DIP
Data, Information and Process Integration with Semantic Web Services

FP6 – 507483

Deliverable

D4.12
Goal-oriented SWS composition specification

Patrick Albert
Walter Binder
Ion Constantinescu
Christian de Sainte Marie
Laurent Henocque
Mathias Kleiner

January 13, 2006
EXECUTIVE SUMMARY

1. Summary
This deliverable and its annexed documents and archives detail the specification of a prototype for a tool achieving automated goal directed composition of Semantic Web Services (SWS). The document and its annexes are intended to allow project partners to document, develop and specify DIP components that may depend in any way from SWS composition.

2. Technical contribution to the project
This work is an early specification step towards a first version of a composer for the DIP project. As an helper application embedded within the WSMO Studio editor, this tool aims to achieve the entire range of functionality required to make SWS composition useful:

- edit composition requests by selecting goals representing target atomic DIP SWS
- formulate a composition request by optionally\textsuperscript{1} telling how input/output messages relate, possibly involving constraints explicitly formulated: for instance when some data is the aggregated sum of others (as is frequent in composition in the presence of cost data).
- run the composer to produce an orchestration satisfying the above requirements, and correct with respect to the choreographies of all participant SWS, by using the DIP discovery module or, optionally, a specialized, efficient indexing directory which we call Hotblu in this document.
- extract a choreography from the resulting orchestration
- invoke a module translating choreographies and orchestrations to ASMs that can potentially be emulated by WSMX
- publish the results as a fully functional DIP SWS,

3. Scientific contribution
The planned implementation specified in the current deliverable is based upon an entirely original scientific contribution to the project. On the one hand, the optional Hotblu directory exploits a translation from types to integers to allow for high speed classification reasoning. On the other hand the option to treat workflow composition as a finite model search (enumerative) problem has first been experimented earlier in the same project (deliverable D4a.9), and will be taken to a fully operational level in D4.15. The choice of using a constraint based configurator to achieve those aims is also completely original.

4. Interactions within DIP
This deliverable is influenced by the WSMO specification for choreography and orchestration, as documented in the deliverables D3.4 and D3.5 and their common annex DIO. Most notably must we mention the following elements:

\textsuperscript{1}The composer can propose answers even when the request is totally underspecified, in which case of course non satisfactory answers may arise.
• a WSMO ontology for workflows, that implements the chosen subset for UML2 activity diagrams. This ontology allows for attaching explicit workflow related data to DIP SWS. This information can then be obtained/stored using WSMO4J² by tools like the currently specified composer.

• a framework for implementing choreography and orchestration ASMs, as well as a tool to convert from WSMO workflows down to valid WSMX ASMs allowing to emulate (semi) automatically generated SWS.

Also, the composer extensively uses mediators, and thus depends on issues and APIs concerning them.

5. Target audience
The deliverable is of particular interest to DIP participants involved in Choreography, Orchestration, Discovery, Process Mediation, Data Mediation, Use Cases.

²WSMO4J: http://wsmo4j.sourceforge.net
**DOCUMENT INFORMATION**

<table>
<thead>
<tr>
<th>IST Project Number</th>
<th>Acronym</th>
<th>DIP</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP6 – 507483</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Full Title</th>
<th>Project URL</th>
<th>Document URL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data, Information, and Process Integration with Semantic Web Services</td>
<td><a href="http://dip.semanticweb.org/">http://dip.semanticweb.org/</a></td>
<td>Daniele Rizzi</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Deliverable Number</th>
<th>Work Package Number</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.12</td>
<td>4</td>
<td>Goal-oriented SWS composition specification</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Date of Delivery</th>
<th>Contractual</th>
<th>Actual</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>M24</td>
<td>31-Dec-05</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Status</th>
<th>Nature</th>
<th>Dissemination Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>version 0.5</td>
<td>prototype</td>
<td>public</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Authors (Partner)</th>
<th>Resp. Author</th>
<th>E-mail</th>
<th>Partner</th>
<th>Phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>Albert, de Sainte Marie, Henocque, Kleiner (ILOG), Binder, Constantinescu (EPFL)</td>
<td>Patrick Albert</td>
<td><a href="mailto:palbert@ilog.fr">palbert@ilog.fr</a></td>
<td>ILOG</td>
<td>+33 (1) 49-08-35-00</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Abstract (for dissemination)</th>
<th>Keywords</th>
</tr>
</thead>
<tbody>
<tr>
<td>This deliverable is for internal usage of the consortium. It specifies a prototype of a tool for goal oriented SWS composition.</td>
<td>composition, configuration, constraint, policy, finite model, search, semantic web service, web service, workflow, modeling</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Version Log</th>
</tr>
</thead>
<tbody>
<tr>
<td>Issue Date</td>
</tr>
<tr>
<td>09-10-05</td>
</tr>
<tr>
<td>11-10-05</td>
</tr>
<tr>
<td>09-11-05</td>
</tr>
<tr>
<td>10-11-05</td>
</tr>
<tr>
<td>11-11-05</td>
</tr>
<tr>
<td>18-11-05</td>
</tr>
<tr>
<td>21-11-05</td>
</tr>
<tr>
<td>22-11-05</td>
</tr>
<tr>
<td>23-11-05</td>
</tr>
<tr>
<td>23-11-05</td>
</tr>
<tr>
<td>28-11-05</td>
</tr>
<tr>
<td>28-11-05</td>
</tr>
<tr>
<td>28-11-05</td>
</tr>
<tr>
<td>28-11-05</td>
</tr>
<tr>
<td>29-11-05</td>
</tr>
<tr>
<td>30-11-05</td>
</tr>
<tr>
<td>05-12-05</td>
</tr>
</tbody>
</table>
## PROJECT CONSORTIUM INFORMATION

