DIP

Data, Information and Process Integration with Semantic Web Services

FP6 - 507483

Deliverable

WP 5: Automatic Service Integration D5.3b

Michael Schumacher
Walter Binder
Ion Constantinescu

June 16th, 2004

EXECUTIVE SUMMARY

Automatic service integration or composition is one important approach to achieve interoperability among processes, making them collaborate in order to achieve given user goals (see state-of-the-art analysis in DIP deliverable 5.1 [Cimpian et al. 2005]). Actually, the goal of automated composition of web services is to compose automatically web services together in order to achieve a new functionality. This composition acts as a middleware, a mediator, between existing services.

In this document, we discuss in more details our specific approach to automatic composition, using type-compatible service composition, i.e. service composition that takes type constraints into account. We base first on the Knowledge Web Deliverable 2.4.2 [Lara et al. 2005][1] that presents a formalism to describe and reason about different service composition algorithms. This formalism uses the composition typing information in order to propose different composition algorithms. Then implementation techniques are considered to enable efficient, scalable service composition in an open environment populated by large numbers of heterogeneous services. In such a setting, the efficient interaction of directory-based service discovery with service composition engines is crucial. We present a directory that offers special functionality enabling effective service composition. In order to optimize the interaction of the directory with different service composition algorithms exploiting application-specific heuristics, the directory supports user-defined selection and ranking functions written in a declarative query language.

[1] The referenced part of this Knowledge Web deliverable has been written by the authors of this document.

Document Information

IST Project Number	FP6 – 507483	**Acronym**	DIP
Full title	Data, Information, and Process Integration with Semantic Web Services		
Project URL	http://dip.semanticweb.org		
Document URL			
EU Project officer	Kai Tullius		

Deliverable	**Number**	5.3b	**Title**	Automatic Service Integration
Work package	**Number**	5	**Title**	Service Mediation

Date of delivery	**Contractual**	M 18	**Actual**	30-Jun-05
Status	version. 1.0		final ☑	
Nature	Prototype ☐ Report ☑ Dissemination ☐			
Dissemination Level	Public ☐ Consortium ☑			

Authors (Partner)	Michael Schumacher (EPFL), Walter Binder (EPFL), Ion Constantinescu (NUIG)			
Responsible Author	Michael Schumacher		**Email**	michael.schumacher@epfl.ch
	Partner	EPFL	**Phone**	(+41 21) 693-6679

Abstract (for dissemination)	DIP D5.3b explores the use of automated service composition in order to integrate semantic web services. Possibilities and limitations of automated service composition are explored in this context. A formalism is defined to describe and reason about different service composition algorithms. Implementation techniques are then considered to enable efficient, scalable service composition.	
Keywords	Automated service composition, semantic web services, mediation.	

Version Log

Issue Date	**Rev No.**	**Author**	**Change**
<dd-mmm-yy>	<nnn starting 001>	<author name>	<Description of the changes that were made to the preceding revision>
01-06-05	**001**	Michael	First version

		Schumacher	
06-06-05	**002**	Michael Schumacher	Several changes
10-06-05	**003**	Michael Schumacher	New introductions to sections, updated executive summary, introduction and conclusion
22-06-05	**004**	Michael Schumacher	Last modifications

Project Consortium Information

Partner	Acronym	Contact
National University of Ireland Galway	NUIG	Prof. Dr. Christoph Bussler Digital Enterprise Research Institute (DERI) National University of Ireland, Galway Galway Ireland Email: chris.bussler@deri.org Tel: +353 91 512460
Fundacion De La Innovacion.Bankinter	Bankinter	Monica Martinez Montes Fundacion de la Innovation. BankInter Paseo Castellana, 29 28046 Madrid, Spain Email: mmtnez@bankinter.es Tel: 916234238
Berlecon Research GmbH	Berlecon	Dr. Thorsten Wichmann Berlecon Research GmbH Oranienburger Str. 32 10117 Berlin, Germany Email: tw@berlecon.de Tel: +49 30 2852960
British Telecommunications Plc.	BT	Dr John Davies BT Exact (Orion Floor 5 pp12) Adastral Park Martlesham Ipswich IP5 3RE, United Kingdom Email: john.nj.davies@bt.com Tel: +44 1473 609583
Swiss Federal Institute of Technology, Lausanne	EPFL	Prof. Karl Aberer Distributed Information Systems Laboratory École Polytechnique Féderale de Lausanne Bât. PSE-A 1015 Lausanne, Switzerland Email : Karl.Aberer@epfl.ch Tel: +41 21 693 4679
Essex County Council	Essex	Mary Rowlatt, Essex County Council PO Box 11, County Hall, Duke Street Chelmsford, Essex, CM1 1LX United Kingdom. Email: maryr@essexcc.gov.uk Tel: +44 (0)1245 436524
Forschungszentrum Informatik	FZI	Andreas Abecker Forschungszentrum Informatik Haid-und-Neu Strasse 10-14 76131 Karlsruhe Germany Email: abecker@fzi.de Tel: +49 721 9654 0

Partner	Acronym	Contact
Institut für Informatik, Leopold-Franzens Universität Innsbruck	UIBK	Prof. Dieter Fensel Institute of computer science University of Innsbruck Technikerstr. 25 A-6020 Innsbruck, Austria Email: dieter.fensel@deri.org Tel: +43 512 5076485
ILOG SA	ILOG	Christian de Sainte Marie 9 Rue de Verdun, 94253 Gentilly, France Email: csma@ilog.fr Tel: +33 1 49082981
inubit AG	Inubit	Torsten Schmale inubit AG Lützowstraße 105-106 D-10785 Berlin Germany Email: ts@inubit.com Tel: +49 30726112 0
Intelligent Software Components, S.A.	iSOCO	Dr. V. Richard Benjamins, Director R&D Intelligent Software Components, S.A. Pedro de Valdivia 10 28006 Madrid, Spain Email: rbenjamins@isoco.com Tel. +34 913 349 797
NIWA WEB Solutions	NIWA	Alexander Wahler NIWA WEB Solutions Niederacher & Wahler OEG Kirchengasse 13/1a A-1070 Wien Email: wahler@niwa.at Tel:+43(0)1 3195843-11 |
The Open University	OU	Dr. John Domingue Knowledge Media Institute The Open University, Walton Hall Milton Keynes, MK7 6AA United Kingdom Email: j.b.domingue@open.ac.uk Tel.: +44 1908 655014
SAP AG	SAP	Dr. Elmar Dorner SAP Research, CEC Karlsruhe SAP AG Vincenz-Priessnitz-Str. 1 76131 Karlsruhe, Germany Email: elmar.dorner@sap.com Tel: +49 721 6902 31

