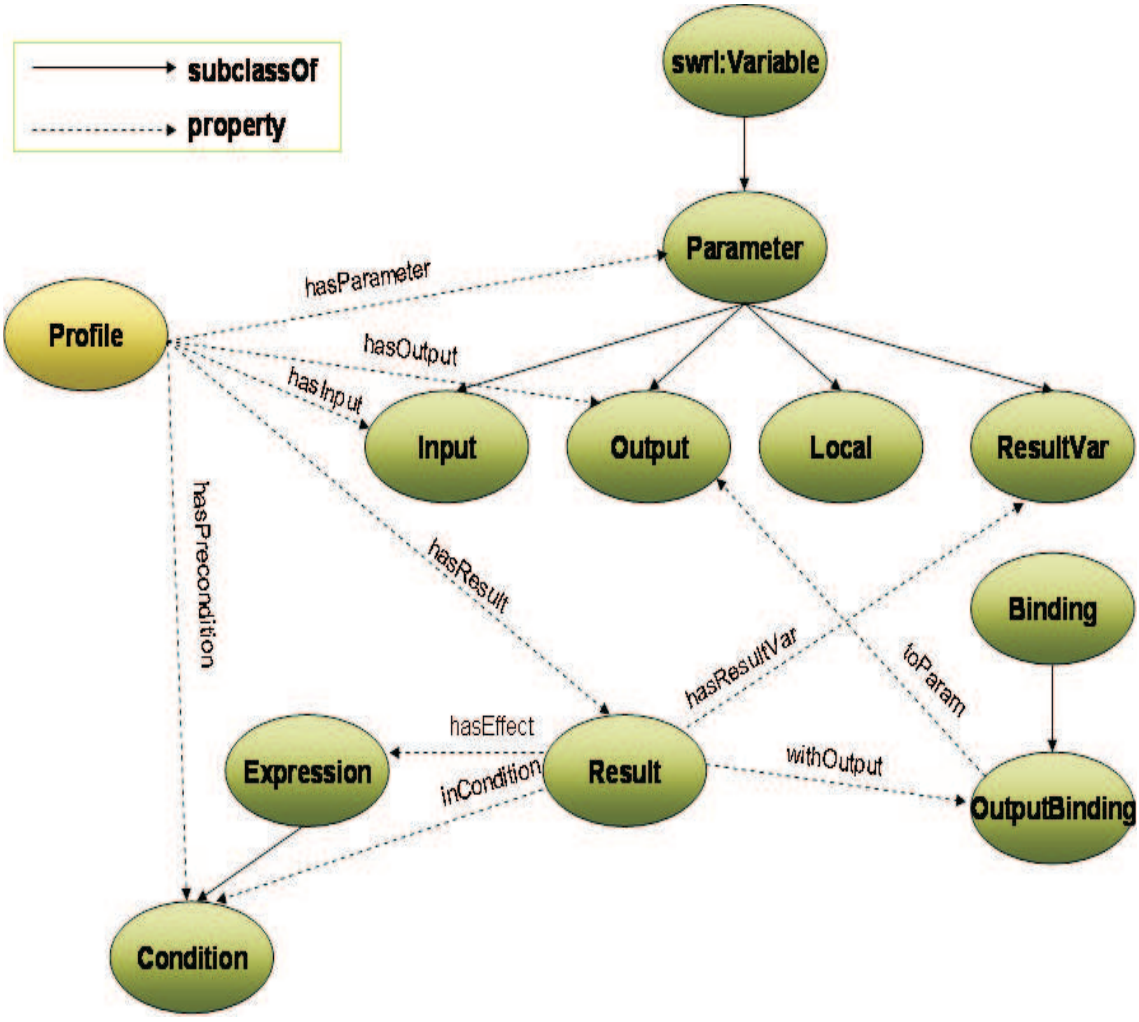


Figure 1.1: Definition of OWL-S profile

Support for composition

Automatic Web service composition and interoperation involves the automatic selection, composition, and interoperation of Web services to perform some complex task, given a high-level description of an objective. For example, the user may want to make all the travel arrangements for a trip to a conference. Currently, the user must select the Web services, specify the composition manually, and make sure that any software needed for the interoperation of services that must share information is custom-created. With OWL-S markup of Web services, the information necessary to select and compose services will be encoded at the service Web sites. Software can be written to manipulate these representations, together with a specification of the objectives of the task, to achieve the task automatically. To support this, OWL-S provides declarative specifications of the prerequisites and consequences of application of individual services (relevant for functional-level composition), and a language for describing service compositions and data flow interactions (relevant for process-level composition).

1.1.2 WSMO

WSMO is a specification by the WSMO working group[7] of the SDK cluster[8]. WSMO relies on four major components inspired by the conceptual work done in the definition of WSMF (see Figure 1.2), namely:

Ontologies. They provide the terminology and formal semantics for describing the other elements in WSMO.

Goals. These elements provide the means to specify the requester-side objectives when consulting a Web service, describing at a high-level a concrete task to be achieved.

Web services. They provide a semantic description of Web services, including their functional and non-functional properties, as well as other aspects relevant for interoperating with them.

Mediators. These modelling elements are connectors that resolve heterogeneity problems in order to enable interoperation between heterogeneous parties.

Support for Discovery

From the elements presented above, the most relevant to the discovery process are:

[7] http://www.wsmo.org/
[8] http://www.sdkcluster.org/

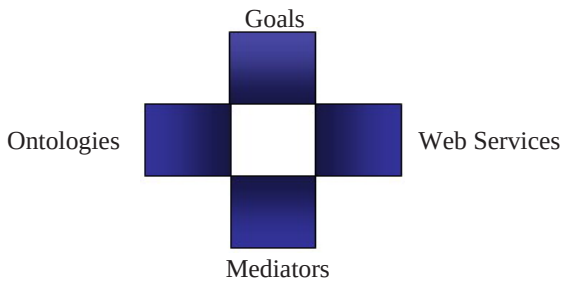


Figure 1.2: WSMO core elements [RL*editors*04]

Goals. Goals are defined in WSMO as the objectives that a client may have when consulting a Web service. They consist of non-functional properties (from a set of pre-defined core properties), imported ontologies, used mediators, postconditions and effects.

Postconditions and effects describe the state of the information space and the world desired by the requester, respectively. Ontologies can be directly imported as the terminology to define the goal when no conflicts need to be resolved. However, if any aligning, merging, or conflict resolution is required, they are imported through *ooMediators*.

Web services. Several aspects of Web services are described in WSMO. The required terminology to describe them, as for goals, can be imported directly or via *ooMediators* when conflicts need to be resolved. In addition, the capability and interfaces of the service are described, from which the capability is the most relevant element for discovery.

WSMO capabilities define the functional aspects of the offered service, modelled in terms of preconditions, assumptions, postconditions and effects. It is defined separately from the requester goals, thus distinguishing between the requester and provider points of view.

The *preconditions* of the capability describe the valid states of the information space prior to the service execution. *Postconditions* describe the state of the information space that is guaranteed to be reached after the service execution. *Assumptions* are similar to preconditions, but they define valid states of the world for a correct service execution. *Effects* describe the state of the world that is guaranteed to be reached after executing the service.

Mediators. As one of its pillars, WSMO introduces the concept of mediators in order to resolve heterogeneity problems[9]. Mediators in WSMO are special elements used to link heterogeneous components involved in the modelling of a Web service. They define the necessary mappings, transformations or reductions between the linked elements. As depicted in Figure 1.1.2, four different types of mediators are defined. From those, of relevance for discovery are:

[9]For an introduction to the concept of Mediators, cf. [Wie92].

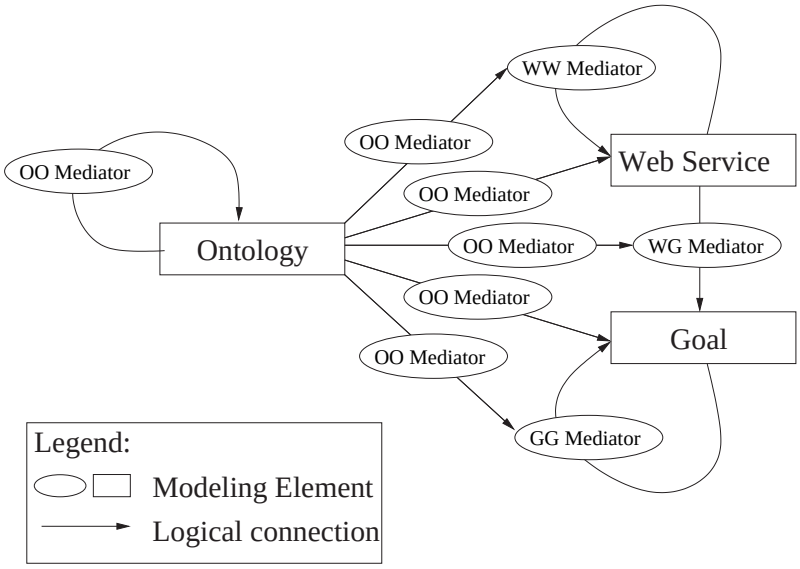


Figure 1.3: Mediators in WSMO [RL*editors*04]

- ggMediators: They link two goals, expressing the reduction of a source goal into a target goal. They can use *ooMediators* to bypass the differences in the terminology employed to define these goals. In addition, WSMO allows linking not only goals, but also goals to *ggMediators*, thus allowing the reuse of multiple goals to define a new one.

- ooMediators: They import ontologies and resolve possible representation mismatches between them such as differences in representation languages or in conceptualizations of the same domain.

- wgMediators: They link a Web service to a goal. This link represents the (total or partial) fulfillment of the goal by the Web service. *wgMediators* can use *ooMediators* to resolve heterogeneity problems between the Web service and the goal.

Support for Composition

Functional-level composition can exploit the formal description of capabilities and goals provided by WSMO. Regarding process-level composition, WSMO provides service interface descriptions that provide information about how the service interacts, and with what other services it cooperates to achieve its functionality. These descriptions are called choreography and orchestration, respectively. Process-level composition can exploit the formal choreography descriptions provided by WSMO to know the external process of a given service, and construct orchestrations for the composite service generated.

WSML

The Web Service Modeling Language working group (WSML)[10] defines a family of languages with its roots in Description Logics, First-Order Logic and Logic Programming [de 04].

The variant WSML-Core semantically corresponds with the intersection of Description Logic and Horn Logic, extended with extensive datatype support in order to be useful in practical applications. WSML-Core is fully compliant with a subset of OWL, albeit that the datatype support in WSML-Core is already beyond OWL, because the datatype support in OWL is very limited. WSML-Core is extended, both in the direction of Description Logics and in the direction of Logic Programming.

WSML-DL extends WSML-Core to an expressive Description Logic, namely $\mathcal{SHOIN}$.

WSML-Flight extends WSML-Core in the direction of Logic Programming with more intuitive value restrictions and cardinality constraints. WSML-Flight is the preferred ontology modelling language for WSMO, because of its rich set of modelling primitives for modelling different aspects of attributes, such as value constraints and integrity constraints, and its rich logical language which allows for writing down arbitrary rules. Furthermore, WSML-Flight incorporates a fully-fledged rule language, while still allowing efficient decidable reasoning.

WSML-Rule extends WSML-Flight to a fully-fledged Logic Programming language, including function symbols and higher-order features of HiLog [CKW93] and possibly Transaction Logic [BK98].

WSML-Full unifies all WSML variants under a common First-Order umbrella with non-monotonic extensions. All WSML variants are described in terms of a normative human-readable syntax. Besides the human-readable syntax, an XML and an RDF syntax for exchange between machines are provided. Furthermore, a mapping to and from OWL for basic inter-operation with OWL ontologies through a common semantic subset of OWL and WSML is also given.

1.1.3 METEOR-S

METEOR-S[11], started in 2002 at the LSDIS Lab at the University of Georgia, aims to integrate Web service standards such as Business Process Execution Language for Web Services (BPEL4WS) [ACD+03], Web Service Description Language (WSDL) [CCMW01] and Universal Description, Discovery and Integration (UDDI) [BCE+02] with Semantic Web technologies.

METEOR-S aims at automating the tasks of publication, discovery, description, and control flow of Web services. In the following, we will focus on how METEOR-S

[10] http://www.wsmo.org/wsml/.

[11] http://lsdis.cs.uga.edu/Projects/METEOR-S/.

supports discovery. Although METEOR-S also provides a Discovery Infrastructure for Web services (MWSDI) [VSSP04], our interest is restricted to how METEOR-S services are described to support such discovery, not in the infrastructure which exploits the METEOR-S descriptions.

Support for Discovery

METEOR-S adds semantics (using ontologies) at two levels [VSSP04]: at the level of individual Web services and at the level of the registries that store the services. For the annotation of individual Web services, a bottom-up approach is followed i.e. WSDL message types, both inputs and outputs, are mapped to the appropriate concepts in domain specific ontologies. In addition to the annotation of WSDL inputs and outputs, WSDL operations are also mapped to ontological concepts from an operations domain ontology [SVSM03]. In the annotation of WSDL operations, preconditions and effects are also added, interpreted as logical conditions that must hold for performing the operation, and as changes in the world caused by the execution of the operation, respectively. The user goals are expressed using service templates based on the concepts from the domain ontologies. In such templates, information about the operation being sought and their inputs and outputs are given, and, optionally, preconditions and effects can also be specified.

The aim of annotating registries is to enable the classification of Web services based on their domain. Registries are specialized in a given domain, and store Web services related to that domain. A specialized ontology, the registries ontology, is used to annotate registries, mapping the registries to a given domain and giving additional information about the registry, relations with other registries and relations to other domains.

Support for Composition

Functional-level composition can exploit the semantic description of the inputs, outputs, postconditions and effects to determine what parts of a request are solved by a given service and what others remain open. For process-level composition, the BPEL4WS descriptions used in METEOR-S can be exploited. However, as such descriptions do not come with formal semantics, process-level composition has to rely on externally defined semantics.

1.1.4 IRS-III

The primary goal of the IRS-III[12] project, being carried out by the Knowledge Media Institute at the Open University[13], is to support the discovery and retrieval of knowledge components (i.e services) from libraries distributed over the Internet and their semi-automatic

[12] http://kmi.open.ac.uk/projects/irs/.
[13] http://kmi.open.ac.uk/

configuration in order to realize specific tasks according to user requirements [MDCG03]. The IRS-III framework can be seen as an adaptation of the UPML [FMB+03] framework to the Web services domain.

Support for Discovery and Composition

IRS-III has adopted the WSMO conceptual model. Therefore, we can assume that the support provided for discovery and composition is the same as in WSMO.

1.2 A Motivating Use Case

The different approaches for describing services presented above aim at supporting the automatic location, composition and interoperation with services. In order to illustrate the problems we aim at solving in this document i.e. the automatic location and composition of services based on such descriptions, in this section we introduce a B2C use case that motivates such problems and illustrates how real applications can benefit from a higher level of automation. This use case will be used in following chapters to illustrate our proposals for achieving dynamic discovery and composition.

1.2.1 Description

Let us imagine a Virtual Traveling Agency (VTA for short) [S*editors*] which is an end user platform providing eTourism services to customers. These services can cover all kinds of information services concerned with tourism information, from information about events and sights in an area to services that support booking of flights, hotels, rental cars, etc. online. Such VTAs are already existent, but they are mostly simple information portals along with some Web-based customer services. By applying semantic Web services, a VTA will invoke Web services provided by several eTourism suppliers and aggregate them into new customer services in a (semi)automatic fashion. Such VTAs providing automated eTourism services represent an evolution of currently existing VTAs, as they can dynamically select (from ALL the available tourism services) and compose the appropriate services to fulfill a given end-user request, not being limited to pre-arranged providers in a pre-arranged way.

Our VTA use case, which aggregates Web services of different tourism service providers, shall provide the following general functionality: a customer uses the VTA service as the entry point for his requests. Such end-user services are provided by the VTA by discovering, invoking and combining Web services offered by several tourism service providers. Figure 1.4 gives an overview (modified and extended from [HHO04]) of the use case.

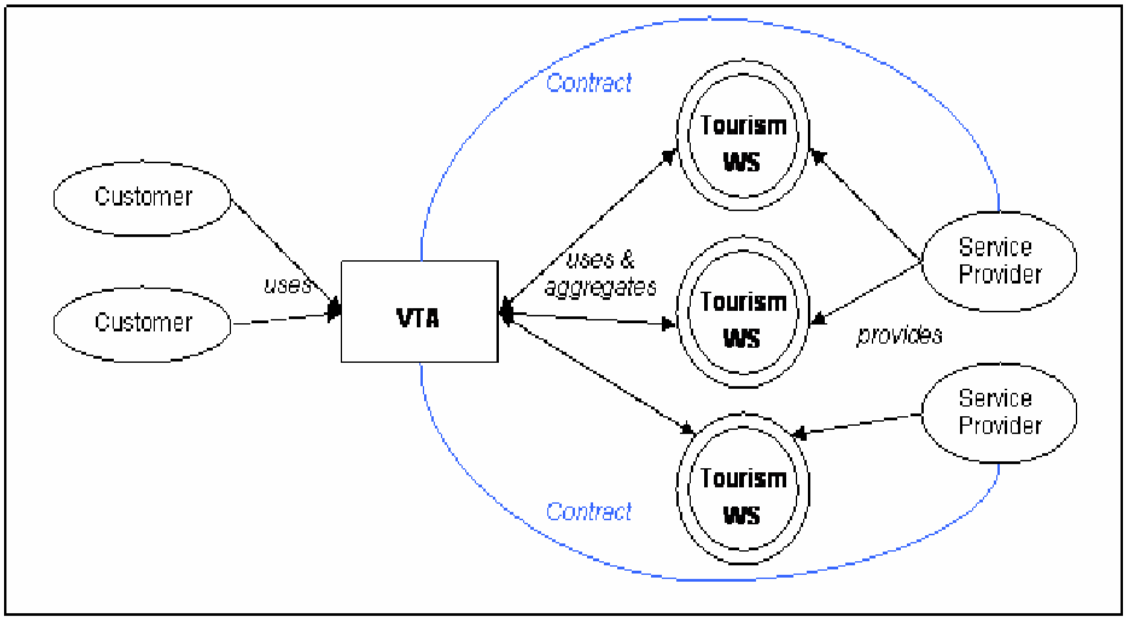


Figure 1.4: Use Case Overview: Virtual Travel Agency based on Semantic Web services

1.2.2 Scope

The scenario outlines a general structure for VTAs that can be extended to more complex scenarios wherein the customer can be a Web service itself, thus creating a network of composed services that offer complex tourism services. For example, one VTA can provide flight booking services for an airline union, another VTA aggregates booking service for a worldwide hotel chain, and a third VTA provides booking services for rental cars by combining the services of several worldwide operating car rental agencies. Then, another VTA uses these services for providing an end-user service for booking complete holiday trips worldwide.

1.2.3 Actors, Roles and Goals

In the general use case there are three actors. The following defines why they participate in this use case (goal) and the particular interactions they are involved in (roles).

- Customer: the end-user that requests a service provided by the VTA.
  - Goal: automated resolution of his tourism service request.
  - Role: end-user, interacts with VTA for service usage, payment, and non-computational assets (e.g. receiving the actual ticket when booking a trip).

- Tourism service providers: commercial companies that provides specific tourism services.
  - Goal: sell their services to customers.
  - Role: provide tourism services as Web services (also provides the necessary semantic descriptions of Web services).
- VTA: the intermediary between the customer and the tourism service providers. It provides high-quality tourism services to customers by aggregating the separate services provided by the single service providers.
  - Goal: provides high-quality end-user tourism services, uses existing tourism services and aggregates these into new services. It represents the union of the available tourism service providers.
  - Role: interacting with the customer, locating, composing (if necessary) and using the services offered by the tourism providers and, ultimately, solving the customer request.

1.2.4 Example Usage Scenarios

Here we include some possible scenarios that, among others, can appear in our use case:

- The customer requests the VTA for searching tourism service offers, the VTA detects and queries suitable tourism service providers and gives the results to the customer.
- The customer selects a concrete offer and requests booking for this offer (interacting with the VTA), VTA discovers and aggregates tourism service providers for booking, payment, etc. and returns the result to the customer.
- The VTA interacts with the customer and (one or more) tourism service providers for checking the delivery status of a given flight ticket.

It can be seen in the above use case that the added value of the VTA resides on its ability to automatically detect appropriate tourism service providers, aggregate them, and offer the aggregated tourism services to the customer in a transparent fashion.

This use case will be used in the next chapters of this document to illustrate how we propose to achieve the automatic discovery of service providers and their composition. Examples mainly related to the domain of this use case will be given.

1.3 Goals and overview of this document

This document presents the problems of automatic service discovery and composition and propose solutions to them. In Chapter 2 we introduce a conceptual model for service discovery and contracting based on [Pre04] (Section 2.1), analyze the state of the art on automatic Web service discovery and software component retrieval (Section 2.2), and introduce our description of services (Section 2.3). With these elements in place, we address the problem of service discovery in Section 2.4, formalizing different notions of match for discovery and discussing means to implement them. Section 2.5 discusses the problem of service contracting and provide different formalizations for it. The relation between discovery and mediation is discussed in Section 2.6. The final sections of Chapter 2 summarize our achievements, open points and future work.

Chapter 3 is devoted to semantic Web service composition. Two different forms of compositions are addressed, corresponding to different levels of abstraction in the description of the Web services participation to the composition. In functional-level composition (Section 3.1) atomic Web services are considered, while in process-level composition (Section 3.2) the interaction with the Web services are defined as complex protocols. For each approach we discuss the state of the art, provide a formal description of the composition problem and an approach to its solution, we discuss implementation techniques, and evaluate the proposed solution.

In Chapter 4 the options for integrating automatic discovery, functional-level composition, and process-level composition are presented.

Finally, our results and plans for future work are summarized in Chapter 5.

Chapter 2

Semantics for Web Service Discovery

In this chapter we address the problem of automatic service discovery by providing a conceptual model for it and proposals for the different steps involved in the model. Such conceptual model is presented in Section 2.1. We analyze in Section 2.2 the work available in the area of automatic Web service discovery and in a strongly related area, software component retrieval. How services and goals will be described is related in Section 2.3. Our proposals for service discovery and contracting are presented in Section 2.4 and Section 2.5, respectively. Section 2.6 briefly discusses the mediation needs for discovery. In Sections 2.7, 2.8, and 2.9, we will briefly discuss our achievements on service discovery, the issues that remain open, and our planned future work, respectively.

2.1 A conceptual model for discovery

A workable approach to service discovery must precisely define its scope and the conceptual model and assumptions behind the proposed solution. While, as will be detailed in Section 2.2, a number of proposals are available in our area of interest, none of them has precisely discussed these aspects. Therefore, we start by addressing this task and providing the scope, model and assumptions that guide our proposal for service discovery.

2.1.1 Definition of service

Precisely defining what we mean by *service* and, therefore, what kind of entities we aim at discovering is needed to reach a common understanding of the problem and to explicitly describe our assumptions. For providing this definition, we look at the definitions given in the conceptual architecture for semantic Web services presented in [Pre04]:

- *Service as provision of value in some domain*. This definition regards a service as a provision of value (not necessarily monetary value) in some given domain,

independently of how the supplier and the provider interact. Examples of a service in this sense are the provision of information about flight tickets by a tourism service provider, or the provision of a booked trip with certain characteristics by a VTA.

- *Service as a software entity able to provide something of value*. This is the common understanding of service in the IT community, regarding the service as a software entity that provides something of value (a service in the sense above). An example would be the software that a VTA employs for providing aggregated tourism services.
- *Service as a means of interacting online with a service provider*. This definition refers to services like negotiation services, that do not provide anything of value by themselves but make the provision of the service possible. An example would be a WSMO choreography provided by a tourism service provider to interact with it for booking a given flight.

In the remainder of this chapter, we will refer to *service* in the first sense, understanding Web services as services that are programmatically accessible over the Web i.e. WSDL-like services. Services offered through human interaction using traditional Web sites will not be explicitly considered as they are less amenable to automation, but notice that they can be discovered and contracted in the same way described above if appropriate descriptions are given.

2.1.2 Levels of abstraction

Having defined what we understand by service, the next step is to define what kinds of service are of relevance for us. For doing so, we will look again at the work presented in [Pre04], where the following types of service are identified:

- A *Concrete Service* is an actual or possible performance of a set of tasks that represent a coherent functionality (and therefore deliver some value) within some domain of interest to its associated requestor and provider entities i.e. a concrete service is an actual service that will be or has been provided, for example the actual booking of a flight by a VTA.
- An *Abstract Service* is some set of concrete services, and an *Abstract Service Description* is some machine-processable description D which has, as its model, an abstract service C i.e. an abstract service description specifies the set of concrete services that can be provided e.g. booking of flights departing from Austria.
- An *Agreed Service* is an abstract service agreed between two parties i.e. it represents the agreement between a requester and a provider to receive and perform, respectively, a given service, for example, the agreement to provide information about flights for a given itinerary between a tourism service provider and a VTA.

- A *Service Contract* is an agreement between a service provider and requestor that the provider will supply an agreed service to the requestor e.g. a tourism service provider will provide the service agreed with the VTA.

From these, the abstract service is identified as relevant for the discovery of services, while the concrete services, agreed services, and service contracts are relevant for the contract agreement phase [Pre04].

2.1.3 Scope

As presented in [KLP+04], we will differentiate between service discovery and service contracting. The the abstract service descriptions presented above will be involved in the discovery phase. In the contracting phase, the concrete services will be used, and this phase will result in a service contract. Therefore, the overall scope for discovery does not only include discovery in the sense presented above, but also the contracting of relevant concrete services.

2.1.4 Assumptions

In order to define a model for service discovery and contracting, we need to make clear our assumptions on the domain. Such assumptions are related below:

- Assumptions on goals:
  - Pre-defined, generic and reusable goals will be available to the requester, defining generic objectives requesters may have.
  - Pre-defined goals are described in a formal manner.
  - Pre-defined goals can be refined (or parameterized) by the requester to reflect his concrete needs.
  - Requesters will not write from scratch formalized goals but will reuse and refine pre-defined goals i.e. we do not expect requesters to be able to completely formalize their goals from scratch.
  - Requesters have to be able to locate pre-defined goals that are relevant to them i.e. as requesters are expected to refine pre-defined goals, we assume that there will be a way for requesters to locate such pre-defined goals.
- Assumptions on abstract service descriptions:
  - Abstract service descriptions will be complete but not always correct [Pre04] i.e. every concrete service that can be provided will be a model of the description, but there will possibly be concrete services that are models of the

description but cannot be actually provided. For example, a tourism service that provides flights within Europe (but not all possible flights) will describe its abstract service as being able to provide any flight within Europe. However, there will be flights that are a model of this description i.e. they are flights within Europe, but that cannot actually be provided by this concrete provider. This incorrecteness is a consequence of the abstraction necessary to make descriptions manageable.

  - Abstract service descriptions will only include the expected results of the service, but not what input is required to achieve them.
  - As abstract service descriptions might not be correct, whether a concrete service can be actually provided and therefore agreed and contracted will be determined during the contracting phase i.e. during the contracting phase service providers will only agree and establish contracts for concrete services that they can actually provide.
  - A service provider will describe the services he is able to provide by making available an abstract service description, called an *abstract capability*.

- Assumptions on concrete services:

  - A service provider will describe the concrete services he can agree on and establish a contract for by providing a *contracting capability*. The contracting capability will also include the description of what conditions have to be fulfilled for a successful service provision, as well as the relation of the required input to the results of the service.

- Assumptions on consistency of descriptions:

  - The abstract capability might be automatically derived from the contracting capability, and both must be consistent.

- Assumptions on the contracting phase:

  - The requester goal resulting from refining a pre-defined goal will include the information necessary for contracting, such as the input information the requester can provide to the service. We do not impose that this (possibly big) set of information has to be listed for every goal, but can be made available to the discovery process by other means e.g. an additional service (in the sense of communication means described in Section 2.1.1) that provides the information that the requester has available and is willing to disclose[1].
  - A contract will not be agreed if the requester is not able to provide all the information required by the provider to actually deliver a concrete service.

[1] As discussed in [OlPL04], this might involve the use of information disclosure policies and a trust negotiation process.

– The communication between the requester and provider to establish a contract will be transparent to us i.e. we will not describe and deal with service choreographies but only with logic predicates that, in practice, will involve communication with either the requester or the provider.

2.1.5 Conceptual Model

Based on the definition of service chosen, the different levels of abstraction identified, and the assumptions on the domain given above, we provide a conceptual model for discovery that includes the reuse of pre-defined goals, the discovery of relevant abstract services and the contracting of concrete services to fulfill a concrete requester goal. Figure 2.1 depicts such conceptual model[2]. In the following, we provide an explanation of the different steps of the conceptual model.

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 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 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 on 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.

Let us take as an example a requester who wants to find information about flights from Innsbruck to Madrid on December 21st, 2004. Such requester can express his desire as a text of the form ”Search information about flights from Innsbruck to Madrid on December 21st, 2004”. This text can be used to perform keyword-based matching of existing pre-defined goals, such as a pre-defined goal for searching flight information.

[2]This model is a revision of the model presented in [KL*editors*04]

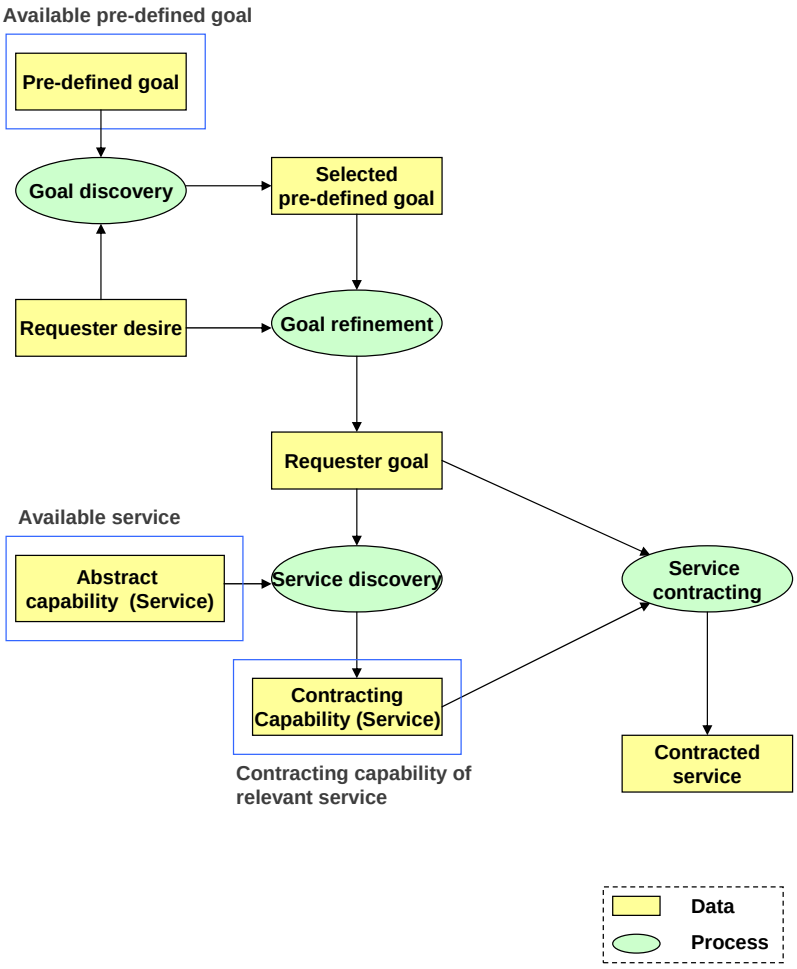


Figure 2.1: Discovery conceptual model

Once such formal pre-defined goal has been located, it will be refined to reflect the concrete origin and destination given by the requester, as well as the date. This refinement can be done manually (supported by appropriate tools) or automatically from the textual desire.

If a VTA service is available describing that it can provide flights from any place in Austria to any other place in Europe (as its abstract capability), this service will be selected and considered in the contracting phase. Notice that at this level we do not expect the VTA to accurately describe all the actual flights it can provide information for, as in general it is not realistic to expect flight information providers to replicate their flight databases in the service description. They will, instead, provide an abstraction of the kind of service they can provide.

During the contracting phase, whether the selected service can actually provide the requested flight information will be checked i.e. whether the selected VTA can provide information about flights from Innsbruck to Madrid on the given date will be tested. For that, the contracting capability of the VTA will be used, and such capability will include logic predicates that will actually query the database of the VTA (or of the aggregated tourism service providers) to check whether the requested flight information is available. In addition, and in case the VTA requires extra information to provide its service e.g. customer details, it will be checked whether the requester can provide such information.

If all the above criteria are fulfilled, the service will be contracted and eventually the concrete service provided.

2.2 State of the art on Software Component Retrieval and Web Service Discovery

In this section, we present and analyze the state on the art on service discovery. Given its strong relation to Web service discovery, we will start by presenting in section 2.2.1 the the work done in the area of software component retrieval. In Section 2.2.2, we will analyze the work available in automatic Web service discovery.

2.2.1 State of the Art on Software Component Retrieval

During the 90's, the computer science research community devoted some efforts to improve the state of the art on reuse of software components. The motivation for this work was to support the reuse of already existing and tested software components as one of the key factors for successful software engineering projects [SF97].

The efficient reuse of reliable software components providing a given functionality obviously requires efficient means to locate such components. A manual approach, in which the software engineer has to browse (possibly a big number of) libraries of compo-

nents to locate a suitable one, clearly does not scale. For this reason, the research efforts were oriented towards the formal specification of the components functionality and the formal description of the sought component in order to enable semi-automatic retrieval of appropriate components.

It can be seen that the problem of semi-automatically retrieving software components is highly similar to the automatic location of services. The concrete relation to the problem of automatically locating Web services will be discussed at the end of this section.

In the following, we briefly describe the work done in the area of software component retrieval to specify the functionality of software components and to formally define the different notions of match that can be of interest.

Specification Matching

Specification matching has been proposed in several works e.g. [JC92, JC93, JC95, RW91, ZW95] to evaluate how software components relate to a given query i.e. user's need. Specification matching relies on the axiomatization of software components and user queries. A formal (logical) relation is then defined and whether a given query and component satisfy this relation is checked. Such a relation must capture the notion of reusability i.e. if the relation holds for formally specified components and queries, it means that the component can be reused to solve the problem captured by the query.

At the formalization level, the following questions must be answered: a) How the components and queries are specified, and b) What is the relation to be checked for determining reusability.

As related in [CC00], a widely used axiomatization of components and queries is based on [Hoa69]. [Hoa69] provides a logical basis to prove some properties of a program, including determining whether a given program provides a certain functionality. The intended functionality of a program (C) is specified in terms of initial preconditions (C_{pre}) i.e. assertions about certain properties of the values taken by the relevant variables before the program initiation and the relations among them, and postconditions (C_{post}) i.e. the same kind of assertions as for preconditions but about the values after execution. The relation between the preconditions and postconditions of a given program is formulated as follows:

$$C_{pre}\{Q\}C_{post}$$

interpreted as "If the assertion C_{pre} is true before initiation of a program Q, then the assertion C_{post} will be true on its completion." [Hoa69]

Based on this type of axiomatization, most of the work done in specification matching specifies a component C as a 2-tuple of predicates (C_{pre}, C_{post}), being C_{pre} the precondition of the component and C_{post} its postcondition. Similarly, a query Q is specified as

(Q_{pre}, Q_{post}). Preconditions of the component are logical formulas that must hold prior to the use of the component[3], and postconditions are logical formulas that are guaranteed to be true after the execution of the component. The preconditions and postconditions of the query give a characterization of the desired component in terms of its preconditions and postconditions. Query preconditions can be interpreted as a description of the initial states for which the sought component must guarantee the fulfillment of the query postconditions.

[ZW97] explores different notions of match for retrieving formally-specified software components. A software component C is described in terms of their signature, C_{sig}, and their behaviour specification, C_{spec}. The former describes statically checkable information (component's type information), while the latter describes the component dynamic behaviour i.e. functionality. Although both aspects have to be matched for retrieving a software component given a query, here we are interested in the specification and matching of the component functionality. C_{spec} is, as explained above, described by the preconditions and postconditions of the component. Similarly, a query Q is described by its preconditions and postconditions. All the preconditions and postconditions are first-order formulas.

Match	**Definition**
1.$M_{exact-pre/post}$	$(Q_{pre} \leftrightarrow C_{pre}) \wedge (C_{post} \leftrightarrow Q_{post})$
2.$M_{plug-in}$	$(Q_{pre} \rightarrow C_{pre}) \wedge (C_{post} \rightarrow Q_{post})$
3.$M_{plug-in-post}$	$C_{post} \rightarrow Q_{post}$
4.$M_{guarded-plug-in}$	$(Q_{pre} \rightarrow C_{pre}) \wedge ((C_{pre} \wedge C_{post}) \rightarrow Q_{post})$
5.$M_{relaxed-plug-in}$	$(Q_{pre} \rightarrow C_{pre}) \wedge ((Q_{pre} \wedge C_{post}) \rightarrow Q_{post})$
6.$M_{guarded-post}$	$(C_{pre} \wedge C_{post}) \rightarrow Q_{post}$
7.$M_{partial-comp}$	$C_{pre} \wedge Q_{pre} \wedge C_{post} \rightarrow Q_{post}$
8.$M_{exact-pred}$	$(C_{pre} \rightarrow C_{post}) \leftrightarrow (Q_{pre} \rightarrow Q_{post})$
9.$M_{gen-pred}$	$(C_{pre} \rightarrow C_{post}) \rightarrow (Q_{pre} \rightarrow Q_{post})$
10.$M_{spe-pred}$	$(Q_{pre} \rightarrow Q_{post}) \rightarrow (C_{pre} \rightarrow C_{post})$
11.$M_{exact-pred-2}$	$(C_{pre} \wedge C_{post}) \leftrightarrow (Q_{pre} \wedge Q_{post})$
12.$M_{gen-pred-2}$	$(C_{pre} \wedge C_{post}) \rightarrow (Q_{pre} \wedge Q_{post})$
13.$M_{guarded-gen-pred}$	$(Q_{pre} \rightarrow C_{pre}) \wedge ((C_{pre} \rightarrow C_{post}) \rightarrow (Q_{pre} \rightarrow Q_{post}))$

Table 2.1: Summary of specification matches

[ZW97] identifies **exact match** based on the pre and postconditions of Q and C (1st row in table 2.1), which corresponds to the case in which C and the desired component specified in Q are equivalent. As the equivalence between specifications is a strong requirement and in many cases a more general or more specific component can be useful, various relaxed notions of match are defined.

In **plug-in match** (2nd row in table 2.1), Q is matched by components with weaker preconditions and stronger postconditions i.e. the component postconditions imply (and, therefore, satisfy) the query postconditions, and the preconditions of the query imply the preconditions of the component. Intuitively, C can be used for obtaining the behaviour

[3]If the preconditions do not hold, the behaviour of the component is undefined.

specified in Q, but not the other way around. Plug-in match is also used in [SF97, PA97]. [JC93, JC94] use a subsumption test which is equivalent to plug-in match, and [LW94, Ame91] uses plug-in match for defining a notion of subtyping.

Plug-in post match (3rd row in table 2.1) does not consider the preconditions for determining a match. This notion correspond to the cases in which we only care about the results of the component execution. The preconditions of the component can be assured at a later stage by the user of the component. In this case, the relation that must hold between C and Q is that the postcondition of C must imply i.e. guarantee the postcondition of Q. This notion is also considered in [PA97].

The **guarded plug-in match** (4th row in table 2.1) restricts the plug-in match by introducing a guard stating that the component postconditions is required to imply the query postconditions only in cases where the component preconditions are satisfied. For an example of a case where this assumption is necessary, we refer the reader to [ZW97]. The same notion is used in [PA97], but called *weak plug-in*.

Guarded post match (6th row in table 2.1) relates to guarded plug-in match in the same way plug-in post match relates to plug-in match i.e. it is the result of dropping the precondition relation checking from guarded plug-in match. [SF97] also describes guarded post match. This notion is equivalent to *weak post* match in [PA97].

In **exact predicate match** (8th row in table 2.1), the relation between the preconditions and postconditions of the component must be equivalent to the relation between pre and postconditions of the query, that is, the functions (relation between pre and postconditions) specified in the query and in the component must be equivalent. As stated in [ZW97], $M_{exact-pre/post} \rightarrow M_{exact-pred}$, and they are equivalent in cases where $C_{pre} = Q_{pre} = true$ i.e. exact predicate match is less strict than exact match.