<table>
<thead>
<tr>
<th>Partner</th>
<th>Acronym</th>
<th>Contact</th>
</tr>
</thead>
</table>
| National University of Galway | NUIG | Dr. Sigurd Harand  
Digital Enterprise Research Institute  
(DERI)  
National University of Ireland, Galway  
Galway  
Ireland  
E-mail: sigurd.harand@deri.org  
Tel: +353 91 495112 |
| 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  
E-mail: 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 Fédérale de Lausanne  
Bât. PSE-A  
1015 Lausanne, Switzerland  
E-mail: Karl.Aberer@epfl.ch  
Tel: +41 21 693 4679 |
| Essex County Council | Essex | Mary Rowlatt,  
Essex County Council,  
PÓ Box 11, County Hall, Duke Street,  
Chelmsford, Essex, CM1 1LX,  
United Kingdom.  
E-mail: 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  
E-mail: abecker@fzi.de  
Tel: +49 721 96540 |
<table>
<thead>
<tr>
<th>Company</th>
<th>Address</th>
<th>Contact</th>
</tr>
</thead>
<tbody>
<tr>
<td>Institut für Informatik,</td>
<td>IFI</td>
<td>Prof. Dieter Fensel</td>
</tr>
<tr>
<td>Leopold-Franzens Universität Innsbruck</td>
<td>University of Innsbruck Technikerstr. 25 A-6020 Innsbruck, Austria</td>
<td>Institute of computer science</td>
</tr>
<tr>
<td></td>
<td></td>
<td>E-mail: <a href="mailto:dieter.fensel@uibk.ac.at">dieter.fensel@uibk.ac.at</a></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Phone: +43 512 507 6486</td>
</tr>
<tr>
<td>ILOG SA</td>
<td>ILOG</td>
<td>Christian de Sainte Marie</td>
</tr>
<tr>
<td></td>
<td></td>
<td>9 Rue de Verdun, 94253, Gentilly, France</td>
</tr>
<tr>
<td></td>
<td></td>
<td>E-mail: <a href="mailto:csma@ilog.fr">csma@ilog.fr</a></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Tel: +33 1 49082981</td>
</tr>
<tr>
<td>inubit AG</td>
<td>inubit</td>
<td>Torsten Schmale, inubit AG, Lützowstraße 105-106 D-10785 Berlin,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Germany</td>
</tr>
<tr>
<td></td>
<td></td>
<td>E-mail: <a href="mailto:ts@inubit.com">ts@inubit.com</a></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Tel: +49 30726112 0</td>
</tr>
<tr>
<td>Intelligent Software Components, S.A.</td>
<td>iSOCO</td>
<td>Dr. V. Richard Benjamins, Director R&amp;D</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Intelligent Software Components, S.A. Pedro de Valdivia 10</td>
</tr>
<tr>
<td></td>
<td></td>
<td>28006 Madrid, Spain</td>
</tr>
<tr>
<td></td>
<td></td>
<td>E-mail: <a href="mailto:rbenjamins@isoco.com">rbenjamins@isoco.com</a></td>
</tr>
<tr>
<td>NIWA WEB Solutions</td>
<td>NIWA</td>