Sirma AI Ltd.	Sirma	Atanas Kiryakov, Ontotext Lab, - Sirma AI EAD Office Express IT Centre, 3rd Floor 135 Tzarigradsko Chausse Sofia 1784, Bulgaria Email: atanas.kiryakov@sirma.bg Tel.: +359 2 9768 303
Unicorn Solution Ltd.	Unicorn	Jeff Eisenberg Unicorn Solutions Ltd, Malcha Technology Park 1 Jerusalem 96951 Israel Email: Jeff.Eisenberg@unicorn.com Tel.: +972 2 6491111
Vrije Universiteit Brussel	VUB	Carlo Wouters Starlab- VUB Vrije Universiteit Brussel Pleinlaan 2, G-10 1050 Brussel ,Belgium Email: carlo.wouters@vub.ac.be Tel.: +32 (0) 2 629 3719

TABLE OF CONTENTS

EXECUTIVE SUMMARY .... **I**

TABLE OF CONTENTS .... **VII**

1 INTRODUCTION **1**

2 FORMALISM AND SEMANTICS **3**

2.1 Service Advertisements and Requests 3

2.2 Matchmaking – Current Approaches 4

2.3 Interval Constraints 4

2.4 Describing Services by Interval Constraints 6

3 TYPE-COMPATIBLE SERVICE COMPOSITION **8**

3.1 Type-Compatible Discovery and Composition 8

3.2 Type-Compatible Service Composition Versus Planning 9

3.3 Computing Type-Compatible Service Compositions 10

3.4 Composition with Complete Type Matches 10

3.5 Composition with Forward Partial Type Matches 11

3.5.1 Discovering full input coverage 11

3.5.2 Discovering solution switch 12

4 IMPLEMENTATION TECHNIQUES FOR AUTOMATED SERVICE INTEGRATION **13**

4.1 Multidimensional Access Methods - GiST 13

4.2 Defining Pruning and Ranking Functions 14

4.3 Efficient Directory Search 17

4.4 Example Queries for Service Composition 20

5 CONCLUSIONS **22**

REFERENCES **23**

LIST OF FIGURES

Figure 1: Service with types and corresponding planning operator 9
Figure 2: The architecture of our service integration engine... 10
Figure 3: Flow of algorithm for composition with partial type matches........................ 11
Figure 4: Generalised Search Tree (GiST)... 14
Figure 5: A grammar for $DirQL_{SE}$ 15
Figure 6: Processing of a directory query. While the given $DirQL_{SE}$ expression is directly applied to leaf nodes (white), it has to be transformed for inner nodes (black).17
Figure 7: Transformation operators $\uparrow$, $\downarrow$, $\oplus$, and $\ominus$ for the generation of inner node code.. 18
Figure 8: Transformation operator $\updownarrow$ for the generation of code in inner nodes of the GiST. 19
Figure 9: Forward chaining (complete matches)... 20
Figure 10: Backward chaining.. 21

1 Introduction

Automatic service integration or composition is one important approach to achieve interoperability among processes, making them collaborate in order to achieve given user goals. Actually, the goal of automated composition of web services is to compose automatically web services together in order to achieve a new functionality. This composition acts as a middleware, a mediator, between existing services. The present document extends this view by directly basing on section 2.2.2.3 of DIP deliverable 5.1 [Cimpian et al. 2005] which discusses process integration by process composition.

Two types of service composition can be considered [PRT05].

In *functional-level composition*, the searched services are selected and combined together in a suitable way with basic composition constructs in order to match a user query. This means that the service composition interacts with the service discovery to dynamically retrieve relevant service descriptions. 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. The query defines the overall functionality that the composed service should implement, again in terms of its inputs, outputs, preconditions, and effects. The composition constructs allow basic interaction schemes of the type request-answer.

Process-level service composition follows the protocols of the different services involved in order to obtain a composed service. The starting point can be the set of services found in the functional-level composition. Here, however, it is insufficient to consider services as only inputs, outputs, preconditions and effects: a more precise web description is needed in the form of a process model. For example, in flight booking, several details should be considered such as authentication, offer negotiation or payment, maybe with conditional or non-nominal outcomes that may influence the following steps. Here, more complex constructs are used that take typically advantage of workflow concepts: atomic interactions are composed with sequences, conditions and iterations. This means that composed processes are stateful processes. This approach to process-level service composition is proposed in work such as [TP04] and [PRT05].

Our specific approach to automatic composition can be considered as functional-level composition. It uses type-compatible service composition, i.e., service composition that takes type constraints into account. Our techniques are very much related to traditional AI planning. The service composition problem is specified by a set of available inputs (preconditions) and provided outputs (effects). The planning results in an arrangement of services in a simple workflow. 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.

This document is organized as follows. Section 2 first presents a formalism to describe and reason about different service composition algorithms. This formalism uses the composition typing information in order to propose different composition algorithms. In section 3, we discuss in more details our specific approach to type-compatible service composition, i.e. service composition that takes type constraints into account. In section 4, implementation techniques are considered to enable efficient, scalable service composition in an open environment populated by large numbers of heterogeneous services. In such a setting, the efficient interaction of directory-based service discovery with service composition engines is crucial. We present a directory that offers special functionality enabling effective service composition. In order to optimize the interaction of the directory with different service composition algorithms exploiting application-specific heuristics, the directory supports user-defined selection and ranking functions written in a declarative query language.

The work presented in this document is cross-integration between the Knowledge Web and the DIP European projects. More concretely, section 2 and 3 have also been published by the authors as a part of part of Knowledge Web Deliverable 2.4.2 [Lara et al. 2005].

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. We review some state-of-the-art regarding matchmaking that is of interest for our composition approach, and we introduce interval constraints, a supporting formalism which we use for describing and matching services.

2.1 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. Primitive data-types can be defined using a language like XSD [W3C].[3] 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.

Service requests are represented in a similar manner but have different semantics:

[3] At the implementation level both primitive data types 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.

2.2 Matchmaking – Current Approaches

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.

2.3 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 S is logically equivalent to the formula formed as the conjunction of the constraints in the store:

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

By combining universal (all) and existential ($some$) quantifiers over a pair of constraint stores Q and S we can define eight predicates (e.g., $all_Q all_S$, $all_Q some_S$,..., $all_S all_Q, all_S some_Q$,..., etc). Each of the predicates holds if the two stores contain constraints accordingly to the quantifications q_Q and q_S that are in a relation as defined above by ent:

$$q_1 q_2(P_Q, P_S, op_1, ..., op_n) \Leftrightarrow$$
$$((\forall | \exists)(P_Q | P_S))((\forall | \exists)(P_S | P_Q))$$
$$(P_Q \in Q)(P_S \in S)\ ent(P_Q, P_S, op_1, ..., op_n),$$

where $q_1, q_2 \in \{all_Q, all_S, some_Q, some_S\}$, $store(q_1) \neq store(q_2)$ and where $store(quant_X) = X$ for $quant \in \{all, some\}$, $X \in \{Q, S\}$.