Generalized match (9th row in table 2.1), corresponds to the intuition that the description of components will be complete, while the description of the queries can be kept simple. This notion of match retrieves the components providing a functionality more general than the one specified in the query. Generalized match is less strict than plug-in match i.e. $M_{plug-in} \rightarrow M_{gen-pred}$.

Alternative versions of exact predicate match and generalized match can be obtained by replacing the implication relation between the preconditions and postconditions in both the component and the query by a conjunction (11th and 12th rows in table 2.1, respectively).

Guarded generalized predicate match (13th row in table 2.1) is derived from generalized match by restricting the constraints of generalized match to the domain defined by the preconditions (Q_{pre}) of the query.

Specialized match (10th row in table 2.1) retrieves components whose functionality is more specialized than the requested functionality. The rationale behind this notion of match is that a function whose specification is weaker than the query might still be interesting as a base to implement the desired functionality [ZW97].

[SF97] introduces the **partial compatibility match** (7th row in table 2.1). Using this notion of match, a component is matched if it computes the required results on a common domain. Such domain is defined by the conjunction of the preconditions of the query and the preconditions of the component. If this conjunction is not satisfiable i.e. the domains of the query and the component are disjoint, the formula will also be true. However, it is expected that disjoint domains will be rejected by signature matching.

Relaxed plug-in (5th row in table 2.1), also called *satisfies match* in [PA97], is introduced in [SF97]. Here, the guard for the relation between the component and query postconditions is the query precondition i.e. the checking of the relation is restricted to the domain given by the query preconditions. Notice that this notion of match is implied by the guarded plug-in match because of the first implication in the formula for relaxed plug-in i.e. the components retrieved under guarded plug-in are a subset of the ones retrieved under relaxed plug-in.

Table 2.1, adapted and extended from [CC00], provides a summary of different notions of match presented in [JC95, ZW95, PA97, SF97, Ame91, LW94, DL96].

Figure 2.2 shows the relation between the notions of match presented above. An arrow from one notion to another means that the first one is stronger i.e. the first one implies the second one. It is an extension of Figure 4 of [ZW97] with the notions of match not considered in that paper.

Chen and Cheng [CC00] discuss a measure of reusability and classify some of the afore-mentioned notions of match according to such measure. Given any component C and query Q, they consider a match $M(C, Q)$ to be *reuse-ensuring* iff the following relation holds:

$$M(C, Q) \wedge \{C_{pre}\}C\{C_{post}\} \rightarrow \{Q_{pre}\}C\{Q_{post}\}$$

where $\{p\}C\{q\}$ means that if p holds at the beginning of C's execution, C's execution will terminate with q satisfied. The intuitive meaning of the formula above is that, if a component is matched, and it has preconditions C_{pre} and postconditions C_{post}, the postconditions of the query will be satisfied by the execution of C provided that the preconditions of the query hold before its execution.

Under this definition, it is shown that exact-pre/post, plug-in, exact-pred-2, relaxed plug-in, and guarded-gen-pred matches are reuse-ensuring matches.

Relation to Semantic Web Service discovery

As has been presented in Section 1.1, Web services are described using a similar approach to Hoare's axiomatization of software components. In fact, the only difference that arise in the description of Web services is that, due to their range of application, they can in general have effects on the real world. Therefore, it is important to separate (information)

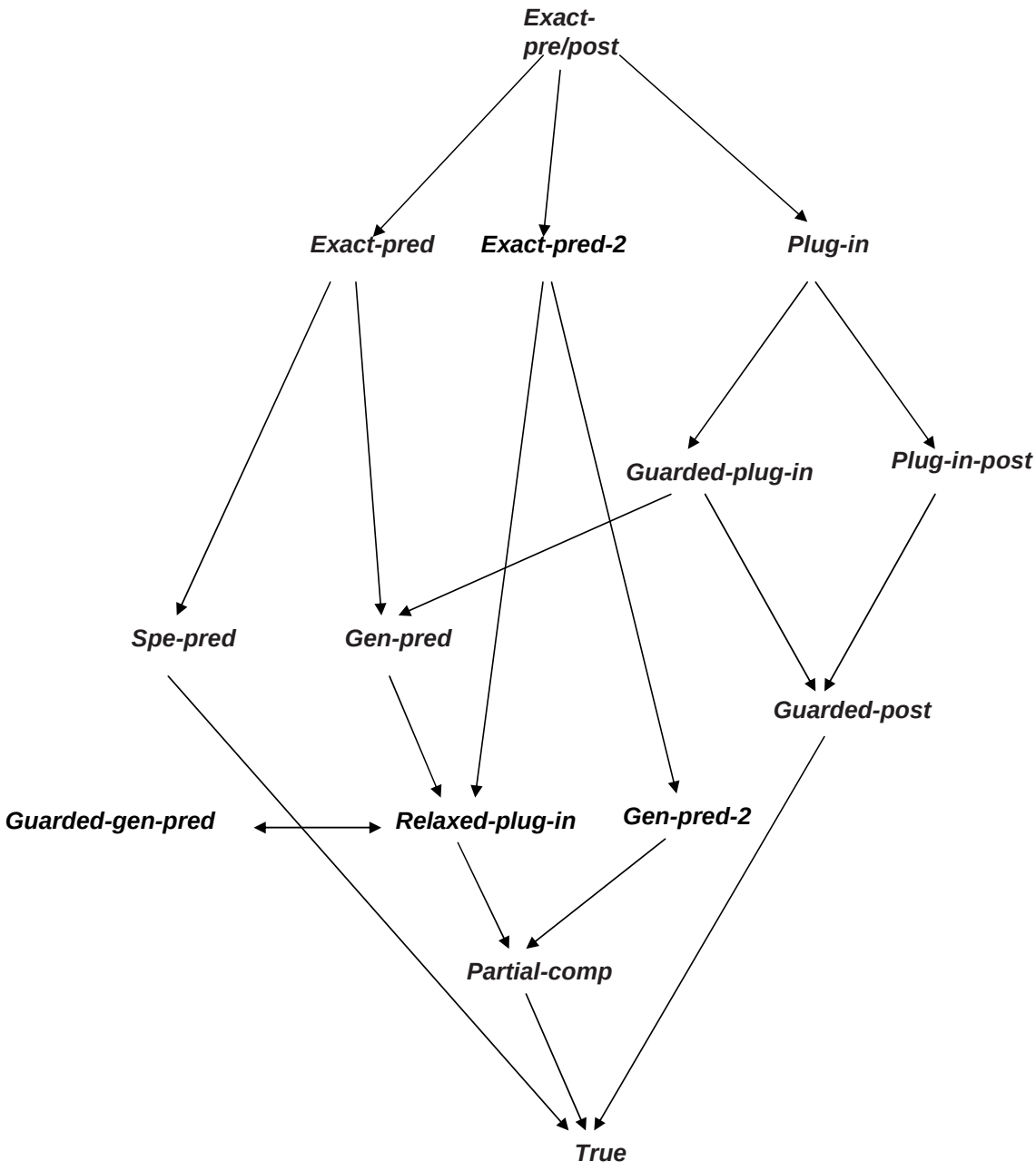


Figure 2.2: Lattice of Function Specification Matches

postconditions from real-world effects, and (information) preconditions from real-world assumptions, as different notions of match can be applied to these. Queries in software component retrieval are described in a similar way to Web service requests or goals.

Some of the notions of match discussed in the context of software component retrieval have been used in semantic Web service discovery, as will be seen in Section 2.2.2, where notions such as exact or plug-in match are applied to the automatic location of Web services. However, and as have been presented in Section 2.1.5, a conceptual model for Web service discovery introduces different steps where some notions of match investigated in software component retrieval are interesting while others do not have practical interest.

Goal Discovery. In goal discovery we do not expect the requester to directly provide a formal specification of the requester desires. In fact, if it is given, we will skip the goal discovery and goal refinement steps and proceed with the formalized goal to the service discovery step. Therefore, the notions of match introduced in this section are not applied.

Service Discovery. During service discovery we will only work with abstract service descriptions, which only contain a description of the results that can be provided by the service. Therefore, we will only consider postconditions and effects at this step, and only notions of match considering the set of results that can be provided by the service wrt. the set of results requested will be of interest. The *plug-in-post* notion of match presented before is the only of this kind and, as will be seen in Section 2.4.2, other notions based on some of the notions investigated in software component retrieval where precondition checking is dropped will be necessary.

Service Contracting. As explained in Section 2.1.5, service contracting involves the contracting, among the services selected during discovery, of a concrete service that fulfills the requester goal. In this case, the contracting capability of a service will describe the actual relation between the inputs and the results of the service, which is the way software components were commonly axiomatized. In addition, the requester goal will also include a description of the information that can be provided, which can be seen as the query preconditions in software component retrieval. The notions of match that will be presented in the following sections are based some of the notions investigated in the software component retrieval area.

Although many of the notions of match presented in this section cannot be directly used in our conceptual model for service discovery and contracting, it will be seen in the remainder of this chapter that most of the notions of match that we will used in our conceptual model have their roots on the work done in software component retrieval, with the adaptations necessary for the Web services domain. Particularly, while the notions of match discussed in this section focus on locating a software component that can be used in the place where the software component represented by the query could, in service

discovery we focus on what results can be delivered by the service. For example, the plug-in notion of match introduced in software component retrieval requires the component postconditions to be more specific than the query postconditions, while in our case we will reverse this relation, as we want a guarantee that the service will deliver what was requested in the goal i.e. every result described by the goal postcondition has to be a result of the service and, therefore, the goal postcondition has to imply the service postcondition.

2.2.2 State of the art on Web Service Discovery

In this section, we will discuss the work done in automatic Web service discovery based on the semantic description of the Web service functionality and the user requests. Other Web service discovery means that offer a low level of automation, such as the use of UDDI [BCE+02] registries to locate Web services, will not be discussed.

Discovery in OWL-S

[PKPS02] proposes a DAML-S-based[4] approach to the semantic matching between service advertisements and requests.

In [PKPS02], only the information transformation aspect of the service is considered i.e. only inputs and outputs are taken into account in the discovery process. A matching of a service profile and a request goal (also modelled as a profile) occurs when all the outputs of the goal are matched by (possibly a subset of) the outputs of the capability, and all the inputs of the capability are matched by (possibly a subset of) the inputs of the goal. Given a request, the matching algorithm performs this checking for all the available services. For each service, all the outputs of the request are matched against the outputs of the capability and, symmetrically, all the inputs of the capability are matched against the inputs of the request.

The match of both inputs and outputs relies on subsumption reasoning in DAML+OIL. Different degrees of match are determined for the checking of inputs and outputs depending on the subsumption relationship that holds between the pairs of outputs (resp. inputs), which in turn result on a ranking of the matching services. The preferred matches are *exact matches*, which correspond to the cases where the outputs (resp. inputs[5]) being matched are equivalent concepts[6] or the output from the request is a direct subclass of the output of the service [7]. The second best cases are *plug in matches*, where the output of the service subsumes the output of the request[8]. The third case, considered worse than the

[4]Unless explicitly stated, we will use the terms DAML-S and OWL-S interchangeably.

[5]Note that the inverse subsumption relation wrt. the one tested for outputs is used for inputs in all the notions of match applied.

[6]Exact-pre/post match in Section 2.2.1.

[7]Restricted case of plug-in match in Section 2.2.1.

[8]Similar to specialized match in Section 2.2.1.

previous ones, is the *subsumes match*, where the output of the request subsumes the output of the service[9]. The worst case is where no subsumption relation exists between the outputs, which corresponds to a *fail*. Different score is given to different types of matches and, taking the worst case match, the matching services are ranked (based on the output match scores, the input match scores are only considered for equally scoring services).

The approach in [PKPS02] presents some limitations and problems. First, every requester output have to be checked against all the outputs of every single service, so the number of necessary output subsumption tests is *n*m*s*, where *n* is the number of outputs in the request, *m* is the number of outputs in the capability, and *s* is the number of available services. The same applies for input checking. Second, only inputs and outputs are considered, thus only modelling the communicative aspects i.e. information flow of the service, excluding the effects on the real world. Third, some limitations are derived from the use of DAML+OIL[10]. DAML+OIL does not offer a formalism to express rules. Although, as described in section 1.1.1, the last OWL-S 1.1 introduces the use of DRS, KIF, or SWRL to express rules, the version of DAML-S used in [PKPS02] does not employ them. Because of this lack of rules, how the outputs of the service relate to its inputs is not defined, and the same applies for effects and assumptions. Therefore the functionality of the service is not completely captured. From the conceptual point of view, this approach does not differentiate discovery and contracting, and does not provide a fully usable mechanism for any of them. If regarded as service discovery, it introduces the consideration of inputs, which is not our intention. If regarded as service contracting, it does not model the relation between inputs and results and, therefore, cannot lead to the contracting of a concrete service whose usability is guaranteed.

In [LH03], also DAML-S and DL subsumption reasoning is used to perform discovery. In this approach, the whole profile (including inputs, outputs, preconditions and effects) is defined as a subconcept of the DAML-S service profile, and then classified in the subsumption hierarchy defined by the advertised service profiles. Also the kind of matches considered are different from [PKPS02]. Exact match is restricted to equivalent concepts[11], the same subsumption relation is applied to inputs and outputs for plug-in and subsumes matches[12], and intersection match is added. Intersection matches correspond to the cases where the intersection of the request and the service profile is satisfiable[13]. A match occurs whenever the requester and the service profile are compatible i.e. there is an intersection match, as the other types of match are special cases of this one.

First, the subsumption relation between the requester and the service profile is computed using RACER[14]. If no subsumption relationship can be established, then the satisfiability of the intersection of these concepts is checked. For that, the negated request

[9] Plug-in match in Section 2.2.1.

[10] The limitations also apply to OWL if OWL-S is used.

[11] Exact-pre/post in Section 2.2.1.

[12] Variant of plug-in and specialized matches in Section 2.2.1, respectively

[13] That is, there exists an assignment of variables for which $C_{pre} \wedge C_{post} \wedge Q_{pre} \wedge Q_{post}$ holds.

[14] http://www.sts.tu-harburg.de/ r.f.moeller/racer/.

is classified in the service profiles hierarchy. Profiles that subsumes but are not equal to the negated request, are considered to be intersection matches. Profiles subsumed by the negated request are considered to be failed matches.

[LH03] offers some experimental results with arbitrarily generated service profiles. It is shown that the computationally hard task is to classify new service profiles into the profile hierarchy. Once the profiles subsumption hierarchy is computed, matching a request is done in less than 20 milliseconds.

However, [LH03] does not solve some of the limitations encountered in [PKPS02]: the lack of rules for DAML+OIL results on an incomplete description of the service functionality and, therefore, it cannot be used for service contracting. This approach can be used for service discovery if the inputs and preconditions are removed, but not for service contracting.

The work in [BHRT03] also relies on the use of DAML-S as the ontology for service descriptions. A subset of DAML+OIL for which the difference operator[15] is semantically unique[16] is used to express the service profile inputs and outputs (preconditions and effects are not considered). However, service discovery is not formulated as a subsumption reasoning problem but as a rewriting problem i.e. how to rewrite the request in terms of available services. Given a service request and a service profile, a combination of Web services that satisfy as much as possible the outputs requested and that require as few as possible inputs not provided in the request are selected. Such combination is the so-called best profile cover of the request using the set of available (advertised) services.

Using the difference operator, the services whose outputs satisfy at least one of the outputs in the request i.e. the difference between the outputs of the request and the outputs of the service is not the whole set of outputs in the request, are identified. The same operation (using the difference operator in reverse order) is performed for the inputs. The set of services that have the smallest set of outputs in the request and inputs of the service not satisfied is selected. In this way, the best combination of services that provide the higher number of requester outputs and requires less inputs not given in the request is selected. The problem of determining such set of services is reduced to the problem of computing minimal traversals with minimal costs in an hypergraph.

This approach has the advantage of enhancing the discovery process using a simple type of service composition. In addition, incomplete matches can be found and information about what outputs or inputs are missing for a complete match is given, serving as an explanation that the requester can use to refine his request. However, the problem of the lack of rules for DAML+OIL and the related problems to capture the service functionality remain. Furthermore, the expressivity of the language is reduced to subsets for which the difference operator is semantically unique. Some experimental results are given for different sizes of the problem and different variants of the algorithm. The results do not clearly show the efficiency of the algorithm, and unacceptable response times are found

[15] Given concepts C and D, C-D is the information expressed in C and not in D.

[16] Please refer to [Tee94] for details.

for certain combinations of experiments data and algorithm variant. This approach, as the ones presented before, can be (if adapted) used for service discovery, but not for service contracting.

In general, all the OWL-S-based works miss the relation between input and output and effects, thus not capturing the functionality of Web Services. Although the use of SWRL, DRS and KIF has been introduced in the latest release of OWL-S, there is no work available so far in exploiting conditions and the relation between IOPEs for discovery. As a consequence, these approaches are not directly suitable for service contracting, and should be adapted for service discovery in our conceptual model.

Discovery in METEOR-S

As presented in Section 1.1.3, METEOR-S semantically annotates WSDL operations, inputs and outputs, and adds preconditions and effects to the operations description, using the extensibility elements of WSDL. The matching process uses subsumption reasoning to match operations, inputs, outputs, preconditions and effects of the annotated services against the ones of the template describing the request.

The annotation of registries and its specialization in domains helps to deal with a potentially huge number of published services. However, the METEOR-S approach to service description and discovery presents some limitations. Similarly to previous releases of OWL-S, the METEOR-S annotation does not relate inputs, outputs, preconditions and effects, therefore providing a not accurate description of the service functionality. Furthermore, the METEOR-S description is too WSDL centered, while approaches such as OWL-S and WSMO take a more flexible approach, not imposing any language for the grounding of Semantic Web Services.

It must be noticed that discovery in METEOR-S uses request templates similar to our pre-defined goals, that are to be parameterized by the requester. However, and as happened with the OWL-based approaches, service contracting is not addressed.

Discovery in LARKS

[SWKL02] defines the so-called Language for Advertisement and Request for Knowledge Sharing (LARKS), used to describe agent[17] capabilities. A capability specification in LARKS defines the context of the specification, the input and output variables, the constraints on these variables, the ontological descriptions used, and a textual description. As in OWL-S, a capability specification can be treated as a request or as an advertisement. Given a request and an advertisement, the matchmaking process can apply five different filters, namely [SWKL02]: 1) Context matching, 2) Profile comparison, 3) Similarity

[17] Although LARKS is a language for describing agent capabilities, it can equally be applied to Web services.

matching, 4) Signature matching, and 5) Constraint matching. The combination of filters that will be actually applied can be selected by the requester. In this way, the trade-off between accuracy and efficiency of the discovery can be selected.

The context in LARKS is defined as a set of keywords that describe the domain of the agent. Context matching computes the distance [Ros94] between the keywords of the request and the advertisement, and the subsumption relation between the concepts corresponding to the pairs of most similar words. The computed similarity, using a given threshold, will determine if there exists a match.

Profile comparison treats the request and advertisements as documents and determines the degree of similarity between them based on frequency and relevance of words in a document. If the similarity exceeds a given threshold, there is a match.

The profile comparison does not consider the structure of the specification, while similarity matching does. Such similarity is computed combining distance values for the pairs of input and output declarations, including their constraints. Again, if the computed similarity exceeds a threshold, the result will be a match.

Signature matching check if the input and output declarations of the request R and the advertisement A match. This is done using subtype inference rules and subsumption reasoning.

Constraint matching uses the notion of plug-in match. The logical implications are checked using subsumption reasoning for Horn clauses. Constraint matching uses the signature filter and, therefore, these two filters work together.

The application of different filters for discovery in LARKS has the advantage of customizing the trade-off between accuracy and efficiency by deciding the filters that will be applied. In addition, the input and output constraints can include the relation between the input and the output, capturing more accurately the functionality of the agent. However, the changes in the information space and in the state of the world are not differentiated, as effects of the action of an agent are implicitly encoded in the output constraints. In addition, discovery in LARKS does not differentiate between service discovery and contracting.

Other approaches

In [GCTB01], DAML+OIL subsumption reasoning is the central reasoning mechanism for discovery. It describes Web services as the boolean combination of a set of restrictions over datatype and abstract properties of the service. The advertised services are classified in a subsumption hierarchy, and different kind of matches are identified for a given request *S* and a service descriptions hierarchy, namely: equivalent concepts to *S*, sub-concepts of *S*, super-concepts of *S* that are subsumed by the general service description concept, and sub-concepts of any direct super-concept of *S* whose intersection with *S* is satisfiable. As for previous DAML+OIL-based approaches, the description of the service functionalities

is not complete, and therefore contracting cannot be achieved. However, as the approach in [LH03], is interesting for service discovery, as it relies on the classification of the Web services in a T-Box, and does not require the checking of each service separately against the given goal.

In [ZK04], an F-Logic-based approach for the description of Web Services is presented. The service concept describes the service in terms of non-functional properties such as provider, location, service type, etc., its behaviour (inputs, outputs, and the relation between them), and its operations (including the inputs and outputs for each operation). A requester expresses its goal as a query in terms of the service non-functional properties, behaviour, and operations. However, the service description cannot model world-altering services i.e. services with effects on the world. Furthermore, as it uses simple query answering, the description of the services is limited to ground facts, which considerably reduces the expressivity allowed for describing the service functionality and does not allow the description of abstract capabilities.

2.3 Description of Web Services and Goals

As presented in Section 2.2.1, software components and queries were described using Hoare's axiomatization [Hoa69]. The intended functionality of a program (C) is specified in terms of initial preconditions (C_{pre}) and postconditions (C_{post}). A query Q is similarly specified as (Q_{pre}, Q_{post}).

Semantic descriptions of Web Services proposed by OWL-S, WSMO, METEOR-S and IRS-III follow a similar approach, but they also differentiate between real-world effects and information postconditions, and between real-world assumptions and information preconditions.

In the following, we will assume Web service capabilities described in terms of their preconditions, assumptions, postconditions and effects, using the WSMO terminology[18] [RL*editors*04]. We will denote these by W_{pre}, W_{ass}, W_{post}, and W_{eff}, respectively. W_{pre} can be regarded as OWL-S and METEOR-S inputs, W_{ass} as preconditions, W_{post} as outputs, and W_{eff} as effects. From these, only postconditions and effects will be part of abstract capabilities of services. Notice that WSMO does not allow multiple capabilities and does not differentiate between abstract and contracting capabilities. However, and as related in our assumptions (see Section 2.1.5), the abstract capability might be automatically extracted from the contracting capability, as the former is precisely an abstraction of the latter and they must be consistent.

Regarding goals, we will also follow the WSMO terminology and denote their postconditions and effects by G_{post} and G_{eff}, respectively, interpreted as the state of the information and state of the world that is desired after the service execution. These can be regarded as OWL-S and METEOR-S outputs and effects of the request. However, it

[18] Such terminology is the same in the IRS-III approach

is important the conceptual differentiation of the requester point of view (goals) and the service point of view (capabilities), which is missing in OWL-S. With respect to WSMO, we introduce a new element $\mathcal{G}_{input}$, which describes the input information that can be provided by the requester and that is (directly or indirectly) part of the goal.

It can be seen that, even though we will adopt a WSMO terminology, the approaches that will be discussed in next sections are neutral with respect to the proposal for describing Web services. The basic requirement is that the descriptions capture the aspects of the Web service functionality and the goal related before. However, different approaches to Web service discovery will require different expressiveness for the description of such aspects, which might make WSMO more suitable than other approaches specially when the relation between inputs and postconditions and effects have to be described.

As shown in [LPL$^+$], the differences between the characterization of service capabilities in OWL-S and WSMO are not many, being the most relevant differences the language used to express them and the separation of goals and service capabilities in WSMO. A formal mapping between OWL-S and WSMO is being provided in [L*editors*04a], which can be used in the future to translate OWL-S descriptions into WSMO descriptions. If it is shown that the mapping does not imply a loss of relevant information, this will confirm that our discovery proposal is also applicable to OWL-S services.

In the following, we will go into the details of the different approaches to Web service discovery, starting with keyword-based discovery and ending with discovery based on precise descriptions of the Web Service functionality.

2.4 Automatic Web Service Discovery

In this section, we briefly discuss the use of keyword-based goal discovery, and we present a set-theoretic approach to Web service discovery. The formalization of this approach first (unrestrictedly) using first-order logic and later restricting the language expressiveness to interesting DL languages will be discussed.

As will be discussed in Section 2.6, the single approaches can require different techniques for mediation. The most important technique for us certainly is ontology merging and alignment. However, the mediation itself is outside the scope of this deliverable. Therefore, we will assume in the following that mediation is available and that it has already been performed.

2.4.1 Keyword-based Discovery

Keyword-based discovery is a basic ingredient in a complete framework for semantic Web service discovery. By performing a keyword-based search, the huge amount of available services can be filtered or ranked efficiently. The focus of service discovery is not on

keyword-based discovery but we consider this kind of discovery a useful technique in a complete semantic Web service discovery framework.

In a typical keyword-based scenario a keyword-based query engine is used to discover services. A query, which is basically a set of keywords, is provided as input to the query engine. The query engine match the keywords from the query against the keywords used to describe the service. A query with the same meaning can be formulated by using a synonyms dictionary, like WordNet [19] [Fel98]. The semantic of the query remains the same but because of the different keywords used, synonyms of previous ones, more services that possible fulfill the user request are found. Moreover, by using dictionaries like WordNet, as well as natural language processing techniques, an increase of the semantic relevance of search results (wrt. to the search request) can be achieved [RS95]; nonetheless, such techniques are inherently restricted by the ambiguity of natural language and the lack of semantic understanding of natural language descriptions by algorithmic systems.

The services descriptions are provided in terms of advertisements along with the keywords for categorization. That means that either keywords have to be extracted from the abstract capabilities and goal descriptions mentioned in the previous section, or such descriptions have to be extended to include relevant keywords.

While keyword-based discovery is an interesting technique that has to be kept in order to deal with already existing WSDL-based services, it can be better exploited for goal discovery. Given that we do not want to pose unrealistic requirements on the knowledge representation skills of the requester, the most likely way a requester will express his desire is by providing a textual description of it. Keyword-based match can be used to locate relevant pre-defined goals that also provide a formalization of the requester desire i.e. a requester goal. Once such pre-defined goal is located, appropriate tools will automatically refine or parameterize the pre-defined goal to actually reflect his desire, or will support him on manually refining such goal.

Pre-defined goals can of course also be browsed, but if the number of those is big, keyword-based match is a useful technique for the purpose of locating the ones relevant for the requester.

In the following, we will assume that a requester goal has already being generated from a pre-defined goal, having a formalization of the requester goal which includes the input information he can provide for the service provision.

2.4.2 Semantic Characterization of Results

Although keyword-based search is a widely used technique for information retrieval, it does not use explicit, well-defined semantics. The keywords used to retrieve relevant information do not have an explicit formalization and, therefore, do not allow inferencing to improve the search results.

[19] The WordNet homepage: http://www.cogsci.princeton.edu/~wn/

For these reasons, as a second approach we consider the use of controlled vocabularies with explicit, formal semantics. Ontologies are well-suited and prominent conceptual means for this purpose. They provide an explicit and shared terminology, explicate interdependencies between single concepts and, thus, are well-suited for the description of abstract capabilities and requester goals. Moreover, ontologies can be formalized in logics which enables the use of inference services for exploiting knowledge about the problem domain during the discovery process.

In this section we present a formal modelling approach for Web services and goals which is based on set theory and exploits ontologies as a formal, machine-processable representation of domain knowledge. We discuss service discovery based on this approach for simple semantic descriptions and how to implement such set-based model in the formal framework of logic.

The framework that we present in this section for Web service discovery based on simple semantic annotations is in fact similar to a model that has recently been proposed in [GMP04] for discovery in an e-Business setting. In some respects our approach is a generalization of the model discussed there.

An example. We will illustrate our approach with simple examples in the VTA context: we model some requester goals and Web service capabilities related to information about flights and train connections. For the sake of simplicity, we will consider in the following only postconditions (outputs) of a service and do not treat effects explicitly. Although they are conceptually different and requesters might want to apply different notions of match to them, our discussion can be equally applied to both.

The informal description of the requester goals is the following:

$\mathcal{G}_1$: I want to know about all flights from any place in Ireland to any place in Austria.

$\mathcal{G}_2$: I want to know about some flights from any place in Ireland to any place in Ireland.

$\mathcal{G}_3$: I want to know about all flights from Galway (Ireland) to Dublin (Ireland).

$\mathcal{G}_4$: I want to know about some flights from Innsbruck (Austria) to some place in Ireland (the client does not necessarily care which one).

$\mathcal{G}_5$: I want to know about some train connections from Galway (Ireland) to Dublin (Ireland).

Let us further assume four available Web services $\mathcal{W}_1, \ldots, \mathcal{W}_4$ exposing the following (informal) abstract capabilities:

$\mathcal{W}_1$ offers information about all flights from any place in Europe to any place in Europe.

$\mathcal{W}_2$ offers information about flights from all places in Ireland to all other places in Ireland, but not necessarily information about all such flights.

$\mathcal{W}_3$ offers information about all flights from all places in Ireland to Innsbruck (Austria).

$\mathcal{W}_4$ offers information about all train connections from Galway (Ireland) to Dublin (Ireland).

A set-based Modelling Approach

Frame-based or object-oriented modelling techniques have become popular approaches to system and domain modelling in both industry and academia. One main characteristic of these approaches is that the problem domain (or the *universe*) is understood as a set of objects and single objects can be grouped together into sets (or *classes*). Each class captures common (syntactic and semantic) features of their elements. Features can be inherited between classes by defining class hierarchies. This way, a problem domain can be structured as classes of objects and is basically understood as a collection of classes (or sets of things). In particular, ontologies are a popular knowledge-representation technique which usually exploit the very same modelling paradigm.

In such modelling approaches the main semantic properties that one is interested in are certain relationships between such sets of objects of the universe. Establishing and checking such relationships is the main reasoning task which allows agents to exploit knowledge formalized in the domain model (or ontology). From a knowledge representation perspective, this is a very natural and simple modelling approach for human beings.

Modelling of Web services and Goals. The provision of a service results in (wrt. a set of input values provided by the client) certain information given to the requester (output) and the achievement of certain changes in the state of the world. Hence, an abstract service capability can be described in terms of the results that can potentially be delivered by the service.

Goals describe the information the client wants to receive as a result of a service provision (outputs), as well as the effects on the state of the world that the client intends to achieve. This desire can be represented as sets of elements which are relevant to the client as the outputs and the effects of a service execution.

According to this view, Web services and goals are represented as *sets of objects*. The descriptions of these sets refer to ontologies that capture general knowledge about the problem domains under consideration.

An important observation in our approach is that the description of a set of objects for representing a goal or a Web service actually can be interpreted in different ways and, thus, the description by means of a set is not semantically unique: A modeler might want

to express that either *all* of the elements that are contained in the set are requested (in case of a goal description) or can be delivered (in case of a Web service description), or that only *some* of these elements are requested (or can be delivered). A modeler has some specific intuition in mind when specifying such set of relevant objects for a goal or Web service description and this intention essentially determines whether we consider two descriptions to match or not. Thus, these intuition should be stated explicitly in the descriptions of requested goals and abstract services.

If we take as an example the goal $\mathcal{G}_4$ and the service $\mathcal{W}_1$, the relevant objects they describe are:

Goal / Web service	Set R of relevant Objects	Intention of R
$\mathcal{G}_4$	$\{f \mid f$ is a flight starting at Innsbruck in Austria and ending at any city c located in Ireland $\}$	existential ($\exists$)
$\mathcal{W}_1$	$\{f \mid f$ is a flight starting at city s and ending at city e, s any city in Europe, e any city in Europe $\}$	universal ($\forall$)

Semantic Matching. The semantics of a goal or an abstract service description $\mathcal{D}$ is represented by a set of objects $R_{\mathcal{D}} \subseteq U$ (in a common universe U) which represent the set of *relevant objects* for the description as well as an explicit specification about the corresponding intention $I_{\mathcal{D}} \in \{\forall, \exists\}$ of the set.

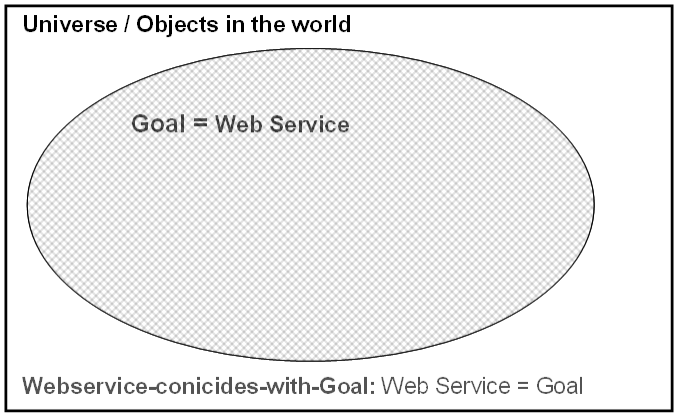
In order to consider a goal $\mathcal{G}$ and a Web service $\mathcal{W}$ to match on a semantic level, the sets $R_{\mathcal{G}}$ and $R_{\mathcal{W}}$ describing these elements have to be interrelated somehow; precisely spoken, we expect that some set-theoretic relationship between $R_{\mathcal{G}}$ and $R_{\mathcal{W}}$ has to exist. The most basic set-theoretic relationships that one might consider are the following:

- Set equality: $R_{\mathcal{G}} = R_{\mathcal{W}}$
- Goal description subset of Web service description: $R_{\mathcal{G}} \subseteq R_{\mathcal{W}}$
- Web service description subset of goal description: $R_{\mathcal{W}} \subseteq R_{\mathcal{G}}$
- Common element of goal and Web service description: $R_{\mathcal{G}} \cap R_{\mathcal{W}} \neq \emptyset$
- No common element of goal and Web service description: $R_{\mathcal{G}} \cap R_{\mathcal{W}} = \emptyset$

These set-theoretic relationships provide the basic means for formalizing our *intuitive understanding of a match* between goals and Web services in the real-world. For this reason, they have been considered to some extent already in the literature, for instance in [LH03] or [PKPS02] in the context of service matchmaking based on Description Logics, as presented in Section 2.2.2.

On the other hand, we have to keep in mind that in our model these sets actually only capture *one part of the semantics* of goals and service description, namely the relevant objects for the service requestor or service provider. The intentions of these sets in the semantic description of the goal or Web service are not considered but clearly affect whether a certain existing set-theoretic relationship between $R_{\mathcal{G}}$ and $R_{\mathcal{W}}$ is considered to actually correspond to (or formalize) an expected match. In the following we will discuss the single set-theoretical relations as well as their interpretation in detail:

- **Set equality**: $R_{\mathcal{G}} = R_{\mathcal{W}}$



Here the objects that are advertised by the abstract capability of $\mathcal{W}$ (and which thus can potentially be delivered by the service) and the set of relevant objects for the requester (given by $\mathcal{G}$) perfectly match, i.e. they coincide.

However, when considering the possible combinations of intentions $I_{\mathcal{W}}$ and $I_{\mathcal{G}}$ for $R_{\mathcal{W}}$ and $R_{\mathcal{G}}$, we get the following intuitive interpretations of the set-theoretic relationship:

1. The service requester wants to get all the objects that he has specified as relevant ($I_{\mathcal{G}} = \forall$), and the service is able to deliver all the objects specified in $R_{\mathcal{W}}$ ($I_{\mathcal{W}} = \forall$). In this case, the requester needs and the advertised service capability match perfectly.
2. The service requester wants to get some of the objects that he has specified as relevant ($I_{\mathcal{G}} = \exists$), whereas the service is able to deliver all the objects

specified in $R_{\mathcal{W}}$ ($I_{\mathcal{W}} = \forall$). In this case, the requester needs and the advertised service capability also match perfectly. In a sense, the service even over-satisfies the needs of the requester.

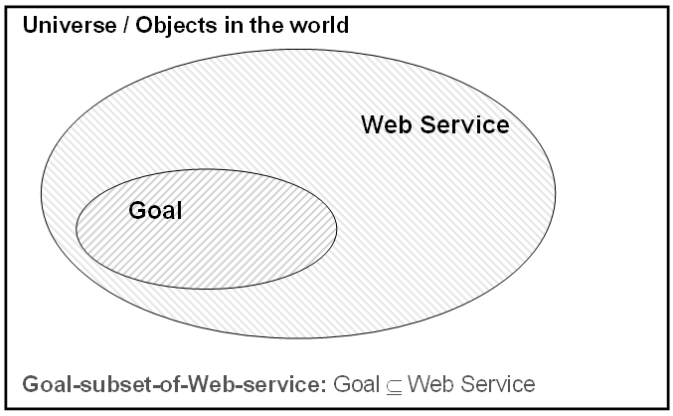
3. The service requester wants to get all the objects that he has specified as relevant ($I_{\mathcal{G}} = \forall$), whereas the service provider claims that the Web service is able to deliver only some of the objects specified in $R_{\mathcal{W}}$ ($I_{\mathcal{W}} = \exists$). In this case, the requester needs cannot be fully satisfied by the service. At best, the service can contribute to resolve the desire of the client. Thus, we consider this case as a *partial match*.
4. The service requester wants to get some of the objects that he has specified as relevant ($I_{\mathcal{G}} = \exists$), and the service provider claims that the Web service is able to deliver only some of the objects specified in $R_{\mathcal{W}}$ ($I_{\mathcal{W}} = \exists$). In this case, the requester needs and the advertised service capability again match. This time, the Web service does not necessarily over-satisfy the needs of the requester.

In [LH03] the situation where $R_{\mathcal{G}} = R_{\mathcal{W}}$ can be established is called *exact match*. However, in our model we do not necessarily consider the goal and the Web service as perfectly matching, as it will depend on the respective intentions of requester and provider.

- **Goal description subset of Web service description**: $R_{\mathcal{G}} \subseteq R_{\mathcal{W}}$

  Here the relevant objects that are advertised by the service provider form a superset of the set of relevant objects for the requester as specified in the goal $\mathcal{G}$.

  In other words, the service might be able to deliver all relevant objects (depending on the respective intention of $R_{\mathcal{W}}$).



When considering the possible combinations of intentions $I_{\mathcal{W}}$ and $I_{\mathcal{G}}$ for $R_{\mathcal{W}}$ and $R_{\mathcal{G}}$, we get the following intuitive interpretations of the set-theoretic relationship:

1. For intentions $I_{\mathcal{G}} = \forall$ and $I_{\mathcal{W}} = \forall$), the requester needs are fully covered by the Web service and we have a match.

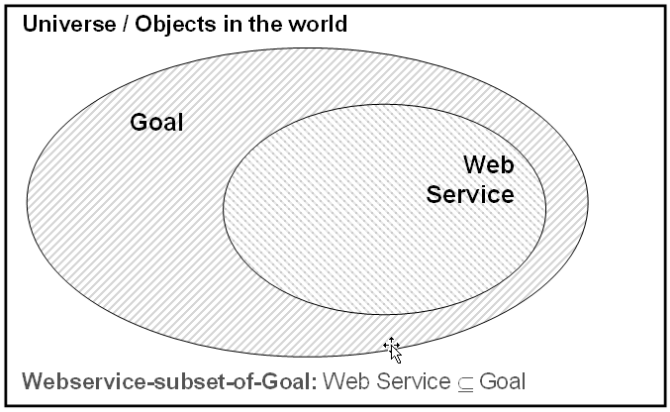
2. For intentions $I_{\mathcal{G}} = \exists$ and $I_{\mathcal{W}} = \forall$, the requester needs are fully covered by the Web service. In a sense, the service even over-satisfies the needs of the requester, since it actually is able to deliver all relevant objects for the client.
3. For intentions $I_{\mathcal{G}} = \forall$ and $I_{\mathcal{W}} = \exists$, we cannot determine whether there is actually a match, since we do not know which elements of $R_{\mathcal{W}}$ are actually delivered. It might happen that the actually delivered set contains all the elements of $R_{\mathcal{G}}$ (in that case we would have indeed a match), but it also might happen that it contains none of them. Hence, we consider this a *possible-match*. At contracting time, we can find three possibilities: a perfect match, a partial match, or a non-match.
4. For intentions $I_{\mathcal{G}} = \exists$ and $I_{\mathcal{W}} = \exists$, it is not guaranteed that a relevant element for the requester will be delivered by the provider. However, it is also possible that the service delivers some object of interest for the requester and, therefore, we consider this situation a possible match. At contracting time, there will be either a perfect match or a non-match.

In [LH03] the situation where $R_{\mathcal{G}} \subseteq R_{\mathcal{W}}$ can be established is called *plug-in match*. However, as can be seen above, it can correspond to a match or to a possible match.