<td>Alexander Wahler</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NIWA WEB Solutions, Niederacher &amp; Wahler OEG, Kirchengasse 13/1a A-1070</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Wien</td>
</tr>
<tr>
<td></td>
<td></td>
<td>E-mail: <a href="mailto:wahler@niwa.at">wahler@niwa.at</a></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Tel.: +43(0)1 3195843-11</td>
</tr>
<tr>
<td>The Open University</td>
<td>OU</td>
<td>Dr. John Domingue</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Knowledge Media Institute, The Open University, Walton Hall, Milton</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Keynes, MK7 6AA, UK</td>
</tr>
<tr>
<td></td>
<td></td>
<td>E-mail: <a href="mailto:j.b.domingue@open.ac.uk">j.b.domingue@open.ac.uk</a></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Tel.: +44 1908 655014</td>
</tr>
<tr>
<td>SAP AG</td>
<td>SAP</td>
<td>Dr. Elmar Dorner</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SAP Research, CEC Karlsruhe</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SAP AG Vincenz-Priesnitz-Str. 1 76131 Karlsruhe, Germany</td>
</tr>
<tr>
<td></td>
<td></td>
<td>E-mail: <a href="mailto:elmar.dorner@sap.com">elmar.dorner@sap.com</a></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Tel: +49 721 6902 31</td>
</tr>
<tr>
<td>Sirma AI Ltd.</td>
<td>Sirma</td>
<td>Atanas Kiryakov, Ontotext Lab, - Sirma AI EAD, Office Express IT Centre,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3rd Floor 135 Tzarigradsko Chaussee, Sofia 1784, Bulgaria</td>
</tr>
<tr>
<td></td>
<td></td>
<td>E-mail: <a href="mailto:atanas.kiryakov@sirma.bg">atanas.kiryakov@sirma.bg</a></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Tel.: +359 2 9768 303</td>
</tr>
<tr>
<td>Company</td>
<td>Contact Information</td>
<td></td>
</tr>
<tr>
<td>------------------------------</td>
<td>--------------------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>Unicorn Solution Ltd.</td>
<td>Jeff Eisenberg&lt;br&gt;Unicorn Solutions Ltd&lt;br&gt;Malcha Technology Park 1&lt;br&gt;Jerusalem 96951, Israel&lt;br&gt;E-mail: <a href="mailto:Jeff.Eisenberg@unicorn.com">Jeff.Eisenberg@unicorn.com</a>&lt;br&gt;Tel.: +972 2 6491111</td>
<td></td>
</tr>
<tr>
<td>Vrije Universiteit Brussel</td>
<td>Carlo Wouters&lt;br&gt;Starlab- VUB&lt;br&gt;Vrije Universiteit Brussel&lt;br&gt;Pleinlaan 2, G-10&lt;br&gt;1050 Brussel, Belgium&lt;br&gt;E-mail: <a href="mailto:carlo.wouters@vub.ac.be">carlo.wouters@vub.ac.be</a>&lt;br&gt;Tel.: +32 (0) 2 629 3719</td>
<td></td>
</tr>
</tbody>
</table>
# Table of Contents