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 $notEnt$:

$$\neg q_1 q_2 (P_Q, P_S, op_1, ..., op_n) \Leftrightarrow$$
$$((\exists \mid \forall)(P_S \mid P_Q))((\exists \mid \forall)(P_Q \mid P_S))$$
$$(P_Q \in Q)(P_S \in S)\ \ notEnt(P_Q, P_S, op_1, ..., op_n),$$

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 S accordingly to their entailment relation with constraints in the store Q :

$$count_{Q,S}(P_Q, P_S, op_1, ..., op_n) = \ |\{P_S \in S : P_Q \in Q, ent(P_Q, P_S, op_1, ..., op_n)\}|.$$

We introduce also $count_Q$ and $count_S$ two functions which return the cardinality of a set of constraints having a given name P from the stores Q or S :

$$count_Q(P) = |\{P(C_1, ..., C_n) \in Q\}|,$$
$$count_S(P) = |\{P(C_1, ..., C_n) \in S\}|.$$

2.4 Describing Services by Interval Constraints

We use constraint stores to define service advertisements or service requests. 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: 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 Q and a service store S this match type can be specified as:

$$match_{PlugIn}(Q,S) =$$

$$\begin{aligned}
&all_S some_Q(IN_Q, IN_S, \sqsubseteq_{role}, \sqsubseteq_{type}) \wedge \\
&all_Q some_S(OUT_Q, OUT_S, \sqsupseteq_{role}, \sqsupseteq_{type}) \wedge \\
&all_S some_Q(PRE_Q, PRE_S, \sqsubseteq) \wedge \\
&all_Q some_S(EFF_Q, EFF_S, \sqsupseteq) .
\end{aligned}$$

In the next section, we introduce our work on functional-level service composition and, for that, we explain in details type-compatible service composition.

3 TYPE-COMPATIBLE SERVICE COMPOSITION

Most works in functional-level service composition assume that the relevant service descriptions are initially loaded into the reasoning engine and that no discovery is performed during composition. However, due to the large number of services and to the loose coupling between service providers and consumers, services are indexed in directories. Consequently, planning algorithms must be adapted to a situation where planning operators are not known a priori, but have to be retrieved through queries to these directories.

We present here our approach to functional-level automated service composition that interacts with such directories. It is based on matching input and output parameters of services using type information in order to constrain the ways how services may be composed. This way, we allow for *partially matching* types and handle them by introducing *switches* in the composition plan.

3.1 Type-Compatible Discovery and Composition

For composition we consider 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 therange 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 $all_S some_Q$ predicate:

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

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

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 holiday's 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:

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

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:

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

3.2 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 Figure 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 preconditions and disjunctive effects. Negation is not required.

```
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 1: Service with types and corresponding planning operator

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.

3.3 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 ser- vice 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 Figure 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.
- **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.

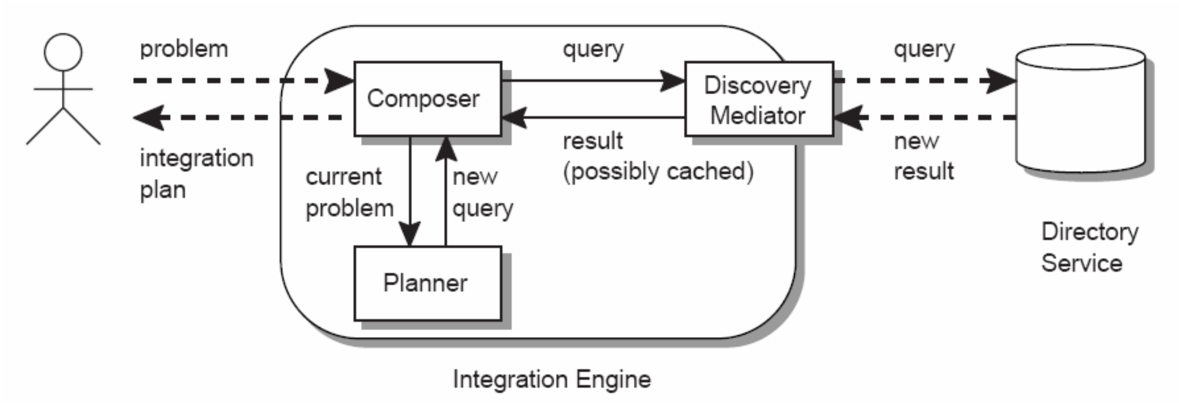


Figure 2: The architecture of our service integration engine.

3.4 Composition with Complete Type Matches

Composing completely matching services using forward chaining is straightforward: once the condition for complete type matches is fulfilled (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*.

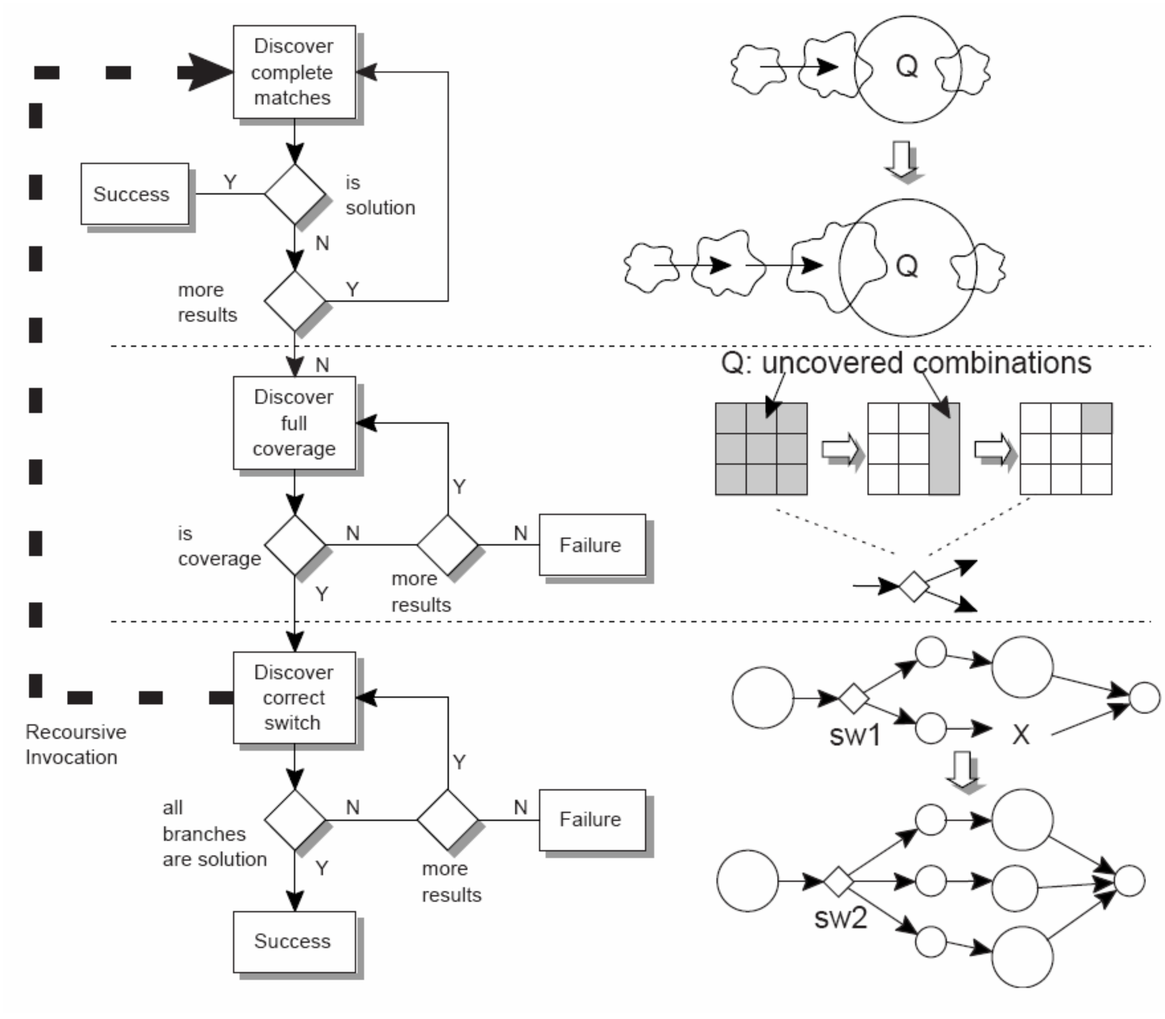


Figure 3: Flow of algorithm for composition with partial type matches

3.5 Composition with Forward Partial Type Matches

Conceptually the algorithm that we use for composing services with forward partial type matches has three steps, see (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.

3.5.1 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 assembled 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.

3.5.2 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.

4 Implementation Techniques for Automated Service Integration

In automatic service integration, the set of service descriptions may be very large and is usually maintained in service directories. Hence, as previously showed, 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 give an overview of implementation techniques to support scalable and efficient automated service integration with service directories. We include 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]. We propose a directory service with specific features to ease service composition. Queries may not only search for complete matches, but may also retrieve *partially matching* directory entries [CF03]. As in a large-scale directory the number of (partially) matching results for a query may be very high, it is crucial to order the result set within the directory according to heuristics and to transfer first the better matches to the client. If the heuristics work well, only a small part of the possibly large result set has to be transferred, thus saving network bandwidth and boosting the performance of a directory client that executes a service composition algorithm (the results are returned incrementally, once a result fulfils the client's requirements, no further results need to be transmitted). However, the heuristics depend on the concrete composition algorithm. For each service composition algorithm (e.g., forward chaining, backward chaining, etc.), a different heuristic may be better adapted. As research on service composition is still in its beginnings and the directory cannot anticipate the needs of all possible service composition algorithms, our directory supports *user-defined selection and ranking heuristics* expressed in a *declarative query language*. The support for application-specific heuristics significantly increases the flexibility of our directory, as the client is able to tailor the processing of directory queries. For efficient execution, the queries are *dynamically transformed* by the directory.

This section is structured as follows: Section 4.1 discusses the directory structure. In section 4.2, we discuss how to express application-specific selection and ranking heuristics in a simple, functional query language. Section 4.3 explains the processing of directory queries and introduces query transformations that enable a best-first search with early pruning. Section 4.4 discusses some sample queries.

4.1 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 Figure 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., file system 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.

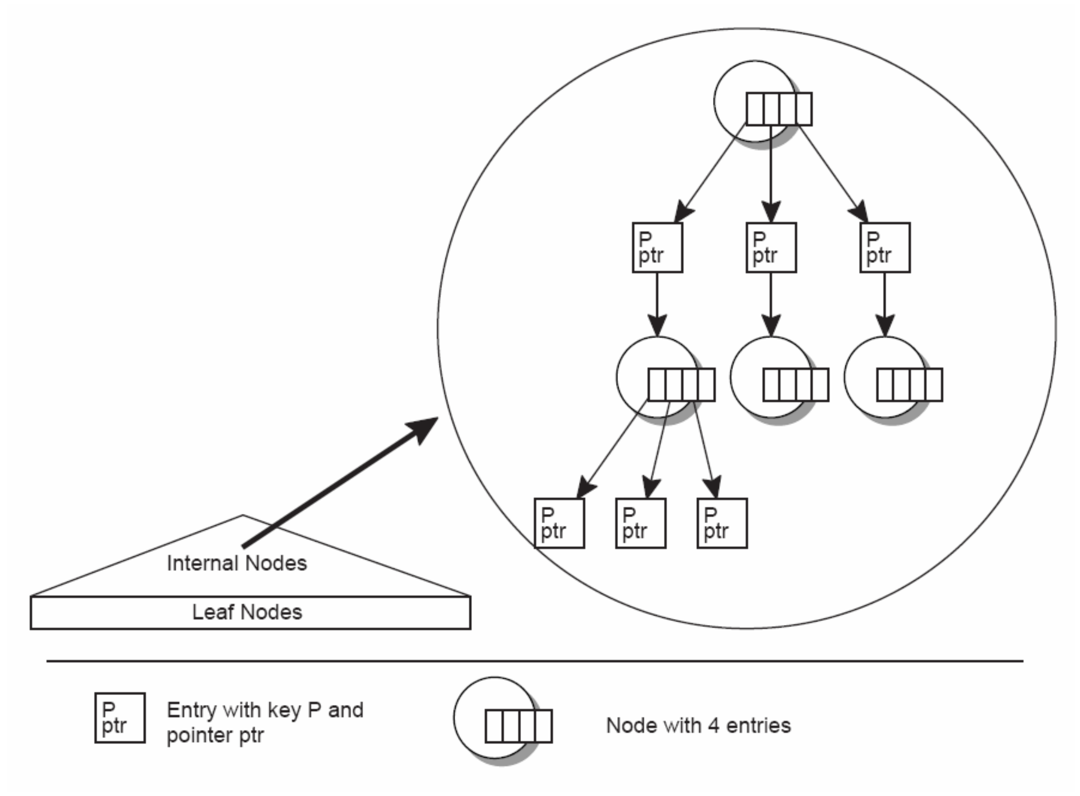


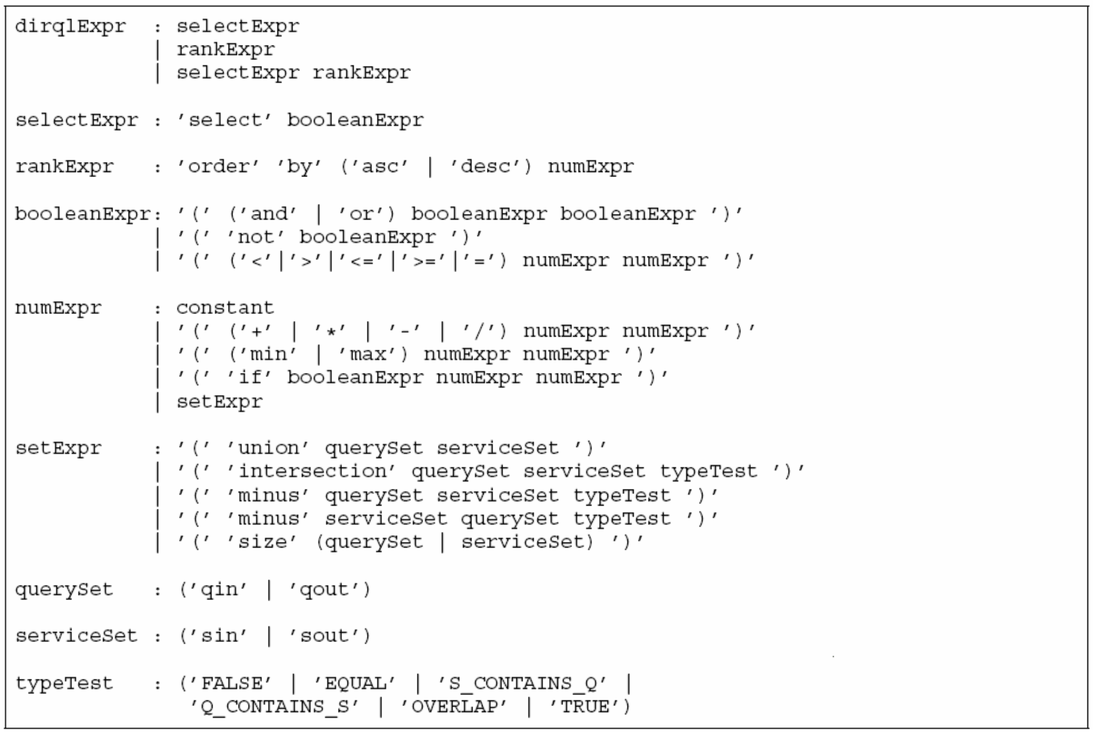
Figure 4: Generalised Search Tree (GiST)

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 parameters 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.

4.2 Defining Pruning and Ranking Functions

As directory queries may retrieve large numbers of matching entries (especially when partial matches are taken into consideration), our directory supports 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 composition 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 composition algorithm a different pruning and ranking heuristic may be better suited, our directory allows its clients to define custom selection 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].

A directory query consists of a set of provided inputs and required outputs (both sets contain tuples of parameter name and associated type), as well as a custom selection and ranking function. The selection and ranking function is written in the simple, high-level, functional query language $DirQL_{SE}$ (Directory Query Language with Set Expressions). An (informal) EBNF grammar for $DirQL_{SE}$ is given in . The non-terminal *constant*, which is not shown in the grammar, represents a non-negative numeric constant (integer or decimal number). The syntax of $DirQL_{SE}$ has some similarities with LISP.[5]