- **Web service description subset of goal description**: $R_{\mathcal{W}} \subseteq R_{\mathcal{G}}$

  Here the relevant objects that are advertised by the service provider only form a subset of the set of relevant objects for the requester as specified in the goal $\mathcal{G}$.

  In other words, the service in general is not able to deliver all objects that are relevant objects for the requester.



When considering the possible combinations of intentions $I_{\mathcal{W}}$ and $I_{\mathcal{G}}$ for $R_{\mathcal{W}}$ and $R_{\mathcal{G}}$, we have the following situations:

1. For intentions $I_{\mathcal{G}} = \forall$ and $I_{\mathcal{W}} = \forall$, the requester needs can not be fully satisfied by the service. At best, the service can contribute to resolve the desire of the client. Thus, we consider this case a *partial match*.

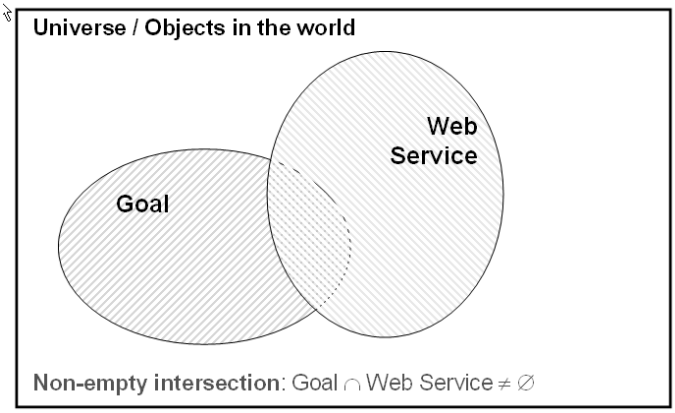
2. For intentions $I_{\mathcal{G}} = \exists$ and $I_{\mathcal{W}} = \forall$, the requester needs are fully covered by the Web service and, therefore, we have a match.
3. For intentions $I_{\mathcal{G}} = \forall$ and $I_{\mathcal{W}} = \exists$, the requester needs can not be fully satisfied by the service but the service can at least contribute to resolve the desire of the client. Thus, we consider this case a *partial match*.
4. For intentions $I_{\mathcal{G}} = \exists$ and $I_{\mathcal{W}} = \exists$, we have the same situation as in the second case and we consider this situation a match.

In [LH03] the situation where $R_{\mathcal{W}} \subseteq R_{\mathcal{G}}$ can be established is called *subsumes match*. However, in our model we can have a total or partial match.

- **Common element of goal and Web service description**: $R_{\mathcal{G}} \cap R_{\mathcal{W}} \neq \emptyset$

  Here there the set of relevant objects that are advertised by the service provider and the set of relevant objects for the requester have a non-empty intersection, i.e. there is at least an object in the common universe which is declared as relevant by both parties.

  In other words, the service in general is not able to deliver all objects that are relevant objects for the requester, but at least one such element can be delivered.



When considering the possible combinations of intentions $I_{\mathcal{W}}$ and $I_{\mathcal{G}}$ for $R_{\mathcal{W}}$ and $R_{\mathcal{G}}$, we distinguish the following cases:

1. For intentions $I_{\mathcal{G}} = \forall$ and $I_{\mathcal{W}} = \forall$, the requester needs can not be fully satisfied by the service. At best, the service can (weakly) contribute to resolve the desire of the client. Thus, we consider this case a *partial match*.
2. For intentions $I_{\mathcal{G}} = \exists$ and $I_{\mathcal{W}} = \forall$, the requester needs are fully covered by the Web service and we have a match.

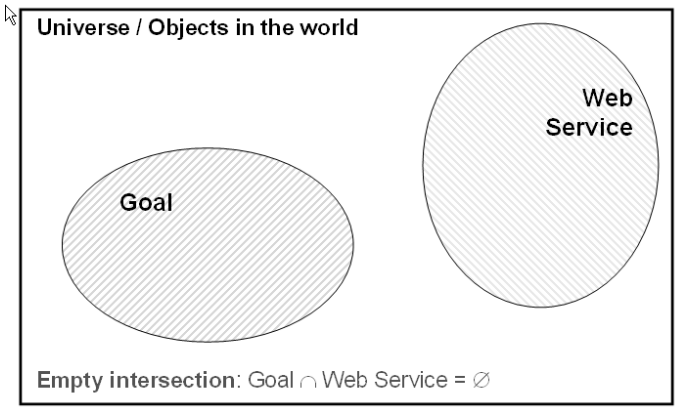
3. For intentions $I_{\mathcal{G}} = \forall$ and $I_{\mathcal{W}} = \exists$, the requester needs are not fully covered. We are not able to determine whether the service actually can deliver a common element of $R_{\mathcal{G}}$ and $R_{\mathcal{W}}$. However it might happen that the service can deliver some requested results, so we consider this situation a *possible-partial-match*. At contracting time, we can at best have a partial match, and in the worst case we will have a non-match.

4. For intentions $I_{\mathcal{G}} = \exists$ and $I_{\mathcal{W}} = \exists$, we consider the situation a possible match. At contracting time, we can have a perfect match or a non-match.

In [LH03] the situation where $R_{\mathcal{G}} \cap R_{\mathcal{W}} \neq \emptyset$ can be established is called *intersection match*. However, in our model we can have different flavors of match.

- **No common element of goal and Web service description**: $R_{\mathcal{G}} \cap R_{\mathcal{W}} = \emptyset$

  Here, the objects the Web service description refers to and the objects the requester goal refers to are disjoint. That means that there is no semantic link between both descriptions and, therefore, there is no element common to both.



  Hence, regardless of the corresponding intuitions for $R_{\mathcal{G}}$ and $R_{\mathcal{W}}$, we consider this situation a non-match.

  In [LH03] the situation where $R_{\mathcal{G}} \cap R_{\mathcal{W}} = \emptyset$ can be established is called *disjointness*.

Given some goal $\mathcal{G}$ and some Web service $\mathcal{W}$, Figure 2.3 summarizes the discussion and shows under which circumstances the presence of which set-theoretic relationship between $R_{\mathcal{G}}$ and $R_{\mathcal{W}}$ is considered as a match (Match), a partial match (PMatch), a possible match (poMatch), a possible partial match (ppMatch), or a non-match (NoMatch).

Intention of $\mathcal{G}$ / $\mathcal{W}$	$I_{\mathcal{W}} = \forall$		$I_{\mathcal{W}} = \exists$	
$I_{\mathcal{G}} = \forall$	$R_{\mathcal{G}} = R_{\mathcal{W}}$	Match	$R_{\mathcal{G}} = R_{\mathcal{W}}$	PMatch
	$R_{\mathcal{G}} \subseteq R_{\mathcal{W}}$	Match	$R_{\mathcal{G}} \subseteq R_{\mathcal{W}}$	poMatch
	$R_{\mathcal{G}} \supseteq R_{\mathcal{W}}$	PMatch	$R_{\mathcal{G}} \supseteq R_{\mathcal{W}}$	PMatch
	$R_{\mathcal{G}} \cap R_{\mathcal{W}} \neq \emptyset$	PMatch	$R_{\mathcal{G}} \cap R_{\mathcal{W}} \neq \emptyset$	ppMatch
	$R_{\mathcal{G}} \cap R_{\mathcal{W}} = \emptyset$	NoMatch	$R_{\mathcal{G}} \cap R_{\mathcal{W}} = \emptyset$	NoMatch
$I_{\mathcal{G}} = \exists$	$R_{\mathcal{G}} = R_{\mathcal{W}}$	Match	$R_{\mathcal{G}} = R_{\mathcal{W}}$	Match
	$R_{\mathcal{G}} \subseteq R_{\mathcal{W}}$	Match	$R_{\mathcal{G}} \subseteq R_{\mathcal{W}}$	poMatch
	$R_{\mathcal{G}} \supseteq R_{\mathcal{W}}$	Match	$R_{\mathcal{G}} \supseteq R_{\mathcal{W}}$	Match
	$R_{\mathcal{G}} \cap R_{\mathcal{W}} \neq \emptyset$	Match	$R_{\mathcal{G}} \cap R_{\mathcal{W}} \neq \emptyset$	poMatch
	$R_{\mathcal{G}} \cap R_{\mathcal{W}} = \emptyset$	NoMatch	$R_{\mathcal{G}} \cap R_{\mathcal{W}} = \emptyset$	NoMatch

Figure 2.3: Interaction between set-theoretic criteria, intentions and our intuitive understanding of matching.

Inconsistent descriptions for goals and services. We have not considered so far the possibility of inconsistent descriptions for goals and Web services i.e. descriptions where $R_{\mathcal{G}}$ or $R_{\mathcal{W}}$ are empty. They might occur in cases where there is a mistake in the description e.g. when the descriptions are quite complex or refer to several complex ontologies which are not themselves designed by the modeler.

Additionally, when just being ignored they can have an undesired impact on matching and thus discovery. Consider for example an inconsistent goal description, i.e. $R_{\mathcal{G}} = \emptyset$. If we check $\mathcal{G}$ for matching Web services using the Plugin-criterium, i.e. $R_{\mathcal{G}} \subseteq R_{\mathcal{W}}$, *every* Web service will match. For a user (who is not aware that his description is inconsistent, since otherwise he would usually not pose the query) the result would seem rather strange and even incorrect because all checked services actually will be matched.

We envision two ways to deal with inconsistent descriptions:

- *Do not allow the advertisement of inconsistent goal and Web service descriptions.* If somebody advertises a goal or service description which denotes an empty set, then just reject the description before it can take part in any discovery. During discovery we will work only with consistent descriptions. One has to be careful when considering the consistency-state of some goal or Web service description: the consistency does not depend exclusively on the description itself but as well on all the ontologies that the description refers to. Hence, changes to such ontologies potentially can lead to inconsistent goal and Web service descriptions. For this reason, one might have to reconsider the consistency of all depending goal and Web service descriptions that refer to some ontology when changing the ontology.

- *Check for inconsistency of goal and Web service descriptions during discovery and*

return acceptable results. In this case we do not assume the consistency of any of the descriptions which are involved, but we check the consistency during the discovery phase and deliver reasonable results or inform the requester or Web service provider that something is wrong with the advertised description. Checking the consistency basically means to determine whether there is some element $e \in U$ such that it satisfies the properties of the given goal (or Web service) description. It is not hard to extent the criteria in Figure 2.3 in this way.

We recommend to follow the first option for service descriptions and pre-defined goals. Although it will require to check the consistency of the service descriptions and pre-defined goals that are already advertised every time an ontology is changed, this can be done off-line i.e. separated from the discovery process itself. Therefore, it will not affect the efficiency of the discovery process.

Intentions. In DL based approaches to service discovery like [PKPS02], [LH03] the notion of ,,intention" has at present not been reflected explicitly. As we have shown above, intentions capture an important aspect of goal and Web service descriptions and determine whether certain set-theoretic criteria represent the expected match results.

The DL approaches available so far can be understood as covering only the lower left part of the table i.e. the situation where a goal has an existential intention and a Web service has universal intention, i.e. $I_{\mathcal{G}} = \exists, I_{\mathcal{W}} = \forall$.

Figure 2.3 shows in detail how the overall picture is affected when intentions come into play. In contrast to existing approaches where intentions of goal and Web service descriptions are fixed (and thus can not be affected by modelers), we believe that it is useful to give modelers additional freedom for precisely capturing the meaning of their descriptions.

Moreover, we believe that certain pairs of intentions will occur more often in practice than others: Web service providers have a strong interest in their Web services being discovered. If we compare the number of possible matches with a given goal under existential and universal intentions, it seems most likely that providers tend to use universal intentions, even if the description does not necessarily model the accurate capability of the service but only an abstraction of it. However, if a service provider wants to be more accurate with his Web service description then in many situations he would have to use the existential intention. Even under existential intention, there are actual matches detectable, when the requester uses goals with existential intention.

For service requesters (in particular in an e-Business setting) we expect that the existential intention will suffice in many situations, however the requester has the freedom to properly express stronger requests than existential goals (using universal intention) if he needs to get more accurate results.

The different intentions discussed above will determine what formal relationship between the requester goal and the abstract capability has to be tested. Figure 2.4 describes the relationship between goal and abstract service descriptions that has to be checked for each notion of match.

Intention of $\mathcal{G}$ / $\mathcal{W}$	$I_{\mathcal{W}} = \forall$		$I_{\mathcal{W}} = \exists$	
$I_{\mathcal{G}} = \forall$	Match	$R_{\mathcal{G}} \subseteq R_{\mathcal{W}}$	Match	**Not possible**
	PMatch	$R_{\mathcal{G}} \cap R_{\mathcal{W}} \neq \emptyset$	PMatch	$R_{\mathcal{G}} \supseteq R_{\mathcal{W}}$
	poMatch	**Not possible**	poMatch	$R_{\mathcal{G}} \subseteq R_{\mathcal{W}}$
	ppMatch	**Not possible**	ppMatch	$R_{\mathcal{G}} \cap R_{\mathcal{W}} \neq \emptyset$
	NoMatch	$R_{\mathcal{G}} \cap R_{\mathcal{W}} = \emptyset$	NoMatch	$R_{\mathcal{G}} \cap R_{\mathcal{W}} = \emptyset$
$I_{\mathcal{G}} = \exists$	Match	$R_{\mathcal{G}} \cap R_{\mathcal{W}} \neq \emptyset$	Match	$R_{\mathcal{G}} \supseteq R_{\mathcal{W}}$
	PMatch	**Not possible**	PMatch	**Not possible**
	poMatch	**Not possible**	poMatch	$R_{\mathcal{G}} \cap R_{\mathcal{W}} \neq \emptyset$
	ppMatch	**Not possible**	ppMatch	**Not possible**
	NoMatch	$R_{\mathcal{G}} \cap R_{\mathcal{W}} = \emptyset$	NoMatch	$R_{\mathcal{G}} \cap R_{\mathcal{W}} = \emptyset$

Figure 2.4: Which formal criteria should be used for checking different degrees of matching.

We can also establish that some notions of match will be preferred. It is clear that a *match* will be preferred over other notions. We also consider a *possible match* preferable to a *partial match*, as the possible match can turn out to be a perfect match at contracting time, while a partial match will never provide all that is requested. Obviously, a *partial match* will be preferred over a *possible partial match*, as the former offers a guarantee that at list some relevant results will be provided, while the latter can turn out to be a non-match. We show this ordering below, with $\preceq$ meaning that the notion on the left-hand side is preferred over the one on the right-hand side:

match $\preceq$ *possible-match* $\preceq$ *partial-match* $\preceq$ *possible-partial-match* $\preceq$ *non-match*

For the goal and Web service examples given before, the following matches will be established:

Match:

$\mathcal{G}_1$-$\mathcal{W}_1$, $\mathcal{G}_2$-$\mathcal{W}_1$, $\mathcal{G}_2$-$\mathcal{W}_2$, $\mathcal{G}_3$-$\mathcal{W}_1$, $\mathcal{G}_4$-$\mathcal{W}_1$, $\mathcal{G}_5$-$\mathcal{W}_4$

Possible-match:

$\mathcal{G}_3$-$\mathcal{W}_2$

Partial-match:

$\mathcal{G}_1$-$\mathcal{W}_3$

Matching scenario. During the discovery process the scenario for matching between goal and Web service descriptions in general is as follows: A requester specifies his goal (the refinement of a pre-defined goal) by means of a set of relevant objects and the respective intention ($(R_{\mathcal{G}}, I_{\mathcal{G}})$), and we check for a match using the respective criteria for intentions $(I_{\mathcal{G}}, I_{\mathcal{W}})$ from Figure 2.4. We will start searching different types of matches following the ordering given below, starting with the most preferred notion of match. .

The discussion shows that during discovery and matching intentions can be dealt with on a meta-level (in comparison to the set-theoretic notions), i.e. they do not directly affect single set-theoretic criteria and the necessary checks. Hence, we just need an implementation of the different set-theoretic criteria in order to realize a system for matchmaking. We will discuss a logic based approach as a candidate for the implementation of these formal criteria in Section 2.4.2.

Realization of the Approach in Logic

In the formalization of the approach presented above, we will not enforce at the moment any specific restriction on the expresiveness of the language used i.e. we will allow a full first-order language for describing requester goals and abstract capabilities.

Formally, we represent the sets $R_{\mathcal{W}}$ and $R_{\mathcal{G}}$ used for describing the semantics of a Web service by means of *unary predicates* in a first-order language. All elements of the universe which satisfy the predicate (according to its definition) are considered to be elements of the set.

The precise semantics of the predicate symbol $W_{post}(x)$[20] is defined by means of a set $\mathcal{W}$ of first-order formulae. In general, we assume that the formulae in $\mathcal{W}$ refer to some set of ontologies. Hence, these ontologies have to also be formally described in order to define the semantics of the predicate symbol $W_{post}(x)$ in a formal manner. The formal representation of these ontologies as a logical theory in our first-order framework is denoted by $\mathcal{O}$.

The set $\mathcal{W}$ of defining formulae must be chosen in such a way that under every interpretation I which is a model of $\mathcal{W}$ as well as the ontologies $\mathcal{O}$ to which the definition refers to, the predicate is interpreted as the respective set of relevant objects, i.e.

$$I \models \mathcal{W} \Leftrightarrow I(W_{post}(x)) = R_{\mathcal{W}} \tag{2.1}$$

for every interpretation I.

In this formal sense, $\mathcal{W}$ (and $\mathcal{O}$) determine the interpretation of the symbol W_{post} in the intended way. Obviously, the symbol W_{post} has to occur in $\mathcal{W}$.

[20] A similar predicate can be used for effects. However, and for simplicity reasons, we will limit the discussion to the postconditions of the service

In the simplest case, we can just use a single formula of the form

$$\mathcal{W} : \forall x.(\psi(x) \leftrightarrow W_{post}(x)) \tag{2.2}$$

where $\psi(x)$ is an arbitrary first-order formula with exactly one free variable x.

For goals $\mathcal{G}$ we do the same and define an unary predicate symbol $G_{post}(x)$ by means of a (closed) set $\mathcal{G}$ of first-order formulae and the formalization $\mathcal{O}$ of the set of ontologies which the definition refers to. Again, the definition $\mathcal{G}$ has to satisfy the property

$$I \models \mathcal{G} \Leftrightarrow I(G_{post}) = R_{\mathcal{G}} \tag{2.3}$$

for every interpretation I.

Again, in the simplest case $G_{post}(x)$ can be defined by a single formula of the form

$$\mathcal{G} : \forall x.(\phi(x) \leftrightarrow G_{post}(x)) \tag{2.4}$$

where $\phi(x)$ is an arbitrary first-order formula with exactly one free variable x.

Logical modelling of the set-theoretic relations. Given the logical definitions $\mathcal{W}$ of an abstract capability and $\mathcal{G}$ of a goal, we now want to show how to represent the set-theoretic criteria from Section 2.4.2 in our logical framework.

Similar to [LH03], we distinguished five types of criteria. Notice that this criteria correspond to the formalization of the set relationships used in Figure 2.4, not to the notions of match themselves i.e. the word *match* below refers to the relation between the sets described without considering intentions:

- **Exact-Match.** Here the sets of relevant objects of the Web service description and the goal description coincide: $R_{\mathcal{G}} = R_{\mathcal{W}}$. Each element $e \in U$ of the universe U which is in $R_{\mathcal{G}} \subseteq U$ is as well in $R_{\mathcal{W}} \subseteq U$ and viceversa.

  Formally, that means that we have to prove the following:

  $$\mathcal{W}, \mathcal{G}, \mathcal{O} \models \forall x.(G_{post}(x) \leftrightarrow W_{post}(x)) \tag{2.5}$$

  where $\mathcal{W}$ is the definition of the Web service, $\mathcal{G}$ is the definition of the goal, and $\mathcal{O}$ is a set of ontologies to which both descriptions refer.

  In this case we write $\mathcal{W} \equiv_{\mathcal{O}} \mathcal{G}$ to indicate this particular kind of match.

- **Subsumption-Match.** Here the set of relevant objects for the Web service is a subset of the set of relevant objects of the goal: $R_{\mathcal{W}} \subseteq R_{\mathcal{G}}$. Each element $e \in U$ of the universe U which is in $R_{\mathcal{W}} \subseteq U$ is as well in $R_{\mathcal{G}} \subseteq U$.

Formally, that means that we have to prove the following:

$$\mathcal{W}, \mathcal{G}, \mathcal{O} \models \forall x.(W_{post}(x) \rightarrow G_{post}(x)) \tag{2.6}$$

In this case we write $\mathcal{W} \sqsubseteq_{\mathcal{O}} \mathcal{G}$ to indicate this particular kind of match.

- **Plugin-Match.** Here the set of relevant objects for the Web service is a superset of the set of relevant objects of the goal: $R_{\mathcal{G}} \subseteq R_{\mathcal{W}}$. Each element $e \in U$ of the universe U which is in $R_{\mathcal{G}} \subseteq U$ is as well in $R_{\mathcal{W}} \subseteq U$.

  Formally, that means that we have to prove the following:

$$\mathcal{W}, \mathcal{G}, \mathcal{O} \models \forall x.(G_{post}(x) \rightarrow W_{post}(x)) \tag{2.7}$$

  In this case we write $\mathcal{W} \sqsupseteq_{\mathcal{O}} \mathcal{G}$ to indicate this particular kind of match.

- **Intersection-Match.** Here the intersection between the set of relevant objects for the Web service and the set of relevant objects of the goal is not the empty set: $R_{\mathcal{W}} \cap R_{\mathcal{G}} \neq \emptyset$. There is an element $e \in U$ of the universe U which is in both $R_{\mathcal{W}} \subseteq U$ and $R_{\mathcal{G}} \subseteq U$.

  Formally, that means that we have to prove the following:

$$\mathcal{W}, \mathcal{G}, \mathcal{O} \models \exists x.(G_{post}(x) \wedge W_{post}(x)) \tag{2.8}$$

  In this case we write $\mathcal{W} \sqcap_{\mathcal{O}} \mathcal{G}$ to indicate this particular kind of match.

- **Non-Match.** Here the intersection between the set of relevant objects for the Web service and the set of relevant objects of the goal is the empty set: $R_{\mathcal{W}} \cap R_{\mathcal{G}} = \emptyset$. There is no element $e \in U$ of the universe U which is in both $R_{\mathcal{W}} \subseteq U$ and $R_{\mathcal{G}} \subseteq U$.

  Formally, that means that we have to prove the following:

$$\mathcal{W}, \mathcal{G}, \mathcal{O} \models \neg\exists x.(G_{post}(x) \wedge W_{post}(x)) \tag{2.9}$$

  In this case we write $\mathcal{W} \|_{\mathcal{O}} \mathcal{G}$ to indicate a non-match.

For the formalization of the examples given at the beginning of the section following the formalizations above we assume that an appropriate set of ontologies $\mathcal{O}$ for geographical data and travelling are in place. In particular, we have a relation $in(x, y)$ which states that object x is geographically located in object y, a relation $flight(f, s, e)$ which states that f is a flight from location s to location e and a relation $train(t, s, e)$ which states that t is a train connection from location s to location e.

The modelling of the set of objects relevant to for example $\mathcal{G}_1$ would be:

$$\mathcal{G}_1 : \forall x.(\exists s, e\,(fligh(x, s, e) \wedge in(s, ireland) \wedge in(e, austria)) \leftrightarrow G_{1post}(x)) \quad (2.10)$$

where *ireland* and *austria* are constants in $\mathcal{O}$. The modelling of the relevant objects for, for example, $\mathcal{W}_3$ would be:

$$\mathcal{W}_3 : \forall x.(\exists s\, fligh(x, s, innsbruck) \wedge in(s, ireland) \leftrightarrow W_{3post}(x)) \quad (2.11)$$

where *innsbruck* is a constant in $\mathcal{O}$ for which the relation $in(innsbruck, austria)$ holds.

As we have universal intentions for both the goal and the abstract capability, according to Figure 2.4, we have to check for a match the following formal relation:

$$\mathcal{W}_3, \mathcal{G}_1, \mathcal{O} \models \forall x.(G_{1post}(x) \rightarrow W_{3post}(x)) \quad (2.12)$$

which corresponds to the relation:

$$\mathcal{W}_3, \mathcal{G}_1, \mathcal{O} \models \forall x.(\exists s, e\,(flight(x, s, e) \wedge in(s, ireland) \wedge in(e, austria)) \rightarrow flight(x, s, innsbruck) \wedge in(s, ireland)) \quad (2.13)$$

which does not hold.

For checking the next preferred possible notion of match i.e. partial match, we have to check that:

$$\mathcal{W}_3, \mathcal{G}_1, \mathcal{O} \models \exists x.(\exists s, e\,(flight(x, s, e) \wedge in(s, ireland) \wedge in(e, austria)) \wedge (\exists s\, flight(x, s, innsbruck) \wedge in(s, ireland))) \quad (2.14)$$

which holds for $e = innsbruck$ if the domain ontology says that $in(innsbruck, austria)$. Therefore, $\mathcal{W}_3$ is a partial match for $\mathcal{G}_1$.

2.4.3 Using DL for Characterizing Results

In this section, we investigate the restriction of the expressiveness allowed to describe abstract capabilities and requester goals in order to increase the efficiency of the service discovery process.

We will briefly introduce Description Logics (DLs), describe how DL-based ontology languages can be applied in Web service discovery, and analyze the computational benefits we get from the use of DL modelling and DL reasoners. We will also discuss DLs that provide customized datatypes and datatype predicates, as they can be interesting for Web service discovery.

Description Logics

Description Logics (DLs) [BN03] are a family of class-based knowledge representation formalisms, equipped with well-defined model-theoretic semantics [BMNPS02]. They were first developed to provide formal, declarative meaning to semantic networks [Qui67] and frames [Min81], and to show how such structured representations can be equipped with efficient reasoning tools. The basic notions of Description Logics are classes, i.e., unary predicates that are interpreted as sets of objects, and properties, i.e., binary predicates that are interpreted as sets of pairs.

Description Logics are characterized by the constructors that they provide to build complex class and property descriptions from atomic ones. For example, 'elephants with their ages greater than 20' can be described by the following DL class description:[21]

$$\mathsf{Elephant} \sqcap \exists age.\ >_{20},$$

where Elephant is an atomic class, age is an atomic datatype property, $>_{20}$ is a customized datatype (treated as a unary datatype predicate) and $\sqcap$, $\exists$ are class constructors. As shown above, datatypes and predicates (such as $=, >, +$) defined over them can be used in the constructions of class descriptions. Unlike classes, datatypes and datatype predicates have obvious (fixed) extensions; e.g., the extension of $>_{20}$ is all the integers that are greater than 20. Due to the differences between classes and datatypes, there are two kinds of properties: (i) object properties, which relate objects to objects, and (ii) datatype properties, which relate objects to data values, which are instances of datatypes.

Class and property descriptions can be used in axioms in DL knowledge bases. DL *Axioms* are statements that describe (i) relations between class (property) descriptions, (ii) characteristics of properties, such as asserting a property is functional, or (iii) instance-of relations between (pairs of) individuals and classes (properties). We can use DL axioms to represent concepts and constraints in an ontology. For example, we can define the class AdultElephant with the following DL axiom

$$\mathsf{AdultElephant} \equiv \mathsf{Elephant} \sqcap \exists age.\ >_{20}; \tag{2.15}$$

we can assert that the property age is functional (e.g., Elephants can have at most 1 age):

$$\mathsf{Func}(age);$$

we can also assert that the object Ganesh is an instance of the class description 'Elephants who are older than 25 years old':

$$\mathsf{Ganesh} : (\mathsf{Elephant} \sqcap \exists age.\ >_{25}). \tag{2.16}$$

[21] Readers are referred to [BN03] for detailed descriptions of DL syntax and semantics.

A DL system not only stores axioms, but also offers *services* that *reason* about them. Typically, reasoning with a DL knowledge base is a process of discovering implicit knowledge entailed by the knowledge base. Reasoning services can be roughly categorized as basic services, which involve the checking of the truth value for a statement, and complex services, which can be built upon basic ones. Let Σ be a knowledge base, $\mathcal{L}$ a Description Logic, C, D $\mathcal{L}$-classes, a an individual name. Principle basic reasoning services include:

Knowledge Base Satisfiability is the problem of checking whether there exists a model $\mathcal{I}$ of Σ.

Concept Satisfiability is the problem of checking whether there exists a model $\mathcal{I}$ of Σ in which $C^{\mathcal{I}} \neq \emptyset$.

Subsumption is the problem of verifying whether in every model $\mathcal{I}$ of Σ we have $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$.

Instance Checking is the problem of verifying whether in every model $\mathcal{I}$ of Σ we have $\mathsf{a}^{\mathcal{I}} \in C^{\mathcal{I}}$.

For instance, given the axioms (2.15) and (2.16), a DL reasoner (with datatype support) should be able to infer that Ganesh is an AdultElephant.

The most common complex services include classification and retrieval. Classification is a problem of putting a new class in the proper place in a taxonomic hierarchy of *class names*; this can be done by subsumption checking between each named class in the hierarchy and the new class. The location of the new class, let us call it C, in the hierarchy will be between the most specific named classes that subsume C and the most general named classes that C subsumes. TBox classification, which computes the taxonomic hierarchy of concept names mentioned in a TBox, is a special case of classification, where $\top$ is chosen as the 'new' concept. Retrieval (or query answering) is a problem of determining the set of individuals that instantiate a given class; this can be done naively by instance checking between each named individual and the given class.

Ontologies and Description Logics

Ontology is a term borrowed from philosophy that refers to the science of describing the kinds of entities in the world and how they are related. In computer science, ontology is, in general, a 'representation of a shared conceptualisation' of a specific domain [Gru93a, UG96]. Ontologies [Gru93a, UG96] have been proposed to provide shared and precisely defined terms and constraints to describe the meaning of Web resources (including Web services).

An *ontology* typically consists of a hierarchical description of important concepts in a domain, along with descriptions of the properties of each concept, and constraints about

these concepts and properties. An ontology usually corresponds to a set of class and property axioms in Description Logics. Vocabulary in an ontology can be expressed by named classes and properties. Background assumptions/constraints can be represented by general class and property axioms. Sometimes, an ontology corresponds to a DL knowledge base. For example, in the OWL Web ontology language [BvHH$^+$04], an ontology also contains instances of important classes and relationships among these instances, which can be represented by DL individual axioms.

High quality ontologies crucially depend on the availability of a well-defined semantics and powerful reasoning tools [BHS02, Pan04a]. Description Logics address *both* these ontology needs. Unsurprisingly, well known ontology languages such as OIL, DAML+OIL and OWL use DL-style model-theoretic semantics.

Among these Web ontology languages, OWL is particularly important. OWL has been adopted as the standard (W3C recommendation) for expressing ontologies in the Semantic Web. There are three sub-languages of OWL: OWL Lite, OWL DL and OWL Full. In this section, when we mention 'OWL' we usually mean 'OWL DL' because OWL DL, equivalent to the $\mathcal{SHOIN}(\mathbf{D}^+)$ DL, is the most expressive decidable sub-language of OWL, while OWL Lite is simply a sub-language of OWL DL and OWL Full can be seen as an unsuccessful attempt at integrating RDF(S) and OWL DL [Pan04b]. The reader is referred to [PSHH04] for details of the abstract syntax and model-theoretic semantics of OWL DL.

Description Logics and Service Discovery

Description Logics have distinguished logical properties. They emphasize on the decidability of key reasoning problems. Modern DL reasoners, such as FaCT [Hor98], FaCT++, RACER[HM01], DLP [PS99], and Pellet[22] have demonstrated that, even with expressive DLs, highly optimized implementations can provide acceptable performance in realistic applications. In other words, thoughtful optimization techniques ([Hor97, HPS98, HS02, Hor03]) have moved the boundaries of 'tractability' to somewhere very close to EXPTIME-hard, or worse ([Don03]).

In particular, it can be seen that computing the subsumption relation between concepts is equivalent to determine the set inclusion relation between the sets described by such concepts. We can rewrite the formal relations introduced in the previous section in terms of DL subsumption of concepts, equivalence of concepts, and concept satisfiability. Given the postconditions of an abstract capability (W_{post}) and a requester goal (G_{post}) modelled as DL concepts, we need to check the following:

1. **Exact-Match** If W_{post} and G_{post} are equivalent concepts, we have an exact match i.e. $C_A \equiv C_R$ has to be checked.

[22] http://www.mindswap.org/2003/pellet/index.shtml

2. **Plugin-Match** If G_{post} is a sub-class of W_{post} i.e. W_{post} subsumes G_{post} we have a Plugin match i.e. $C_R \sqsubseteq C_A$ has to be checked.

3. **Subsumption-Match** If G_{post} is a super-class of W_{post} i.e. G_{post} subsumes W_{post}, we have a Subsumes match i.e. $C_A \sqsubseteq C_R$ has to be checked.

4. **Intersection-Match** If the intersection of W_{post} and G_{post} is satisfiable, we have an intersection match i.e. $\neg(C_A \sqcap C_R \sqsubseteq \bot)$ has to be checked.

5. **Non-match** If none of the above relations hold, we have a non-match i.e. $C_A \sqcap C_R \sqsubseteq \bot$ has to be checked.

If we restrict the expressivity allowed for describing abstract capabilities and requester goals (for discovery) to the DL languages that current DL reasoners can efficiently deal with, we can efficiently exploit subsumption reasoning. RACER provides efficient subsumption reasoning for $\mathcal{SHIQ}$ with incomplete reasoning for nominals. FACT++ supports $\mathcal{SHIF}(\mathcal{D})$. Pellet provides sound and complete reasoning for $\mathcal{SHIN}(\mathcal{D})$ and $\mathcal{SHON}(\mathcal{D})$, and sound but incomplete for $\mathcal{SHOIN}(\mathcal{D})$.

In [LH03], RACER is used to classify Web services in the T-Box, which is time-consuming but can be done off-line at publishing time. Once the TBox is classified, experimental results show that checking the subsumption relation between the user request and the Web services in the TBox can be done within 20 milliseconds. In additon, and as it was shown in the Semantic Web Fred (SWF)[23] project ([KSF04]), using more expressive logics (First-order Logic) for set based modelling and a theorem prover for set-based discovery in principle implies checking every available service individually, as theorem provers are not optimized for this reasoning task. For a reduced test set of four available services, the theorem prover required between one and two seconds to determine the existence of a matching service. Under the assumption that eventually a big number of services will be available to the discovery engine, a faster filtering of relevant services before evaluating them one by one is essential to make the discovery scalable. These results clearly suggest that using DL reasoners to index Web services at publication time can be a useful, restricting more detailed and more expensive contracting to a (ideally small) subset of all the available Web services. Notice that a theorem prover could be also optimized for classification, but it would require a bigger effort, and it is not clear whether additional expressivity is required for abstract capabilities.

What to classify? One of the major benefits of using DL reasoners is that we can have the available Web services classified before the discovery process takes place, as DL reasoners are optimized for this task. Another possibility is to classify the pre-defined goals. These two options have the following advantages and disadvantages:

[23] http://www.deri.at/research/projects/swf/

- The use of pre-defined, formalized goals is expected in our conceptual model. If we restrict such goals to be described in a DL language, we can classify them off-line in a DL reasoner TBox. When publishing a Web service, its subsumption relationship with the classified pre-defined goals can be checked, and a wgMediator [RL*editors*04] can be generated to link the Web service to a pre-defined goal it (totally or partially, depending on the subsumption relation computed) fulfills. In this way, and as goal discovery should result on a (refined) pre-defined goal, and Web service discovery is expected to require matching these refined pre-defined goals against published Web services, Web service discovery is reduced to exploring the Web services linked via wgMediators to the used pre-defined goal.

- Another option is to classify Web services when they are published. It is expected that the number of Web services available will be considerably higher than the number of pre-defined goals and, therefore, we would have to deal with bigger TBoxes and worse classification times. However, these classification times do not necessarily affect the time to answer an incoming goal. Another consideration is that in this case the subsumption relation between a refined pre-defined goal and the classified Web services would have to be computed for each discovery request, while in the previous solution subsumption checking is only required once for each Web service, and only at publication time. However, notice that in this case we directly obtain the subsumption relation between the concrete refined pre-defined goal and the Web services, while in the previous case the direct relation between those is not known but only their relation via the pre-defined goal.

We suggest following the second path, as computing the direct relation between the concrete requester goal and the abstract capabilities of published services will provide a better filtering than only checking the relation of these to the pre-defined goal. In addition, as shown in [LH03], computing the relation of the requester goal with respect to a classified TBox containing available services can be done efficiently[24]. Notice that this choice is under the assumption that abstract capabilities of available services will constitute a structured domain and not a set of hardly related concepts.

Notice that the intention of the provider cannot be directly modelled using DL. Therefore, the intention will be an annotation of the concept describing the abstract results of the service, and it will be used to select what services having a given relation to the goal will be selected.

In [L*editors*04b] the use of RACER and Pellet has been investigated to classify abstract service capabilities and goals. The results show that the treatment of nominals is not fully supported for the $\mathcal{SHOIN}$ DL, which limits their use in abstract capabilities and requester goals for service discovery. However, two possible solutions are foreseen:

[24]Notice that instead of checking the satisfiability of the intersection of the goal and each available service, the negated goal can be classified and non-Matches will correspond to the abstract capabilities subsumed by the negated goal.

- Forbidding the use of nominals for service discovery and considering instance values only in contracting, or
- providing an algorithm for replacing instances in service descriptions and requested goals by corresponding concepts.

Analyzing which option is more adequate for service discovery is out of the scope of this deliverable and subject of future work. More detailed discussions along this line can be found in the section of "Discussion: Reasoning with OWL-E" (page 60).

OWL-E: Supporting Customised Datatypes and Datatype Predicates in Ontologies

One of the most useful features of the above Web ontology languages is the support of datatypes (such as strings and integers) and datatyped values (such as integers 1,2,3 etc). Specifically, one can describe not only relationships between terms, but also relationships between terms and datatypes, e.g., the age of an Elephant, the $price$ of a Product, the $date$ of a Conference and the $postcode$ of a Restaurant. It is these kinds of 'real world' relationships, which provide useful concrete information, that ontology applications often require.

Customised datatypes and datatype predicates (such as $>, +$) are very helpful because it is often necessary to enable users to define their own datatypes and datatype predicates for their applications. For instance, in a computer sales ontology, a service requirement may ask for a PC with memory size greater than 512Mb, unit price less than 700 pounds and delivery date earlier than 15/03/2004. Here 'greater than 512', 'less than 700' and 'earlier than 15/03/2004' are customised datatypes of base datatypes integer, integer and date, respectively. Similar use cases arise in the VTA domain, where a client can ask for a flight cheaper than 300 euros on 15/03/2004 and arriving before 20:00.

OWL DL has a strong limitation on datatype support [Pan04a]. It supports neither datatype predicates with arbitrary arities, nor customised datatypes, not to mention customised datatype predicates. To overcome these limitations, [PH04, Pan04b] propose OWL-E, equivalent to the $\mathcal{SHOIQ}(\mathcal{G})$ DL, which is a decidable extension of both OWL DL and DAML+OIL, which provides customised datatypes and predicates; in fact, all the basic reasoning services of OWL-E are decidable.