1 Introduction
   1.1 Scope ................................................. 1
   1.2 Context for goal oriented SWS composition .............. 2
   1.3 Brief introduction to configuration ...................... 4
   1.4 From configuration to workflow composition .......... 5
   1.5 Related work ........................................... 6
      1.5.1 Describing Services ............................... 6
      1.5.2 Service Discovery ................................. 7
      1.5.3 Service Composition ............................... 8
      1.5.4 Execution of Composite Services .................. 8

2 Problem statement and requirements 13
   2.1 Prototype objectives .................................... 13
   2.2 Requirements .......................................... 13
   2.3 Restrictions ............................................ 13
   2.4 Operating conditions ................................... 14

3 Prototype Validation ........................................ 15

4 Composer operation .......................................... 16
   4.1 Use case diagrams ...................................... 16
      4.1.1 Phase 1: Editing of a composition goal ............ 16
      4.1.2 Phase 2: Semantical discovery and matching candidate SWS
            from subgoals present in the composition directive ... 16
      4.1.3 Phase 3: Automatic generation of an orchestration ... 17
      4.1.4 Phase 4: Publishing .................................. 18

5 Indexing Service Descriptions ................................ 19
   5.1 Representing Service Descriptions for Indexing ........ 19

6 An UML2AD subset for expressing choreographies and orches-
   tration ......................................................... 23

7 Goal composer input .......................................... 24
   7.1 Context .................................................. 24
   7.2 Atomic goals ............................................ 24
   7.3 Composition goals ...................................... 24
   7.4 A metamodel for composition goals ....................... 25
      7.4.1 composition goal constraints ....................... 25
   7.5 About using Z: preliminary ................................ 26
   7.6 Z counterpart of the UML class diagrams ................ 27
      7.6.1 Classes ............................................... 27
      7.6.2 Relations ............................................ 27
   7.7 Additional metamodel constraints ........................ 28
      7.7.1 UnaryConstraint constraints ......................... 28
      7.7.2 Relational constraints ................................ 28
7.7.3 Functional constraints ................................................. 29
7.8 An example of a composition goal instance ......................... 29

8 Goal composer output .................................................. 32
8.1 Automatic publication to DIP ........................................... 32
8.2 Automatic extraction of the choreography from the orchestration .......................... 32
8.3 Automatic generation of a goal representing the composition ..................... 33
8.4 Output Example .......................................................... 33

9 Conclusion and references ................................................ 36

10 Annex – Indexing Directory .............................................. 43
10.1 Flexible Directories ...................................................... 43
10.1.1 Flexible Selection and Ranking of Service Descriptions ............ 43
10.1.2 Directory Query Language Requirements .......................... 46
10.1.3 Directory Query Language Grammar ................................ 46
10.1.4 Example Queries ..................................................... 49
10.2 Efficient Directories ...................................................... 49
10.2.1 Multidimensional Indexing for Service Directories ................. 50
10.2.2 Efficient Propositional Inference .................................... 52
10.2.3 Two Bound Search Trees ............................................. 54
10.2.4 Best-First Search ..................................................... 55
10.2.5 Query Transformation ................................................ 56
10.2.6 Relaxation of Set Inclusions ........................................ 57
1 Introduction

This deliverable specifies a prototype goal oriented SWS composer for DIP based upon constraint programming technologies, an unambiguous subset of UML2 activity diagrams as a choreography and orchestration language, and scalable classification based matching.

1.1 Scope

We place ourselves in the scope of automatic or computer aided goal oriented SWS composition, with immediate applications to Business Process Modeling or the Semantic Web. The basic assumptions for composing SWS is that there exists a form of directory listing of SWS that document their choreography using the workflow ontology defined in D3.4, as well as a directory listing of transformations that are usable to mediate between workflows having incompatible message type requirements. How and when a proper list of elementary workflows and transformations can be produced is beyond the scope of this specification, and is treated as if it was available to the program from the start. In other words, the current prototype does not cover the functionalities involved in helping an end user to produce, document and/or publish the related elements. To clarify this issue, since the use made of WSML in DIP does not require an executable SWS to document its choreography as a workflow, nor to propose mediators to ontologies used by other (potentially interoperable) SWS, the availability of these elements require extra publication effort. The prototype specified here will hence populate a test set of accurately workflow documented SWS.

We also assume that the composition process is goal oriented in the DIP sense: a user may list the message types he can possibly input to the system (e.g. credit card number, expiry date, budget, yes/no answer etc...) involving restrictions on their types and attributes, and the same user may formulate the precise (set of) message(s) that must be output by the system (e.g. a plane ticket reservation electronic confirmation: his “objective”), again involving restrictions on their types and attributes. More accurately, in the proposed specification, the user may formulate a composition request by manipulating, organizing and placing constraints on abstractions of such messages and SWS. The binding to actual SWS and messages will either be performed by the program or be left open for runtime (lazy) evaluation, whereby the composer generates what is known to DIP as an abstraction of goals.

In order to be sufficiently self contained, the present introduction borrows some material from the articles [2] and [3]. A reader aware of configuration techniques and of configuration applied to web service (WS) composition may skip this part or read it quickly. A previous work [2] proved the feasibility of using a configurator program to solve this problem, and presented a constrained object model adequate for this purpose, using the semi-formal language UML/OCL. The current specification uses a Z based specification of the constrained object system involved in the configurator, as was proved feasible in [3]. An essential part of the workflow metamodel and constraints exploited by the current composer were inserted in the D3.4/D3.5 annex called DIO, because they strongly impact on other ontology related issues, most notably the conversion to WSMX compliant choreography and orchestration ASMs and to the Cashew language now developed at Open University (OU).
A rationale for the use of both UML2AD, and of Z to specify object model constraints, is presented in the DIO annex. Concerning UML2AD, the core arguments are a strong market pull and the acknowledged high level of support of workflow patterns in UML2AD. Concerning Z, in a few words, although indeed it was shown in [23] that the use of UML/OCL is generally viable, the language is also known as having limitations, notably concerning relational operators and recursive structures. Z was shown suitable for such a usage in [3, 33], via a framework for the Z specification of constrained object models\(^1\). This research heads towards the complete formal specification of a constrained object model for workflow composition.