```
dirqlExpr   : selectExpr
            | rankExpr
            | selectExpr rankExpr

selectExpr : 'select' booleanExpr

rankExpr   : 'order' 'by' ('asc' | 'desc') numExpr

booleanExpr: '(' ('and' | 'or') booleanExpr booleanExpr ')'
            | '(' 'not' booleanExpr ')'
            | '(' ('<'|'>'|'<='|'>='|'=') numExpr numExpr ')'

numExpr     : constant
            | '(' ('+' | '*' | '-' | '/') numExpr numExpr ')'
            | '(' ('min' | 'max') numExpr numExpr ')'
            | '(' 'if' booleanExpr numExpr numExpr ')'
            | setExpr

setExpr     : '(' 'union' querySet serviceSet ')'
            | '(' 'intersection' querySet serviceSet typeTest ')'
            | '(' 'minus' querySet serviceSet typeTest ')'
            | '(' 'minus' serviceSet querySet typeTest ')'
            | '(' 'size' (querySet | serviceSet) ')'

querySet    : ('qin' | 'qout')

serviceSet : ('sin' | 'sout')

typeTest    : ('FALSE' | 'EQUAL' | 'S_CONTAINS_Q' |
              'Q_CONTAINS_S' | 'OVERLAP' | 'TRUE')
```

Figure 5: A grammar for $DirQL_{SE}$

We have designed the language considering the following requirements:

- Simplicity: $DirQL_{SE}$ offers only a minimal set of constructs, but it is expressive enough to write relevant selection and ranking heuristics.
- Declarative: $DirQL_{SE}$ is a functional language and does not support destructive assignment. The absence of side-effects eases program analysis and transformations.
- Safety: As the directory executes user-defined code, $DirQL_{SE}$ expressions must

[5]In order to simplify the presentation, the operators '`and`', '`or`', '<', '>', '<=', '>=', '=', '+', '`*`', '`-`', '`min`', and '`max`' are binary, whereas in the implementation they may take an arbitrary number arguments, similar to the definition of these operations in LISP.

not interfere with internals of the directory. Moreover, the resource consumption (e.g., CPU, memory) needed for the execution of $DirQL_{SE}$ expressions is bounded in order to prevent denial-of-service attacks: $DirQL_{SE}$ supports neither recursion nor loops, and queries can be executed without dynamic memory allocation.

- Efficient directory search: $DirQL_{SE}$ has been designed to enable an efficient best-first search in the directory GiST. Code transformations automatically generate selection and ranking functions for the inner nodes of the GiST (see 4.3).

A $DirQL_{SE}$ expression defines custom selection and ranking heuristics. The evaluation of a $DirQL_{SE}$ expression is based on the 4 sets `qin` (available inputs specified in the query), `qout` (required outputs specified in the query), `sin` (required inputs of a certain service S), and `sout` (provided outputs of a certain service S). Each element in each of these sets represents a query/service parameter identified by its unique name within the set and has an associated type.

A $DirQL_{SE}$ expression may involve some simple arithmetic. The result of a numeric $DirQL_{SE}$ expression is always non-negative. The '`-`' operator returns 0 if the second argument is bigger than the first one. The $DirQL_{SE}$ programmer may use the '`if`' conditional to ensure that the first argument of '`-`' is bigger or equal than the second one. For division, the second operand (divisor) has to evaluate to a constant for a given query. That is, it is a numeric expression with only numeric constants, as well as `size(qin)` and `size(qout)` at the leaves. Before a query is executed, the directory ensures that the $DirQL_{SE}$ expression will not cause a division by zero. For this purpose, all subexpressions are examined. The reason for these restrictions will be explained in the following section.

A $DirQL_{SE}$ query may comprise a selection and a ranking expression. Service descriptions (inputs/outputs defined by `sin`/`sout`) for which the selection expression evaluates to *false* are not returned to the client (pruning). The ranking expression defines the custom ranking heuristics. For a certain service description, the ranking expression computes a non-negative value. The directory will return service descriptions in ascending or descending order, as specified by the ranking expression.

The selection and ranking expressions may make use of several set operations. `size` returns the cardinality of any of the sets `qin`, `qout`, `sin`, or `sout`. The operations `union`, `intersection`, and `minus` take as arguments a query set (`qin` or `qout`) as well as a service set (`sin` or `sout`). For `union` and `intersection`, the query set has to be provided as the first argument. All set operations return the cardinality of the resulting set.

union: Cardinality of the union of the argument sets. Type information is irrelevant for this operation.

intersection: Cardinality of the intersection of the argument sets. For a parameter to be counted in the result, it has to have the same name in both argument sets and the type test (third argument) has to succeed.

minus: Cardinality of the set minus of the argument sets (first argument set minus second argument set). For a parameter to be counted in the result, it has to occur in the first argument set and, either there is no parameter with the same name in the second set, or in the case of parameters with the same name, the type test has to fail.

The type of parameters cannot be directly accessed, only the operations `intersection` and `minus` make use of the type information. For these operations, a type test is applied to parameters that have the same name in the given query and service set. The following type tests are supported (T_S denotes the type of a common parameter in the service set, while T_Q is the type of the parameter in the query set): `FALSE` (always fails), `EQUAL` (succeeds if $T_S = T_Q$), `S_CONTAINS_Q` (succeeds if T_S subsumes T_Q), `Q_CONTAINS_S` (succeeds if T_Q subsumes T_S), `OVERLAP` (succeeds if there is an overlap between T_S and T_Q, i.e., if a common subtype of T_S and T_Q exists), and `TRUE` (always succeeds).

4.3 Efficient Directory Search

Processing a user query requires traversing the GiST structure of the directory starting from the root node. The given $DirQL_{SE}$ expression is applied to leaf nodes of the directory tree, which correspond to concrete service descriptions (i.e., `sin` and `sout` represent the exact input/output parameters of a service description). For an inner node I of the GiST, `sin` and `sout` are supersets of the input/output parameters found in any node of the subtree whose root is I. The type of each parameter in I is a supertype of the parameter found in any node (which has a parameter with the same name) in the subtree. Therefore, the user-defined selection and ranking function cannot be directly applied to inner nodes.

In order to prune the search (as close as possible to the root of the GiST) and to implement a best-first search strategy which expands the most promising branch in the tree first, appropriate selection (pruning) and ranking functions are needed for the inner nodes of the GiST. In our approach, the client defines only the selection and ranking function for leaf nodes (i.e., to be invoked for concrete service descriptions), while the corresponding functions for inner nodes are automatically generated by the directory. The directory uses a set of simple transformation rules that enable a very efficient generation of the selection and ranking functions for inner nodes (the execution time of the transformation algorithm is linear with the size of the query). Figure 6 illustrates the processing of a directory query.

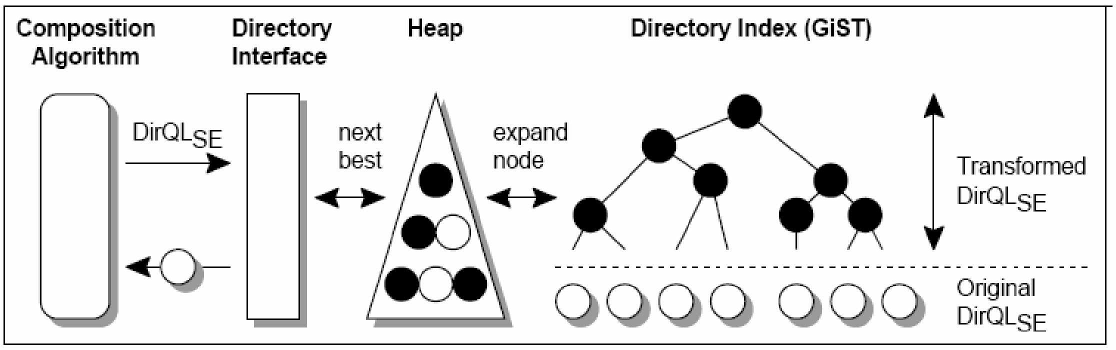


Figure 6: Processing of a directory query. While the given $DirQL_{SE}$ expression is directly applied to leaf nodes (white), it has to be transformed for inner nodes (black).

If the client desires ranking in ascending order, the generated ranking function for inner nodes computes a lower bound of the ranking value in any node of the subtree; for ranking in descending order, it calculates an upper bound. While the query is being processed, the visited nodes are maintained in a heap (or priority queue), where the node with the most promising heuristic value comes first. Always the first node is expanded; if it is a leaf node, it is returned to the client. 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 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 approach 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 the number of retrieved results exceed a certain threshold defined by the directory service provider.

Figure 7 shows the transformation operators $\uparrow$ and $\downarrow$ which allow to generate the code for calculating upper and lower bounds in inner nodes of the GiST. The variables a and b are arbitrary numeric expressions, c is a numeric expression that is guaranteed to be constant throughout a query, x is a boolean expression, q may be `qin` or `qout`, s may be `sin` or `sout`, and t is a type test. The operator $\oplus$ relaxes certain type tests, the operator $\ominus$ constrains them. For a $DirQL_{SE}$ ranking expression '`order by asc` E', the code for inner node ranking is '`order by asc` $\downarrow E$'; for a ranking expression '`order by desc` E', the inner node ranking code is '`order by desc` $\uparrow E$'.