OWL-E provides datatype expressions based on the datatype group approach [Pan04b], which can be used to represent customised datatypes and datatype predicates. Table 2.2 shows the kind of datatype expression OWL-E supports, where u is a datatype predicate URIref, "s_i"^^d_i are typed literals, $v_1, \ldots, v_n$ are (possibly negated) unary supported predicate URIrefs, P, Q are datatype expressions and $\Phi_{\mathcal{G}}$ is the set of supported predicate URIrefs in a datatype group $\mathcal{G}$. OWL-E provides some new classes descriptions, which are listed in Table 2.3, where $T, T_1, \ldots, T_n$ are datatype properties (where $T_i \not\sqsubseteq T_j, T_j \not\sqsubseteq T_i$

Abstract Syntax	DL Syntax	Semantics
rdfs:Literal	$\top_{\mathrm{D}}$	$\Delta_{\mathbf{D}}$
owlx:DatatypeBottom	$\bot_{\mathrm{D}}$	$\emptyset$
u a predicate URIref	u	$u^{\mathbf{D}}$
$\mathtt{not}(u)$	$\overline{u}$	if $u \in \mathbf{D}_{\mathcal{G}}$, $\Delta_{\mathbf{D}} \setminus u^{\mathbf{D}}$ if $u \in \Phi_{\mathcal{G}} \setminus \mathbf{D}_{\mathcal{G}}$, $(\mathsf{dom}(u))^{\mathbf{D}} \setminus u^{\mathbf{D}}$ if $u \notin \Phi_{\mathcal{G}}$, $\bigcup_{n>1}(\Delta_{\mathbf{D}})^n \setminus u^{\mathbf{D}}$
$\mathtt{oneOf}(\text{“}s_1\text{”}\hat{}\hat{}d_1 \ldots \text{“}s_n\text{”}\hat{}\hat{}d_n)$	$\{\text{“}s_1\text{”}\hat{}\hat{}d_1, \ldots, \text{“}s_n\text{”}\hat{}\hat{}d_n\}$	$\{(\text{“}s_1\text{”}\hat{}\hat{}d_1)^{\mathbf{D}}\} \cup \cdots \cup \{(\text{“}s_n\text{”}\hat{}\hat{}d_n)^{\mathbf{D}}\}$
$\mathtt{domain}(v_1, \ldots, v_n)$	$[v_1, \ldots, v_n]$	$v_1^{\mathbf{D}} \times \cdots \times v_n^{\mathbf{D}}$
$\mathtt{and}(P, Q)$	$P \wedge Q$	$P^{\mathbf{D}} \cap Q^{\mathbf{D}}$
$\mathtt{or}(P, Q)$	$P \vee Q$	$P^{\mathbf{D}} \cup Q^{\mathbf{D}}$

Table 2.2: OWL-E datatype expressions

Abstract Syntax	DL Syntax	Semantics
restriction($\{T\}$ someTuplesSatisfy(E))	$\exists T_1, \ldots, T_n.E$	$\{x \in \Delta^{\mathcal{I}} \mid \exists t_1, \ldots, t_n.\langle x, t_i\rangle \in T^{\mathcal{I}}$ (for all $1 \leq i \leq m) \wedge \langle t_1, \ldots, t_n\rangle \in E^{\mathbf{D}}\}$
restriction($\{T\}$ allTuplesSatisfy(E))	$\forall T_1, \ldots, T_n.E$	$\{x \in \Delta^{\mathcal{I}} \mid \forall t_1, \ldots, t_n.\langle x, t_i\rangle \in T^{\mathcal{I}}$ (for all $1 \leq i \leq m) \rightarrow \langle t_1, \ldots, t_n\rangle \in E^{\mathbf{D}}\}$
restriction($\{T\}$ minCardinality(m) someTuplesSatisfy(E))	$\geqslant m T_1, \ldots, T_n.E$	$\{x \in \Delta^{\mathcal{I}} \mid \sharp\{\langle t_1, \ldots, t_n\rangle \mid \langle x, t_i\rangle \in T^{\mathcal{I}}$ (for all $1 \leq i \leq m) \wedge \langle t_1, \ldots, t_n\rangle \in E^{\mathbf{D}}\} \geq m\}$
restriction($\{T\}$ maxCardinality(m) someTuplesSatisfy(E))	$\leqslant m T_1, \ldots, T_n.E$	$\{x \in \Delta^{\mathcal{I}} \mid \sharp\{\langle t_1, \ldots, t_n\rangle \mid \langle x, t_i\rangle \in T^{\mathcal{I}}$ (for all $1 \leq i \leq m) \wedge \langle t_1, \ldots, t_n\rangle \in E^{\mathbf{D}}\} \leq m\}$
restriction(R minCardinality(m) someValuesFrom(C))	$\geqslant m R.C$	$\{x \in \Delta^{\mathcal{I}} \mid \sharp\{y \mid \langle x, y\rangle \in R^{\mathcal{I}} \wedge y \in C^{\mathcal{I}}\} \geq m\}$
restriction(R maxCardinality(m) someValuesFrom(C))	$\leqslant m R.C$	$\{x \in \Delta^{\mathcal{I}} \mid \sharp\{y \mid \langle x, y\rangle \in R^{\mathcal{I}} \wedge y \in C^{\mathcal{I}}\} \leq m\}$

Table 2.3: OWL-E introduced class descriptions

for all $1 \leq i < j \leq n$),[25] R is an object property, C is a class, E is a datatype expression or a datatype expression URIref, and $\sharp$ denotes cardinality. Note that the first four are datatype group-based class descriptions, and the last two are qualified number restrictions.

Case Study: Matchmaking in OWL-E

Let us use an example to illustrate how to use OWL-E for service discovery.

We can use the OWL-E to describe both abstract service capabilities and requester goals. More precisely, they can be represented as an OWL-E class or class restriction, e.g. the capability that memory size should be either 256Mb or 512Mb, can be represented the datatype expression-related concept $\exists memoryUnitSizeInMb.(=_{256} \vee =_{512})$.

To gain a further insight into the above five relations to be checked to establish a match, it is often helpful to have some working examples. Suppose, in a scenario of computer selling, that an agent would like to buy a PC with the following capabilities:

[25] $\sqsubseteq^*$ is the transitive reflexive closure of $\sqsubseteq$.

- the $processor$ must be Pentium4;
- the $memoryUnitSizeInMb$ must be 128;
- the $priceInPound$ must be less than 500.

This can be represented by the following OWL-E class description:

$$C_{R1} \equiv \mathsf{PC} \sqcap \exists processor.\mathsf{Pentium4} \sqcap \exists memoryUnitSizeInMb.\ =^{int}_{[128]} \sqcap \exists priceInPound.\ <^{int}_{[500]}$$

Exact match: $C_{A1} \equiv \mathsf{PC} \sqcap \exists processor.\mathsf{Pentium4} \sqcap \geqslant 1 memoryUnitSizeInMb.\ =^{int}_{[128]} \sqcap \exists priceInPound.\ <^{int}_{[500]}$

PulgIn match: $C_{A2} \equiv \mathsf{PC} \sqcap \exists processor.\mathsf{Pentium4} \sqcap \geqslant 1 memoryUnitSizeInMb.(=^{int}_{[128]} \vee =^{int}_{[256]}) \sqcap \exists priceInPound.\ <^{int}_{[700]}$

Subsume match: $C_{A3} \equiv \mathsf{PC} \sqcap \exists processor.\mathsf{Pentium4} \sqcap \geqslant 1 memoryUnitSizeInMb.\ =^{int}_{[128]} \sqcap \exists priceInPound.\ <^{int}_{[500]} \sqcap \forall orderDate.((\geq^{int}_{[20040801]} \wedge \leq^{int}_{[20040831]}) \vee (\geq^{int}_{[20040901]} \wedge \leq^{int}_{[20040930]})) \sqcap \forall orderDate, deliverDate.\ <^{int}$

Intrsect. match: $C_{A4} \equiv \mathsf{PC} \sqcap \exists processor.\mathsf{Pentium4} \sqcap \geqslant 1 memoryUnitSizeInMb.\ =^{int}_{[128]} \sqcap \exists priceInPound.\ >^{int}_{[400]} \sqcap \leqslant 1 priceInPound.\top_{\mathrm{D}} \sqcap\ \exists CPUFreqInGHz.\ =^{real}_{[2.8]}$

Disjoint match: $C_{A5} \equiv \mathsf{PC} \sqcap \exists processor.\mathsf{Pentium4} \sqcap \geqslant 2 memoryUnitSizeInMb.\ =^{int}_{[256]} \sqcap \exists priceInPound.\ <^{int}_{[500]} \leqslant 2 memoryUnitSizeInMb.\ =^{int}_{[256]} \sqcap \forall HardDiskBrand, USBKeyBrand.\ =^{str}$

Figure 2.5: Example matching advertisements

Figure 2.5 presents five example relations for C_{R1} and the advertised services. Among them, C_{A1} is the exact match. In realistic situations, however, it is not to easy have an exact match, since advertisements might provide more general or more specific information. For example, C_{A2} states that $priceInPound$ is only less than 700 and that the $memoryUnitSizeInMb$ can be either 128 or 256 (represented by the datatype expression $=^{int}_{[128]} \vee =^{int}_{[256]}$). C_{A3} adds two restrictions on $orderDate$s: firstly, the order date must be in August and September of 2004, which is represented by the datatype expression $(\geq^{int}_{[20040801]} \wedge \leq^{int}_{[20040831]}) \vee (\geq^{int}_{[20040901]} \wedge \leq^{int}_{[20040930]})$; secondly, $orderDate$s should be sooner than (represented by the binary predicate $<^{int}$) $DeliverDate$s, indicating that PCs will be delivered on some date *after* orders are made. As a result, C_{A2} and C_{A3} are

PlugIn match and Subsume match of C_{R1}, respectively. C_{A4} says the $priceInPound$ is greater than 400, and the $CPUFreqInGHz$ of their PCs is 2.8;[26] it is an Intersection match. Finally, C_{A5} advertises that their PCs have exactly two memory chips, with the $memoryUnitSizeInMb$ of each chip is 256, and the $HardDiskBrand$ and $USBKeyBrand$ in their PCs are the same (represented by the binary predicate $=^{str}$); hence it is a disjoint (failed) match.

Discussion: Reasoning with OWL-E

In this section, we briefly discuss how to provide practical decision procedures for OWL-E, the $\mathcal{SHOIQ}(\mathcal{G})$ DL. [Pan04a] shows that the $\mathcal{SHOIQ}(\mathcal{G})$ DL is decidable, and that if we have a tableaux algorithm for $\mathcal{SHOIQ}$, we can easily upgrade it to one for $\mathcal{SHOIQ}(\mathcal{G})$. The problem is that, to the best of our knowledge, there exist no published tableaux algorithms to handle the $\mathcal{SHOIQ}$ DL.

'Divide and Conquer' Approach The motivation of this approach is that in ontology applications we might not often need the full $\mathcal{SHOIQ}(\mathcal{G})$ DL. Therefore, instead of providing a decision procedure for the full $\mathcal{SHOIQ}(\mathcal{G})$ DL, we can provide decision procedures for a set of sub-languages of the $\mathcal{SHOIQ}(\mathcal{G})$ DL so that they can cover all features of the $\mathcal{SHOIQ}(\mathcal{G})$ DL. Now we have to decide which set of sub-languages we should consider. Since the $\mathcal{SHOQ}(\mathbf{D})$ DL has been argued to be useful in the context of the Semantic Web [HS01], and the $\mathcal{SHIQ}$ DL is the underpinning of the OIL Web ontology language and is implemented in popular DL systems like FaCT [Hor98] and RACER [HM01], a possible choice would be the following set of sub-languages $\{\mathcal{SHOQ}(\mathcal{G}), \mathcal{SHIQ}(\mathcal{G}), \mathcal{SHIO}(\mathcal{G})\}$ of $\mathcal{SHOIQ}(\mathcal{G})$. Tableaux algorithms for the $\mathcal{SHOQ}(\mathcal{G})$, $\mathcal{SHIQ}(\mathcal{G})$ and $\mathcal{SHIO}(\mathcal{G})$ DLs are presended in [Pan04a]. Among them, the one for $\mathcal{SHIQ}(\mathcal{G})$ has been implemented in an extended FaCT reasoner.

FOL Approach This approach is based on the observation that the $\mathcal{SHOIQ}$ DL can be translated into the $\mathcal{C}^2$ fragment of first order logic (FOL). Therefore, decision procedures of $\mathcal{C}^2$ can be used as the ones for $\mathcal{SHOIQ}$. To support the $\mathcal{SHOIQ}(\mathcal{G})$ DL, we need to extend the decision procedure(s) for $\mathcal{C}^2$, and support also datatypes. To the best of our knowledge, there exist no such published decision procedures for the datatype extension of $\mathcal{C}^2$; there exists no published FOL theorem provers that implement decision procedures of $\mathcal{C}^2$, neither.

Notice that in this section we have not introduced the annotation of registries as proposed for METEOR-S (see Section 1.1.3). However, this annotation can be useful at the

[26] Note that $=^{real}_{[2.8]}$ is not a supported predicate for our prototype: our prototype will not reject it and even provides minimum checking for it; i.e., if $=^{real}_{[2.8]}$ and its relativised negation $\overline{=^{real}_{[2.8]}}$ are both in a predicate conjunction, this conjunction is *unsatisfiable*.

discovery level, when abstract service capabilities can be further abstracted to a common domain of service and therefore grouped into appropriate registries. Although this can be an interesting technique to improve the efficiency, providing an additional filter to limit the service discovery process proposed to a smaller subset of services, it is not in the core of the discovery problem so we will not elaborate more on this issue.

2.5 Automatic Web Service Contracting

In the previous section we have studied how to model and how to match abstract service capabilities and requester goals. At that level, we were interested on an abstract characterization of the services that can be provided, without considering the precise description of concrete services and the relation between the information available from the requester and the service provided.

However, at the contracting phase a concrete service to be agreed and eventually delivered needs to be located and, therefore, precise details of such service have to be (directly or inderectly) captured, including what information has to be provided by the requester in order to get the desired service.

In the following, we will discuss the extension of our previous model to capture in the contracting capability the relation between the input to the service and the results provided, as well as how the requester goal and such capability are matched.

Using simple semantic annotations for a service, as it was described in the previous section, adds machine-processable semantic information to service descriptions which allows a discovery mechanism to exploit this semantics during the discovery process and deliver results with high precision and recall.

Nonetheless, the kind of semantic information that can be expressed in that approach is limited wrt. the details of the concrete services that can be agreed and contracted, as it represents, in the general case, solely an abstraction of such concrete services.

In the following we show how to extend the set-based modelling approach discussed in Section 2.4.2 in the direction of service contracting. In addition, we will discuss an approach based on Transaction Logic [BK98], an extension of First-order Logic that enables to specify the dynamics of logical theories (or knowledge bases) in a declarative way. Here, the state of the world is represented by a logical theory and since the provision of a service changes the state of the world and the information space, it results in an update of the logical theory. The described approach can be implemented using the $\mathcal{F}$LORA-2 system [KLP+04].

A set-based Modelling Approach for Rich Service Descriptions

The informal Service Model revisited. The provision of a service generates (wrt. a set of input values) certain information as an output and achieves certain effects on the state of the world. Both an output and an effect can be considered as objects which can be embedded in some domain ontology.

So far we ignored inputs and their relation to outputs and effects of the Web service provision. However, when considering concrete services, these will *depend on the provided input values*. Hence, a contracting capability can be described by the sets of outputs and effects for *specific* input values. In addition, the information the requester is able to provide, denoted by $\mathcal{G}_{input}$ will play a role in determining whether a concrete service can be agreed and contracted. Finally, the contracting capability of a given service will in the general case capture some details that were not considered previously and that will involve actual communication with the provider e.g. checking if a requested flight information can be actually be provided i.e. whether a concrete service fulfilling the concrete requester goal can be agreed by the provider and, therefore, provided.

Additionally, we can enrich our set of matching notions given in Section 2.4.2 with an orthogonal dimension: we can express that we can satisfy a particular matching notion wrt. a *single concrete service* as well as wrt. an *arbitrary number of concrete services*. This results in additional matching notions that capture additional semantics in a given requester goal.

Let us illustrate the difference with a simple example. Imagine the following (informal) requester goal:

> $\mathcal{G}$: "I want to know about all flights from Innsbruck and Salzburg to Madrid on December 28th, 2004"

and a Web service with the following (informal) capability

> $\mathcal{W}$: "The service provides information about all flights from any place in Austria to Madrid, on any date. However, it provides information about a single itinerary e.g. Innsbruck-Madrid at a time"

Therefore, a single concrete service cannot fulfill the requester goal, but two concrete services (one for the itinerary Innsbruck-Madrid and one for the itinerary Salzburg-Madrid) can be agreed and they fulfill together the requester goal. These two concrete services correspond to different input information provided by the requester. This can be seen as a simple form of composition, but it can still be captured in our contracting framework and in the definition of the formal proof obligations that have to be checked to determine whether concrete services fulfilling the goal can be contracted.

In the following paragraphs we will show how to formalize the extended service capability and the corresponding matching notions.

Formalizing the extended Service Model. Instead of using an unary predicate $W_{post}(x)$ for describing the contracting capability of a Web service $\mathcal{W}$, we have to express the dependency of the concrete services that can be delivered on the concrete input required for the provision of such services $i_1, \ldots, i_n$.

Let $\mathcal{W}$ be a Web service with input parameters $i_1, \ldots, i_n$, then we formalize the contracting capability of the service as follows:

$$\mathcal{W} : \forall x, i_1 \ldots i_n.(ws(x, i_1 \ldots i_n) \leftrightarrow W_{pre}(i_1 \ldots i_n) \wedge W_{post}(i_1 \ldots i_n, x)) \tag{2.17}$$

where $W_{pre}(i_1 \ldots i_n)$ is an arbitrary first-order formula describing the preconditions of the Web service and $W_{post}(i_1 \ldots i_n)$ is an arbitrary first-order formula describing its post-conditions. In $W_{post}(i_1 \ldots i_n, x)$ the variable x refers to the output value(s) resulting from the service provision. The defined predicate $ws(x, i_1 \ldots i_n)$ has a natural interpretation: The value x will be delivered by the Web service $\mathcal{W}$ for the input values $i_1 \ldots i_n$.

Goals are just described in the same way as in Section 2.4.2.

Adapting the Matching Notions. Since we adapted the way we describe service capabilities for contracting, we have to consequently adapt the formal criteria used to determine a match. In the following we will show how to adapt the single relations accordingly and give a definition for the case in which we only consider *single concrete services* as well as the case of considering *sets of concrete services*.

The inputs considered in the formalizations below have to be provided by the requester i.e. $i_1 \ldots i_n \in \mathcal{G}_{input}$

- **Exact-Match ($\mathcal{W} \equiv^1_{\mathcal{O}} \mathcal{G}$, $\mathcal{W} \equiv^+_{\mathcal{O}} \mathcal{G}$)**.

  Formally, the relation we have to prove if we restrict ourselves to a single concrete service is the following:

  $$\mathcal{W}, \mathcal{G}, \mathcal{O} \models \exists i_1, \ldots, i_n.(\forall x.(g(x) \leftrightarrow ws(x, i_1 \ldots i_n)) \tag{2.18}$$

  where $\mathcal{W}$ is the definition of the service contracting capability, $\mathcal{G}$ is the definition of the requester goal, and $\mathcal{O}$ is a set of ontologies to which both descriptions refer.

  In this case we write $\mathcal{W} \equiv^1_{\mathcal{O}} \mathcal{G}$ to indicate this particular kind of relation.

  For multiple concrete services we instead would have to prove the following:

  $$\mathcal{W}, \mathcal{G}, \mathcal{O} \models \forall x.(\exists i_1, \ldots, i_n.(g(x) \leftrightarrow ws(x, i_1 \ldots i_n)) \tag{2.19}$$

  In this case we write $\mathcal{W} \equiv^+_{\mathcal{O}} \mathcal{G}$ to indicate this particular kind of relation.

- **Subsumption-Match ($\mathcal{W} \sqsubseteq^1_{\mathcal{O}} \mathcal{G}$, $\mathcal{W} \sqsubseteq^+_{\mathcal{O}} \mathcal{G}$)**.

  Formally, the relation we have to prove if we restrict ourselves to a single concrete service is the following:

$$\mathcal{W}, \mathcal{G}, \mathcal{O} \models \exists i_1, \ldots, i_n.(\forall x.(g(x) \leftarrow ws(x, i_1 \ldots i_n)) \tag{2.20}$$

  We denote it by $\mathcal{W} \sqsubseteq^1_{\mathcal{O}} \mathcal{G}$.

  And for multiple concrete services:

$$\mathcal{W}, \mathcal{G}, \mathcal{O} \models \forall x.(\exists i_1, \ldots, i_n.(g(x) \leftarrow ws(x, i_1 \ldots i_n)) \tag{2.21}$$

  Denoted by $\mathcal{W} \sqsubseteq^+_{\mathcal{O}} \mathcal{G}$.

- **Plugin-Match ($\mathcal{W} \sqsupseteq^1_{\mathcal{O}} \mathcal{G}$, $\mathcal{W} \sqsupseteq^+_{\mathcal{O}} \mathcal{G}$)**.

  Formally, the relation we have to prove if we restrict ourselves to a single concrete service is the following:

$$\mathcal{W}, \mathcal{G}, \mathcal{O} \models \exists i_1, \ldots, i_n.(\forall x.(g(x) \rightarrow ws(x, i_1 \ldots i_n)) \tag{2.22}$$

  Denoted by $\mathcal{W} \sqsupseteq^1_{\mathcal{O}} \mathcal{G}$.

  For multiple concrete services we have to prove the following:

$$\mathcal{W}, \mathcal{G}, \mathcal{O} \models \forall x.(\exists i_1, \ldots, i_n.(g(x) \rightarrow ws(x, i_1 \ldots i_n)) \tag{2.23}$$

  Written $\mathcal{W} \sqsupseteq^+_{\mathcal{O}} \mathcal{G}$.

  Notice that inconsistent definitions, both in Plugin and Subsumption-Match can be dealt with in the very same way as we discussed in Section 2.4.2.

- **Intersection-Match($\mathcal{W} \sqcap^1_{\mathcal{O}} \mathcal{G}$, $\mathcal{W} \sqcap^+_{\mathcal{O}} \mathcal{G}$).**

  The relation we have to prove for a single concrete service is:

$$\mathcal{W}, \mathcal{G}, \mathcal{O} \models \exists i_1, \ldots, i_n.(\exists x.(g(x) \wedge ws(x, i_1 \ldots i_n)) \tag{2.24}$$

  Written $\mathcal{W} \sqcap^1_{\mathcal{O}} \mathcal{G}$.

  For multiple concrete services we have to prove the following:

$$\mathcal{W}, \mathcal{G}, \mathcal{O} \models \exists x.(\exists i_1, \ldots, i_n.(g(x) \wedge ws(x, i_1 \ldots i_n)) \tag{2.25}$$

  Denoted by $\mathcal{W} \sqcap^+_{\mathcal{O}} \mathcal{G}$.

Notion of match. As discussed for service discovery, the formal relations above do not represent themselves the notion of match we expect but the relation that has to be checked to establish a given match relation. In Figure 2.4 the relations that have to be checked for the different notions of match were presented. Notice that the relations that have to be checked for contracting are essentially the same but extended with the consideration of the input. As an example, for checking a perfect match if both requester and provider have a universal intention, we will have to check the Plug-in or Exact-Match relations above.

Requester input. Notice that the input involved in the proof obligations above has to be made available by the requester. This does not impose that the requester has to list for every goal all the information he has available, but he can for example offer a service that provides his available information on demand. In addition, some input information can automatically be extracted from the goal description e.g. if the requester wants to fly from Innsbruck to Madrid we already know that he can provide Innsbruck as the departure location and Madrid as the arrival location. However, how this information is made available to the contracting process is beyond the scope of the deliverable, and we assume that it will be available in some way during the contracting phase.

An example. As an example, let us consider the goal $\mathcal{G}$ and the informal contracting capability of service $\mathcal{W}$ given above. In addition, let us suppose that the Web service requires the nationality and such nationality has to be from any European country, and the requester can provide the start location, end location, date and nationality:

$$\mathcal{G} : \forall x.((\exists s\ flight(x, s, madrid, Dec282004) \wedge (s = innsbruck \vee s = salzburg)) \leftrightarrow G_{post}(x)) \tag{2.26}$$

$$startlocation(innsbruck), startlocation(salzburg), endlocation(madrid), \\ date(Dec282004), nationality(austrian) \in \mathcal{G}_{input} \tag{2.27}$$

$$\begin{aligned} \mathcal{W} : \forall x, i_1, i_2, i_3, i_4 (ws(x, i_1, i_2, i_3, i_4) \leftrightarrow \\ (startlocation(i_1) \wedge in(i_1, austria) \wedge endlocation(i_2) \wedge i_2 = madrid \\ \wedge date(i_3) \wedge nationality(i_4) \wedge european(i_4)) \\ \wedge flight(x, i_1, i_2, i_3) \wedge availableinfo(x)) \end{aligned} \tag{2.28}$$

where $availableinfo(x)$ is a predicate that involves checking whether information about flight x is available in the provider knowledge base.

As both requester and provider have universal intentions, we can have a match involving multiple concrete services if the formal relation $\mathcal{W} \sqsupseteq^{+}_{\mathcal{O}} \mathcal{G}$ is fulfilled, which is the case if the predicate $availableinfo(x)$ evaluates to true. However, if we restrict ourselves to single concrete services, we will only be able to prove the relation $\mathcal{W} \sqcap^{1}_{\mathcal{O}} \mathcal{G}$ and, therefore, only a partial match will be possible. In conclusion, if $availableinfo(x)$ is true, we will be able to agree on and contract two concrete services (one for the itinerary Innsbruck-Madrid and one for the itinerary Salzburg-Madrid), and otherwise we will only agree on a single concrete service partially fulfilling the goal (for only one of the itineraries).

Notice that the evaluation of the predicate $availableinfo(x)$ will require the access to the provider knowledge base. However, and as stated before, we consider this process to be transparent to us i.e. we just require that we get a truth value from the predicate but we do not care how the communication with the provider is actually performed. This will be subject of future work.

Discovery with Transaction Logic

The approach related above does not restrict the expressiveness allowed and requires in principle the use of a theorem prover, which efficiency is not clear for arbitrary capabilities and goals.

In order to overcome these problems, we propose the use of transaction logic to model the proof obligations for service contracting. Transaction logic is an extension of predicate calculus that provides a logical foundation for state changes in logic programs and databases. It comes with a model theory and sound and complete proof theory. Remarkably, the proof does not only verify programs but also executes them. Interesting features for us are hypothetical updates, constraints on transaction execution, and bulk updates [BK98].

Proof obligations If the contracting capability of a service models the relation between its input and its results i.e. if the postconditions and effects delivered by the service are dependent on the input information made available by the requester to it, we can check whether a given input information will lead to postconditions and effects that satisfy the postconditions and effects requested in the goal.

For this purpose, we first need to check that there exists input information $Input$ available to the service i.e. $Input \in \mathcal{G}_{input}$ that satisfies the preconditions (W_{pre}) of a given service $\mathcal{W}$.

If the above holds, we can hypothetically assume that the postconditions and effects of the Web Service for this input ($W_{post}(Input)$ and $W_{eff}(Input)$ hold. Under this assumption, we have the effects and postconditions of the candidate service available; if the goal postconditions and effects hold in this state, it means that the Web Service provides the desired results for the available input and, therefore, the concrete service corresponding

to this input can be agreed and contracted.

This can be formalized by the following proof obligation using transaction logic, where $\diamond$ is the hypothetical operator, $\otimes$ is the sequence operator, and $\mathcal{O}$ is the set of domain ontologies the description of the requester goal ($\mathcal{G}$) and the contracting capability of service $\mathcal{W}$ refer to:

$$\begin{aligned} &\mathcal{O}, \mathcal{W}, \mathcal{G} \models \exists Input \\ \diamond(W_{pre}(Input) \otimes insert\{W_{post}(Input), W_{eff}(Input)\} \\ &\otimes G_{post} \wedge G_{eff}) \end{aligned} \tag{2.29}$$

Since we use the hypothetical operator ($\diamond$), the assertion of the service postconditions and effects will be rolled back i.e. retracted after the checking is finished.

The use of the sequence operator ($\otimes$) means that the preconditions must be tested before the postconditions and effects of the service can be asserted, and that the goal will only be tested after this. In this way, we distinguish between the pre and post-state of the concrete service corresponding to input $Input$. This is especially relevant for checking the preconditions, as the assertion of the Web Service results will change the pre-state[27] (both of the information and real-world spaces) and, by definition, the preconditions must be checked before such changes happen. Similarly, the goal must be tested only in the post-state i.e. when the postconditions and effects of the Web Service have been asserted.

The proof obligation is illustrated in Figure 2.6. On the left hand side of the picture, we represent the input information available. From this, $Input$ satisfying the Web contracting capability preconditions $W_{pre}(Input)$ is considered. Depending on this input, we hypothetically assume (represented by an arrow in the figure) the postconditions and effects of the concrete service corresponding to $Input$. These, represented by the box on the right hand side of the figure, are the results provided by the concrete service. Finally, we check whether the goal is satisfied by these results i.e. the results requested in the goal are generated.

For the description of the goal, we allow the use of both universally quantified goals i.e. all the results satisfying the conditions stated in the goal have to be delivered by the Web Service, as well as existentially quantified goals i.e. some results satisfying such conditions have to be delivered. Whether the request is existential or universal will be part of the description of the goal itself. Similarly, the contracting capability will also specify whether it can provide one or all the elements described[28]. If existential quantification is used, Web Services providing at least one result satisfying the conditions stated in the goal will be matched, while only Web Services providing all such results will be matched if universal quantification is used.

[27] This is true for every service which really provides something. Otherwise, the service does not have any interest for the requester and it should not even be considered.

[28] This can be done by introducing a meta-annotation of the results, declaring local complete knowledge [HMA02].

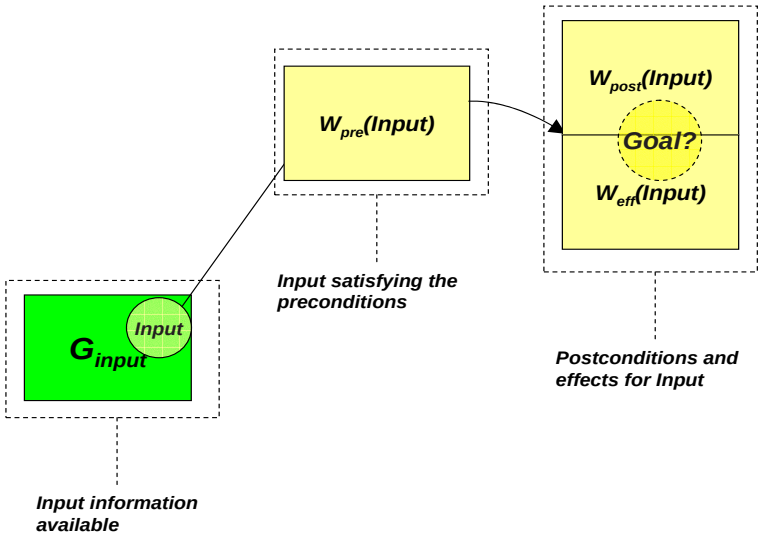


Figure 2.6: Hypothetically assuming Web Service postconditions and effects

Single concrete service vs set of concrete services. The proof obligation (2.29) only considers a single concrete service fulfilling the goal but, as discussed above, it might be interesting to also consider a set of concrete services fulfilling the request. For considering this case, we have to modify our previous proof obligation, which leads to proof obligation (2.30):

$$\begin{aligned} &\mathcal{O}, \mathcal{W}, \mathcal{G} \models \\ &\Diamond(insert\{W_{post}(Input), W_{eff}(Input) | W_{pre}(Input)\} \\ &\otimes G_{post} \wedge G_{eff}) \end{aligned} \tag{2.30}$$

$insert\{W_{post}(Input), W_{eff}(Input) | W_{pre}(Input)\}$ corresponds in the previous formula to a *bulk update* i.e. the assertion of the postconditions and effects will occur for every Input satisfying the conditions in $W_{pre}(Input)$. Therefore, we hypothetically assume the results of the service for every possible input that satisfies the Web Service preconditions, and we test the goal with all such results hypothetically asserted i.e. we test the goal over the union of these results. This allows to match multiple concrete services corresponding to all the inputs that lead to required results.

Let us retake the example used for illustrating the formalization based on first-order logic. In this case, and using proof obligation 2.30 to allow the agreement and contracting of multiple concrete services:

1. Inputs matching the service preconditions will be selected. In this case, $startlocation(innsbruck)$, $startlocation(salzburg)$, $endlocation(madrid)$, $date(Dec282004)$ and $nationality(austrian)$ fulfill the preconditions.

2. The postconditions of the service for this input will be hypothetically asserted i.e. $k(flight(f1, innsbruck, madrid, Dec282004))$ and $k(flight(f2, innsbruck, salzburg, Dec282004$, being $f1$ and $f2$ generated flight information and the k logical connective meaning that all the information about the flights with those characteristics will be provided[29].

3. The requester goal will be checked and, in this case, fulfilled.

The use of transaction logic introduces the main advantage of explicitly considering the state in the proof obligations for Web Service discovery. As discussed before, the preconditions of the Web Service must be checked in the pre-state i.e. before the post-conditions and effects of the Web service are generated. Not considering the state in the discovery process might lead to problems in some cases e.g. if the Web service requires the balance of the requester's account to be higher than a given amount and one of its effects is to decrease the balance of such account. If preconditions and effects are checked in the same state, some expected matches might fail.

The notions of match previously identified are covered by the two proof obligations introduced in this section. Although partial matches i.e. $\mathcal{W} \sqsubseteq^1_{\mathcal{O}} \mathcal{G}$ and $\mathcal{W} \sqcap^1_{\mathcal{O}} \mathcal{G}$ are not directly covered, they can be relaxed to $\mathcal{W} \sqsubseteq^+_{\mathcal{O}} \mathcal{G}$ and $\mathcal{W} \sqcap^+_{\mathcal{O}} \mathcal{G}$, respectively. In fact, partially matching in the first two cases is equivalent to relaxing the intention of the requester from universal intention to existential intention. Therefore, we can see that this approach covers all the cases discussed before.

As shown in [KLP+04], an implementation of a very similar proof obligation has been done using $\mathcal{F}$LORA-2.

2.6 Relation between Discovery and Mediation

In a distributed environment, different users and Web services can use different terminologies, which leads to the need for mediation in order to make heterogeneous parties communicate. In this section we analyze the relation between discovery and mediation, identifying what kind of mediation is required in different scenarios. Providing the mediation support required is out of the scope of this deliverable. However, we discuss the assumptions necessary in different technological scenarios and to what extent these are realistic. Notice that a strongly similar mediation problem arises in composition, and the assumptions and requirements presented here are applicable to composition or any other process involving Web Services.

[29] The k logical connective used is inspired in the work on Local Closed World reasoning (cf. [EGW94, HMA02]) and in the K operator introduced in [DLNS98], meaning complete knowledge about some information.

2.6.1 Assumptions on Mediation

One could assume that Web services and goals are described using the same terminology. In that case no mediation problem exists during the discovery process. However, it is unlikely that a potentially huge number of distributed and autonomous parties will agree before-hand on a common terminology.

Alternatively, one could assume that goals and Web services are described using completely independent vocabularies. Although this case might happen in a real setting, discovery would be impossible to achieve. In consequence, we consider an intermediate scenario where we do not ignore the heterogeneity inherent to the domain, and where mediation and therefore discovery are possible. Such scenario relies on three main assumptions:

- Goals and Web services use most likely different vocabularies, or in other words, we do not restrict our approach to the case where both need to use the same vocabulary.
- Goals and Web services use some controlled vocabulary or ontology to describe requested and provided services.
- There is some mediation service in place. Given the previous assumption, we can optimistically assume that a mapping has already been established between the used terminologies, not exclusively to facilitate our specific discovery problem but rather to support general information exchange between parties using these terminologies.

Under these assumptions, we assume a minimal mediation support that is a prerequisite for successful discovery.

Notice that IBROW (cf. [BPM+98]), a project in the area of internet-based matchmaking of task descriptions and competence definitions of problem-solving methods, has adopted a similar approach. Both tasks and methods used different ontologies to describe their requests and services. However, both description ontologies are grounded in a common basic ontology that allow to rewrite requests and services in terms of a common ontology.

2.6.2 Mediation requirements

In the following, we will discuss different technological scenarios for the discovery process and how this reshapes our assumption on the underlying mediation support that needs to be in place.

Natural Language Processing and Keywords

Processing written natural language input coming from a human user is commonly used to derive information for computer consumption. Stemming, part-of-speech tagging, phrase

recognition, synonym detection etc. can be used to derive machine-processable semantics from goals and service descriptions. In this way, a set of keywords extracted from a requester goal are matched against a set of keywords extracted from a service description or pre-defined goal. Alternatively, both requester and provider can directly use keywords to describe their requests and offers.

Stemming and synonym recognition already try to reduce different words to their common conceptual meaning. Still, this mediation support is very generic and only relies on general rules of language processing. Mediation can also be required for translating from a given human language into another one. Also in this case the mediation support required is generic and it can be realistically assumed.

Controlled vocabularies & Ontologies

A different scenario is to assume that requester and provider use (not necessarily the same) controlled vocabularies[30]. In order to illustrate such scenario we will refer to controlled vocabularies for products and services. Efforts like UNSPSC[31] or eCl@ss[32] (among others) provide controlled vocabularies for describing products and services with around 15,000 concepts each. A service is described by a reference to (one or more) classes in one of the classification schemas and, similarly, a goal is described by a concept taken from (not necessarily the same) controlled vocabulary.

A mediation service is needed in case the requester and provider use different vocabularies. Since they use controlled instead of ad-hoc vocabularies, requiring such mediation service is not an unrealistic assumption. Notice that we do not assume a customized mediation service for our specific discovery task but rather alignments of generic business terminologies that may be used for other information exchanges using such terminologies[33].

The border between controlled vocabularies and ontologies is thin and open for a smooth and incremental evolvement. Ontologies are consensual and formal conceptualizations of a domain (cf. [Gru93b]). Controlled vocabularies organized in taxonomies and with defined attributes like eCl@ss resemble all necessary elements of an ontology. Ontologies may simply add some logical axioms for further restricting the semantics of their elements. Notice that a service requester or provider gets these logical definitions ”for free”. He can select concepts for annotating his service or goal, but he does not need to write logical expression as long as he only transparently reuses the ones already included in the ontology.

[30] Notice that this does not necessarily imply that service provider and requester use directly these controlled vocabularies. The goal discovery process can provide this service. Trials to mechanize such a process are described in [DKO+02].

[31] http://www.unspsc.org/

[32] http://www.eclass.de/

[33] Establishing this alignments is not an easy and straight-forward task as discussed in [Fen03].

This scenario looks quite appealing since it adds semantic Web facilities to the Web service discovery eliminating two major risk factors of logic-based techniques:

- Effort in writing logical expressions. Writing down correct logical formulas is a cumbersome process. A discovery approach that is assuming this on a large scale for requesters and providers leads to scalability problems.

- Effort in reasoning about logical expressions. Reasoning over complex logical statements is computationally hard in case the formal language provides a certain level of expressivity. Therefore, it is important to decouple the reasoning process from the actual discovery process. This is possible by using ontologies, as the goal as well as the service descriptions are abstracted from concrete specifications and inputs to a generic description at the level of an ontology. The logical relationship between these concepts can be derived independently from a concrete goal and the materialized inferences can simply be reused during the discovery process.

As for controlled vocabularies, mediation support is needed in case the requester and provider use different ontologies. However, since generic ontologies can be used, such mediation service can be assumed. Again, we do not assume a customized meditation service for our specific discovery task but rather generic alignments of generic ontologies[34].

Full-fledged logic

Simply reusing existing concept definitions as described in the previous section has the advantage of the simplicity in annotating services and in reasoning about them. However, this approach has limited flexibility, expressivity, and grain-size in describing services and request. Therefore, it is only suitable for scenarios where a more precise description of goals and services is not required. Furthermore, describing the functionality of a Web service requires an expressivity which goes beyond the capabilities of current ontology languages. For these reasons, a full-fledged logic is required when a higher precision is required for the discovery process. This is the case for service contracting, where concrete services have to be agreed and contracted.

Mediation can only be provided if the terminology used in the logical expressions is grounded on ontologies. Otherwise, it is impossible to prove the given logical relation between goals and service descriptions. Therefore, the mediation support required is the same as for ontologies.

[34] See [dMRME04] for a survey on Ontology mapping and alignment.

2.7 Achievements

In this chapter, we have introduced the problem of automatically locating services that fulfill a given requester desire. In order to provide a comprehensive solution to the problem, we have introduced in Section 2.1 a conceptual model for discovery and contracting of services that takes into account pragmatical issues such as the degree of accuracy in the formal descriptions of services that can be expected.

In Section 2.2 we have analyzed the work done so far in the area of automatic service discovery and a strongly related area: software component retrieval.

After introducing our description of goals and Web services in Section 2.3, the formal semantics for requested goals and abstract capabilities of services have been introduced in Section 2.4, as well as the formal relation between those that have to be checked in order to establish different kinds of match. In addition, we have discussed the influence of the requester and service intentions when describing their goals and abstract capabilities, which has been ignored so far in the literature, and how they affect the relations that have to be checked for detecting different types of match. Finally, we have provided a formalization based on first-order logic and an adaptation that restricts the expressiveness allowed for describing goals and capabilities and that bring desirable computational characteristics.

In Section 2.5 we have addressed the last step in our conceptual model i.e. service contracting, again providing a formalization of the notion of match and introducing the kind of modelling necessary for contracting capabilities and requester goals. Such formalization has been provided using first-order logic and transaction logic, the latter being implementable using $\mathcal{F}$LORA-2.