In this document we also describe techniques to efficiently index a large number of service advertisements within a directory. The service advertisements are treated as multi-dimensional data and the directory leverages a balanced search tree for multi-dimensional data. A flexible query language allows to express complex matching relations between a given service request and the service advertisements stored in the directory. A service composer may dynamically obtain relevant service advertisements by issuing queries to the directory. This approach improves scalability, because the service composer does not have to keep copies of a potential large number of service advertisements, but it can retrieve only matching service advertisements on demand.

### 1.2 Context for goal oriented SWS composition

We consider SWS choreographies and orchestrations defined using a variant of extended workflow nets, as are UML2 activity diagrams [30] or the YAWL language [66]. In order to adapt to market trends, the choice made focused on a relevant subset of UML2 Activity Diagrams as a language for describing choreographies and orchestrations. The underlying semantic model is that of colored\(^2\) Petri nets, restricted to match “traverse-to-completion” semantics\(^3\). UML diagrams generally receive poor acceptance from the scientific community, because of patent ambiguities, both at the syntactic and semantic level. By isolating a useful subset of UML2AD, we have the possibility to wipe out all ambiguities. At the syntactic level, this is achieved by taking the appropriate decisions wherever needed, and documenting them by constraints (using the Z mathematical (relational) language). At the semantic level, the translation from activity diagrams to abstract state machines results in applying unambiguous operational semantics to the chosen subset. We do not need however to consider here the operational semantics of the workflow language, since we solely need to compose them syntactically. We hence focus on the properties of the corresponding metamodel. Indeed, we treat workflow composition as the process of connecting input and output message flows to preexisting or added workflow items, like fork, join nodes or auxiliary user input handling actions. Hence the only element retained for composition are the structural properties of argument workflows, messages, and transformations. We do not need to emulate workflows in any case, but can however formulate some constraints that to some extent guarantee the viability of the result (some constraints guarantee that a composite workflow will not be subject to starvation).

\(^1\)Also called Object Oriented Constraint Programs
\(^2\)In a CPN (colored Petri net), messages (tokens) have types.
\(^3\)Essentially, these semantics as introduced in the UML superstructure aim at preventing flow starvation in case of token competition.
The same general context is envisioned in several research communications [74, 18, 48]. As an archetypal problem, the “producer-shipper” use case which originates from [48] was selected as a reference use case in the D3.4 and D3.5 ontology deliverables. The problem there is to compose a valid workflow from a producer workflow and a shipper workflow. One difficulty is that the execution of both workflows must be interleaved. Briefly stated (a more thorough description comes later in the deliverable), the producer outputs results that must be fed into the shipper so that both their “offers” can be aggregated and presented to the user. This inter-connection remains unknown to the external user. Experimental evidence on the possibility to address this problem using constraint based configuration was published in [3]. The producer-shipper example is interesting because:

- both the shipper and the producer make an offer corresponding to the user request, which are aggregated to make a global offer that may later be accepted or rejected by the user. Note that the shipper needs input data from the producer to build its offer,
• both the producer and the shipper are specified using full fledged partial\textsuperscript{4} workflows, and do not simply amount to simple isolated activities,

• the two workflows cannot be executed one after the other, but they must be interleaved, as each one must wait for the other offer to obtain an OfferAcceptance and therefore complete the transaction,

• the ShipperWorkflow needs a size as input, which can only be obtained by extraction (i.e. transformation) on the ProducerOffer,

• finally, the goal is decomposed into two sub-goals: the producer and the shipper order confirmations.