$\uparrow constant$	$\longrightarrow constant$	$\downarrow constant$	$\longrightarrow constant$
$\uparrow(+\ a\ b)$	$\longrightarrow (+\ \uparrow a\ \uparrow b)$	$\downarrow(+\ a\ b)$	$\longrightarrow (+\ \downarrow a\ \downarrow b)$
$\uparrow(*\ a\ b)$	$\longrightarrow (*\ \uparrow a\ \uparrow b)$	$\downarrow(*\ a\ b)$	$\longrightarrow (*\ \downarrow a\ \downarrow b)$
$\uparrow(-\ a\ b)$	$\longrightarrow (-\ \uparrow a\ \downarrow b)$	$\downarrow(-\ a\ b)$	$\longrightarrow (-\ \downarrow a\ \uparrow b)$
$\uparrow(/\ a\ c)$	$\longrightarrow (/\ \uparrow a\ c)$	$\downarrow(/\ a\ c)$	$\longrightarrow (/\ \downarrow a\ c)$
$\uparrow(min\ a\ b)$	$\longrightarrow (min\ \uparrow a\ \uparrow b)$	$\downarrow(min\ a\ b)$	$\longrightarrow (min\ \downarrow a\ \downarrow b)$
$\uparrow(max\ a\ b)$	$\longrightarrow (max\ \uparrow a\ \uparrow b)$	$\downarrow(max\ a\ b)$	$\longrightarrow (max\ \downarrow a\ \downarrow b)$
$\uparrow(if\ x\ a\ b)$	$\longrightarrow (max\ \uparrow a\ \uparrow b)$	$\downarrow(if\ x\ a\ b)$	$\longrightarrow (min\ \downarrow a\ \downarrow b)$
$\uparrow(union\ q\ s)$	$\longrightarrow (union\ q\ s)$	$\downarrow(union\ q\ s)$	$\longrightarrow (size\ q)$
$\uparrow(intersection\ q\ s\ t)$	$\longrightarrow (intersection\ q\ s\ \oplus t)$	$\downarrow(intersection\ q\ s\ t)$	$\longrightarrow 0$
$\uparrow(minus\ q\ s\ t)$	$\longrightarrow (size\ q)$	$\downarrow(minus\ q\ s\ t)$	$\longrightarrow (minus\ q\ s\ \oplus t)$
$\uparrow(minus\ s\ q\ t)$	$\longrightarrow (minus\ s\ q\ \ominus t)$	$\downarrow(minus\ s\ q\ t)$	$\longrightarrow 0$
$\uparrow(size\ q)$	$\longrightarrow (size\ q)$	$\downarrow(size\ q)$	$\longrightarrow (size\ q)$
$\uparrow(size\ s)$	$\longrightarrow (size\ s)$	$\downarrow(size\ s)$	$\longrightarrow 0$
$\oplus TRUE$	$\longrightarrow TRUE$	$\ominus TRUE$	$\longrightarrow TRUE$
$\oplus OVERLAP$	$\longrightarrow OVERLAP$	$\ominus OVERLAP$	$\longrightarrow FALSE$
$\oplus Q\_CONTAINS\_S$	$\longrightarrow OVERLAP$	$\ominus Q\_CONTAINS\_S$	$\longrightarrow Q\_CONTAINS\_S$
$\oplus S\_CONTAINS\_Q$	$\longrightarrow S\_CONTAINS\_Q$	$\ominus S\_CONTAINS\_Q$	$\longrightarrow FALSE$
$\oplus EQUAL$	$\longrightarrow S\_CONTAINS\_Q$	$\ominus EQUAL$	$\longrightarrow FALSE$
$\oplus FALSE$	$\longrightarrow FALSE$	$\ominus FALSE$	$\longrightarrow FALSE$

Figure 7: Transformation operators $\uparrow$, $\downarrow$, $\oplus$, and $\ominus$ for the generation of inner node code.

If I is an inner node on the path to the leaf node L and E is a $DirQL_{SE}$ ranking expression, $\uparrow E$ (resp. $\downarrow E$) applied to I has to compute an upper (resp. lower) bound for E applied to L. We exemplarily explain 2 rules in an informal way:

First we consider computing an upper bound for $E = (intersection\ q\ s\ t)$. In an inner node I, the service set s_I is a superset of s_L in a leaf node, while the query set q remains constant. Moreover, the type of each parameter in s_L is subsumed by the type of the parameter with the same name in s_I. Not considering the parameter types, applying E to I would compute an upper bound for E applied to L, as intuitively the intersection of q with the bigger set s_I will not be smaller than the intersection of q with s_L. Taking parameter types into consideration, we must ensure that whenever a type test succeeds for L, it will also succeed for I. That is, if a common parameter is counted in the intersection in L, it must be also counted in the intersection in I. As it can be seen in Figure 7, $\oplus t$ will succeed in I, if t succeeds in L (remember that parameter types are guaranteed to be non-empty). For instance, if the type of a parameter in s_L is subsumed by the type of the parameter with the same name in q (`Q_CONTAINS_S` succeeds for that parameter in L), the type of the corresponding parameter in s_I (which subsumes the type in s_L) will overlap with the parameter type in q. If the types in s_L and q are equal, the type in s_I will subsume the type in q.

As a second example we want to compute an upper bound for $E = (minus\ s\ q\ t)$. Without considering parameter types, applying E to I would give an upper bound for E applied to L, as s_I is a superset of s_L. In contrast to intersection, a common parameter is counted in the result if the type test fails. That is, if the type test fails in L, it has also to fail in I. As shown in Figure 7, $\ominus t$ will fail in I, if t fails in L. For example, if the type of a parameter in q does not subsume the type of the parameter with the same name in s_L (`Q_CONTAINS_S` fails for that parameter in L), it will also not subsume the type of that parameter in s_I (which subsumes the type of the parameter in s_L). If the type test is `TRUE`, it will never fail, neither in L nor in I. In all other cases, no matter whether the type test fails in L or not, it will fail in I (because $\ominus t$ will be `FALSE`). Hence, '$\uparrow$ `(minus` s q t`)`' may result in '`(minus` s q `FALSE)`', which is equivalent to '`(size` s`)`'.

Considering the upper bound operator $\uparrow$, the reason why we require the divisor of '`/`' to evaluate to a constant becomes apparent: If c was not constant, for division the operator $\uparrow$ would have been defined as '$\uparrow(/\ a\ c) \longrightarrow (/\ \uparrow a\ \downarrow c)$'. Hence, even if the ranking expression provided by the client did not divide by zero ($c > 0$), the automatically generated code for computing an upper bound in inner nodes might possibly result in a division by zero ($\downarrow c = 0$). For this reason, c must depend neither on `sin` nor on `sout`.

In order to automatically generate the code for inner node selection (pruning), we define the transformation operator $\updownarrow$ for boolean expressions (see Figure 8). If E is *true* for a leaf node L, $\updownarrow E$ has to be *true* for all nodes on the path to L. In other words, if $\updownarrow E$ is *false* for an inner node, it must be guaranteed that E will be *false* for each leaf in the subtree. This condition ensures that during the search an inner node may be discarded (pruning) only if it is sure that all leaves in the subtree are to be discarded, too. For a $DirQL_{SE}$ selection expression '`select` E', the code for inner node selection is '`select` $\updownarrow E$'. In Figure 8 a and b are numeric expressions, while x and y are boolean expressions.

$\updownarrow(and\ x\ y)$	$\longrightarrow (and\ \updownarrow x\ \updownarrow y)$	$\updownarrow(or\ x\ y)$	$\longrightarrow (or\ \updownarrow x\ \updownarrow y)$
$\updownarrow(<\ a\ b)$	$\longrightarrow (<\ \downarrow a\ \uparrow b)$	$\updownarrow(<=\ a\ b)$	$\longrightarrow (<=\ \downarrow a\ \uparrow b)$
$\updownarrow(>\ a\ b)$	$\longrightarrow (>\ \uparrow a\ \downarrow b)$	$\updownarrow(>=\ a\ b)$	$\longrightarrow (>=\ \uparrow a\ \downarrow b)$

Figure 8: Transformation operator $\updownarrow$ for the generation of code in inner nodes of the GiST.

The alert reader may have noticed that the operators '`not`' and '`=`' have been omitted in Figure 8. The reason for this omission is that initially we transform all boolean expressions in the query according to De Morgan's theorem, moving negations towards the leaves, removing double negations, and changing the comparators if needed. The resulting expressions are free of negations. Moreover, an expression of the form `(=` *a* *b* `)` is transformed to the equivalent expression `(and (<=` *a* *b* `) (<=` *b* *a* `))`.

Related to our work are SS trees [Aoki1998], a GiST extension for directed stateful search. The main difference between SS trees and our approach is that we use a declarative query language which makes the internal organization of the directory transparent to the user. In our system, search is still very efficient thanks to query transformation.

4.4 Example Queries for Service Composition

In this section we discuss two simple selection and ranking heuristics: The first one is suited for service composition algorithms using forward chaining, the second one for algorithms based on backward chaining.

For forward chaining with complete type matches (see Figure 9 (a)), we want that all inputs required by the service are provided by the query (and the service has to be able to handle the parameter types of the provided inputs, i.e., the types in the query have to be more specific than in the service). Moreover, we require that the service provides new outputs which are not already available as query inputs. The results are sorted in ascending order according to the remaining outputs that are required by the query, but not provided by the service (services that provide more of the required outputs come first). In order to support partial type matches, only `S_CONTAINS_Q` has to be replaced with `OVERLAP` in the first line of the selection expression in Figure 9 (a).