Finally, we have discussed the relation between service discovery and mediation, analyzing the assumptions we have to make on the mediation support.

In summary, we have presented a comprehensive analysis of the problem of automatic service discovery and a conceptual model for it, providing formal semantics for the most relevant steps in the model and analyzing the use of different formalisms to achieve it. Furthermore, our model is independent on the underlying semantic Web service description model, although it does put some requirements on the conceptual elements that have to be in place and what expressiveness is required for the modelling language used.

2.8 Open points

During our work, we have found several points that remain open and have to be solved for an implementation of our discovery model. In the following, we list the most relevant ones:

- The process of refining pre-defined goals based on a user desire has to be either

automated or at least facilitated.

- An efficient discovery model needs to perform complex and expensive reasoning only for a small subset of all available services. DL modelling and reasoning is regarded as a good candidate for service discovery, as it is expected to efficiently filter out a big number of irrelevant services. However, current DL reasoners present problems when dealing with nominals for certain DL languages e.g. $\mathcal{SHOIN}$. Therefore, a way to overcome this problem has to be found and, eventually, performance tests have to be carried out to select the most suitable DL reasoner.

- As DL modelling does not allow the differentiation of different intentions, annotation of DL concepts has to be introduced at a meta-level. How to precisely implement this is still open.

- Abstract and contracting service capabilities are required. However, the provision of service descriptions covering these two aspects must be consistent. Whether the abstract capability can be automatically abstracted from the concrete capability has not been proved. If this is not the case, a way to ensure the consistency of both descriptions is required. Furthermore, these capabilities pose different requirements on the language used to describe them. How different languages interact is an open issue. The WSML family of languages is expected to be a good candidate for solving it. We foresee that the WSML-DL variant will be required for describing abstract capabilities, while WSML-Rule is the most likely candidate for contracting capabilities. As WSML-Full is expected to provide a common unifying language for WSML-DL and WSML-Rule, this would almost solve the problem. WSML-Core can also serve as the language for describing domain ontologies, as it corresponds to the maximal common subset of WSML-DL and WSML-Rule. The expressiveness provided by WSML-Core is expected to suffice for most domains [Vol04], but this has to be tested. A similar problem has to be resolved for the description of requester goals.

- Contracting capabilities rely on the existence of predicates that (transparently) communicate with the service provider e.g. querying the provider knowledge base. However, communication issues are involved that need to be resolved e.g. the dynamic use of service choreographies to communicate with the provider. Similarly, the information a requester can provide has been assumed to be provided transparently, without addressing communication issues.

- Mediation support is assumed to make discovery and contracting work in a heterogeneous environment. Actual mediation support has to be provided for a real implementation of discovery and contracting. WSMO provides an appropriate conceptual framework for it, and wgMediators, ggMediators and ooMediators [RL*editors*04] can be employed for this purpose.

- A discovery engine that implements the proposed approaches has to be implemented and its efficiency tested.

2.9 Future work

Our future work will concentrate on solving the open issues mentioned in the previous section, particularly the interaction of abstract and contracting capabilities of services and the required languages and the evaluation of candidate reasoners. This will eventually lead to the implementation of a discovery engine based on this conceptual model. This work has already started (see [L*editors*04b]).

A second major line in our future work is to integrate the planned discovery engine with the approaches for composition that will be presented in the next chapters. A first conceptual analysis of the nature of such integration will be provided in Chapter 4.

Chapter 3

Semantics for Web Service Composition

The automatic composition of Web services, namely the problem of providing an "automatic selection, composition, and interoperation of [existing] Web services to perform some complex task, given a high-level description of an objective" [Coa04], is one of the motivating goals of the research in Semantic Web Services. In this chapter we provide a conceptual model for describing Web service composition problems and we discuss techniques and tools supporting the automatic service composition.

We distinguish two different forms of Web service composition, namely a "Functional-Level Composition" and a "Process-Level Composition". **Functional-level** composition addresses the problem of selecting a set of services that, combined in a suitable way, are able to match a given query. Each existing service is defined in terms of an atomic interaction, i.e., in terms of its input and output parameters as well as of its preconditions and effects. (Such description is provided for instance in the OWL-S service profile, or in the WSMO service capability model, see Section 1.1). The query defines the overall functionality that the composed service should implement, again in terms of its inputs, outputs, preconditions, and effects. An example of functional-level composition in the domain of the Virtual Travel Agency (see Section 1.2) is that of identifying the Web services that need to be integrated in order to answer to a customer request. The result of the composition can be the identification of "Flight Booking" service, a "Hotel Booking" service, and a "Car Rental" service which are adequate for the specific request of the customer (i.e., the specific destination of the trip).

Process-level composition covers a later phase of the overall composition task. Here 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. In this phase it is not sufficient to define the Web services in terms of inputs, outputs, preconditions and effects. A more detailed description of each Web service is necessary in order to generate the exact sequence of operations for interacting with the service provider. (Such description is provided for instance in the OWL-S process model, or in the WSMO service interface).

In the case of the "Flight Booking" service, for instance, this interaction requires several steps including authentication, submission of a specific request, negotiation of an offer, acceptance (or refusal) of the offer, and payment. Moreover, these steps may have conditional, or non-nominal outcomes (e.g., there may be no offer available from an existing service...) that may affect the following steps (e.g., if there is no offer available, an order cannot be submitted...). All these details are inessential for defining the functional-level composition, but become important for the generation of the actual code implementing it. The necessity of doing service composition at the process level is recognized in several works, notably in [NM02], where an approach is proposed for simulation, verification and composition of complex services described in OWL-S. An automated approach for the process-level composition of Web service is described in [TP04]; also in this case OWL-S is used as modeling language.

The two forms of automatic compositions require different conceptual frameworks and different techniques. For this reason, in this chapter, functional-level and process-level composition are discusses separately, in Sections 3.1 and 3.2 respectively. We postpone the discussion on how to combine them until Chapter 4, where we will also discuss how to compose these two functionalities with automatic service discovery.

3.1 Functional-Level Composition

Here we discuss approaches for the fully automated, functional composition of services according to user constraints. This approach to service composition is very much related to traditional AI planning. The service composition problem is specified by a set of available inputs and a set of required outputs. The services correspond to planning operators: They require certain inputs (preconditions) and provide outputs (effects). The planning results in an arrangement of services in a workflow to fulfill the user requirements. One important difference to planning is that the set of service descriptions (i.e., the planning operators) may be very large and is usually maintained in service directories. Hence, it is crucial for service composition algorithms to interact with service directories in order to dynamically retrieve relevant services. In order to achieve reasonable composition performance, the interaction between composition algorithm and service directory has to be carefully crafted. In this section, we will consider several types of matching that are used by the functional-level composition. The notions of match used do not totally follow the conceptual model defined in Chapter 2, and are limited to discovery, not involving contracting. The relation between the model presented in Chapter 2 and composition will be discussed in Chapter 4, and a complete alignment between the discovery and contracting model and composition will be subject of future work in the Knowledge Web project.

The remainder of this section is structured as follows: In Section 3.1.1 we briefly review existing planning formalisms and service composition systems based on planning. In Section 3.1.2 we introduce our formalism to describe service advertisements and requests. We also formalize different forms of matching requests with advertisements.

In Section 3.1.3 we introduce type-compatible service composition, i.e., service composition that takes type constraints into account. We also present a concrete service composition algorithm that supports partial type matches. In Section 3.1.4 we give an overview of implementation techniques of service directories to support scalable and efficient automated service composition, including techniques for multidimensional indexing, the support for large result sets (incremental retrieval of results), efficient concurrency control, and the support for user-defined search heuristics. Finally, in Section 3.1.5 we present our testbed that offers several models to simulate large service directories. We present experimental results that underline the benefits of supporting partial type matches in the process of service composition. Thanks to this support for partial type matches, a much larger part of the problem set can be solved by automated composition. We also present results that stress the importance of supporting user-defined heuristics in the directory search.

3.1.1 Background and State of the Art

In this section we briefly review some related work in the area of service composition. First, we discuss STRIPS planning, which is the basis for many research work in the area of functional-level service composition. We also review Golog, a composition approach based on the situation calculus, and SHOP-2, a hierarchical planning system.

STRIPS Planning

STRIPS was adopted by the research community as the basis for the majority of planning formalisms used today. Different extensions were provided to it and a number of STRIPS flavors exist [FN71, Lif87, McD03], but the high-level description presented next is common to all approaches.

In STRIPS the world is described in terms of its state. Actions are described in terms of precondition, ‘add’ and ‘delete’ lists. The precondition list of an action describes state that has to be true in the world before the action can be applied. The ‘add’ list describes state in the world that will be true after the planning operator has been applied. The ‘delete’ list describes state in the world that will be false after the operator has been applied.

Systems Supporting Automated Service Composition

Golog An initial approach to process composition in [MSZ01] and [MS02b] was to use a planning formalism based on the situation calculus, a firstorder logical language for reasoning about action and change. In the situation calculus, the state of the world is expressed in terms of functions and relations relativized to a particular situation. The advantage of this approach is that complex control constructs like loops can be modelled

using this framework. The drawback of this approach is its high computational complexity.

This work builds on and extends Golog, a highlevel, logic programming language, developed at the University of Toronto. Golog supports the specification and execution of complex actions in dynamical domains.

SHOP-2 In [Wu,03] the authors describe SHOP2, a hierarchical planning formalism for encoding the composition domains. This approach is more efficient but it doesn't support complex constructs like loops.

SHOP2 is a domain-independent HTN planning system. HTN planning is an AI planning methodology that creates plan by task decomposition. This is a process in which the planning system decomposes tasks into smaller and smaller subtasks, until primitive tasks are found that can be performed directly. The concept of task decomposition in HTN is very similar to the concept of process decomposition in OWL-S.

One difference between SHOP2 and most other HTN planning systems is that SHOP2 plans for tasks in the same order that they will later be executed. Planning for tasks in the order they will be performed makes it possible to know the current state of the world at each step in the planning process, which makes it possible for SHOP2's precondition-evaluation mechanism to incorporate significant inferencing and reasoning power, including the ability to call external programs. This allows SHOP2 to integrate planning with external information sources as in the Web environment.

In order to do planning in a given planning domain, SHOP2 needs to be given the knowledge about that domain. SHOP2's knowledge base contains operators and methods. Each operator is a description of what needs to be done to accomplish some primitive task, and each method tells how to decompose some compound task into partially ordered subtasks.

3.1.2 Formalism and Semantics

In this section we give some basic definitions and introduce our formalism for describing service advertisements and service requests together with their associated semantics. This formalism is very similar to the description of services used in Chapter 2. However, some differences arise and will be discussed. We briefly repeat from Chapter 2 some state-of-the-art regarding matchmaking that is of interest for our composition approach. We introduce interval constraints, a supporting formalism which we use for describing and matching services.

Service Advertisements and Requests

The functional aspects of service advertisements and service requests are specified as parameters and states of the world [CCMW01, DS04]. Parameters can be either *input* or *output*, and states of the world can be either *preconditions* (required states) or *effects* (generated by the execution of the service). We presume that terms in the service descriptions are defined using a class/ontological language like OWL [DS04]. Primitive data-types can be defined using a language like XSD [W3C].[1] As specified by the latest version of OWL-S [Coa04], in our formalism each parameter has two elements:

- A *role* describing the actual semantics of the parameter (e.g., in a travel domain the role of a parameter could be $departure$ or $arrival$).
- A *type* defining the actual datatype of the parameter (e.g., the datatype for both $departure$ and $arrival$ could be $location$).

We define states of the world through preconditions and effects. We extend the normal semantics of concepts that can be included in preconditions or effects.

In service advertisements input and output parameters, as well as preconditions and effects, have the following semantics:

- In order for the service to be invokable, a value must be known for each of the service input parameters and it has to be consistent with the respective semantic role and syntactic type of the parameter. The parameter provided as input has to be semantically more specific than what the service is able to accept. Regarding the parameter type, in the case of primitive data types the invocation value must be in the range of allowed values, or in the case of classes the invocation value must be subsumed by the parameter type. The preconditions define in which state the world has to be before the service can be invoked. All preconditions must be entailed by the conditions specified by the current state of the world.
- Upon successful invocation the service will provide a value for each of the output parameters and each of these values will be consistent with the respective parameter role and datatype. After invocation the state of the world will be modified such that all effects listed in the service advertisement will be added to the new world state. Terms in the original state conflicting with terms in the new state will be removed from the new state.

The above semantics is consistent with the descriptions introduced in Section 2.3. Service requests are represented in a similar manner but have different semantics:

[1] At the implementation level both primitive datatypes and classes are represented as sets of numeric intervals [CF03].

- The service request inputs represent available parameters (e.g., provided by the user or by another service). Each of these input parameters has attached a semantic role description and either some description of its datatype or a concrete value. Preconditions in a request represent the state of the world available for any matching service advertisement. They are equivalent to initial conditions in a classic planning environment. This state has to entail the state required in the precondition of any compatible service.
- The service request outputs represent parameters that a compatible (composed) service must provide. The parameter role defines the actual semantics of the required information and the parameter type defines what ranges of values can be handled by the requester. The compatible (composed) service must be able to provide a value for each of the parameters in the output of the service request, semantically more specific than the requested role, and having values in the range defined by the requested parameter type. Effects represent the change of the world desired by the requester of the service or the goals that the service request needs to be fulfilled. In order for any of the goals or effects of the service request to be considered fulfilled, the state of the world after the invocation of a given service will have to contain an effect entailing the respective goal.

Notice that in the descriptions presented in Section 2.3 inputs are not used for discovery but only for contracting, and at contracting time instance data is expected to be made available by the requester. In addition, the conditions over the state of the world are not represented in the goal i.e. request. These differences will be discussed in Chapter 4.

Matchmaking – Current Approaches

As presented in Section 2.2, previous work regarding the matching of software components [ZW97] has considered several possible match types based on the implication relations between preconditions and postconditions of a library component S and a query Q. For example the **PlugIn** match, one of the most useful match types is defined as:

$$match_{PlugIn}(Q, S) = (pre_Q \Rightarrow pre_S) \wedge (post_S \Rightarrow post_Q).$$

In LARKS [SWKL02] the above condition has been adapted such that the implication was replaced my a more tractable operation, the θ subsumption over sets of constraints ($\preceq_\theta$):

$$match_{PlugIn}(Q, S) = (pre_Q \preceq_\theta pre_S) \wedge (post_S \preceq_\theta post_Q).$$

A set of constraints pre_S θ-subsumes a set of constraints pre_Q ($pre_Q \preceq_\theta pre_S$ or otherwise $pre_Q \sqsubseteq pre_S$ or $pre_Q \Rightarrow pre_S$), if every constraint in pre_Q is subsumed by a

constraint in pre_S (similarly for postconditions):

$$pre_Q \preceq_\theta pre_S \Leftrightarrow$$
$$(\forall C_Q \in pre_Q)(\exists C_S \in pre_S)(C_Q \preceq_\theta C_S).$$

Most recent work regarding matchmaking [PKPS02, LH03, CF03] has extended these approaches by using description logic based languages [BS01, DS04] for defining terms of service advertisements or requests.

Interval Constraints

For describing service advertisements and requests we use constraints on sets of intervals (possibly generated from class descriptions [CF03]). A constraint is a special form of first order predicate that universally quantifies over the values of the interval sets ; in the case that an interval represents the encoding of a class the constraint will correspond to a quantification over all the individuals in the class:

$$P(C_1, C_2, ..., C_n) \Leftrightarrow$$
$$(\forall x_1 \in C_1)(\forall x_2 \in C_2)...(\forall x_n \in C_n)P(x_1, x_2, ..., x_n).$$

We define a number of possible relations between two interval sets C_1 and C_2:

$$C_1 \sqsubseteq C_2 \Leftrightarrow\ (\forall i_1 \in C_1)\ (\exists i_2 \in C_2)(i_1 \subseteq i_2),$$
$$C_1 \equiv C_2 \Leftrightarrow\ C_1 \sqsubseteq C_2 \wedge C_2 \sqsubseteq C_1,$$
$$C_1 \dot{\sqcap} C_2 \Leftrightarrow (\exists i_1 \in C_1)\ (\exists i_2 \in C_2)(i_1 \cap i_2 \neq \emptyset).$$

The relation $\neg\dot{\sqcap}$ is the logical negation of $\dot{\sqcap}$ and holds when the argument interval sets are disjoint. We define also two special relations: top $\dot{\top}$ that always holds and bottom $\dot{\bot}$ that never holds. There is a similarity between the θ subsumption relation between sets of clauses and the interval set subsumption relation $\sqsubseteq$.

We assume that constraints have unique arities - that is constraints with the same name have always the same number of terms.

We define ent, a complex entailment relation between two constraints $P_1(C_{11}, ..., C_{1n})$ and $P_2(C_{21}, ..., C_{2n})$ having same arity n but possibly different names P_1 and P_2. The predicate $ent(P_1, P_2, op_1, ..., op_n)$ holds when each of the terms C_{1i} and C_{2i} of the two constraints are in the relation specified by the respective operator op_i:

$$ent(P_1, P_2, op_1, ..., op_n) \Leftrightarrow \bigwedge_{i=1}^{n} C_{1i}\ op_i\ C_{2i}$$
where $op_i \in \{\equiv, \sqsubseteq, \sqsupseteq, \dot{\sqcap}, \neg\dot{\sqcap}, \dot{\top}, \dot{\bot}\}, i = 1..n.$

We define $notEnt$, a non-entailment relation having semantics in concordance with those of ent - the predicate holds when at least one of the terms C_{1i} and C_{2i} is not in the relation specified by the respective operator op_i:

$$notEnt(P_1, P_2, op_1, ..., op_n) \Leftrightarrow \bigvee_{i=1}^{n} \neg(C_{1i}\ op_i\ C_{2i})$$
where $op_i \in \{\equiv, \sqsubseteq, \sqsupseteq, \dot{\sqcap}, \neg\dot{\sqcap}, \dot{\top}, \dot{\bot}\}, i = 1..n$.

Constraints can be grouped in constraint stores. A constraint store $\mathcal{S}$ is logically equivalent to the formula formed as the conjunction of the constraints in the store:

$$\mathcal{S} = \{P_1(C_{11}, ..., C_{1n}), ..., P_k(C_{k1}, ..., C_{km})\} \Leftrightarrow$$
$$P_1(C_1, ..., C_n) \wedge ... \wedge P_k(C_{k1}, ..., C_{km}).$$

By combining universal (all) and existential ($some$) quantifiers over a pair of constraint stores $\mathcal{Q}$ and $\mathcal{S}$ we can define eight predicates (e.g., $all_{\mathcal{Q}}all_{\mathcal{S}}$, $all_{\mathcal{Q}}some_{\mathcal{S}}$,..., $all_{\mathcal{S}}all_{\mathcal{Q}}, all_{\mathcal{S}}some_{\mathcal{Q}}$,..., etc). Each of the predicates holds if the two stores contain constraints accordingly to the quantifications $q_{\mathcal{Q}}$ and $q_{\mathcal{S}}$ that are in a relation as defined above by ent:

$$q_1q_2(P_{\mathcal{Q}}, P_{\mathcal{S}}, op_1, ..., op_n) \Leftrightarrow$$
$$((\forall \mid \exists)(P_{\mathcal{Q}} \mid P_{\mathcal{S}}))((\forall \mid \exists)(P_{\mathcal{S}} \mid P_{\mathcal{Q}}))$$
$$(P_{\mathcal{Q}} \in \mathcal{Q})(P_{\mathcal{S}} \in \mathcal{S})\ \ ent(P_{\mathcal{Q}}, P_{\mathcal{S}}, op_1, ..., op_n),$$

where $q_1, q_2 \in \{all_{\mathcal{Q}}, all_{\mathcal{S}}, some_{\mathcal{Q}}, some_{\mathcal{S}}\}$, $store(q_1) \neq store(q_2)$ and where $store(quant_{\mathcal{X}}) = \mathcal{X}$ for $quant \in \{all, some\}$, $\mathcal{X} \in \{\mathcal{Q}, \mathcal{S}\}$.

We also explicitly define the negation of the quantification predicates with semantics that can be straightforwardly deduced by the application of DeMorgan's laws for quantifier transformation. After applying these transformations (assumed to be already done on the right part of the expression below) the formula can be written in terms of the non-entailment predicate $notEnt$:

$$\neg q_1q_2(P_{\mathcal{Q}}, P_{\mathcal{S}}, op_1, ..., op_n) \Leftrightarrow$$
$$((\exists \mid \forall)(P_{\mathcal{S}} \mid P_{\mathcal{Q}}))((\exists \mid \forall)(P_{\mathcal{Q}} \mid P_{\mathcal{S}}))$$
$$(P_{\mathcal{Q}} \in \mathcal{Q})(P_{\mathcal{S}} \in \mathcal{S})\ \ notEnt(P_{\mathcal{Q}}, P_{\mathcal{S}}, op_1, ..., op_n),$$

where q_1 and q_2 are as above and the negation is propagated over the quantifiers using the extended DeMorgan laws: $\neg all \rightarrow some\neg$, $\neg some \rightarrow all\neg$, $\neg ent \rightarrow notEnt$.

We define $count$ a function which returns the cardinality of a set of constraints selected from the constraint store $\mathcal{S}$ accordingly to their entailment relation with constraints in the store $\mathcal{Q}$:

$$count_{\mathcal{Q},\mathcal{S}}(P_{\mathcal{Q}}, P_{\mathcal{S}}, op_1, ..., op_n) = \\ \mid \{P_{\mathcal{S}} \in \mathcal{S} : P_{\mathcal{Q}} \in \mathcal{Q}, ent(P_{\mathcal{Q}}, P_{\mathcal{S}}, op_1, ..., op_n)\} \mid.$$

We introduce also $count_{\mathcal{Q}}$ and $count_{\mathcal{S}}$ two functions which return the cardinality of a set of constraints having a given name P from the stores $\mathcal{Q}$ or $\mathcal{S}$:

$$count_{\mathcal{Q}}(P) = \mid \{P(C_1, ..., C_n) \in \mathcal{Q}\} \mid, \\ count_{\mathcal{S}}(P) = \mid \{P(C_1, ..., C_n) \in \mathcal{S}\} \mid.$$

Describing Services by Interval Constraints

We use constraint stores to define service advertisements or service requests. In this deliverable we will consider the latter as user queries but this doesn't necessarily have to be so. Input and output constraints are defined over the two kind of elements that describe a parameter - roles for semantics and types for syntactic compatibility. Preconditions and effects are defined over concepts describing features of the world. The exact semantics of input, output parameters and preconditions and effects are defined above, depending if they are in the scope of a service advertisement or a service request. Four kinds of constraints are used in service descriptions:

- $IN(R, T)$ - which defines an input parameter through its role R and type T.
- $OUT(R, T)$ - which defines an output parameter through its role R and type T.
- $PRE(F)$ - which defines a precondition through the world state F.
- $EFF(F)$ - which defines an effect through the world state F.

Let's consider as an example a service description with two input parameters having roles A and B and types a1-a2, b1, output parameters having roles C, D and types c1, d1-d2 with preconditions p1 and p2 and effects g1. This service description would be represented as the following constraint store: $\mathcal{S}$ = { $IN(A, a1 - a2)$, $IN(B, b1)$, $OUT(C, c1)$, $OUT(D, d1 - d2)$, $PRE(p1)$,$PRE(p2)$, $EFF(g1)$ }.

In order to illustrate our approach we show below how the basic **PlugIn** match type is expressed in our formalism. For a query store $\mathcal{Q}$ and a service store $\mathcal{S}$ this match type can be specified as:

$$match_{PlugIn}(\mathcal{Q}, \mathcal{S}) = \\ all_{\mathcal{S}} some_{\mathcal{Q}}(IN_Q, IN_S, \sqsubseteq_{role}, \sqsubseteq_{type}) \wedge$$

$$all_{\mathcal{Q}}some_{\mathcal{S}}(OUT_Q, OUT_S, \sqsupseteq_{role}, \sqsupseteq_{type}) \wedge$$
$$all_{\mathcal{S}}some_{\mathcal{Q}}(PRE_Q, PRE_S, \sqsubseteq) \wedge$$
$$all_{\mathcal{Q}}some_{\mathcal{S}}(EFF_Q, EFF_S, \sqsupseteq).$$

3.1.3 Automated Service Composition

In this section we introduce type-compatible service composition, i.e., service composition that takes type constraints into account. We also present a concrete service composition algorithm that supports partial type matches.

Type-Compatible Discovery and Composition

For composition we considering two kinds of possible approaches: forward chaining and backward chaining. Informally, the idea of forward chaining is to iteratively apply a possible service S to a set of input parameters provided by a query Q (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 query Q are available) then a new query Q' can be computed from Q and S and the whole process is iterated. This part of our framework corresponds to the planning techniques currently used for service composition [TKAS02]. In the case of backward chaining we start from the set of parameters required by the query Q and at each step of the process we choose a service S that will provide at least one of the required parameters. Applying S might result in new parameters being required which can be formalised as a new query Q'. Again the process is iterated until a solution is found.

Now we consider the conditions needed for a service S to be applied to the inputs available from a query Q using forward chaining: for all of the inputs required by the service S, there has to be a compatible parameter in the inputs provided by the query Q. Compatibility has to be achieved both for roles, where the role of any parameter provided by the query Q has to be semantically more specific ($\sqsubseteq$) than the role of the parameter required by the service S, and for types, where the range provided by the query Q has to be more specific ($\sqsubseteq$) than the one accepted by the service S. In the formalism introduced above the forward complete chaining condition would map to the $all_{\mathcal{S}}some_{\mathcal{Q}}$ predicate:

$$fwdComp(\mathcal{Q}, \mathcal{S}) =$$
$$all_{\mathcal{S}}some_{\mathcal{Q}}(IN_Q, IN_S, \sqsubseteq_{role}, \sqsubseteq_{type}) \wedge all_{\mathcal{S}}some_{\mathcal{Q}}(PRE_Q, PRE_S, \sqsubseteq).$$

A similar kind of **PlugIn** match between the inputs of query Q and of service S has been identified by Paolluci [PKPS02] for the matchmaking of DAML-S services.

Forward complete matching of types is too restrictive and might not always work, because the types accepted by the available services may partially overlap the type specified in the query. For example, a VTA might offer restaurant recommendations when booking

a full holidays trip. When using a restaurant recommendations provider, a query given by the VTA for restaurant recommendation services across all Switzerland could specify that the integer parameter zip code could be in the range [1000,9999] while an existing service providing recommendations for the french speaking part of Switzerland could accept only integers in the range [1000-2999] for the zip code parameter.

A major novelty of our approach regarding composition is in that the above condition for forward chaining is modified such that services with *partial type matches* can be supported. For doing that we relax the type inclusion to a simple overlap:

$$fwdPart(\mathcal{Q},\mathcal{S}) = \\ all_{\mathcal{S}}some_{\mathcal{Q}}(IN_Q, IN_S, \sqsubseteq_{role}, \dot{\sqcap}_{type}) \wedge all_{\mathcal{S}}some_{\mathcal{Q}}(PRE_Q, PRE_S, \sqsubseteq).$$

This kind of matching between the inputs of query Q and of service S corresponds to the **overlap** or **intersection** match identified by Li [LH03] and Constantinescu [CF03].

We will also consider the condition needed for a backward chaining approach. The service S has to provide at least one output which is required by the query Q. This corresponds to the **plugIn** match for query and service outputs. Using the formal notation above this can be specified as:

$$backComp(\mathcal{Q},\mathcal{S}) = some_{\mathcal{Q}}some_{\mathcal{S}}(OUT_Q, OUT_S, \sqsupseteq_{role}, \sqsupseteq_{type}) \vee \\ some_{\mathcal{Q}}some_{\mathcal{S}}(EFF_Q, EFF_S, \sqsupseteq).$$

As discussed in Chapter 2, other notions of match at discovery time are also possible, and the intentions of requesters and providers might be considered when determining a match. In addition, contracting can be seen as a step in the selection of services suitable for the problem at hand. Finally, in the composition approach presented in this section, goals i.e. user requests are assumed to be already formalized. However, a goal selection and refinement might be required for realistic description of user requests. These issues will be discussed in Chapter 4 and future work will address them.

Type-Compatible Service Composition Versus Planning

As the majority of service composition approaches today rely on planning we will analyze the correspondence between our formalism for service descriptions with types and an hypothetic planning formalism using symbol-free first order logic formulas for preconditions and effects.

As an example (see Table 3.1 let's consider the service description S which has two input parameters A and B and two output parameters C and D. Their types are represented as sets of accepted and provided values and are a1, a2 for A, respectively b1, b2 for B, c1, c2 for C, and d1, d2 for D. This corresponds to an operator S that has disjunctive

```
S = {                                :action S
                                        :precondition
                                           (and
   IN(A,[a1, a2]),                            (or a1 a2)
   IN(B,[b1, b2]),                            (or b1 b2))

                                         :effect
                                           (and
   OUT(C,[c1, c2]),                           (or c1 c2)
   OUT(D,[d1, d2])                            (or d1 d2))

}
```

Figure 3.1: Service with types and corresponding planning operator.

preconditions and disjunctive effects. Negation is not required.

Written in this way our formalism has some correspondence with existing planning languages like ADL [Ped89] or more recently PDDL [McD98] (concerning the disjunctive preconditions) and planning with non-deterministic actions [KHW95] (regarding the disjunctive effects), but the combination as a whole (positive-only disjunctive preconditions and effects) stands as a novel formalism.

Computing Type-Compatible Service Compositions

In this section we will present algorithms for computing type-compatible service compositions. Their design is motivated by two aspects specific to large scale service directories operating in open environments:

- **large result sets** - for each query the directory could return a large number of service descriptions.
- **costly directory accesses** - being a shared resource accessing the directory (possibly remotely) will be expensive.

We address these issues by interleaving discovery and composition and by computing the “right” query at each step. For that, the integration engine (see Fig. 3.2) uses three separate components:

- **planner** - a component that computes what can be currently achieved from the current query using the current set of discovered services. From that the problem that remains to be solved is derived and a new query is returned.

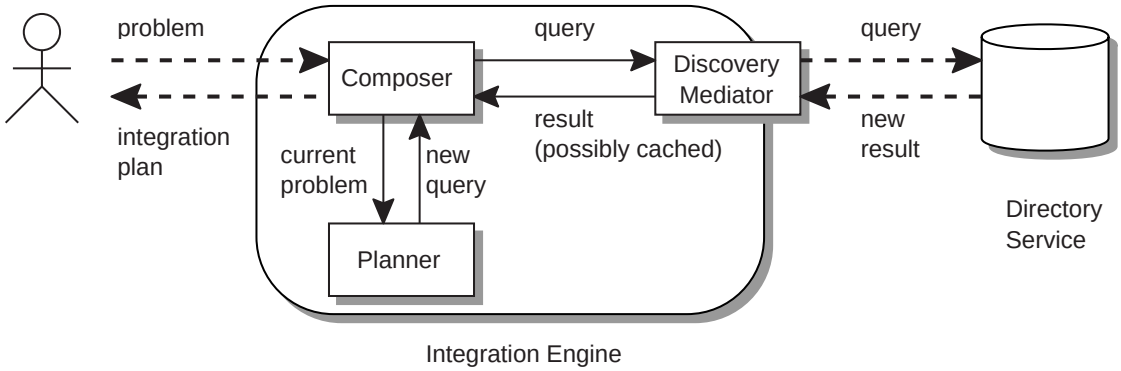


Figure 3.2: The architecture of our service integration engine.

- **composer** - a component that implements the interleaving between planning and discovery. It decides what kind of queries (partial/complete) should be sent to the directory and it deals with branching points and recoursive solving of sub-problems.

- **discovery mediator** - a component that mediates composer accesses to the directory by caching existing results and matching new queries to already discovered services.

Composition with Complete Type Matches

Composing completely matching services using forward chaining is straightforward: once the condition for complete type matches in Section 3.1.3 is fullfilled (all inputs required by the service S are present in the query Q and the types in the query are more specific than the types accepted by the service) a new query Q' can be computed by adding to the set of available inputs of the current query Q all the outputs provided by the service S.

Composition with Forward Partial Type Matches

Conceptually the algorithm that we use for composing services with forward partial type matches has three steps (for more details see [CFB04b]):

- Discovery of completely matching services.

- Discovery of services for full coverage of available inputs.

- Discovery of services for correct switch handling.

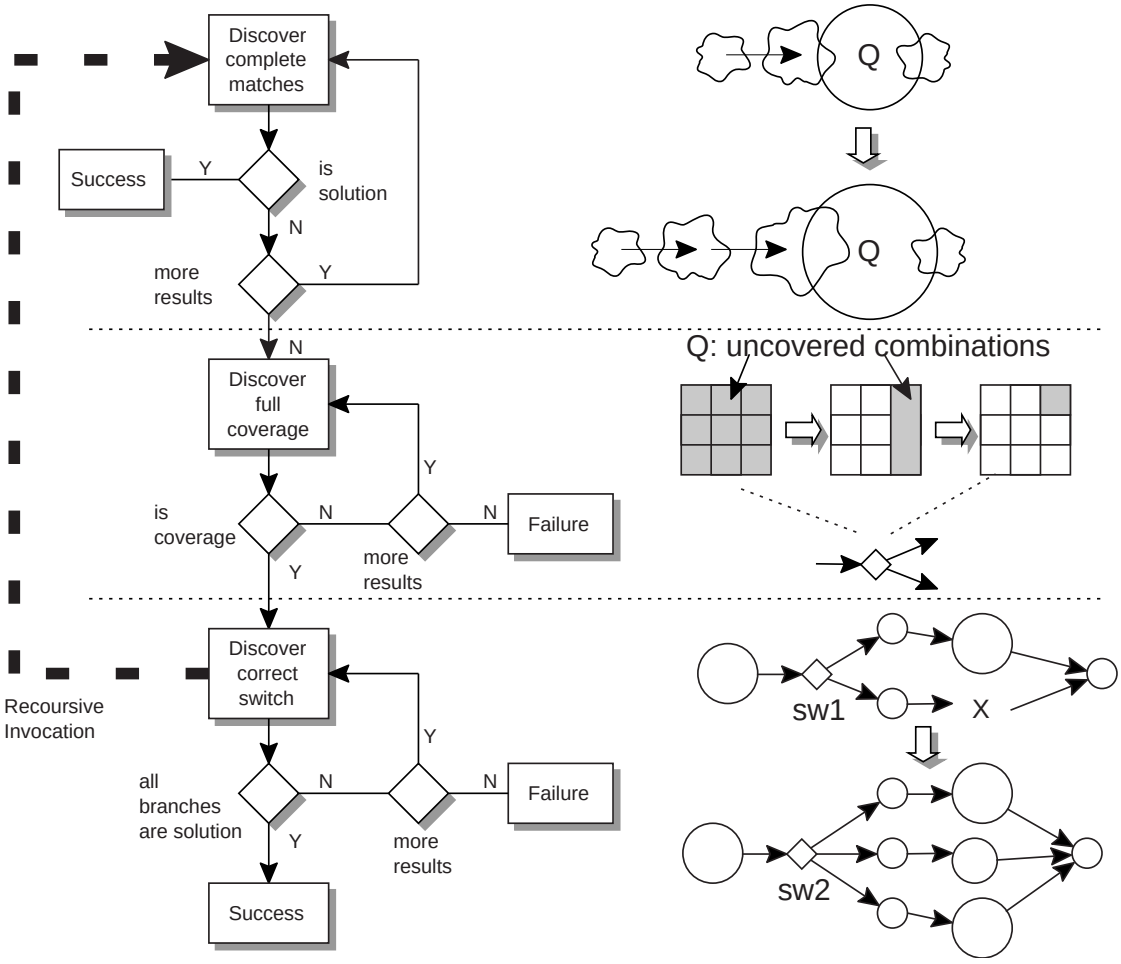


Figure 3.3: Flow of algorithm for composition with partial type matches.

Discovering full input coverage The second step of the algorithm assumes that a solution using only complete matches was not found and that services with partial type matches have to be assambled in order to solve the problem. By definition any of the partially matching services is able to handle only a limited sub-space of the values available as inputs. In order to ensure that any combination of input values can be handled, the space of available inputs is first discretized in parameter value cells. One cell is a rectangular hyperspace containing all dimensions of the space of available inputs but only a single interval for each dimension. A cell corresponds to the guard condition of the switch. Cells are built in such a way that any of the required inputs for the retrieved partially matching services could be expressed as a collection of cells. Each of the retrieved partially matching services is assigned to the cells that it can accept as input. The coverage is considered complete when all cells have assigned one or more services. When all cells are covered the algorithm proceeds at the next step. If no more partially matching services can be found and a complete coverage was not achieved the algorithm returns failure.

Discovering solution switch The last step of the algorithm assumes that a coverage was found and a first switch can be created. The goal of this step is to ensure that the switch will function correctly for each of its branches. For each cell and its set of assigned services the algorithm will compute the set of output parameters that those services will provide. Then a new query is computed, having as available inputs the output parameters of the cell and as required outputs the set of required outputs of the complete matching phase. The whole composition procedure is then invoked recursively. In the case that all cells return a successful result the switch is considered to be correct and the algorithm returns success. Otherwise a new service is retrieved and the process continues. When no more services can be retrieved the algorithm returns failure.

3.1.4 Implementation Techniques (Directory Support for Automated Service Composition)

Here we give an overview of implementation techniques of service directories to support scalable and efficient automated service composition, including techniques for multidimensional indexing, the support for large result sets (incremental retrieval of results), efficient concurrency control, and the support for user-defined search heuristics [CBF04b, CBF04a, BCF04].

Multidimensional Access Methods - GiST

The need for efficient discovery and matchmaking leads to a need for search structures and indexes for directories. We consider numerically encoded service descriptions as

multidimensional data and use techniques related to the indexing of such kind of information in the directory. Our directory index is based on the Generalized Search Tree (GiST), proposed as a unifying framework by Hellerstein [HNP95] (see Fig. 3.4). The design principle of GiST arises from the observation that search trees used in databases are balanced trees with a high fanout in which the internal nodes are used as a directory and the leaf nodes point to the actual data.

Each internal node holds a key in the form of a predicate P and a number of pointers to other nodes (depending on system and hardware constraints, e.g., filesystem page size). To search for records that satisfy a query predicate Q, the paths of the tree that have keys P satisfying Q are followed.

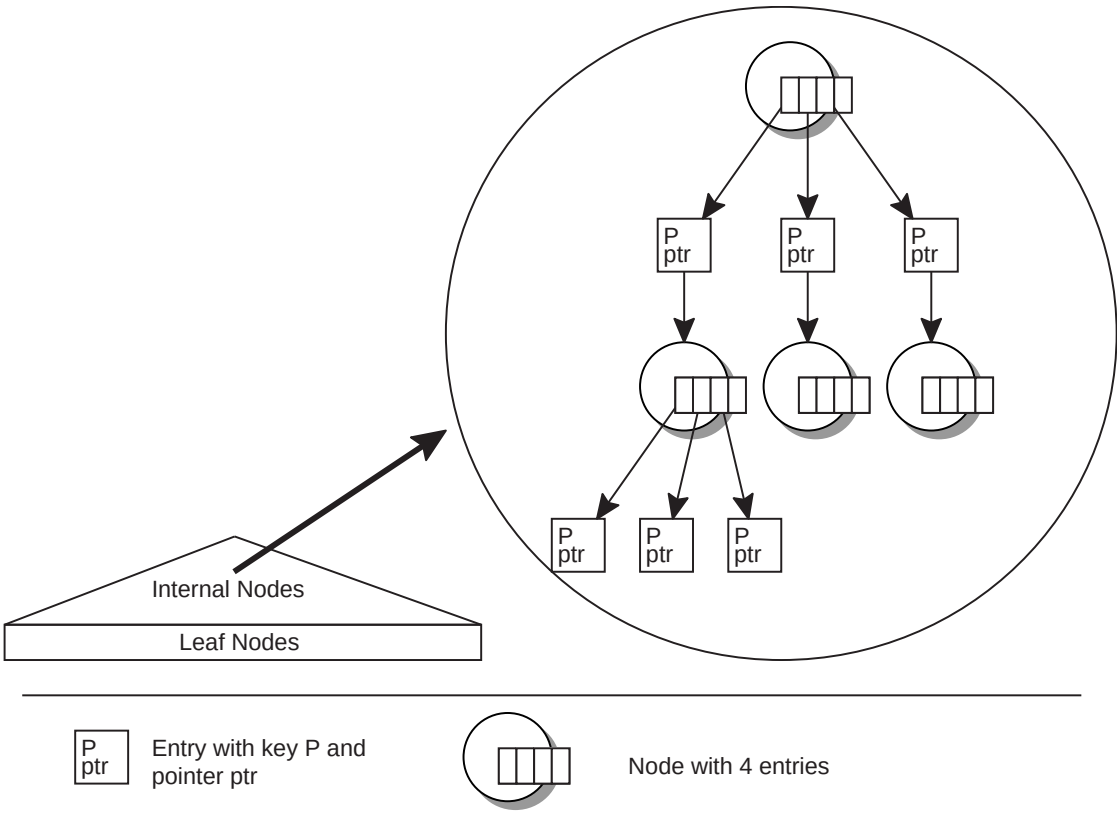


Figure 3.4: Generalised Search Tree (GiST).