Figures 1.1 and 1.2 both illustrate the shipper’s partial workflow\textsuperscript{5}, in two different cases:

• a/ when the user answer is “packed”, meaning that the same answer includes both “ok”, “notok” details,

• b/ when two separate messages treat the “ok”, “notok” cases

In the first case, the shipper SWS behaves as a two operation Web Service (WS), whereas in the latter case the shipper Semantic Web Service (SWS) behaves as a three operation WS. The composition results are shown in Figures 1.3 and 1.4 for both the “packed” vs. “unpacked” cases. This composed workflow involves synchronization, interleaving, mediators (we use a stereotype for those) and it should be noted that some execution paths lead to a point where the goal cannot be fulfilled. In these figures, all the workflow elements that do not belong to the shipper workflow above or its producer counterpart are introduced automatically in order to obtain a valid composition.

1.3 Brief introduction to configuration

A configuration task consists in building (a simulation of) a complex product from components picked from a catalog of types. Neither the number nor the actual types of the required components are known beforehand. Components are subject to relations, and their types are subject to inheritance relationships. Constraints (also called well-formedness rules) generically define all the valid products. A configurator expects as input a fragment of a target object structure, and expands it to a solution of the configuration problem, if any. This problem is semi-decidable in the general case.

A configuration program is well described using a constrained object model in the form of a standard class diagram (as illustrated by the simplified UML2AD metamodel fragment in Figure 1.5), together with well-formedness rules or constraints. Technically solving the associated enumeration problem can be made using various formalisms or technical approaches: extensions of the CSP paradigm \cite{42, 24}, knowledge based approaches \cite{60}, terminological logics \cite{45}, logic programming (using forward or backward

\textsuperscript{4}Here, “partial” means that the diagram is not complete in the UML sense, since it lacks actual connections to the external participants. Normally, a diagram where input/output pins aren’t connected is not valid.

\textsuperscript{5}The producer’s workflow is similar, modulo the message type ontologies.
chaining, and non standard semantics) [57], object-oriented approaches [38, 60]. Our experiments were conducted using the object-oriented configurator Ilog JConfigurator [38].

There currently exists no universally accepted language for specifying constrained object models. The choice of UML/OCL is advocated [23], and is realistic in many situations, but has some drawbacks due to a number of weaknesses. As shown in [3, 33] the Z relational language has enough expressive power and extensibility to properly address the task of specifying a constrained object model, without requiring to use an ad hoc object oriented extension of Z.

UML diagrams mention some of the model constraints, most notably relation cardinalities, as e.g. that a message relates to at most one Activity in Figure 1.5, but there is no possibility to graphically cover the whole range of constraints that may occur in an object model.

1.4 From configuration to workflow composition

Configuration emerges as an AI technique with applications in many different areas, where the problem can be formulated as the production of a finite instance of an object model subject to constraints. Reasoning about workflows falls into this category, because a workflow description is an instance of a given metamodel (as is the UML metamodel for activity diagrams [30]). Composing workflows is a configuration problem in that in so doing, one must introduce an arbitrary number of previously non existent transitions (fork, join, split, merge, transformations, pre-defined user-interactions sequences), and interconnect input and output message pins provided they have compatible types. An SWS composition system in our sense expects:

- a list of potentially usable workflows, implemented in the form of partially defined instances of the workflow metamodel used (e.g. a producer for some good, a shipper, a payment service, etc.). When workflow composition is performed with the aim of composing web services, the input workflows stem from the SWS choreography descriptions.

- the ontologies for the data types linked to the input/output messages present in the workflows (e.g. types of journeys: by train, plane, etc. , methods of payment: cheque, credit card, cash etc.).

- a goal to be satisfied by the result composition (e.g. a train ticket reservation), plus arbitrary extraneous constraints applying to the result. In the context of the current specification, the goal composition request may mention constraints that map to the presence of predefined mediators in the solution (as e.g. data aggregation (packing or unpacking), data transformation, summing up, etc.)

- a list of the data input the end user can provide (e.g. simple “yes/no” answers, credit card number, expiry date, selected item etc.)

In our approach the simplest form of a composition request is defined as a single message, according to an appropriate / specific ontology, connected to the final node of the composite workflow.

The user of a workflow composition system expects in return for his input a complete “composite” workflow, that interleaves the execution of several of the elementary
argument workflows, while ensuring that all possible integrity constraints remain valid. Among such constraints are those that stem from the metamodel itself: for instance some constraints state that two or more workflows should not be inter-blocking, all waiting for some other to send a message. Other constraints are more problem specific, like those stating for instance that an item being shipped is indeed the one that was produced.