```
select (and (<= (minus sin  qin S_CONTAINS_Q) 0)
            (>  (minus sout qin Q_CONTAINS_S) 0))
order by asc (minus qout sout Q_CONTAINS_S)
```

(a) User-defined selection and ranking function.

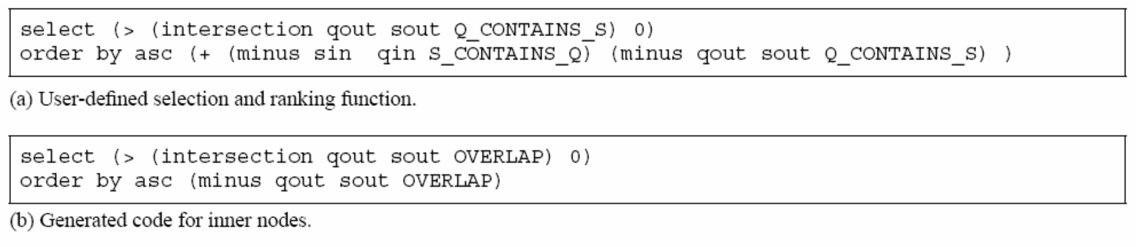
```
select (> (minus sout qin Q_CONTAINS_S) 0)
order by asc (minus qout sout OVERLAP)
```

(b) Generated code for inner nodes.

Figure 9: Forward chaining (complete matches).

For backward chaining (see Figure 10 (a)), we expect that the service provides at least one output that is required by the query. The results are sorted in ascending order according to the number of missing parameters after application of the service, i.e., the missing inputs of the service and the missing outputs as required by the query.

The code for inner nodes is generated according to the transformation scheme presented in the previous section, as illustrated in Figure 9 (b) and Figure 10 (b). Note that after applying the transformation rules, the resulting expressions have been simplified according to simple algebraic rules, such as '`(<= 0 0) =` *true*', '`(and` *true* *X* `) =` *X*', '`(+ 0` *X* `) =` *X*', etc.