More concretely, Each leaf node in the GiST of our directory holds references to all service descriptions with a certain input/output behaviour. The required inputs of the service and the provided outputs (sets of parameter names with associated types) are stored in the leaf node. For inner nodes of the tree, the union of all inputs/outputs found in the subtree is stored. More precisely, each inner node I on the path to a leaf node L contains all input/output parameters stored in L. The type associated with a parameter in I subsumes the type of the parameter in L. That is, for an inner node, the input/output pa-

rameters indicate which concrete parameters may be found in a leave node of the subtree. If a parameter is not present in an inner node, it will not be present in any leave node of the subtree.

Service Integration Sessions and Concurrency Control

As directory queries may retrieve large numbers of matching entries (especially when partial matches are taken into consideration), it is important to support incremental access to the results of a query in order to avoid wasting network bandwidth. Our directory service offers *sessions* which allow a user to issue queries to the directory and to retrieve the results one by one (or in chunks of limited size).

The session guarantees a consistent view of the directory, i.e., the directory structure and contents as seen by a session does not change. Concurrent updates (service registration, update, and removal) do not affect the sequence of query results returned within a session; sessions are isolated from concurrent modifications.

Previous research work has addressed concurrency control in generalized search trees [KMH97]. However, these concurrency control mechanisms only synchronize individual operations in the tree, whereas our directory supports long-lasting sessions during which certain parts of the tree structure must not be altered. This implies that insertion and deletion operations may not be performed concurrently with query sessions, as these operations may significantly change the structure of the tree (splitting or joining of nodes, balancing the tree, etc.).

The following assumptions underly the design of our concurrency control mechanism:

1. Read accesses (i.e., queries within sessions and the incremental retrieval of the results) will be much more frequent than updates.
2. High concurrency for read accesses (high number of concurrent query sessions).
3. Read accesses shall not be delayed.
4. Updates may become visible with a significant delay, but feedback concerning the update (success/failure) shall be returned immediately.
5. The duration of a session may be limited (timeout).

In order to meet the assumptions above, we have designed a mechanism which guarantees that sessions operate on read-only data structures that are not subject to changes. In our approach the in-memory structure of the directory tree (i.e., the directory index) is replicated up to 3 times, while the actual service descriptions are shared between the replicated trees.

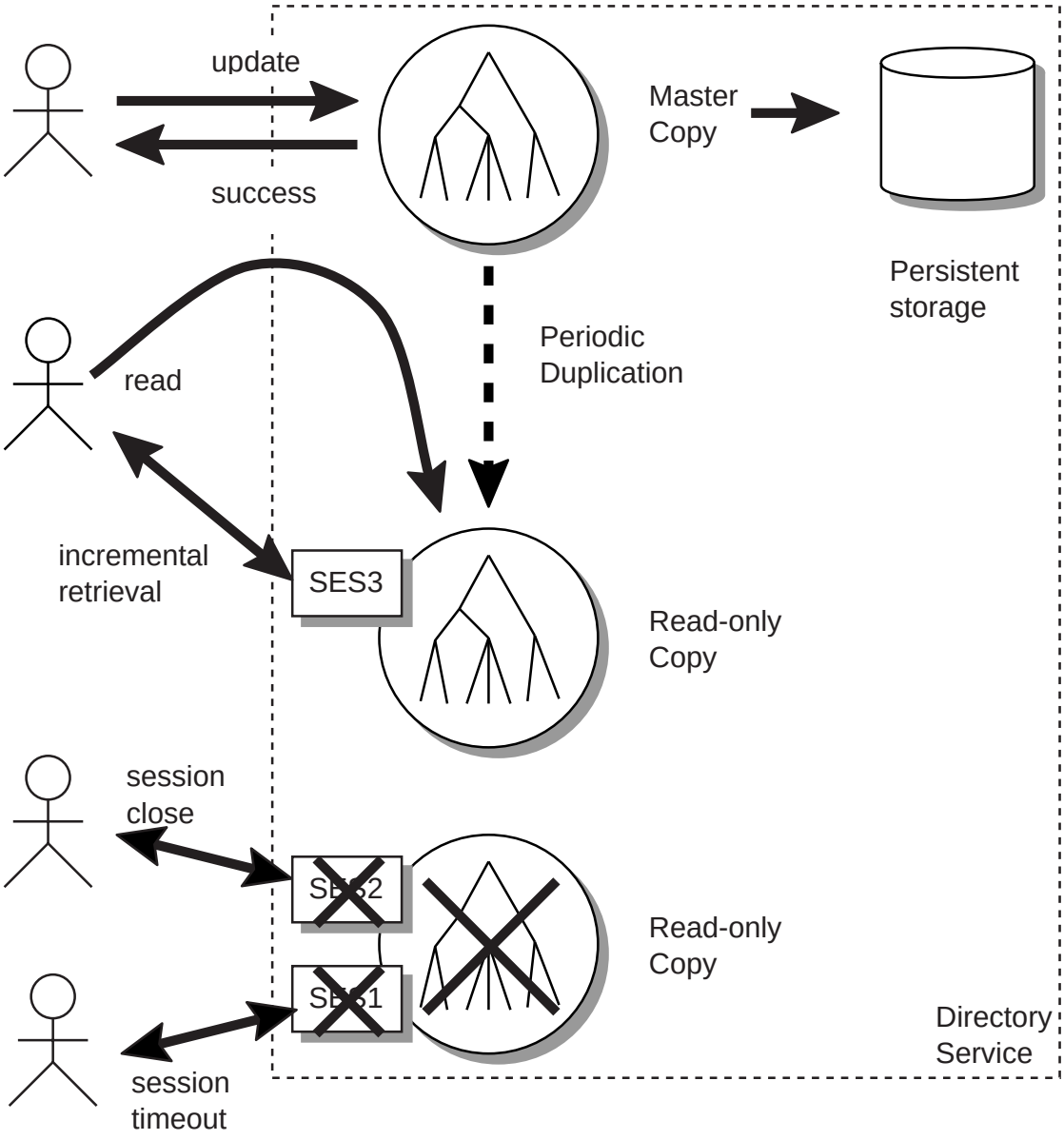


Figure 3.5: Session life-cycle.

When the directory service is started, the persistent representation of the directory tree is loaded into memory. This master copy of the directory tree is always kept up to date, i.e., updates are immediately applied to that master copy and are made persistent, too. Upon start of the directory service, a read-only copy of the in-memory master copy is allocated. Sessions operate only on this read-only copy. Hence, session management is trivial, there are no synchronization needs. Periodically, the master copy is duplicated to create a new read-only copy.[2] Afterwards, new sessions are redirected to the new read-only copy. Garbage collection frees the old read-only copy when the last session operating on it completes (either by an explicit session termination by the client or by a timeout).

We require the session timeout to be smaller than the update frequency of the read-only copy (the duplication frequency of the master copy). This condition ensures that there will be at most 3 copies of the in-memory representation of the directory at the same time: The master where updates are immediately applied (but which is not yet visible to sessions), as well as the previous 2 read-only copies used for sessions. When a new read-only copy is created, the old copy will remain active until the last session operating on it terminates; this time span is bounded by the session timeout.

In our approach only updates to the master copy are synchronized. Updates are immediately applied to the master copy (yielding immediate feedback to the client requesting an update). Only during copying the directory is blocked for further updates. In accord with the third assumption, the creation of sessions requires no synchronization.

Custom Pruning and Ranking Functions

As directory queries may retrieve large numbers of matching entries (especially when partial matches are taken into consideration), our directory support sessions in order to incrementally access the results of a query [CBF04b]. By default, the order in which matching service descriptions are returned depends on the actual structure of the directory index (the GiST structure discussed before). However, depending on the service integration algorithm, ordering the results of a query according to certain heuristics may significantly improve the performance of service composition. In order to avoid the transfer of a large number of service descriptions, the pruning, ranking, and sorting according to application-dependent heuristics should occur directly within the directory. As for each service integration algorithm a different pruning and ranking heuristic may be better suited, our directory allows its clients to define custom pruning and ranking functions which are used to select and sort the results of a query. This approach can be seen as a form of remote evaluation [FPV98]. Within a service integration session multiple pruning and ranking functions may be defined. Each query in a session may be associated with a different one.

[2] Parts of the tree that have not been changed since the last duplication are not copied; these parts of the tree are shared with the previous read-only copy.

API for Pruning and Ranking Functions The pruning and ranking functions receive as arguments information concerning the matching of a service description in the directory with the current query. They return a value which represents the quality of the match. The bigger the return value, the better the service description matches the requirements of the query. The sequence of results as returned by the directory is sorted in descending order of the values calculated by the pruning and ranking functions (a higher value means a better match). Results for which the functions evaluate to zero come at the end, a negative value indicates that the match is too poor to be returned, i.e., the result is discarded and not passed to the client (pruning).

As arguments the client-defined pruning and ranking functions take four `ParamSet` objects corresponding to the input and output parameter sets of the query, and respectively of the service. The `ParamSet` object provides methods like size, membership, union, intersection, and difference (see Fig. 3.6). The size and membership methods require only the current `ParamSet` object, while the union, intersection, and difference methods use two `ParamSet` objects – the current object and a second `ParamSet` object passed as argument.

It is important to note that some of the above methods address two different issues at the same time:

1. Basic *set operations*, where a set member is defined by a parameter name and its type; for deciding the equality of parameters with same name and different types a user-specified expression is used.

2. *Computation of new types* for some parameters in the resulting sets; when a parameter is common to the two argument sets its type in the resulting set is computed with a user-specified expression.

The explicit behavior of the `ParamSet` methods is the following:

size: Returns the number of parameters in the current set.

containsParam: Returns true if the current set contains a parameter with the same name as the method argument (regardless of its type).

union: Returns the union of the parameters in the two sets. For each parameter that is common to the two argument sets the type in the resulting set is computed according to the user-specified expression `newTypeExpr`.

intersection: Returns the parameters that are common to the two sets *AND* for which the respective types conform to the equality test specified by the `eqTestExpr`. The type of these parameters in the resulting set is computed according to the user-specified expression `newTypeExpr`.

minus: Returns the parameters that are present only in the current set and in the case of common parameters only those that *DO NOT* conform to the equality test specified by the `eqTestExpr`. For the latter kind of parameters the type in the resulting set is computed according to the user-specified expression `newTypeExpr`.

Parameters whose `newTypeExpr` would be the empty type (called `NOTHING` in the table below) are removed from the resulting set.

The expressions used in the `eqTestExpr` and `newTypeExpr` parameters have the same format and they are applied to parameters that are common to the two `ParamSet` objects passed to a `union`, `intersection`, or `minus` method. For such kind of parameter we denote its type in the two argument sets as `A` and `B`. The expressions are created from these two types by using some extra constructors based on the Description Logic language OWL [DS04] like $\top$, $\bot$, $\neg$, $\sqcap$, $\sqcup$, $\sqsubseteq$, $\equiv$. The expressions are built by specifying a constructor type and which of the argument types `A` or `B` should be negated. For the single type constructors $\top$ and $\bot$ negation cannot be specified and for the constructors A and B the negation is allowed only for the respective type (e.g., for the constructor type A, only $\neg A$ can be set).

Constructor type	$\neg A$?	$\neg B$?	Possible expressions
`THING`	-	-	$\top$
`NOTHING`	-	-	$\bot$
`A`	Y/N	-	$A, \neg A$
`B`	-	Y/N	$B, \neg B$
`UNION`	Y/N	Y/N	$A \sqcup B$, $A \sqcup \neg B$, $\neg A \sqcup B$, $\neg A \sqcup \neg B$
`INTERSECTION`	Y/N	Y/N	$A \sqcap B$, $A \sqcap \neg B$, $\neg A \sqcap B$, $\neg A \sqcap \neg B$
`SUBCLASS`	Y/N	Y/N	$A \sqsupseteq B$, $A \sqsupseteq \neg B$, $\neg A \sqsupseteq B$, $\neg A \sqsupseteq \neg B$
`SUPERCLASS`	Y/N	Y/N	$A \sqsubseteq B$, $A \sqsubseteq \neg B$, $\neg A \sqsubseteq B$, $\neg A \sqsubseteq \neg B$
`SAMECLASS`	Y/N	Y/N	$A \equiv B$, $A \equiv \neg B$, $\neg A \equiv B$, $\neg A \equiv \neg B$

We represent an expression as a bit vector having a value corresponding to its respective constructor type. For encoding the negation of any of the types that are arguments to the constructor, two masks can be applied to the constructor types: `NEG_A` and `NEG_B`. For the actual encoding, see Fig. 3.6. For example, $A \sqcap \neg B$ will be expressed as `ParamSet.INTERSECTION | ParamSet.NEG_B`.

As an example of API usage, assume we need to select the parameters that are common to the two sets X and Y, which have in X a type that is more specific than the one in Y. In the result set we would like to preserve the type values in X. The following statement can be used for this purpose: `X.intersection(Y, ParamSet.SUPERCLASS, ParamSet.A)`.

The directory supports pruning and ranking functions written in a subset of the Java programming language. The code of the functions is provided as a compiled Java class. The class has to implement the `Ranking` interface shown in Fig. 3.6.

```
public interface Ranking {
  double rankLeaf(  ParamSet qin, ParamSet qout, ParamSet sin, ParamSet sout );
  double rankInner( ParamSet qin, ParamSet qout, ParamSet sin, ParamSet sout );
}

public interface ParamSet {
  static final int THING=1, NOTHING=2, A=3, B=4, UNION=5, INTERSECTION=6,
                   SUBCLASS=7, SUPERCLASS=8, SAMECLASS=9, NEG_A=16, NEG_B=32;

  int size();
  boolean containsParam( String paramName );
  ParamSet union( ParamSet p, int newTypeExpr );
  ParamSet minus( ParamSet p, int eqTestExpr, int newTypeExpr );
  ParamSet intersection( ParamSet p, int eqTestExpr, int newTypeExpr );
}
```

Figure 3.6: The API for ranking functions.

Processing a user query requires traversing the GiST structure of the directory starting from the root node. While `rankInner()` is invoked for inner nodes of the directory tree, `rankLeaf()` is called on leaf nodes. `rankLeaf()` receives as arguments `sin` and `sout` the exact parameter sets as defined by the service description stored in the directory. Hence, `rankLeaf()` has to return a concrete heuristic value for the given service description. In contrast, `rankInner()` receives as arguments `sin` and `sout` supersets of the input/output parameters found in any leaf node of its subtree. The type of each parameter is a supertype of the parameter found in any leaf node (which has the parameter) in the subtree. `rankInner()` has to return a heuristic value which is bigger or equal than all possible ranking values in the subtree. That is, for an inner node the heuristic function has to return an upper bound of the best ranking value that could be found in the subtree. If the upper bound of the heuristic ranking value in the subtree cannot be determined, `Double.POSITIVE_INFINITY` may be used.

The pruning and ranking functions enable the lazy generation of the result set based on a *best-first search* where the visited nodes of the GiST are maintained in a heap or priority queue and the most promising one is expanded. If the most promising node is a leaf node, it can be returned. Further nodes are expanded only if the client needs more results. This technique is essential to reduce the processing time in the directory until the the first result is returned, i.e., it reduces the response time. Furthermore, thanks to the incremental retrieval of results, the client may close the result set when no further results are needed. In this case, the directory does not spend resources to compute the whole result set. Consequently, this approaches reduces the workload in the directory and increases its scalability. In order to protect the directory from attacks, queries may be terminated if the size of the internal heap or priority queue or the number of retrieved results exceed a certain threshold defined by the directory service provider.

Exemplary Ranking Functions In the example in Fig. 3.7 two basic ranking functions are shown, the first one more appropriate for service composition algorithms using

forward chaining (considering only complete matches), the second for algorithms using backward chaining. Note the weakening of the pruning conditions for the inner nodes.

Safe and Efficient Execution of Ranking Functions Using a subset of Java as programming language for pruning and ranking functions has several advantages: Java is well known to many programmers, there are lots of programming tools for Java, and, above all, it integrates very well with our directory service, which is completely written in Java.

Compiling and integrating user-defined ranking functions into the directory leverages state-of-the-art optimizations in recent JVM implementations. For instance, the HotSpot VM [Sun] first interprets JVM bytecode [LY99] and gathers execution statistics. If code is executed frequently enough, it is compiled to optimized native code for fast execution. In this way, frequently used pruning and ranking functions are executed as efficiently as algorithms directly built into the directory.

The class containing the ranking function is analyzed by our special bytecode verifier which ensures that the user-defined ranking function always terminates within a well-defined time span and does not interfere with the directory implementation. Efficient, extended bytecode verification to enforce restrictions on JVM bytecode for the safe execution of untrusted mobile code has been studied in the JavaSeal [VBB98] and in the J-SEAL2 [Bin01, BHVV01] mobile object kernels. Our bytecode verifier ensures the following conditions:

- The `Ranking` interface is implemented.
- Only the methods of the `Ranking` interface are provided.
- The control-flow graphs of the `rankLeaf()` and `rankInner()` methods are acyclic. The control-flow graphs are created by an efficient algorithm with execution time linear with the number of JVM instructions in the method.
- No exception handlers (using malformed exception handlers, certain infinite loops can be constructed that are not detected by the standard Java verifier, as shown in [BR02]). If the ranking function throws an exception (e.g., due to a division by zero), its result is to be considered zero by the directory.
- No JVM subroutines (they may result from the compilation of `finally`{} clauses).
- No explicit object allocation. As there is some implicit object allocation in the set operations in `ParamSet`, the number of set operations and the maximum size of the resulting sets are limited.
- Only the interface methods of `ParamSet` may be invoked, as well as a well-defined set of methods from the standard mathematics package.

```
public final class RankingForward implements Ranking {
  public double rankLeaf( ParamSet qin, ParamSet qout,
                          ParamSet sin, ParamSet sout ) {
    // discard service if it requires parameters that are not in the query;
    // the provided input has to be more specific than the required one
    if (sin.minus(qin, ParamSet.SUBCLASS, ParamSet.A).size() > 0) return -1.0d;

    // services that provide more required parameters are better;
    // the provided output has to be more specific than the required one
    return (double)
             sout.intersection(qout, ParamSet.SUPERCLASS, ParamSet.A).size();
  }

  public double rankInner( ParamSet qin, ParamSet qout,
                           ParamSet sin, ParamSet sout ) {
    // for forward chaining, pruning inner nodes is not possible,
    // but an upper bound of the overlap of the outputs can be easily computed
    return (double)
             sout.intersection(qout, ParamSet.INTERSECTION, ParamSet.A).size();
  }
}

public final class RankingBackward implements Ranking {
  public double rankLeaf( ParamSet qin, ParamSet qout,
                          ParamSet sin, ParamSet sout ) {
    // discard service if it does not provide any required output
    if (sout.intersection(qout, ParamSet.SUPERCLASS, ParamSet.A).size() == 0)
      return -1.0d;

    // services that reduce most the number of required outputs are better
    ParamSet remaining = qout.minus(sout, ParamSet.SUBCLASS, ParamSet.A);
    ParamSet newRequired = sin.minus(qin, ParamSet.SUBCLASS, ParamSet.A);
    ParamSet required = remaining.union(newRequired, ParamSet.INTERSECTION);
    return 1 / (double)(1+required.size());
  }

  public double rankInner( ParamSet qin, ParamSet qout,
                           ParamSet sin, ParamSet sout ) {
    if (sout.intersection(qout, ParamSet.INTERSECTION, ParamSet.A).size() == 0)
      return -1.0d;

    ParamSet remaining = qout.minus(sout, ParamSet.INTERSECTION, ParamSet.A);
    return 1 / (double)(1+remaining.size());
  }
}
```

Figure 3.7: Exemplary pruning and ranking functions.

- Only the static fields defined in the interface `ParamSet` may be accessed.
- No fields are defined.
- No synchronization instructions.

These restrictions ensure that the execution time of the custom pruning and ranking function is bounded by the size of its code. Hence, an attacker cannot crash the directory by providing, for example, a pruning and ranking function that contains an endless loop. Moreover, these functions cannot allocate memory. Our extended bytecode verification algorithm is highly efficient, its performance is linear with the size of the pruning and ranking methods. As a prevention against denial-of-service attacks, our directory service allows to set a limit for the size of custom functions.

Pruning and ranking functions are loaded by separate classloaders, in order to support multiple versions of classes with the same name (avoiding name clashes between multiple clients) and to enable garbage collection of the class structures. The loaded class is instantiated and casted to the `Ranking` interface that is loaded by the system classloader. The directory implementation (which is loaded by the system classloader) accesses the user-defined functions only through the `Ranking` interface.

As service integration clients may use the same ranking functions in multiple sessions, our directory keeps a cache of ranking functions. This cache maps a hashcode of the function class to a structure containing the function bytecode as well as the loaded class. In case of a cache hit the user-defined function code is compared with the cache entry, and if it matches, the function in the cache is reused, skipping verification and avoiding to reload it with a separate classloader. Due to the restrictions mentioned before, multiple invocations of the same ranking function cannot influence each other. The cache employs a least-recently-used replacement strategy. If a function is removed from the cache, it becomes eligible for garbage collection as soon as it is not in use by any service integration session.

3.1.5 Evaluation

In this section we present our testbed that offers several models to simulate large service directories [CFB04a]. We present experimental results that underline the benefits of supporting partial type matches in the process of service composition. Thanks to this support for partial type matches, a much larger part of the problem set can be solved by automated composition. We also present results that stress the importance of supporting user-defined heuristics in the directory search.

Testbed and Simulation Models

Here we present our testbed for large scale service composition. The testbed is build on the assumption that the majority of future Web services will be created by exposing in a machine readable form applications and systems that are currently accessible via human-level interfaces.

From the technology perspective there are several kinds of options for building Web sites ranging from static Web pages to more dynamic one developed using languages specific to server side scripting (e.g., ASP, PHP or JSP). For generating the content to be presented, the dynamic pages can access directly a data layer (e.g., a relational database) or can use and intermediate objectual layer that implements the business logic and encapsulates data (e.g., an application server).

At a higher level, different applications are usually organized accordingly to their domain (e.g., traveling, entertainment, real-estate or medical). Understanding, developing and using the terminology required for a specific domain requires usually a significant knowledge engineering effort.

Our testbed builds on the concepts of data distribution and domain distribution by using application domains as the core idea. In our framework an application domain represents a collection of terms and their associated data types. Then services are defined as transformations between sets of terms in two application domains. Formally this is captured as a directed graph structure, where each node represents an application domain and each edge represents one or more Web services. Each Web service performs a transformation between two given set of terms from the two application domains associated with the two ends of the graph edge.

We will present next in more detail the application domain nodes and the service graph structure.

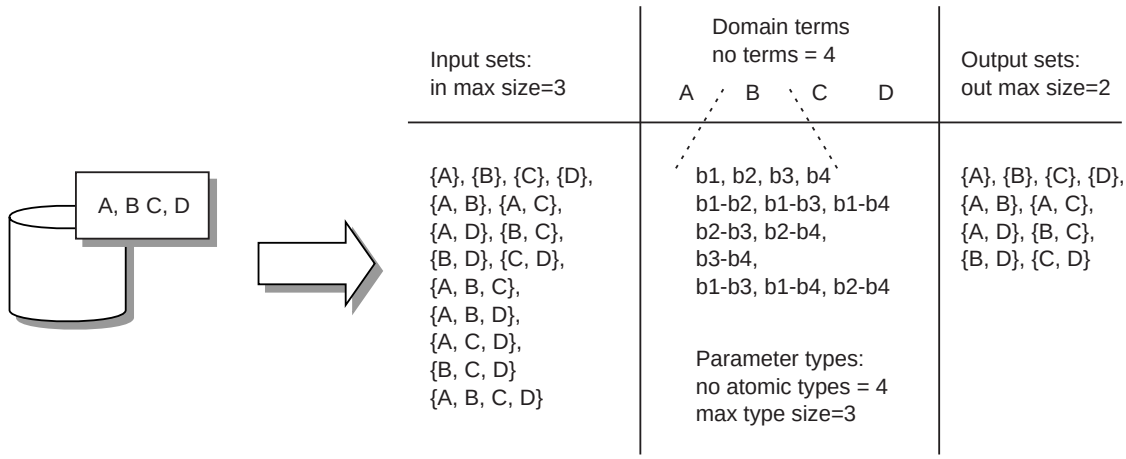


Figure 3.8: The application domain node.

Application Domain Nodes In our framework terms are a first order concept used for the specification of an application domain. Each term maps in the frame of a service description specifiying it either to an input or to an output parameter. For each term an application domain defines a set of possible data types.

For speeding up the generation process we exhaustively generate all possible input or output parameter sets by generating the power set of the domain terms. Since we consider that the number of terms in a domain will be of an order of magnitude larger that the number of parameters that a service will usually use as input or output, we will establish a *maximum size* of the parameter sets and we will we filter the initial power-set accordingly. For example in the case of a domain with terms A, B, C, D with the maximum number of parameters for a service of 2, we will have as possible parameters the sets A, B, C, D, A, B, A, C, A, D, B, C, B, D and C, D. We will filter out the sets A, B, C, A, B, D, A, C, D, B, C, D and A, B, C, D since their cardinality passes 2.

As a given service could use a given set of terms in a domain either as input parameters or as outputs parameters, we make the same differentiation regarding the sets of possible terms such that by having different *maximum sizes* for possible inputs and possible outputs, we will have a different fan-in and fan-out regarding the number of services that could make transformation from or to an application domain. For example the domain previously mentioned could have 2 as the *output maximum size* but could have 3 as the *input maximum size*, which would result in a fan-in/fan-out rapport of 3/2.

For each term in a domain we define a number of possible data types. First, for each term a number of "atomic" types is specified. We consider that the number of occurrences for each of the atomic types obeys a power-law distribution of the form $1/i^a$ where i the index of the type and a is close to the unit. Using this we compute occurrence frequencies for each of the atomic types. We then create an "zipf atomic set" of a given size in which the atomic types appear once or more, accordingly to their frequencies. Then from the "zipf atomic set" we create as above the power set of the atomic types while keeping a higher bound for the cardinality of the obtained sets. These sets will represent "composite types" obtained from the concatenation of one or more "atomic types". The "composite types" are normalized such that double occurrences of the same atomic type are discarded (e.g., $\{b1, b1, b2\} \Rightarrow \{b1, b2\}$) and that consecutive types are merged (e.g., $\{b1, b2, b3\} \Rightarrow \{b1 - b3\}$).

Service Graph Structure We generate services in our testbed as transformation between sets of terms in two application domains. For doing that for each service we first randomly pick two application domains. Then we randomly pick a parameter set from the set of input parameters of the first application domain and a parameter set from the set of output parameters of the second application domain. Then for each of the parameters in the two sets we randomly pick a parameter data-type from the respective application domains.

The main constraint that we enforce while making the choices above is to pick dif-

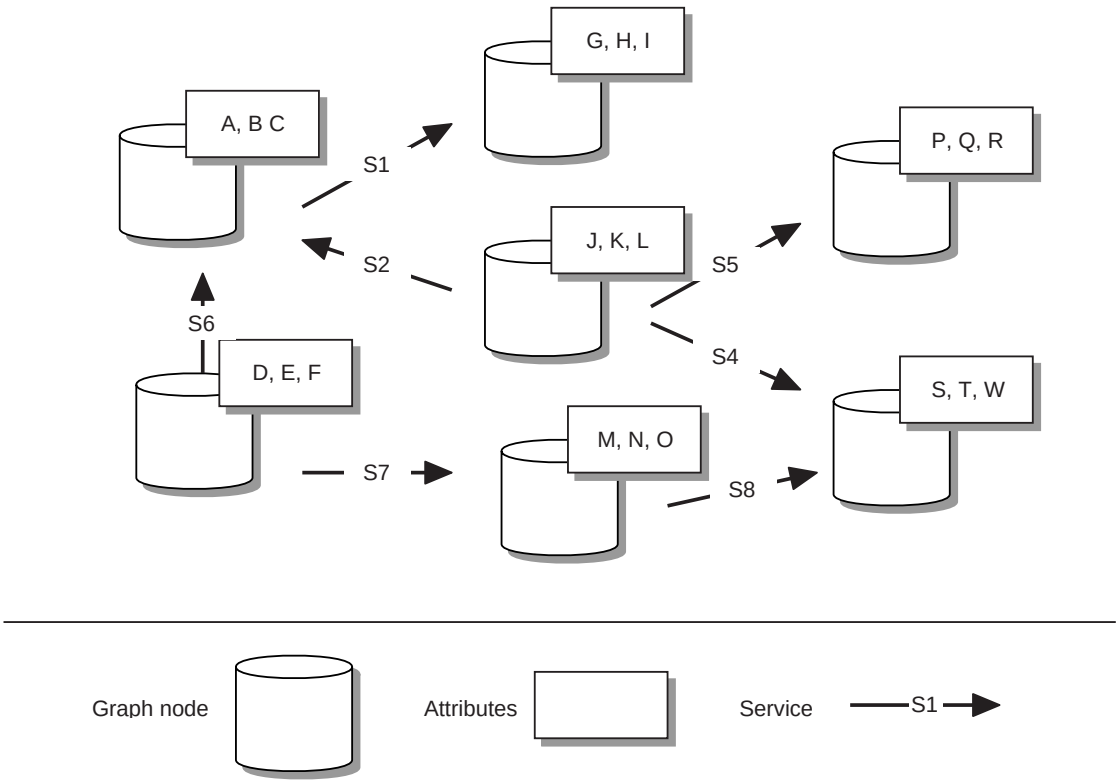


Figure 3.9: The service graph structure.

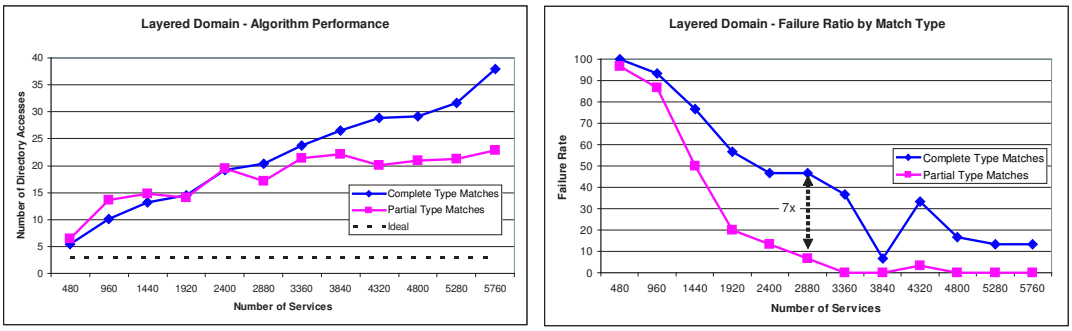
ferent application domains, as we are interested mainly in the cross-domain chaining of services. Another constraint that we enforce is the filtering of duplicates, services that have exactly the same inputs and outputs.

For generating test problems we use a similar algorithm as for creating services, by randomly picking domains, parameter sets and parameter data-types. The major difference stands in the different interpretation of input and output parameters in the case of a query (see the "Type compatible service composition" section). The parameters selected from the first domain input set are "available inputs" and the parameters selected from the second domain output set are "required outputs".

Composition Effectiveness and Performance

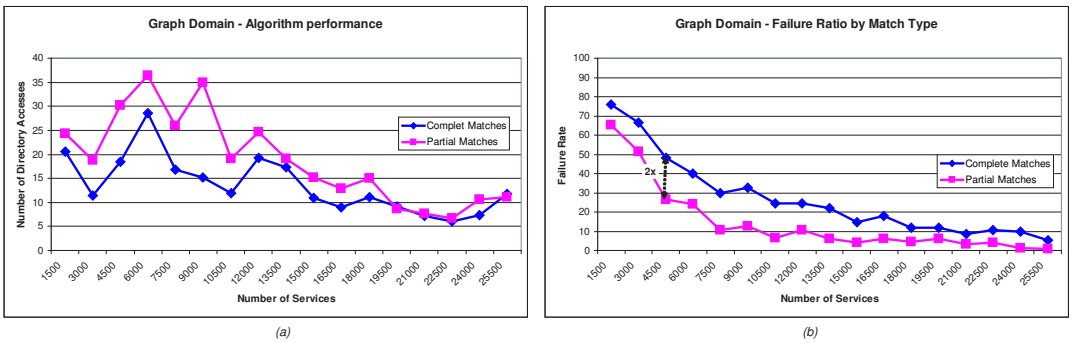
For both domains, we have randomly generated services and queries. We then solved the queries using first an algorithm that handles only *complete* type matches and then an algorithm that handles *partial* type matches (and obviously includes *complete* matches). We have measured the number of directory accesses and the failure ratio of the integration algorithms.

Fig. 3.10 (a) and Fig. 3.11 (a) show the average number of directory accesses for the



(a) (b)

Figure 3.10: The layered domain.



(a) *(b)*

Figure 3.11: The graph domain.

algorithm using *complete* type matching versus the average number of directory accesses for the algorithm also using *partial* type matching. Both algorithms scale well, as there is at most a slow increase in the number of directory accesses as the number of services in the directory grows. As it results from the experimental data for both domains the overhead induced by the usage of partial matches is not very significant and decreases as the directory gets saturated with services. This is due to the fact that having more choices makes the coverage problem intrinsic to the partial algorithm easier. More than that in the layered domain from some point the partial algorithm even performs better that the complete one (Fig. 3.10 (a) after 3000 services). This is due to the fact that the algorithm using partial matches fails less on sub-problems and as such makes better usage of already retrieved services.

The most important result concerns the number of extra problems that can be solved by using partial matches and can be seen in Fig. 3.10 (b) and Fig. 3.11 (b). The graph show that the failure rate in the case of using only *complete matches is much bigger than the failure rate when* partial matches are used: up to **7 times** in the case of the Layered domain and **2 times** in the case of the Graph domain. This shows that using partial matches opens the door for solving many problems that were unsolvable by the complete type matching algorithm.

Effects of Service Composition Heuristics

We have evaluated the impact of user-defined pruning and ranking functions on the composition performance. As we consider directory accesses to be a computationally expensive operation we use them as a measure of efficiency.

The problems have been solved using two forward chaining composition algorithms: One that handles only complete type matches and another one that can compose partially matching services, too [CFB04b]. When running the algorithms we have used two different directory configuration: The first configuration was using the extensible directory described in this deliverable which supports custom pruning and ranking functions, in particular using the forward chaining ranking function described in Fig. 3.7. In the second configuration we used a directory which is not aware of the service composition algorithm (e.g., forward complete, backward, etc.) and cannot be extended by client code. This directory implements a generic ordering heuristic by considering the number of overlapping inputs in the query and in the service, plus the number of overlapping outputs in the query and in the service.

For both directories we have used exactly the same set of service descriptions and at each iteration we have run the algorithms on exactly the same random problems. As it can be seen in Fig. 3.12, using custom pruning and ranking functions consistently improves the performance of our algorithms. In the case of complete matches the improvement is up to a factor of 5 (for a directory of 10500 services) and in the case of partial matches the improvement is of a factor of 3 (for a directory of 9000 of services).

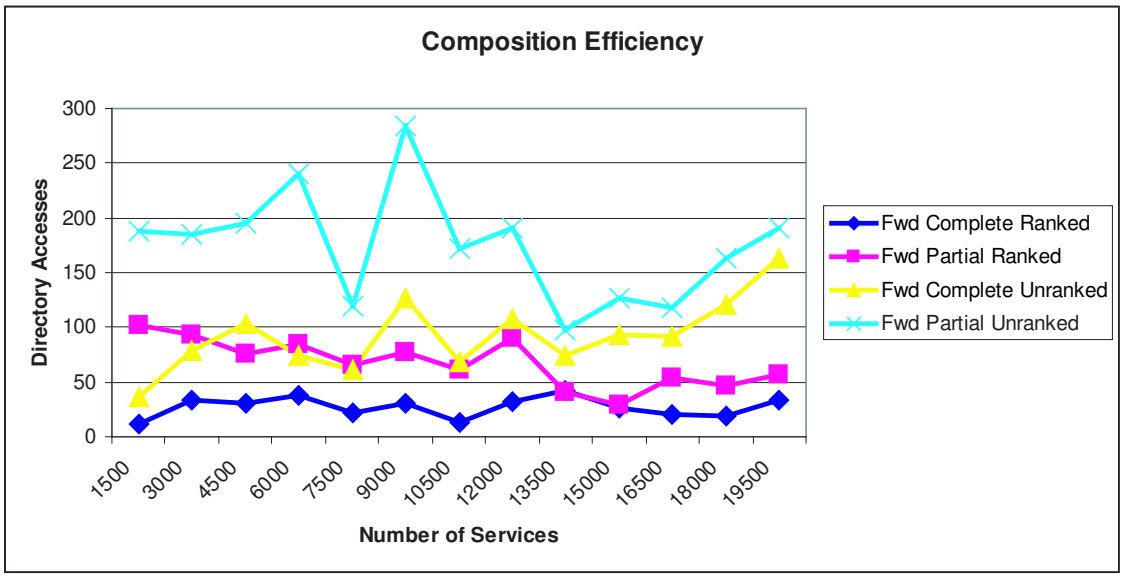


Figure 3.12: Impact of ranking functions on composition algorithms.

3.2 Process-Level Composition

Given a set of existing Web services $W_1, \ldots, W_n$, 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.

Let us consider for instance the case of the Virtual Travel Agency presented in Section 1.2, 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 and a "Hotel Booking" service that are adequate for the specific request of the customer, e.g., the specific destination.[3] 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 payment. 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

[3]The selection of such Web services can be the result of a functional level composition, as described in Section 3.1.

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.

While the details on the exact sequence of operations required to interact with an existing service are not important in discovery and functional-level composition, they become 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. 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.

"On-The-Fly" Composition Versus "Service Generation"

Two different scenarios of process-level composition are possible. In the first scenario, called **on-the-fly composition**, the composition is created in order to satisfy a specific request of the customer (e.g., a trip to a specific location in a specific period of time). In this scenario, a composition task is performed for each submitted request: every time the Virtual Travel Agency receives a customer's request, it generates and executes the composition, thus obtaining an offer that can be given back to the customer. In the second scenario, called **service generation**, the objective of the composition if to define a generic program that is able to answer to generic requests from customers. That is, instead of composing the hotel booking and the flight booking services *for a specific request*, a composition of these services is defined so that it can answer *a generic request*. The latter scenario is called "service generation" since the generated executable code implements a service provided by the Virtual Travel Agency to answer generic requests of customers, and it can deployed and executed within the Travel Agency information system.

We will consider both scenarios of process-level composition, stressing commonalities and differences. We remark here a first difference among them, namely the kind of interactions they support with the user of the composed service. In the case of on-the-fly composition, the interactions with the user are forced to be atomic: in the case of the Virtual Travel Agency, for instance, after the customer has sent a request, the agency interacts in a suitable way with hotel and flight services and sends an offer back to the customer. In the case of service generation, more general interactions with the customer can be considered. For instance, it is possible to generate a composition that, after receiving a request from the customer, asks flight and hotel services for possible offers to be combined and sent back to the customer; the customer can now inspect the offers and decide whether to accept of refuse them; the virtual travel agency can confirm of cancel the hotel and flight offers after it receives the feedback from the customer. This way, also the interaction with

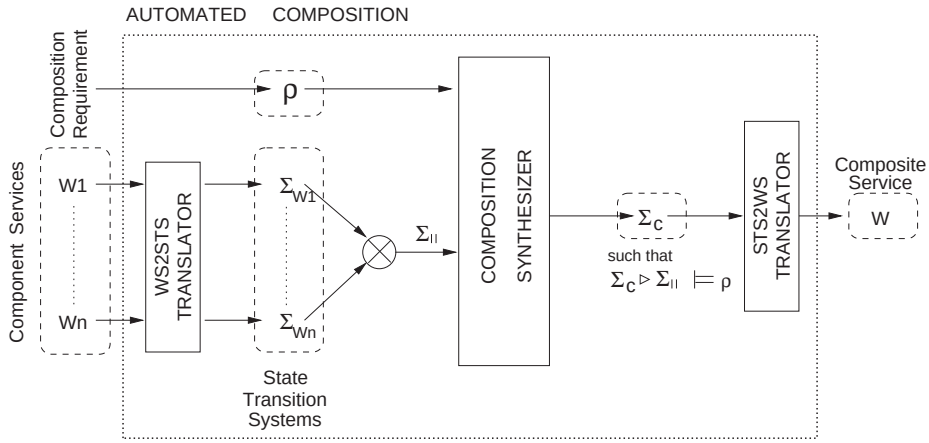


Figure 3.13: Process-Level Service Composition: An Overview

the customer becomes a complex sequence of operations, possibly with conditional and iterative behaviors. Technically speaking, the customer becomes one of the existing Web services $W_1, \ldots, W_n$ the generated executable code has to interact with in order to carry out the composition.