1.5 Related work

Automated workflow composition is a field of intense activity, with applications to at least two wide areas: Business Process Modeling and the (Semantic) Web Services. Tentative techniques to address this problem are experimented using many formalisms and techniques, among which Situation calculus [39], Logic programming [55], Type matching [13], Coloured Petri nets: [74, 18], Linear logic: [51], Process solving methods [7, 28, 62], AI Planning [11], Hierarchical Task Network (HTN) planning [56, 68], Markov decision processes [22].

1.5.1 Describing Services

Currently, the de-facto industrial standard for describing services is the Web Service Description Language (WSDL) [70]. A WSDL description is formulated as an XML document which describes a Web service. Such a description specifies the location of the service, the operations that it exposes, the data expected to carry the exposed operations, the results that are delivered after the execution ends (typically using XML schema data types), and the communication protocols and transport it supports. The main drawback of WSDL is that it addresses the description of services only at a syntactic level.

OWL-S is a language that provides the means to mark-up Web services describing their capabilities and properties in an unambiguous, computer-interpretable way [46]. It is based on the Ontology Web Language OWL [69]. The aim of OWL-S is to make Web services computer-interpretable, enabling their automated use. OWL-S relies on WSDL in order to facilitate the syntactic description of services. OWL-S uses WSDL to describe knowledge about a service in terms of what the service does (represented by messages exchanged across the wire between service participants) and how it does it. From version 0.9 OWL-S is built on OWL (Web Ontology Language), while previous releases were based on DAML+OIL.

WSDL-S [1] aims at providing a lightweight approach for creating semantic Web service descriptions. The approach is considered lightweight because:

• it provides simple extensions to WSDL to add semantics, thereby allowing semantic descriptions of Web services;

• the WSDL-S service ontology is designed to be more aligned with WSDL 2.0 and more compact than OWL-S, without losing significant expressivity;

• it allows integration of semantic and non-semantic descriptions of Web services.

The Web Service Modeling Ontology (WSMO) [72] is a formal ontology and language for modeling Web services, aiming at providing the facilities required by its usage.
process, namely discovery, invocation, composition, execution, monitoring, mediation, and replacement (in WSMO terms: compensation). All these activities are to be supported by WSMO through logical inference-mechanisms, because of the well-known competences of suchlike techniques. In order to allow suitable logic-based reasoning on Semantic Web Services (SWS), the description language has to provide reasonable expressiveness for describing relevant aspects of the services, together with well-defined formal semantics that support effective reasoning. WSMO applies F-Logic [35] as the underlying logical formalism, since it provides a standard model theory, it is a full first order logic language, it provides second order syntax while staying in the first order logic semantics, and it has minimal model semantics.

One of the important initiatives in Semantic Web Services (SWS) area is the Semantic Web Services Initiative (SWSI) [61]. The members of this initiative have identified two major aspects regarding SWS: the architecture to support Semantic Web Services (SWSA) and the language to specify information about Semantic Web Services (SWSL).

SWSL is built upon OWL-S and therefore the main features of SWSL are very similar to those of OWL-S. Using OWL-S as reference point, SWSL can be used to specify information about Web services, but not in a complete manner.

While WSDL lacks rich semantics, the reasoning procedures required by formalisms like OWL-S, WSDL-S, and WSMO are usually computationally expensive. Hence, in our approach we use a formalism that can be seen as a subset of OWL-S, WSDL-S, and WSMO. Our formalism provides enough expressivity so that on one hand, the discovery and composition processes are supported well enough, and on the other hand, the required computational complexity is reduced.

### 1.5.2 Service Discovery

UDDI has become the de-facto standard to provide a general framework to describe and discover services and Web service providers. More specifically, [65] is a technical note that details how WSDL descriptions of web services can be mapped to UDDI data structures and provides examples of how one could find web service descriptions using the standard UDDI query interface.

Within the academic world, a number of approaches exist that try to build semantically enhanced discovery components on top of UDDI. [34] augments the standard UDDI registry APIs with semantic annotations. [67] uses a set of distributed UDDI registries as a storage layer, where each registry is mapped to a specific domain based on a registry ontology.

A very active field in research is the development of discovery algorithms. There, the main focus is on finding good “approximations” when a perfect match cannot be found, i.e., the advertised capabilities of a Web service do not fully match the request. [47] defines an ordered scoring function based on input and output parameters.

A similar approach is taken in [37] and applied to concept-based reasoning in Description Logics: Here, the above categorization is extended by an intersection match, where the intersection of the concepts of the request and the concepts of the advertisement is satisfiable. This intersection match is ranked below a subsumed match.