```
select (> (intersection qout sout Q_CONTAINS_S) 0)
order by asc (+ (minus sin  qin S_CONTAINS_Q) (minus qout sout Q_CONTAINS_S) )
```

(a) User-defined selection and ranking function.

```
select (> (intersection qout sout OVERLAP) 0)
order by asc (minus qout sout OVERLAP)
```

(b) Generated code for inner nodes.

Figure 10: Backward chaining.

5 Conclusions

Process mediation can take advantage of process composition, which targets the automated composition of web services to compose automatically web services together in order to achieve a new functionality. Two kinds of composition are possible: functional-level and process-level composition. The first one selects services according to a goal and creates workflows with basic interaction scheme. The second one handles the protocols of the different services involved in order to obtain an executable composed service.

The deliverable presents our specific view on service composition and discusses thoroughly efficient implementation techniques for composition in an open environment populated by a large number of services. This requires a highly optimized interaction between large-scale directories and service composition engines. The presented directory service addresses this need with special features for service composition: Indexing techniques allowing the efficient retrieval of (partially) matching services, incremental data retrieval, as well as user-defined selection and ranking functions to support application-specific search heuristics within the directory.

The results achieved in this deliverable will be further elaborated in the DIP work package 4.12a.

REFERENCES

[Aoki1998] P. M. Aoki. Generalizing “search” in generalized search trees. In Proc. 14th IEEE Conf. Data Engineering, ICDE, pages 380–389. IEEE Computer Society, 23–27 1998.

[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.

[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.

[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.

[Cimpian et al. 2005] E. Cimpian, C. Drumm, M. Stollberg, I. Constantinescu, L. Cabral, J. Domingue, F. Hakimpour and A. Kiryakov. Report on the State-of-the-Art and Requirement Analysis (WP5 – Service Mediation). DIP Deliverable 5.1, http://dip.semanticweb.org/, 2005.

[DS04] Mike Dean and Guus Schreiber, editors. *OWL Web Ontology Language Reference*. 2004. W3C Recommendation 10 February 2004.

[FPV98] Alfonso Fuggetta, Gian Pietro Picco, and Giovanni Vigna. *Understanding Code Mobility*. IEEE Transactions on Software Engineering, 24(5):342–361, May 1998.

[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.

[KHW95] Nicholas Kushmerick, Steve Hanks, and Daniel S. Weld. *An algorithm for probabilistic planning*. Artificial Intelligence, 76(1–2):239–286, 1995.

[Lara et al. 2005] R. Lara, W. Binder, I. Constantinescu, D. Fensel, U. Keller, J. Pan, M. Pistote, A. Polleres, I. Toma, P. Traverso, M. Zaremba. Semantics for Web Service Discovery and Composition. Knowledge Web Deliverable 2.4.2, http://knowledgeweb.semanticweb.org/, January 2005.

[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.

[McD98] Drew McDermott. *The planning domain definition language manual*. Technical Report 1165, Yale Computer Science, 1998.

[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.

[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.

[PRT05] M. Pistore, P. Roberti, and P. Traverso. Process-Level Composition of Executable Web Services: "On-thefly" Versus "Once-for-all" Composition. The Semantic Web: Research and Applications. Proceedings of the second European Semantic Web Conference, ESWC 2005, Heraklion, Crete, Greece, May/June 2005. LNCS 3532, Springer Verlag, Heidelberg, Germany.

[PTB05] M. Pistore, P. Traverso, P. Bertoli. Automated Composition of Web Services by Planning in Asynchronous Domains. The Fiftteenth Interational Conferece on Automated Planning and Scheduling (ICAPS2005), June 5-10, Monterey, California, USA.

[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.

[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, M. Pistore. Automated Composition of Semantic Web Services into Executable Processes. Third International Semantic Web Conference (ISWC2004), November 9-11, 2004, Hiroshima, Japan.

[W3C] W3C. *XML Schema Part 2: Datatypes*, http://www.w3.org/tr/xmlschema-2/.

[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.