Automated Service Composition: An Overview

Defining the process-level composition, while usually not very complex, is a time consuming and error prone activity, since it requires an analysis of the protocols defining the existing services, and the production of a code that takes into account all possible different scenarios that can occur during the interaction with these services. For this reason, an automated support is desirable to reduce time and errors in the process-level composition of Web services. We now describe an approach to the automated composition of Web service which is able to deal with the features described above. An overview of the approach is defined in the following, while further details are described in the rest of the section.

Our goal is to automatically generate a new service W (called the *composite service*) that interacts with a set of published Web services $W_1, \ldots, W_n$ (called the component services) and satisfies a given composition requirement. More specifically (see Figure 3.13), we start from n process-level descriptions of Web services $W_1, \ldots, W_n$, the WS2STS module automatically translates each of them into a *state transition system* (STS form now on) $\Sigma_{W_1}, \ldots, \Sigma_{W_n}$. Intuitively, each Σ_{W_i} is a compact representation of all the possible behaviors, evolutions of the component service W_i. Each Σ_{W_i} is described in terms of states, input and output actions, and internal actions.

We then construct a *parallel STS* $\Sigma_{\|}$ that combines $\Sigma_{W_1}, \ldots, \Sigma_{W_n}$. Formally, this combination is a parallel product, which allows the n services to evolve concurrently. $\Sigma_{\|}$ represents therefore all the possible behaviors, evolutions of the different component services, without any control by and interaction with the composite service that will be

generated, i.e., W. The second kind of input to the automated composition consists of the requirement ρ for the composite service, which describes what is the functionality that the composed service is supposed to achieve. Given $\Sigma_{\|}$ and ρ, we automatically generate a STS Σ_c that encodes the new service W that has to be generated, which dynamically receives and sends invocations from/to the composite services $W_1, \ldots, W_n$ and behaves depending on responses received from the external services. Σ_c is such that $\Sigma_c \triangleright \Sigma_{\|}$ satisfies the requirement ρ, where $\Sigma_c \triangleright \Sigma_{\|}$ represents all the evolutions of the component services as they are controlled by the composite service. The STS Σ_c is then given in input to the STS2WS module which translates it into the concrete, executable code that implements the desired composite Web service.

3.2.1 Background and State of the Art

In this section we give an overview of existing techniques for performing a process-level composition of Web-services. We distinguish the approaches based on planning techniques (the majority) and the approaches based on automata.

Planning Techniques

There is a large number of works where different planning techniques are used for achieving automated composition of Web services, both within the semantic Web service research area and outside it (see, e.g., [BDG03, WPS+03, Der98, SdF03, MS02a, MF02, PBB+04, TP04]). In these works, existing services can be used to construct the planning domain, composition requirements can be formalized as planning goals, and planning algorithms can be used to generate plans that compose the existing services.

Most of these works, however, are limited to the case the existing Web services are atomic (i.e. they are described in terms of their inputs/outputs and of preconditions/postconditions), and hence perform a functional-level composition. We refer to Section 3.1.1 for a discussion of the state of the art in functional-level composition, and we focus here on the few approaches that consider process-level descriptions of the existing services.

In [NM02], the authors propose an approach to the simulation, verification, and automated composition of Web services based a translation of DAML-S to situation calculus and Petri Nets. However, while the framework allows for modeling complex, non-atomic Web services, the automated composition is limited to sequential composition of atomic services.

The only planning approach we are aware of which allows for an automated process-level composition of Web services is the one proposed in [PBB+04, TP04]. This approach will be described in details in Sections 3.2.2 and 3.2.3.

Automata-Based Techniques

In [HBCS03], a formal framework is defined for composing e-services from behavioral descriptions given in terms of automata. The problem addressed in this work is at the process level, since considers services that can perform more complex interactions than a simple invoke-response operation. However, the composition problem solved by [HBCS03] is different from the problem we are considering in this section. Indeed, in their approach, the composition problem is seen as the problem of coordinating the executions of a given set of available services, and not as the problem of generating a new composite Web service that interacts with the available ones. Solutions to the former problem can be used to deduce restrictions on an existing (composition automaton representing the) composed service, but not to generate executable code for implementing the composition. This is the main conceptual difference also with the work described in [BCG+03], where automated reasoning techniques, based on Description Logic, are used to address the problem of automated composition of e-services described as finite state machines.

More in general, Web service composition shares some ideas with work on the automata-based synthesis of controllers (see, e.g., [PR89, Var]). Indeed, the composite service can be seen as a module that controls an environment which consists of the component services. However, most of the work in this area focuses on the theoretical foundations, without providing practical implementations. Moreover, it is based on rather different technical assumptions on the interaction with the environment.

3.2.2 Formal Definition of the Problem

We now present a formal framework for Web service composition, taken from [PTB04]. The central concept is that of *state transition systems*, a general model for describing dynamic systems which is used for representing both the existing Web services and the executable code defining the process-level composition.

State Transition Systems

A state transition system defines a dynamic system that can be in several possible *states* (some of which are marked as *initial states*) and can evolve to new states as a result of performing some *actions*. Actions are distinguished in *input actions*, which represent the reception of messages, *output actions*, which represent messages sent to external services, and a special action τ, called *internal action*. The action τ is used to represent internal evolutions that are not visible to external services, i.e., the fact that the state of the system can evolve without producing any output, and independently from the reception of inputs. A *transition relation* describes how the state can evolve on the basis of inputs, outputs, or of the internal action τ. Finally, a *labeling function* associates to each state the set

of properties $\mathcal{P}rop$ that hold in the state. These properties will be used to define the composition requirements.

Definition 1 (State transition system (STS))
A state transition system Σ *is a tuple* $\langle \mathcal{S}, \mathcal{S}^0, \mathcal{I}, \mathcal{O}, \mathcal{R}, \mathcal{L} \rangle$ *where:*

- $\mathcal{S}$ *is the set of states;*
- $\mathcal{S}^0 \subseteq \mathcal{S}$ *is the set of initial states;*
- $\mathcal{I}$ *is the set of input actions;*
- $\mathcal{O}$ *is the set of output actions;*
- $\mathcal{R} \subseteq \mathcal{S} \times (\mathcal{I} \cup \mathcal{O} \cup \{\tau\}) \times \mathcal{S}$ *is the transition relation;*
- $\mathcal{L} : \mathcal{S} \rightarrow 2^{\mathcal{P}rop}$ *is the labeling function.*

We assume that infinite loops of τ-transitions cannot appear in the system. Indeed, an infinite τ-loop would describe a divergent behavior of the system, i.e., a behavior where the service is not interacting with the environment.

Figure 3.14 shows the STS corresponding to a very simplified Hotel Booking Web service, corresponding to a hotel chain booking system. In this simplified model, the service waits for a `request` from the user, specifying the period of the trip and the visited location. The service then checks availability of rooms. If there are no rooms available, a negative (`not_avail`) answer is sent to the user. If rooms are available, instead, the service sends an `offer` to the user, providing detailed information on the hotel and a cost for the stay. The user can now confirm (`ack`) or cancel (`nack`) the room reservation.

The set of states $\mathcal{S}$ models the steps of the evolution of the process and the values of its variables. The special variable `pc` implements a "program counter" that holds the current execution step of the service (e.g., `pc` has value `getRequest` when the process is waiting to receive a booking request, and value `checkAvailable` when it is ready to check whether the booking is possible). Other variables like `offer_hotel` or `offer_cost` correspond to those used by the process to store significant information. Finally, arrays like `AvailableHotel` or `CostOfRoom` describe predicates and functions expressing properties of the Web service (e.g., the fact that there is an hotel available for a given period and in a given location, or the cost of a room in a given hotel for a given period). In the initial states $\mathcal{S}^0$ the `pc` is set to `START`, while all the other basic variables are undefined. The initial values of the arrays `AvailableHotel` are `CostOfRoom` unspecified, since they can assume any value in the domain.

The evolution of the process is modeled through a set of possible transitions. Each transition defines its applicability conditions on the source state, its firing action, and the destination state. For instance, "`pc = checkAvailable & AvailableHotel[req_period,req_loc]` $\neq$ `UNDEF -[TAU]-> pc = isAvailable`" states that an action τ can be executed in state `checkAvailable`,

```
PROCESS
    HotelBooking;
TYPE
    Period; Location; Hotel; Cost;
STATE
    pc: { START, getRequest, checkAvailable, isAvailable, isNotAvailable,
          prepareOffer, sendOffer, waitAnswer, prepareNotAvail, sendNotAvail, SUCC, FAIL };
    req_period: Period ∪ { UNDEF };
    req_loc: Location ∪ { UNDEF };
    offer_hotel: Hotel ∪ { UNDEF };
    offer_cost: Cost ∪ { UNDEF };
    AvailableHotel[Period,Location]: Hotel ∪ { UNDEF };
    CostOfRoom[Period,Hotel]: Cost;
INIT
    pc = START;
    req_period = UNDEF;
    req_loc = UNDEF;
    offer_hotel = UNDEF;
    offer_cost = UNDEF;
INPUT
    request(p: Period, l: Location);
    ack();
    nack();
OUTPUT
    offer(h: Hotel, c: Cost);
    not_avail();
TRANS
    pc = START -[TAU]-> pc = getRequest;
    pc = getRequest -[INPUT request(p,l)]-> pc = checkAvailable, req_period = p, req_loc = l;
    pc = checkAvailable & AvailableHotel(req_period,req_loc) ≠ UNDEF -[TAU]-> pc = isAvailable;
    pc = checkAvailable & AvailableHotel(req_period,req_loc) = UNDEF -[TAU]-> pc = isNotAvailable;
    pc = isAvailable -[TAU]-> pc = prepareOffer;
    pc = prepareOffer -[TAU]-> pc = sendOffer,
                               offer_hotel = AvailableHotel(req_period,req_location),
                               offer_cost = CostOfRoom(req_period,offer_hotel);
    pc = sendOffer -[OUTPUT offer(offer_cost, offer_hotel)]-> pc = waitAnswer;
    pc = waitAsnwer -[INPUT nack]-> pc = FAIL;
    pc = waitAsnwer -[INPUT ack]-> pc = SUCC;
    pc = isNotAvailable -[TAU]-> pc = prepareNotAvail;
    pc = prepareNotAvail -[TAU]-> pc = sendNotAvail;
    pc = sendNotAvail -[OUTPUT not_avail]-> pc = FAIL;
```

Figure 3.14: The STS for the Hotel Booking process.

leading to the state `isAvailable`, if there is some hotel with available rooms for the specified period and location. We remark that each `TRANS` clause of Figure 3.14 corresponds to different elements in the transition relation $\mathcal{R}$: e.g., "`pc = checkAvailable & AvailableHotel[req_period,req_loc]` $\neq$ `UNDEF -[TAU]->` $>$ `pc = isAvailable`" generates different elements of $\mathcal{R}$, depending on the values of variables `req_period` and `req_loc`.

According to the formal model, we distinguish among three different kinds of actions. The input actions $\mathcal{I}$ model all the incoming requests to the process and the information they bring (i.e., `request` is used for the receiving of the booking request, while `ack` models the confirmation of the order and `nack` its cancellation). The output actions $\mathcal{O}$ represent the outgoing messages (i.e., `not_avail` is used when the booking is not possible, while `offer` is used to deliver an offer to the user of the service). The action τ is used to model internal evolutions of the process, as for instance assignments and decision making (e.g., when the HotelBooking process is in the state `checkAvailable` and performs internal activities to decide whether there are available rooms or not, or when, in the state `prepareOffer`, it must decide the hotel and the cost of the room).

Finally, the properties of the STS are expressions of the form `<variable> = <value>` or `<array>[idx_1,...,idx_n] = <value>`, and the labeling function is the obvious one.

The definition of STS provided in Figure 3.14 is parametric w.r.t. the types `Period`, `Location`, `Hotel`, and `Cost` used in the messages. In order to obtain a concrete STS and to apply the automated synthesis techniques described later in this paper, specific ranges have to be assigned to these types. Different approaches are possible for defining these ranges. The first, simpler approach is to associate finite (and possibly small) ranges to each type. This approach, exploited in [PTB04], makes the definition of the STS easy (and, as we will see, allows for an efficient automated composition), however, it has the disadvantage of imposing unrealistic assumptions on the data types handled by the Web services. A more realistic (but more complex) approach consists of using abstract models for the data types, avoiding an enumeration of all concrete values that these types can assume, and representing explicitly only those aspects of the data type that are relevant for the task at hand.

Controller for a Process-Level Composition

The automated composition problem has two inputs (see Figure 3.13): the formal composition requirement ρ and the parallel STS $\Sigma_{\|}$, which represents the services $\Sigma_{W_1}, \ldots, \Sigma_{W_n}$. We now formally define the *parallel product* of two STSs, which models the fact that both systems may evolve independently, and which is used to generate $\Sigma_{\|}$ from the component Web services.

Definition 2 (parallel product)
Let $\Sigma_1 = \langle \mathcal{S}_1, \mathcal{S}_1^0, \mathcal{I}_1, \mathcal{O}_1, \mathcal{R}_1, \mathcal{L}_1 \rangle$ *and* $\Sigma_2 = \langle \mathcal{S}_2, \mathcal{S}_2^0, \mathcal{I}_2, \mathcal{O}_2, \mathcal{R}_2, \mathcal{L}_2 \rangle$ *be two STSs with*

$(\mathcal{I}_1 \cup \mathcal{O}_1) \cap (\mathcal{I}_2 \cup \mathcal{O}_2) = \emptyset$. *The parallel product* $\Sigma_1 \parallel \Sigma_2$ *of* Σ_1 *and* Σ_2 *is defined as:*

$$\Sigma_1 \| \Sigma_2 = \langle \mathcal{S}_1 \times \mathcal{S}_2, \mathcal{S}_1^0 \times \mathcal{S}_2^0, \mathcal{I}_1 \cup \mathcal{I}_2, \mathcal{O}_1 \cup \mathcal{O}_2, \mathcal{R}_1 \| \mathcal{R}_2, \mathcal{L}_1 \| \mathcal{L}_2 \rangle$$

where:

- $\langle (s_1, s_2), a, (s_1', s_2) \rangle \in (\mathcal{R}_1 \| \mathcal{R}_2)$ *if* $\langle s_1, a, s_1' \rangle \in \mathcal{R}_1$*;*
- $\langle (s_1, s_2), a, (s_1, s_2') \rangle \in (\mathcal{R}_1 \| \mathcal{R}_2)$ *if* $\langle s_2, a, s_2' \rangle \in \mathcal{R}_2$*;*

and $(\mathcal{L}_1 \| \mathcal{L}_2)(s_1, s_2) = \mathcal{L}_1(s_1) \cup \mathcal{L}_2(s_2)$.

The system representing (the parallel evolutions of) the component services $W_1, \ldots, W_n$ of Figure 3.13 is formally defined as $\Sigma_{\|} = \Sigma_{W_1} \parallel \ldots \parallel \Sigma_{W_n}$.

We remark that this definition only applies to the specific case where inputs/outputs of Σ_1 and those of Σ_2 are disjoint. This is a reasonable assumption in the case of Web service composition, where the different components are independent (e.g., in the Virtual Travel Agency domain, there is no direct communication between the Hotel Booking and Flight Booking services). It is however possible to extend the approach to the more general case where Σ_1 and Σ_2 can send messages to each other (i.e., $(\mathcal{I}_1 \cup \mathcal{O}_1) \cap (\mathcal{I}_2 \cup \mathcal{O}_2) \neq \emptyset$) by modifying in a suitable way the definition of parallel product.

The automated composition problem consists in generating a STS Σ_c that controls $\Sigma_{\|}$ by satisfying ρ. We now define formally the STS describing the behaviors of a STS Σ when controlled by Σ_c.

Definition 3 (controlled system)
Let $\Sigma = \langle \mathcal{S}, \mathcal{S}^0, \mathcal{I}, \mathcal{O}, \mathcal{R}, \mathcal{L} \rangle$ *and* $\Sigma_c = \langle \mathcal{S}_c, \mathcal{S}_c^0, \mathcal{O}, \mathcal{I}, \mathcal{R}_c, \mathcal{L}_\emptyset \rangle$ *be two state transition systems, where* $\mathcal{L}_\emptyset(s_c) = \emptyset$ *for all* $s_c \in \mathcal{S}_c$*. The STS* $\Sigma_c \triangleright \Sigma$*, describing the behaviors of system* Σ *when controlled by* Σ_c*, is defined as:*

$$\Sigma_c \triangleright \Sigma = \langle \mathcal{S}_c \times \mathcal{S}, \mathcal{S}_c^0 \times \mathcal{S}^0, \mathcal{I}, \mathcal{O}, \mathcal{R}_c \triangleright \mathcal{R}, \mathcal{L} \rangle$$

where:

- $\langle (s_c, s), \tau, (s_c', s') \rangle \in (\mathcal{R}_c \triangleright \mathcal{R})$ *if* $\langle s_c, \tau, s_c' \rangle \in \mathcal{R}_c$*;*
- $\langle (s_c, s), \tau, (s_c, s') \rangle \in (\mathcal{R}_c \triangleright \mathcal{R})$ *if* $\langle s, \tau, s' \rangle \in \mathcal{R}$*;*
- $\langle (s_c, s), a, (s_c', s') \rangle \in (\mathcal{R}_c \triangleright \mathcal{R})$*, with* $a \neq \tau$*, if* $\langle s_c, a, s_c' \rangle \in \mathcal{R}_c$ *and* $\langle s, a, s' \rangle \in \mathcal{R}$*.*

Notice that we require that the inputs of Σ_c coincide with the outputs of Σ and vice-versa. Notice also that, although the systems are connected so that the output of one is associated to the input of the other, the resulting transitions in $\mathcal{R}_c \triangleright \mathcal{R}$ are labelled by input/output actions. This allows us to distinguish the transitions that correspond to τ actions of Σ_c or

Σ from those deriving from communications between Σ_c and Σ. Finally, notice that we assume that the plan has no labels associated to the states.

A STS Σ_c may not be adequate to control a system Σ. Indeed, we need to guarantee that, whenever Σ_c performs an output transition, then Σ is able to accept it, and vice-versa. We define the condition under which a state s of Σ is able to accept a message according to our asynchronous model, which abstracts away queues. We assume that s can accept a message a if there is some successor s' of s in Σ, reachable from s through a chain of τ transitions, such that s can perform an input transition labelled with a. Vice-versa, if state s has no such successor s', and message a is sent to Σ, then a deadlock situation is reached.[4]

In the following definition, and in the rest of the paper, we denote by τ-closure(s) the set of the states reachable from s through a sequence of τ transitions, and by τ-closure(S) with $S \subseteq \mathcal{S}$ the union of τ-closure(s) on all $s \in S$.

Definition 4 (deadlock-free controller)
Let $\Sigma = \langle \mathcal{S}, \mathcal{S}^0, \mathcal{I}, \mathcal{O}, \mathcal{R}, \mathcal{L} \rangle$ be a STS and $\Sigma_c = \langle \mathcal{S}_c, \mathcal{S}_c^0, \mathcal{O}, \mathcal{I}, \mathcal{R}_c, \mathcal{L}_\emptyset \rangle$ be a controller for Σ. Σ_c is said to be deadlock free *for Σ if all states $(s_c, s) \in \mathcal{S}_c \times \mathcal{S}$ that are reachable from the initial states of $\Sigma_c \triangleright \Sigma$ satisfy the following conditions:*

- *if $\langle s, a, s' \rangle \in \mathcal{R}$ with $a \in \mathcal{I}$ then there is some $s'_c \in \tau$-closure(s_c) such that $\langle s'_c, a, s''_c \rangle \in \mathcal{R}$ for some $s''_c \in \mathcal{S}_c$; and*
- *if $\langle s_c, a, s'_c \rangle \in \mathcal{R}_c$ with $a \in \mathcal{O}$ then there is some $s' \in \tau$-closure(s) such that $\langle s', a, s'' \rangle \in \mathcal{R}$ for some $s'' \in \mathcal{S}$.*

Process-Level Composition Problems

In a Web service composition problem, we need to generate a Σ_c that guarantees the satisfaction of a composition requirement ρ (see Figure 3.13). This is formalized by requiring that the controlled system $\Sigma_c \triangleright \Sigma_\|$ must satisfy ρ, which is defined in terms of the executions that $\Sigma_c \triangleright \Sigma_\|$ can perform.

In order to define formally when $\Sigma_c \triangleright \Sigma_\|$ satisfies ρ, we need to define first the executions of $\Sigma_c \triangleright \Sigma_\|$. In doing this, we need to take into account that the state transition system $\Sigma_\|$ models a domain that is only partially observable by Σ_c. That is, at execution time, the composite service Σ_c cannot in general get to know exactly what is the current state of the component services modeled by $\Sigma_{W_1}, \ldots, \Sigma_{W_n}$. Consider for instance the STS corresponding to the HotelBooking service (see Figure 3.14). The composite Virtual Travel Agency service has no access to the values of the hotel service's internal variables, and can only deduce their values from the messages exchanged with the HotelBooking

[4] We remark that, if there is such a successor s' of s, a deadlock can still occur. This can happen if a different chain of τ transitions is executed from s that leads to a state s'' for which a cannot be executed anymore. In this case, the deadlock is recognized in s''.

process. This uncertainty has two different sources. The first one is the standard source of uncertainty in planning, namely the presence of non-deterministic transitions (e.g., the two τ transitions of the HotelBooking from "`pc = checkAvailable`", which model the fact that rooms may be available or not for a given customer's request). The second source of uncertainty is due to the fact that we are modeling an asynchronous framework and that, therefore, it is not possible for the Virtual Travel Agency to know when internal τ transitions are performed in the HotelBooking service. Due to this uncertainty, after a message "`request(s,l)`" has been sent to the HotelBooking service, it is impossible to distinguish whether the booking service is still checking whether the delivery is possible (`pc = checkAvailable`), or whether this task has terminated positively (`pc` is `isAvailable`, `prepareOffer`, or `sendOffer`) or negatively (`pc` is `isNotAvailable`, `prepareNotAvail`, or `sendNotAvail`). This uncertainty disappears only when an "`offer`" or a "`not_avail`" message is received by the Virtual Travel Agency.

In the definition of the executions of $\Sigma_c \triangleright \Sigma_{\|}$ (and, more in general, of a state transition system Σ) we take into account this uncertainty by considering, at each step of the execution, a set of possible states, each equally plausible given the partial knowledge that we have of the system. Such a set of states is called a *belief state*, or simply *belief*. The initial belief for the execution is the set of initial states $\mathcal{S}^0$ of Σ. This belief is updated whenever Σ performs an observable (input or output) transition. More precisely, if $B \subseteq \mathcal{S}$ is the current belief and an action $a \in \mathcal{I} \cup \mathcal{O}$ is observed, then the new belief $B' = \mathit{Evolve}(B, a)$ is defined as follows: $s \in \mathit{Evolve}(B, a)$ if, and only if, there is some state s' reachable from B by performing a (possibly empty) sequence of τ transitions, such that $\langle s', a, s\rangle \in \mathcal{R}$. That is, in defining $\mathit{Evolve}(B, a)$ we first consider every evolution of states in B by internal transitions τ, and then, from every state reachable in this way, their evolution caused by a.

Definition 5 (belief evolution)
Let $B \subseteq \mathcal{S}$ be a belief on some state transition system Σ. We define the evolution of B under action a as the belief $B' = \mathit{Evolve}(B, a)$, where

$$\mathit{Evolve}(B, a) = \{s' : \exists s \in \tau\text{-}\mathit{closure}(B).\langle s, a, s'\rangle \in \mathcal{R}\}.$$

In the definition of goal satisfaction, we use beliefs to describe the different "configurations" reached during execution. In order to characterize goal satisfaction, we need to define when a belief B satisfies a given state property p. In planning under partial observability, B is said to satisfy p simply if all states $s \in B$ satisfy p. The definition becomes more complex in our asynchronous setting, due to the presence of τ transitions. Let us consider again the HotelBooking service. After a request message has been sent to the service, it is not yet possible to predict whether the booking can be fulfilled or not; therefore, we expect that conditions `pc = prepareOffer` and `pc = notAvailable` are both false in B. On the other hand, after an acknowledge message has been received by the booking service, it is unavoidable to reach a successful state; therefore, we want to

be able to conclude that condition `pc = SUCC` is true in the corresponding belief. Informally, we are assuming that the execution of τ transitions cannot be postponed forever. Therefore, if the execution of τ transitions is guaranteed to reach a state satisfying a given condition, then we can assume that the condition holds also in belief state B. Conversely, if there is some sequence of τ transitions that does not contain states satisfying p and that cannot be further extended with other τ transitions so that p is reached, then the property p is not satisfied in B.

Definition 6 (belief satisfying a property)
Let $\Sigma = \langle \mathcal{S}, \mathcal{S}^0, \mathcal{I}, \mathcal{O}, \mathcal{R}, \mathcal{L} \rangle$ be a STS, $p \in \mathcal{P}rop$ be a property for Σ, and $B \subseteq \mathcal{S}$ be a belief. We say that B satisfies p, written $B \models_\Sigma p$, if the following condition holds. Let $s_0, s_1, \ldots, s_n$ be such that $s_0 \in B$, $\langle s_i, \tau, s_{i+1} \rangle \in \mathcal{R}$ and either s_n has no outgoing transitions or there exists s_{n+1} such that $\langle s_n, a, s_{n+1} \rangle$ with $a \neq \tau$. Then $p \in \mathcal{L}(s_i)$ for some $0 \leq i \leq n$.

We are now ready to define the STS that defines the executions of $\Sigma_c \triangleright \Sigma_\parallel$ and, more in general, of a STS Σ. We call it "belief-level" STS, since its states are beliefs of Σ and its transitions describe belief evolutions.

Definition 7 (belief-level system)
Let $\Sigma = \langle \mathcal{S}, \mathcal{S}^0, \mathcal{I}, \mathcal{O}, \mathcal{R}, \mathcal{L} \rangle$ be a STS. The corresponding belief-level STS is $\Sigma_\mathcal{B} = \langle \mathcal{S}_\mathcal{B}, \mathcal{S}^0_\mathcal{B}, \mathcal{I}, \mathcal{O}, \mathcal{R}_\mathcal{B}, \mathcal{L}_\mathcal{B} \rangle$, where:

- *$\mathcal{S}_\mathcal{B}$ are the beliefs of Σ reachable from the initial belief $\mathcal{S}^0$;*
- *$\mathcal{S}^0_\mathcal{B} = \{\mathcal{S}^0\}$;*
- *transitions $\mathcal{R}_\mathcal{B}$ are defined as follows: if Evolve$(B, a) = B' \neq \emptyset$ for some $a \in \mathcal{I} \cup \mathcal{O}$, then $\langle B, a, B' \rangle \in \mathcal{R}_\mathcal{B}$;*
- *$\mathcal{L}_\mathcal{B}(B) = \{p \in \mathcal{P}rop : B \models_\Sigma p\}$.*

We remark that a belief-level STS is a very restricted case of STS, since it only has one initial state, there are no τ transitions, and, for all beliefs B and actions a there is at most one belief B' such that $\langle B, a, B' \rangle \in \mathcal{R}_\mathcal{B}$. For these reasons, it is straightforward to reinterpret on $\Sigma_\mathcal{B}$ the definitions of requirements satisfaction proposed in the literature for a large set of requirements specification languages, such as temporal logics like CTL or LTL ([Eme90]) or goal languages for planning problems (e.g., strong and strong cyclic reachability goals [CPRT03], or temporally extended goals like EAGLE [DLPT02]). In the following, we write $\Sigma_\mathcal{B} \models \rho$ whenever the belief-level STS $\Sigma_\mathcal{B}$ satisfies requirement ρ.

We can now characterize formally a (process-level) composition problem.

Definition 8 (composition problem)
Let $\Sigma_1, \ldots, \Sigma_n$ be a set of state transition systems, and let ρ be a composition requirement.

The composition problem for $\Sigma_1, \ldots, \Sigma_n$ *and* ρ *is the problem of finding a controller* Σ_c *that is deadlock-free and such that* $\Sigma_{\mathcal{B}} \models \rho$ *where* $\Sigma_{\mathcal{B}}$ *is the belief-level STS of* $\Sigma_c \triangleright (\Sigma_1 \parallel \ldots \parallel \Sigma_n)$.

3.2.3 Automated Process-Level Service Composition

We now address the problem of automatically generating a process-level Web service composition. According to Figure 3.13, we distinguish four separated steps:

1. **From Web Services to State Transition Systems.** This step consists of acquiring the process-level descriptions of the existing Web services that are composed, and of translating the description of each of such service into a State Transition System.

2. **Expressing Composition Requirements.** This step concerns the definition of the requirement ρ that defines the functionality that the composed Web service should provide.

3. **Synthesis of the Composition.** During this step, the State Transition System implementing the composition of the Web services is automatically generated starting from the STS built is step 1 and from the requirement defined in step 2.

4. **Deployment and Execution of the Composed Service.** In this step, the STS generated in step 3 is translated into executable code.

In the following, we detail the four steps described above.

From Web Services to State Transition Systems

This phase is performed separately for each existing Web service participating to the composition. It can be seen as the combination of two steps: the acquisition of the process level description of the Web service; and the translation of this description into a state transition system.

We do not describe the technical details on the acquisition of the process level description of a Web service, since they are strongly technology and language dependent. We remark that, since such a description is necessary for allowing the interactions with the Web service, each technology will have to provide a mechanism for publishing this process level description. While a natural language description of the protocol that has to be followed for interacting with a Web service is sufficient for hand-written composition, it is clear that a formal description of such protocol is necessary for supporting this composition with automatic tools. Indeed, languages providing a process-level description of Web services and supporting the automated composition have been proposed by the main the Semantic Web Service approaches, see for instance the *OWL-S process models*

or the *WSMO interface* model of Web services. The necessity of process-level description languages is also recognized outside the Semantic Web Service community. For instance, the BPEL4WS language, an industrial standard language for Web service modeling and execution, allows for the publication of the so-called *abstract* description of a Web service process, i.e., the description of the sequence of interactions that are necessary in order to perform a long term interaction with a given service provider.

Once the process level description of an existing Web service is obtained, this description is translated into State Transition Systems. This translation depends on the specific process-level specification language. Specific translations have been defined in [TP04] for the case of OWL-S process models and in [PBB+04] for the case of process-level descriptions expressed in BPEL4WS. Translations from other languages are easy to define, due to the generality and expressiveness of State Transition Systems.

Expressing Composition Requirements

This is a critical step, since it is responsibility of the composition requirement to define precisely what are the expected behaviors of the composition. In the introductory part of Section 3.2 we distinguished two different scenarios of application for process-level composition, namely an on-the-fly scenario and a service generation scenario. These two scenarios require different kinds of composition requirements, so we consider them separately.

In the case of the **on-the-fly composition** of Web services, we have to generate executable code that satisfies a specific customer's request. In the case of the Virtual Travel Agency, for instance, the customer's request specifies a given location l to be visited in a given period p. In order to satisfy this request, the travel agency has to find flight and hotel room compatible with the request of the customer. The composition requirement could be the something like:

> if a travel offer is possible,
> then sell a travel offer to the customer.

In our framework, the offer is possible if it is possible to book flight and/or hotel for the period and for the location specified by the customer. The fact that the offer is sold is described by requiring that the HotelBooking and the FlightBooking processes reach the success state. The goal condition can hence be specified by the formula:

```
   HotelBooking.AvailableHotel[p,l] ≠ UNDEF &
   FlightBooking.AvailableFlight[p,l] ≠ UNDEF
→  HotelBooking.pc = SUCC & FlightBooking.pc = SUCC &
   h = HotelBooking.AvailableHotel[p,l] &
   f = FlightBooking.AvailableFlight[p,l] &
   c = HotelBooking.CostOfRoom[p,h] +
         FlightBooking.CostOfFlight[f]
```

where c, f, and h describe, respectively, the cost of the offer, and the specific flight and hotel information. This requirement has to be interpreted as a condition that must hold at the end of the execution of the code implementing the composition.

The case of the **service generation** is more complex. Consider the following example, where we want to automatically generate the composite service implementing the Virtual Travel Agency. The goal of the travel agency is to "sell a travel offer to the customer". This means we want the Virtual Travel Agency service to reach the situation where an offer has been made to the customer, the customer has confirmed this offer, and the service has confirmed the corresponding (sub-)offers to the HotelBooking and FlightBooking services. However, the hotel may have no available rooms, the flight may not be possible, the user may not accept the offer due to its cost... We cannot avoid these situations, and we therefore cannot ask the composite service to guarantee this requirement. Nevertheless, we would like the Virtual Travel Agency service to try (do whatever is possible) to satisfy it. Moreover, in the case the "sell a travel offer to the customer" requirement is not satisfied, we would like that the Virtual Travel Agency does not commit to an order for a room or for a flight, since we do not want the service to book and pay rooms and flights that will not be accepted by the customer. Let us call this requirement "never a single commit". Our global requirement would therefore be something like:

> try to "sell a travel offer to the customer";
> upon failure,
> do "never a single commit".

Notice that the secondary requirement ("never a single commit") has a different strength w.r.t. the primary one ("sell a travel offer to the customer"). We write "do" satisfy, rather than "try" to satisfy. Indeed, in the case the primary requirement is not satisfied, we want the secondary requirement to be guaranteed.

We need a formal language that can express requirements as those of the previous example, including conditions of different strengths (like "try" and "do"), and preferences among different (e.g., primary and secondary) requirements. For this reason, we cannot simply use a state formula as we did for the case of on-the-fly composition. We use instead the EAGLE language, which has been designed with the purpose to satisfy such expressiveness. A detailed definition and a formal semantics for the EAGLE language can be found in [DLPT02]. Here we just explain how EAGLE can express the composition requirement of the running example.

The EAGLE formalization of the requirement is the following:

TryReach

```
HotelBooking.pc = SUCC & FlightBooking.pc = SUCC &
Customer.pc = SUCC &
Customer.h = HotelBooking.AvailableHotel[Customer.p,Customer.l] &
Customer.f = FlightBooking.AvailableFlight[Customer.p,Customer.l] &
Customer.c = HotelBooking.CostOfRoom[Customer.p,Customer.h] +
```

```
        FlightBooking.CostOfFlight[Customer.f]
Fail DoReach
   HotehlBooking.pc = FAIL & FlightBooking.pc = FAIL &
   Customer.pc = FAIL
```

The goal is of the form "**TryReach** c **Fail DoReach** d". **TryReach** c requires a service that tries to reach condition c, in our case the condition "sell a travel offer to the customer". During the execution of the service, a state may be reached from which it is not possible to reach c, e.g., since the flight is not available. When such a state is reached, the requirement **TryReach** c fails and the recovery condition **DoReach** d, in our case "never a single commit" is considered.

Synthesis of the Composition

This is the core step of the automated composition: starting from a set of STS modeling the existing Web services and from a composition requirement, a new STS is generated which implements the composed service. (A formal definition of the functionality performed in this step is given in Definition 8.) The approach adopted in [PBB+04, TP04, PTB04] for achieving this automated synthesis is based on the "planning as model checking" [BCPT03, BCP+01] framework.

As discussed in Section 3.2.1, there is a large number of works where different planning techniques are used for achieving automated composition of Web services. Most of these works, however, are limited to the case the existing Web services are atomic, i.e., they perform functional-level composition. This limitation is also enforced by the fact that the planning techniques exploited in those approaches are not able to deal with the advanced features required for performing process-level composition.

More precisely, a planning technique capable of performing process-level composition should provide ways of dealing with the following difficulties.

- **Nondeterminism**: The planner cannot foresee the actual interaction that will take place with external processes, e.g., it cannot predict a priori whether the answer to a request for availability will be positive or negative, whether a user will confirm or not acceptance of a service, etc.

- **Partial Observability**: The planner can only observe the communications with external processes; that is, it has no access to their internal status and variables. For instance, the planner cannot know a priori the list of items available for selling from a service.

- **Extended Goals**: As we have discussed previously, composition requirements often involve complex conditions on the behavior of the process, and not only on its final state. For instance, we might require that the process never gets to the state

where it buys an item costing more than the available budget. Moreover, requirements need to express conditional preferences on different goals to achieve. For instance, a process should try first to reserve and confirm both a flight and an hotel from two different service providers, and only if one of the two services is not available, it should fall back and cancel both reservations.

We remark that, while the third problem only occurs if one is interested in "service generation", the first two problems are general of process-level composition and also occur in the case of "on-the-fly" composition.

We address these problems by exploiting planning techniques based on the "planning as model checking" approach, which has been devised to deal with nondeterministic domains, partial observability, and extended goals, and is hence suitable for process-level composition of Web services.

- The "planning as model checking" framework exploits a family of nondeterministic transition systems for modeling the planning domain as well as the generated plans. The STS $\Sigma_{\|}$ used to described the "composition" domain can be translated into a planning domain modeled as nondeterministic transition systems and, vice-versa, a converse mapping can be defined mapping the generated plan into an STS defining the composed service.

- The "planning as model checking" framework deals with partial observability in the planning domain by performing a transformation of the nondeterministic transition system defining the domain into a "belief-level" nondeterministic transition system. This transformation is similar to the one described in Definition 7, and can hence be applied to Web service STS as well.

- The "planning as model checking" framework supports different kinds of planning goals, including classical goals consisting of conditions that must hold at the end of the plan execution and temporally extended goals defined in the EaGLe language [DLPT02]. In the case of Web service composition, the former kind of goal is used for "on-the-fly" composition while the latter kind is used for "service generation".

- Finally, the "planning as model checking" framework has shown to offer a very efficient approach for plan generation, since it takes advantage of BDD-based symbolic mechanisms originally developed for solving model checking problems [BCM$^+$92]. These symbolic mechanisms allow for a compact representation and for an efficient exploration of the search space in nondeterministic and partially observable planning domains. They permit an efficient generation also when applied to the synthesis of composed Web services.

A detailed description of the exact correspondence between service composition and planning problems, and the proof of correctness of adopting planning techniques for solving service composition problems, can be found in [PTB04].

Deployment and Execution of the Composed Service

In this step, the STS which defined the composed service is transformed into executable code. In the case we are interested in an on-the-fly composition, the code is immediately executed. In the case of service generation, the code is deployed, and is ready to answer to users' requests.

Different languages can be used as target languages of this translation. One possibility is to exploit generic languages like Java for implementing the control constructs of the plans and Web service calls for implementing the interactions with the component Web services.

A more interesting approach consists in generating BPEL4WS code (see [PBB+04, PTB04]). BPEL4WS is a language designed for describing and deploying executable Web services that perform complex tasks through long term interactions with one of more external services. It is hence a natural target language in the case of a "service generation" composition. The translation of an STS into BPEL4WS is easy, since BPEL4WS provides the standard control flow constructs (if-them-else, switch, while...) and special constructs for interacting with external Web services.

Finally, the generated STS can be translated into semantic languages such as OWL-S or WSMO. In the case of OWL-S, in particular, a *process model* can be used to define the sequence of operations that defined the composed service. We remark, however, that this translation is possible only in the case of "on-the-fly" composition. Indeed, the OWL-S process model allows for defining a suitable combination of invocations of atomic operations provided by the component Web services (e.g., the Hotel Booking and the Flight Booking service). However, it does not allow for intermixing these invocations with interactions with the "user" (i.e., the Customer of the Virtual Travel Agency), since OWL-S does not allow to model a process that is both invoked by and invoker of external services.

3.2.4 Evaluation and Assessment

In order to test the performance of the automated composition techniques described in the previous section, we have conducted some experiments.[5] We have considered two different classes of experiments, which we called synthetic domains and real domains. The former are artificial domains, and are used to test the performance of automated composition on scalable problems. The latter describe realistic scenarios for automated composition and are used to validate the results of the experiments on synthetic domains.

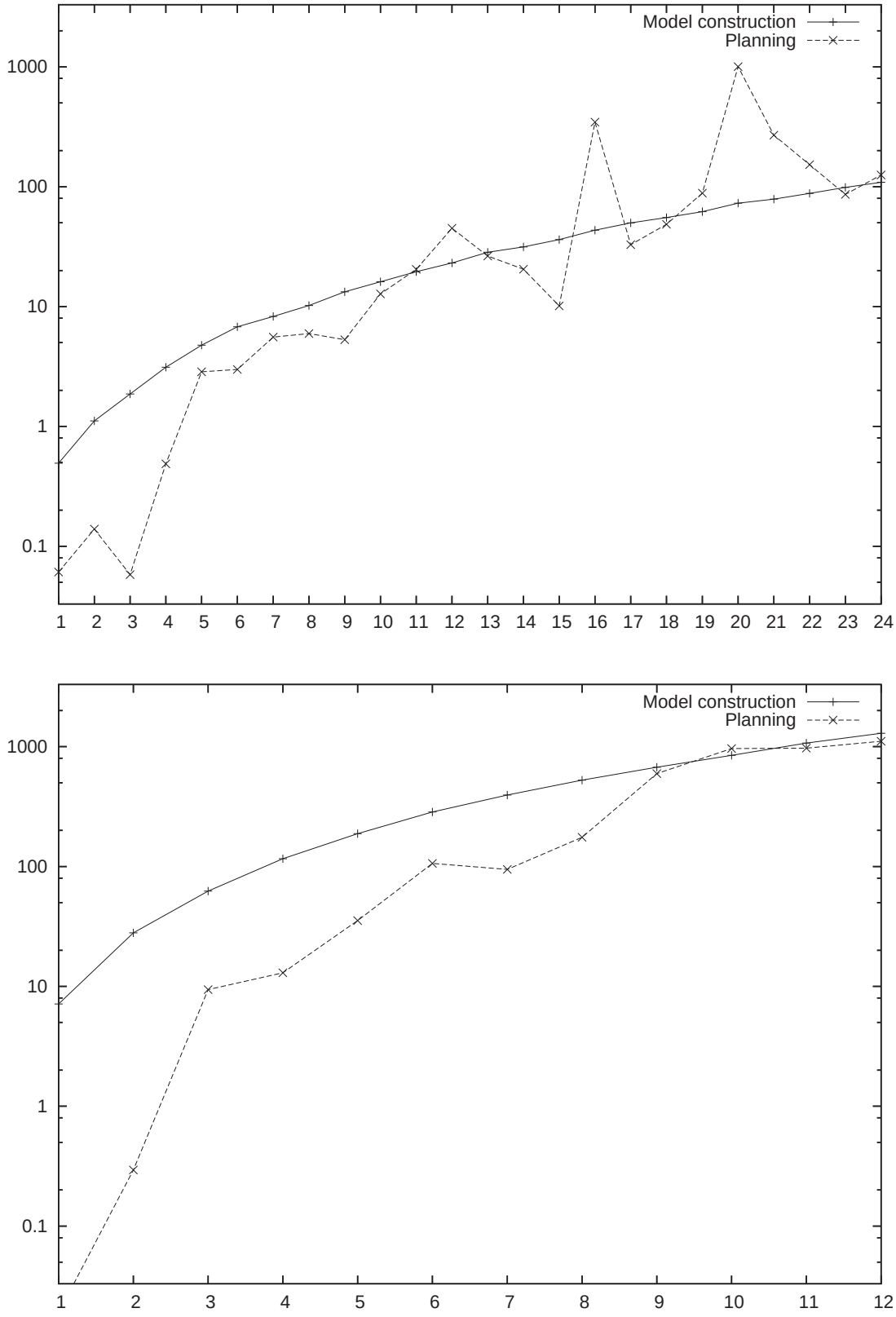


Figure 3.15: Experiments with parametrized domains

Synthetic domains

We consider two different sets of synthetic domains. In the first set of experiments, we test automated synthesis w.r.t. the number of services to be composed. Each component is represented by a very simple process that is requested to provide a service and can respond either positively or negatively. The composition requirement is also very simple: either all services end successfully or a failure is reported to the invoker of the composed service. The results are shown in the left side of Figure 3.15. In the horizontal axis we have the number of components. In the vertical axis, we report in seconds the time for *model construction* (i.e., the time to build the parallel state transition system $\Sigma_{\|}$ in Figure 3.13 starting from the component Web services) and the time for automated *composition* (i.e., the time spent to generate the composite STS Σ_c and to emit is as Web service).

As expected, the time for model construction increases regularly with the number of components. Also the composition time increases with a similar trend, but less regularly. This depends on the BDD-based symbolic mechanisms adopted in the synthesis of the composition, which are responsible of constructing a compact internal representations of the search space for the composition phase. This internal representation can be more of less efficient, depending on the specific problem instance, with strong impacts on the performance of the composition. With these examples, the time required for the automated synthesis increases less than exponentially[6] and manages to deal with a rather high number of components in a rather short time. The case with the worst performance among the considered experiments is that of 20 components, where model construction takes about 70 seconds, while automated composition takes about 1000 seconds.

We remark that the component Web services used in the previous experiment are very elementary, as they implement essentially an invoke-response protocol. In the second set of synthetic experiments, we have considered the case of a composition of Web services which require a complex interaction. More precisely, we have complicated the parameterized domain by imposing a composition that requires a high degree of interleaving between components. Here, the interactions with each component are more complex than a single invoke-response step, and, to achieve the composition, it is necessary to carry out interactions with all components in an interleaved way. Such interleaving is common in the Virtual Travel Agency example where, e.g., the an offer can be sent to the customer only after both the Hotel Booking and the Flight Booking services are confirmed that there are available places, and the flight and hotel bookings are confirmed only if the customer accepts the Travel Agency offer.

As shown in the graph on the right side of Figure 3.15, automated synthesis in this case is more difficult than in the previous set of experiments. While in the previous experiment model construction and automated composition with 12 components took, respectively,

[5] All experiments have been done over a 1.8 GHz Pentium machine, equipped with 1 GByte memory, and running a Linux 2.6.7 operating system.

[6] A precise analysis of the times shows that they grow proportionally to n^4, where n is the number of components.

25 and 45 seconds, now they take both about 1200 seconds. In spite of the fact that the required interleaving reduces performances, the technique still manages to deal with a rather large number of components, since we expect that realistic compositions will not include more than 12 components.

Real domains

To validate the results of the experimental evaluation on the synthetic domains, we have also conducted some experiments of automated composition on problems extracted from realistic Web service domains. The results are reported in Figure 3.16. The first two cases correspond to a furniture purchase and shipping (P&S) domain. It combines two separate, independent, and existing services, a furniture PRODUCER and a delivery service (SHIPPER), so that the user may directly ask the composed service P&S to purchase and deliver a given item at a given place. In the first variant of the P&S domain, automated composition is very fast, since in spite of the interleaving required, we have just three components (shipper, producer, and user of the P&S). We have then experimented with a more complex version of P&S. We have added a further service, taking into account that, in realistic cases, the composite service may require the payment to be dealt by a third party, i.e., a Bank, that is delegated to receiving the money from the client. In this example, the interleaving of interactions is increased by the necessity of receiving a payment confirmation from the Bank before the order can be confirmed to Producer and Shipper. The experimental result confirms that the problem is more difficult than in the previous P&S example, and automated composition time increases of one order of magnitude.

	number of components	model construction	composition
P&S	3	8.4 sec.	1.0 sec.
P&S + BANK	4	39.6 sec.	35.4 sec.
WMO1	5	187.5 sec.	31.6 sec.
WMO2	5	173.1 sec.	48.6 sec.
WMO3	5	174.9 sec.	120.6 sec.

Figure 3.16: Experiments with different applications.

Finally, we experiment with a case study taken from a real e-government application we are developing for a private company. We aim at providing a service that implements a (public) waste management office (WMO), i.e. a service that manages user requests to open sites for the disposal of dangerous waste. According to the existing Italian laws, such a request involves the interaction of different offices of the public administration. In particular, a Protocol Office is in charge of providing a unique identifier (the 'protocol') to the request; a Technical Committee must be invoked to produce an evaluation regarding the technical feasibility and ecological impact of the site, and a Province Board is

responsible to take a final decision. In the application we consider also a Citizen Service that interacts with the citizen or the company that makes the request, and a Secretary Service that interacts with a secretariat office. The composition requirement here can be described with a set of constraints on the order of execution of different procedural steps performed by different offices. We consider three variants of this domain, corresponding to an increasing interleaving between the different services, getting in all cases a very good performance.

In all the realistic examples, automated composition has shown to be feasible and take a rather low amount of time, surely much faster than manual development of the composite processes. Moreover, the times required for the composition confirm the trends of the experiments reported in Figure 3.15.

Chapter 4

Integration of Discovery and Composition

In this chapter we discuss possible approaches for combining service discovery, functional-level service composition, and process-level service composition.

The organic composition of these functionalities within an integrated process is outside the scope of this document (it is actually one of the objectives of the KnowledgeWeb work-plan for the next months). For this reason, we give here only a preliminary description of possible ways for combining these functionalities, and we postpone their detailed analysis to further deliverables.

The rest of the chapter is organized in several sections, each analyzing a different scenario of possible combination of discovery and composition.

4.1 Discovery within Composition

This scenario corresponds to the invocation of discovery functionalities within a composition task. This master-slave integration of the two functionalities is already adopted by different functional-level composition approaches. Notably it is adopted by the approach described in Section 3.1.3, where a discovery mediator is exploited to find existing services that match a new generated (partial) composition goal (see for instance the architecture of the service integration engine in Figure 3.2).

Such an integration of discovery within composition is less natural in the case of process-level composition. Indeed, the assumption in this case is that the component services are already known when the composition starts. On the other hand, we can assume that the process-level service that are composed are “generic”, in the sense that they describe the operations needed to interact with a generic service providing a given functionality (e.g., a generic Hotel Booking service). Once the process-level composition

has been achieved, discovery functionalities are used to find "concrete" instances of these processes (i.e., the Hotel Booking service for a specific hotel chain). This is particularly interesting in the case of the "service generation" case of process-level composition: in this case, it is reasonable to foresee that different concrete implementations for the generic services exploited in the composition need to be discovered for each of the different customer requests (e.g., different flight booking services need to be exploited depending on whether the customer travels in Europe or outside Europe).

4.2 Incremental approach

In this scenario, discovery, function-level composition, and process-level composition are applied in sequence, in order to find solutions of different complexity for a "discovery" query. More precisely, given a query defining a service, the following steps are performed:

1. first, we look for a single, atomic Web service that matches the query (service discovery);
2. if no suitable service is found in step 1, then we search for a composition of atomic Web services matching the query (functional-level composition);
3. if no suitable composition is found in step 2, then we search for a process-level composition of Web services matching the query (process-level composition).

We remark that the composition in the third step is more general than the one in the second step, since it allows for a general interleaving among the composed services. For instance, if we compose a Hotel Booking service and a Flight Booking service, we can interleave the interactions with the two services, what we cannot do in the case of functional-level composition.

An important advantage of this scenario is that it permits a black-box integration of the three functionalities. The only constraint is that a common query (or composition goal) has to be used for the three functionalities.

4.3 Iterative Approach

The goal of this scenario is to incrementally refine a service composition. Initially, the composition is defined as a single abstract service that corresponds to the composition goal. During the incremental composition, the partially developed composition will include concrete services as well as abstract services that represent goals for further refinements. During each refinement step, an abstract service of the partially developed composition is selected and further refined. The incremental composition terminates when the generated process contains no abstract service.

Different forms of refinement steps are possible during the incremental generation of the composition:

- the abstract service is replaced by a single concrete service (complete matching);
- the abstract service is replaced by a concrete service and by a new abstract service describing the "remaining" of the original abstract process (complete type matching);
- the abstract service is replaced by a switch, where each branch consists of a concrete service and a "remaining" abstract service (switch matching);
- a process-level refinement is done, i.e. the abstract process is refined into a general composition of other (abstract and concrete) Web services.

Different strategies can be adopted for choosing the transformation to be performed during a refinement step, and suitable heuristics can be exploited to guide this process.

Incremental composition can be seen as a generalization of the algorithm for composition with partial type matching described in Figure 3.3, which already includes the first three kinds of refinement steps.

We remark that this scenario can also be seen as a generalization of the incremental approach defined previously. In this case, however, the different discovery and composition functionalities are not combined black-box, but are integrated in a single composition algorithm.

4.4 Two-level Composition

In this scenario, the two kinds of composition are executed in sequence, in order to define the composition at the two levels of abstraction. Functional-level composition is performed first, in order to identify a set of Web services that have to be composed, and to obtain a high-level description of their composition. Then, process-level composition is performed in order to refine this high-level description of the composed service into a low-level, executable composition.

In this scenario, a single composition goal has to be specified, namely the goal for the functional-level composition. Indeed, the high-level specification obtained after the first step is then used as "goal" for the process-level composition. A problem that has to be solved for implementing this scenario is to understand how to use the high-level composed service to guide the low level composition. Another challenge is to understand how to pass information back to high-level composition from low-level composition whenever the latter discovers that a given high-level composition cannot be refined into an executable composition and a modification of the high-level composition has to be triggered.

4.5 Process-level discovery

This scenario is different from the previous ones as it corresponds to a different kind of "integration" between discovery and composition. The discovery described in Chapter 2 can be considered at the functional level. Indeed, Web services to be discovered are described in terms of their inputs, outputs, preconditions, and effects. Similarly to what happens for composition, we can think to a lower-level type of discovery, namely process-level discovery. In this case, in the query we do not specify only the functional parameter of a service to be discovered, but we also specify conditions on the whole protocol that need to be followed to interact with the service.

Process-level discovery is useful to express in the search query conditions on the interaction flow with the services (e.g., we can specify that we want a Hotel Booking service that allows us to cancel our reservation in any moment; or a service that requires our credit card number only if we are interested to accept an offer). Process-level discovery can also be used to refine and restrict the set of services returned by a functional-level discovery step.

Chapter 5

Conclusions

The description of Web services in a machine-understandable fashion 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, both within and across enterprise boundaries, is the mechanization of service discovery. Automatically locating and contracting available services to perform a given business activity can considerably reduce the cost of integration and can enable a much more flexible integration, where providers are dynamically selected based on what they provide and possibly other non-functional properties such as trust, security, etc.

A practical and realistic approach to service discovery has to be based on a conceptual model that considers the characteristics of the domain of application, and that make realistic assumptions. We have presented such a model and analyzed how current proposals for service discovery and software component retrieval relate to it. It turns out that they only cover partial aspects of it. For this reason, we have proposed approaches for service discovery and composition and formalized the notions of match involved, identifying the required reasoning support.

Service composition allows to integrate existing services into more complex added-value services, which can be tailored to the particular needs of a user in a certain situation. In this document we have focused on automated service composition, which we addressed at different abstraction levels. From a high-level point of view, we have developed techniques for functional service composition, which interacts with the service discovery to dynamically retrieve relevant service descriptions. In order to obtain an executable composed service, it is necessary to follow the protocols of the different services involved. We propose process-level service composition based on model checking techniques for this purpose.

5.1 Future work

Future work will focus on solving the open issues described in Section 2.8, especially the interaction of abstract and contracting capabilities of services and the required languages, and the evaluation of candidate reasoners. This will eventually lead to the implementation of a discovery engine based on this conceptual model. This work has already started (see [L*editors*04b]).

A second major line in our future work is to integrate the planned discovery engine with the approaches for composition presented in this document. For that purpose, we will further elaborate our first ideas presented in Chapter 4 and provide an integrated approach to automatically discover, contract and compose available services to fulfill a given goal.

Bibliography

[ACD+03] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana. Business Process Execution Language for Web Services Version 1.1. http://www.ibm.com/developerworks/library/ws-bpel/, May 2003.

[ACKM03] G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services*. Springer, 2003.

[Ame91] P. America. *Designing an object-oriented programming language with behavioural subtyping*, volume 489 of *Lecture Notes in Computer Science*, pages 60–90. Springer-Verlag, 1991.

[BCE+02] T. Bellwood, L. Clement, D. Ehnebuske, A. Hately, Maryann Hondo, Y.L. Husband, K. Januszewski, S. Lee, B. McKee, J. Munter, and C. von Riegen. UDDI version 3.0. http://uddi.org/pubs/uddi-v3.00-published-20020719.htm, July 2002.

[BCF04] Walter Binder, Ion Constantinescu, and Boi Faltings. A directory for web service integration supporting custom query pruning and ranking. In *European Conference on Web Services (ECOWS 2004)*, Erfurt, Germany, September 2004.

[BCG+03] D. Berardi, D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Mecella. Automatic composition of E-Services that export their behaviour. In *Proc. ICSOC'03*, 2003.

[BCM+92] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. *Information and Computation*, 98(2):142–170, June 1992.

[BCP+01] P. Bertoli, A. Cimatti, M. Pistore, M. Roveri, and P. Traverso. MBP: a Model Based Planner. In *Proc. of IJCAI-2001 workshop on Planning under Uncertainty and Incomplete Information*, 2001.

[BCPT03] P. Bertoli, A. Cimatti, M. Pistore, and P. Traverso. A Framework for Planning with Extended Goals under Partial Observability. In *Proc. ICAPS'03*, 2003.

[BDG03] J. Blythe, E. Deelman, and Y. Gil. Planning for workflow construction and maintenance on the grid. In *Proc. of ICAPS'03 Workshop on Planning for Web Services*, 2003.

[BHRT03] B. Benatallah, M-S. Hacid, C. Rey, and F. Toumani. Request rewriting-based Web service discovery. In *The Semantic Web - ISWC 2003*, pages 242–257, October 2003.

[BHS02] Franz Baader, Ian Horrocks, and Ulrike Sattler. Description Logics for the Semantic Web. *KI – Künstliche Intelligenz*, 16(4):57–59, 2002.

[BHVV01] Walter Binder, Jarle Hulaas, Alex Villazón, and Rory Vidal. Portable resource control in Java: The J-SEAL2 approach. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA-2001)*, Tampa Bay, Florida, USA, October 2001.

[Bin01] Walter Binder. Design and implementation of the J-SEAL2 mobile agent kernel. In *The 2001 Symposium on Applications and the Internet (SAINT-2001)*, San Diego, CA, USA, January 2001.

[BK98] A.J. Bonner and M. Kifer. A logic for programming database transactions. In J. Chomicki and G. Saake, editors, *Logics for Databases and Information Systems*, chapter 5, pages 117–166. Kluwer Academic Publishers, March 1998.

[BMNPS02] F. Baader, D. L. McGuiness, D. Nardi, and P. Patel-Schneider, editors. *Description Logic Handbook: Theory, implementation and applications*. Cambridge University Press, 2002.

[BN03] F. Baader and W. Nutt. Basic description logics. In Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors, *The Description Logic Handbook: Theory, Implementation, and Applications*, pages 43–95. Cambridge University Press, 2003.

[BPM+98] V. R. Benjamins, E. Plaza, E. Motta, D. Fensel, R. Studer, B. Wielinga, G. Schreiber, Z. Zdrahal, and S. Decker. IBROW3: An intelligent brokering service for knowledge-component reuse on the world-wide web. In *Proceedings of the 11th Banff Knowledge Acquisition for Knowledge-Based System Workshop (KAW98)*, Banff, Canada, 1998.

[BR02] Walter Binder and Volker Roth. Secure mobile agent systems using Java: Where are we heading? In *Seventeenth ACM Symposium on Applied Computing (SAC-2002)*, Madrid, Spain, March 2002.

[BS01] F. Baader and U. Sattler. An overview of tableau algorithms for description logics. *Studia Logica*, 69:5–40, 2001.

[BvHH+04] Sean Bechhofer, Frank van Harmelen, James Hendler, Ian Horrocks, Deborah L. McGuinness, Peter F. Patel-Schneider, and Lynn Andrea Stein eds. OWL Web Ontology Language Reference. URL http://www.w3.org/TR/owl-ref/, Feb 2004.

[CBF04a] Ion Constantinescu, Walter Binder, and Boi Faltings. An Extensible Directory Enabling Efficient Semantic Web Service Integration. In *3rd International Semantic Web Conference (ISWC04)*, Hiroshima, Japan, November 2004.

[CBF04b] Ion Constantinescu, Walter Binder, and Boi Faltings. Directory services for incremental service integration. In *First European Semantic Web Symposium (ESWS-2004)*, Heraklion, Greece, May 2004.

[CC00] Y. Chen and B.H.C. Cheng. *A Semantic Foundation for Specification Matching*. Cambridge University Press, Eds. M. Sitaraman and G. Leavens, eds. m. edition, 2000.

[CCMW01] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web Services Description Language (WSDL) 1.1. http://www.w3.org/TR/wsdl, March 2001.

[CF03] Ion Constantinescu and Boi Faltings. Efficient matchmaking and directory services. In *The 2003 IEEE/WIC International Conference on Web Intelligence*, 2003.

[CFB04a] Ion Constantinescu, Boi Faltings, and Walter Binder. Large scale testbed for type compatible service composition. In *ICAPS 04 workshop on planning and scheduling for Web and grid services*, 2004.

[CFB04b] Ion Constantinescu, Boi Faltings, and Walter Binder. Large scale, type-compatible service composition. In *IEEE International Conference on Web Services (ICWS-2004)*, San Diego, CA, USA, July 2004.

[CKW93] W. Chen, M. Kifer, and D.S. Warren. Hilog: A foundation for higher-order logic programming. *Journal of Logic Programming*, 15(3):187–230, 1993.

[Coa04] The OWL Services Coalition. OWL-S 1.1 beta release. Available at http://www.daml.org/services/owl-s/1.1B/, July 2004.

[CPRT03] A. Cimatti, M. Pistore, M. Roveri, and P. Traverso. Weak, Strong, and Strong Cyclic Planning via Symbolic Model Checking. *Artificial Intelligence*, 147(1-2):35–84, 2003.

[de 04] Jos de Bruijn (*editor*). The WSML Family of Representation Languages. Working draft, Digital Enterprise Research Insitute (DERI), March 2004. Look at `http://www.wsmo.org/2004/d16/d16.1/`.

[Der98] D. Mc Dermott. The Planning Domain Definition Language Manual. Technical Report 1165, Yale Computer Science University, 1998. CVC Report 98-003.

[DKO+02] Ying Ding, M. Korotkiy, Boris Omelayenko, V. Kartseva, V. Zykov, Michael Klein, Ellen Schulten, and Dieter Fensel. Goldenbullet: Automated classification of product data in e-commerce. In *Proceedings of Business Information Systems Conference (BIS 2002)*, Poznan, Poland, 2002.

[DL96] K.K. Dhara and G.T. Leavens. Forcing behavioral subtyping through specification inheritance. In *Proceedings of the 18th International Conference on Software Engineering (ICSE'18)*, March 1996.

[DLNS98] Francesco M. Donini, Maurizio Lenzerini, Daniele Nardi, and Andrea Schaerf. Al-log: integrating datalog and description logics. *Journal of Intelligent Information Systems*, 10:227–252, 1998.

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

[dMRME04] Jos de Bruijn, Francisco Martin-Recuerda, Dimitar Manov, and Marc Ehrig. State-of-the-art survey on ontology merging and aligning v1. Technical report, SEKT project IST-2003-506826, 2004.

[Don03] Francesco M. Donini. Complexity of Reasoning. In Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors, *The Description Logic Handbook: Theory, Implementation, and Applications*, pages 96–136. Cambridge University Press, 2003.

[DS04] Mike Dean and Guus Schreiber, editors. *OWL Web Ontology Language Reference*. 2004. W3C Recommendation 10 February 2004.

[EGW94] Oren Etzioni, Keith Golden, and Daniel Weld. Tractable closed world reasoning with updates. In *KR'94: Principles of Knowledge Representation and Reasoning*, 1994.

[Eme90] E. A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*. Elsevier, 1990.

[FB02] D. Fensel and C. Bussler. The Web Service Modeling Framework WSMF. *Electronic Commerce Research and Applications*, 1(2), 2002.

[Fel98] Christiane Fellbaum, editor. *WordNet: An Electronic Lexical Database*. MIT Press, May 1998.

[Fen03] D. Fensel. *Ontologies: Silver Bullet for Knowledge Management and Electronic Commerce, 2nd edition*. Springer-Verlag, Berlin, 2003.

[FMB+03] D. Fensel, E. Motta, V.R. Benjamins, S. Decker, M. Gaspari, R. Groenboom, W. Grosso, M. Musen, E. Plaza, G. Schreiber, R. Studer, and B. Wielinga. The Unified Problem-solving Method development Language UPML. *Knowledge and Information Systems (KAIS): An international journal*, 5(1), 2003.

[FN71] Richard Fikes and Nils J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. In *IJCAI*, pages 608–620, 1971.

[FPV98] Alfonso Fuggetta, Gian Pietro Picco, and Giovanni Vigna. Understanding Code Mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, May 1998.

[GCTB01] J. Gonzalez-Castillo, D. Trastour, and C. Bartolini. Description logics for matchmaking of services. In *KI-2001 Workshop on Applications of Description Logics*, September 2001.

[GMP04] Stephan Grimm, Boris Motik, and Chris Preist. Variance in e-business service discovery. *Semantic Web Services: Preparing to Meet the World of Business Applications, Workshop at ISWC 2004*, November 2004.

[Gru93a] T. R. Gruber. Towards Principles for the Design of Ontologies Used for Knowledge Sharing. In N. Guarino and R. Poli, editors, *Formal Ontology in Conceptual Analysis and Knowledge Representation*, Deventer, The Netherlands, 1993. Kluwer Academic Publishers.

[Gru93b] T. R. Gruber. A translation approach to portable ontology specification. *Knowledge Acquisition*, 5(2):199–220, 1993.

[HBCS03] R. Hull, M. Benedikt, V. Christophides, and J. Su. E-Services: A Look Behind the Curtain. In *Proc. PODS'03*, 2003.

[HHO04] H. He, H. Haas, and D. Orchard. Web services architecture usage scenarios. W3C working group note, W3C, February 2004.

[HM01] Volker Haarslev and Ralf Möller. RACER System Description. volume 2083, 2001.

[HMA02] J. Heflin and H. Munoz-Avila. LCW-based agent planning for the semantic web. In AAAI Press, editor, *Ontologies and the Semantic Web. Papers from the 2002 AAAI Workshop WS-02-11*, pages 63–70, 2002.

[HNP95] Joseph M. Hellerstein, Jeffrey F. Naughton, and Avi Pfeffer. Generalized search trees for database systems. In Umeshwar Dayal, Peter M. D. Gray, and Shojiro Nishio, editors, *Proc. 21st Int. Conf. Very Large Data Bases, VLDB*, pages 562–573. Morgan Kaufmann, 11–15 1995.

[Hoa69] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10), October 1969.

[Hor97] I. Horrocks. *Optimising Tableaux Decision Procedures for Description Logics*. PhD thesis, The University of Manchester, 1997. URL http://www.cs.man.ac.uk/ horrocks/Publications/download/1997/phd-2sss.ps.gz.

[Hor98] I. Horrocks. Using an Expressive Description Logic: FaCT or Fiction? pages 636–647, 1998.

[Hor03] I. Horrocks. Implementation and Optimisation Techniques. In Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors, *The Description Logic Handbook: Theory, Implementation, and Applications*, pages 306–346. Cambridge University Press, 2003.

[HPS98] I. Horrocks and P. F. Patel-Schneider. Comparing subsumption optimizations. 1998.

[HPSB+04] I. Horrocks, P.F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, and M. Dean. SWRL: A semantic web rule language combining OWL and RuleML. Available at http://www.w3.org/Submission/2004/SUBM-SWRL-20040521/, May 2004.

[HS01] Ian Horrocks and Ulrike Sattler. Ontology reasoning in the $\mathcal{SHOQ}$(D) description logic. pages 199–204, 2001.

[HS02] Ian Horrocks and Ulrike Sattler. Optimised Reasoning for $\mathcal{SHIQ}$. In *Proc. of the 15th Eur. Conf. on Artificial Intelligence (ECAI 2002)*, pages 277–281, July 2002.

[JC92] J.J Jeng and B.H.C. Cheng. Using automated reasoning techniques to determine software reuse. *International Journal of Software Engineering and Knowledge Engineering*, 2(4), December 1992.

[JC93] J.J Jeng and B.H.C. Cheng. *Using Formal Methods to Construct a Software Component Library*, volume 717 of *Lecture Notes in Computer Science*, pages 397–417. Springer Verlag, September 1993.

[JC94] J.J Jeng and B.H.C. Cheng. A formal approach to reusing more general components. In *IEEE 9th Knowledge-Based Software Engineering*, September 1994.

[JC95] J.J Jeng and B.H.C. Cheng. Specification matching for software reuse: A foundation. In *SSR'95. ACM SIGSOFT*. ACM Press, April 1995.

[KHW95] Nicholas Kushmerick, Steve Hanks, and Daniel S. Weld. An algorithm for probabilistic planning. *Artificial Intelligence*, 76(1–2):239–286, 1995.

[kif98] KIF. Knowledge Interchange Format: Draft proposed american national standard (dpans). Technical Report 2/98-004, available at http://www.daml.org/services/owl-s/1.1B/DRSguide.pdf, 1998.

[KL*editors*04] Uwe Keller, Rubén Lara, and Axel Polleres (*editors*). WSMO web service discovery. Technical report, DERI, November 2004.

[KLP+04] Michael Kifer, Rubén Lara, Axel Polleres, Chang Zhao, Uwe Keller, Holger Lausen, and Dieter Fensel. A logical framework for web service discovery. In *"Semantic Web Services: Preparing to Meet the World of Business Applications" worshop at ISWC 2004*, 2004.

[KMH97] Marcel Kornacker, C. Mohan, and Joseph M. Hellerstein. Concurrency and recovery in generalized search trees. In Joan M. Peckman, editor, *Proceedings, ACM SIGMOD International Conference on Management of Data: SIGMOD 1997: May 13–15, 1997, Tucson, Arizona, USA*, 1997.

[KSF04] Uwe Keller, Michael Stollberg, and Dieter Fensel. WOOGLE meets Semantic Web Fred. In *Proceedings of the Workshop on WSMO Implementations (WIW 2004)*, CEUR-WS.org/Vol-113/, September 2004.

[L*editors*04a] Rubén Lara and Axel Polleres (*editors*). Formal mapping and tool to owl-s. Technical report, DERI, December 2004.

[L*editors*04b] Rubén Lara and Holger Lausen (*editors*). WSMO discovery engine. Technical report, DERI, November 2004.

[LH03] L. Li and I. Horrocks. A software framework for matchmaking based on semantic web technology. In *Proceedings of the 12th International Conference on the World Wide Web*, Budapest, Hungary, May 2003.

[Lif87] V. Lifschitz. On the semantics of strips. In M. P. Georgeff and A. L. Lansky, editors, *Reasoning about Actions and Plans*, pages 1–9. Kaufmann, Los Altos, CA, 1987.

[LPL+] Rubén Lara, Axel Polleres, Holger Lausen, Dumitru Roman, Jos de Bruijn, and Dieter Fensel. A comparison of WSMO and OWL-S. *World Wide Web Journal, Special issue on Web Services: Theory and Practice*. Submitted.

[LW94] B.H. Liskov and J.M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages*, 16(10), November 1994.

[LY99] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, MA, USA, second edition, 1999.

[McD98] Drew McDermott. The planning domain definition language manual. Technical Report 1165, Yale Computer Science, 1998.

[McD03] Drew V. McDermott. Pddl2.1 - the art of the possible? commentary on fox and long. *J. Artif. Intell. Res. (JAIR)*, 20:145–148, 2003.

[McD04] D. McDermott. DRS: A set of conventions for representing logical languages in RDF. Available at http://www.daml.org/services/owl-s/1.1B/DRSguide.pdf, January 2004.

[MDCG03] E. Motta, J. Domingue, L. Cabral, and M. Gaspari. IRS-II: A framework and infrastructure for semantic web services. In *2nd International Semantic Web Conference (ISWC2003)*. Springer Verlag, October 2003.

[MF02] S. McIlraith and R. Fadel. Planning with Complex Actions. In *Proc. NMR'02*, 2002.

[Min81] Marvin Minsky. A Framework for Representing Knoledge. In J. Haugeland, editor, *Mind Design*. The MIT Press, 1981.

[MS02a] S. McIlraith and S. Son. Adapting Golog for composition of semantic Web Services. In *Proc. KR'02*, 2002.

[MS02b] Sheila A. McIlraith and Tran Cao Son. Adapting golog for composition of semantic web services. In Dieter Fensel, Fausto Giunchiglia, Deborah McGuinness, and Mary-Anne Williams, editors, *Proceedings of the 8th International Conference on Principles and Knowledge Representation and Reasoning (KR-02)*, pages 482–496, San Francisco, CA, April 22–25 2002. Morgan Kaufmann Publishers.

[MSZ01] S. McIlraith, T.C. Son, and H. Zeng. Mobilizing the semantic web with daml-enabled web services. In *Proc. Second International Workshop on the Semantic Web (SemWeb-2001)*, Hongkong, 2001.

[NM02] S. Narayanan and S. McIlraith. Simulation, Verification and Automated Composition of Web Services. In *Proc. WWW2002*, 2002.

[OlPL04] Daniel Olmedilla, Rubén lara, Axel Polleres, and Holger Lausen. Trust negotiation for semantic web services. In *First International Workshop on Semantic Web Services and Web Process Composition (SWSWPC 2004) at ICWS 2004*, July 2004.

[PA97] J. Penix and P. Alexander. Towards automated component adaptation. In *Proceedings of the 9th International Conference on Software Engineering and Knowledge Engineering*, June 1997.

[Pan04a] Jeff Z. Pan. *Description Logics: Reasoning Support for the Semantic Web*. PhD thesis, School of Computer Science, The University of Manchester, Oxford Rd, Manchester M13 9PL, UK, Sept 2004.

[Pan04b] Jeff Z. Pan. Reasoning Support for OWL-E (Extended Abstract). In *Proc. of Doctoral Programme in the 2004 International Joint Conference of Automated Reasoning (IJCAR2004)*, July 2004.

[PBB+04] M. Pistore, P. Bertoli, F. Barbon, D. Shaparau, and P. Traverso. Planning and Monitoring Web Service Composition. In *Proc. AIMSA'04*, 2004.

[Ped89] Edwin P. D. Pednault. Adl: Exploringthe middle ground between strips and the situation calculus. In *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning (KR'89)*, pages 324–332, Morgan Kaufmann Publishers, 1989.

[PH04] Jeff Z. Pan and Ian Horrocks. OWL-E: Extending OWL with Expressive Datatype Expressions. Technical report, School of Computer Science, the University of Manchester, April 2004.

[PKPS02] M. Paolucci, T. Kawamura, T. Payne, and K. Sycara. Semantic matching of web services capabilities. In I. Horrocks and J. Handler, editors, *1st Int. Semantic Web Conference (ISWC)*, pages 333–347. Springer Verlag, 2002.

[PR89] A. Pnueli and R. Rosner. On the synthesis of an asynchronous reactive module. In *Proc. ICALP'89*, 1989.

[Pre04] Chris Preist. A conceptual architecture for semantic web services. In *Proceedings of the International Semantic Web Conference 2004 (ISWC 2004)*, November 2004.

[PS99] Peter F. Patel-Schneider. DLP. In *Description Logics*, 1999.

[PSHH04] Peter F. Patel-Schneider, Patrick Hayes, and Ian Horrocks. OWL Web Ontology Language Semantics and Abstract Syntax. Technical report, W3C, Feb. 2004. W3C Recommendation, URL `http://www.w3.org/TR/2004/REC-owl-semantics-20040210/`.

[PTB04] M. Pistore, P. Traverso, and P. Bertoli. Automated composition of web services by planning in asynchronous domains, 2004. Submitted for publication.

[Qui67] M. Ross Quillian. Word Concepts: A Theory and Simulation of Some Basic Capabilities. *Behavioral Science*, 12:410–430, 1967.

[RL*editors*04] D. Roman, H. Lausen, and U. Keller (*editors*). Web service modeling ontology - standard (WSMO-Standard). Working draft, Digital Enterprise Research Institute (DERI), September 2004. http://www.wsmo.org/2004/d2/v1.0/.

[Ros94] R. Rosenfield. *Adaptive statistic language model*. PhD thesis, Carnegie Mellon University, 1994.

[RS95] Ray Richardson and Alan F. Smeaton. Using WordNet in a knowledge-based approach to information retrieval. Technical Report CA-0395, Dublin, Ireland, 1995.

[RW91] E.J. Rollings and J.M. Wing. Specifications as search keys for software libraries. In *Proceedings of the Eighth International Conference on Logic Programming*, June 1991.

[SdF03] M. Sheshagiri, M. desJardins, and T. Finin. A Planner for Composing Services Described in DAML-S. In *Proc. AAMAS'03*, 2003.

[S*editors*] M. Stollberg and R. Lara (*editors*). D3.3 v0.1 wsmo use case. Technical report.

[SF97] J. Schumann and B. Ficher. Nora/hammr: Making deduction-based software component retrieval practical. In *Proceedings of the 12th IEEE International Automated Software Engineering Conference (ASE97)*, November 1997.

[Sun] Sun Microsystems, Inc. Java HotSpot Technology. Web pages at `http://java.sun.com/products/hotspot/`.

[SVSM03] Kaarthik Sivashanmugam, Kunal Verma, Amit Sheth, and John Miller. Adding semantics to web services standards. In *Proceedings of the International Conference on Web Services (ICWS'03)*, June 2003.

[SWKL02] K. Sycara, S. Widoff, M. Klusch, and J. Lu. LARKS: Dynamic matchmaking among heterogeneous software agents in cyberspace. *Autonomous Agents and Multi-Agent Systems*, pages 173–203, 2002.

[Tee94] G. Teege. Making the difference: A substraction operation for description logics. In *KR'94*, 1994.

[TKAS02] S. Thakkar, Craig A. Knoblock, Jose Luis Ambite, and Cyrus Shahabi. Dynamically composing web services from on-line sources. In *Proceeding of the AAAI-2002 Workshop on Intelligent Service Integration*, pages 1–7, Edmonton, Alberta, Canada, July 2002.

[TP04] P. Traverso and M. Pistore. Automated Composition of Semantic Web Services into Executable Processes. In *Proc. ISWC'04)*, 2004.

[UG96] M. Uschold and M. Gruninger. Ontologies: Principles, Methods and Applications. *The Knowledge Engineering Review*, 1996.

[Var] M. Y. Vardi. An automata-theoretic approach to fair realizability and synthesis. In *Proc. CAV'95*. 1995.

[VBB98] Jan Vitek, Ciarán Bryce, and Walter Binder. Designing JavaSeal or how to make Java safe for agents. Technical report, University of Geneva, July 1998.

[Vol04] Raphael Volz. *Web Ontology Reasoning with Logic Databases*. PhD thesis, AIFB, Karlsruhe, 2004.

[VSSP04] K. Verma, K. Sivashanmugam, A. Sheth, and A. Patil. METEOR-S WSDI: A scalable P2P infrastructure of registries for semantic publication and discovery of web services. *Journal of Information Technology and Management*, 2004.

[W3C] W3C. XML Schema Part 2: Datatypes, http://www.w3.org/tr/xmlschema-2/.

[W3C03] W3C. SOAP version 1.2 part 0: Primer. http://www.w3.org/TR/2003/REC-soap12-part0-20030624/, June 2003.

[Wie92] G. Wiederhold. Mediators in the architecture of future information systems. *IEEE Computer*, 25(3):38–49, March 1992.

[WPS+03] D. Wu, B. Parsia, E. Sirin, J. Hendler, and D. Nau. Automating DAML-S Web Services Composition using SHOP2. In *Proc. ISWC'03*, 2003.

[Wu,03] Wu, Dan and Parsia, Bijan and Sirin, Evren and Hendler, James and Nau, Dana. Automating DAML-S Web Services Composition Using SHOP2. In *Proceedings of 2nd International Semantic Web Conference (ISWC2003)*, 2003.

[ZK04] O.K. Zein and Y. Kermarrec. An approach for describing/discovering services and for adapting them to the needs of users in distributed systems. In *Semantic Web Services. Papers from 2004 AAAI Spring Symposium*, March 2004.

[ZW95] A.M. Zaremski and J.M. Wing. Specification matching of software components. In *3rd ACM SIGSOFT Symposium on the Foundations of Software Engineering*, October 1995.

[ZW97] A.M. Zaremski and J.M. Wing. Specification matching of software components. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 6:333–369, 1997.