[5] describes an interesting different approach where discovery is treated as a query rewriting problem on hypergraphs: It attempts to find the so-called best profile cover...
which is defined as the set of web services that satisfies as much as possible the outputs of the request while requiring as few non-requested inputs as possible.

While the above approaches to discovery allow for a number of different match types to be supported by the directory, the query formalism is quite coarse and in particular lacks support for a good integration of the discovery process with composition, in particular concerning the formulation of composition search states as queries.

Also current directory services do not explicitly support concurrency which can be critical for the correctness of the composition algorithms in open environments where a directory may be concurrently accessed by a large number of clients.

1.5.3 Service Composition

Service composition has been an active field in AI and database research communities for several years. Thakkar and Knoblock [63] presented an approach, where a number of manually defined data sources, such as the Microsoft Terraservice, the U.S. Census Bureau information files, as well as geocoding information and different real estate property tax sites, were composed using a forward chaining technique.

Some approaches to composition require an explicit specification of the control flow between basic services in order to provide value-added services. For instance, in the eFlow system [12], a composite service is modeled as a graph that defines the order of execution of different processes. The Self-Serv framework [6] uses a subset of statecharts to describe the control flow within a composite service. The Business Process Execution Language for Web Services (WS-BPEL) [8] addresses compositions where the control flow of the process and the bindings between services are known in advance.

There is a good body of work which tries to address the service composition problem by using planning techniques based either on theorem proving (e.g., ConGolog [40, 41] and SWORD [49]) or on hierarchical task planning (e.g., SHOP-2 [73]). Such approaches do not require a pre-defined process model of the composite service and return a possible control flow as result.

As main drawback, all these approaches to composition based on planning assume that the relevant service descriptions are initially loaded into a reasoning engine and that no discovery is performed during composition. Lassila [36] addressed the problem of interleaving discovery and integration in more detail, but he considered only simple workflows where services had one input and one output. Constantinescu [17, 16] introduced service composition algorithms that access a specialized service directory to dynamically retrieve relevant service descriptions. The directory supports a dedicated query language that allows to express powerful matching relations, including also partial matches.

A different approach, known as ‘planning as model checking’ [27], is being considered for web service composition as well, and would allow more complex constructions such as loops [64].

1.5.4 Execution of Composite Services

There is a large amount of related work concerning the execution of services composed as workflows. For example, the AMOR system [31] used mobile agents for executing
ad-hoc processes. The internet indirection infrastructure i3 uses triggers to decouple sender and receiver [59].

In [44] the authors point out the inefficiencies of the centralized orchestration of BPEL4WS programs [8] by engines such as BPWS4J [9]. They describe an algorithm to decompose BPEL4WS programs for decentralized orchestration.

Another relevant approach is the SELF-SERV system [54]. The system architecture identifies three kinds of service: elementary, composite, and community. The execution of composite services is managed by coordinators.

A number of approaches based on Semantic Web Services (SWS) also already exists. Among others is the Web Services Execution Environment (WSMX), the reference implementation of WSMO [72], which aims at creating an execution environment for the dynamic discovery, selection, mediation, invocation, and inter-operation of Semantic Web Services.

Another SWS based approach is IRS (Internet Reasoning Server) [43] that has as components the IRS Server, the IRS Publisher, and the IRS Client. The IRS Server holds descriptions of SWS. IRS Publisher links Web services to their semantic descriptions within the IRS Server and generates a wrapper which allows a Web service to be invoked. An IRS user asks through the IRS Client for a task to be achieved, and accordingly the IRS Server selects and invokes an appropriate Web service. Recently, IRSIII provides infrastructure for creating WSMO-based SWS, building upon the previous implementation of IRS.

For OWL-S [46], there are a set of tools rather than a complete execution environment. For example, the OWL-S Editors helps maintaining OWL-S service descriptions, the OWL-S Matcher provides an algorithm for different degrees of matching for individual elements of OWL-S, the OWL-S Axis plugin advice service providers how to provide service description using OWL Services, etc.

METEOR-S [50] extends existing Web service standards, such as WSDL and BPEL4WS, with Semantic Web technologies to achieve greater dynamism and scalability. The main components of METEOR-S are the Abstract Process Designer, the Semantic Publication and Discovery Engine, the Constraint Analyzer, and the Execution Environment. The Abstract Process Designer allows a user to create the control flow of the process using BPEL constructs. The Semantic Publication and Discovery Engine provides semantic matching based on subsumption and property matching. The Constraint Analyzer dynamically selects services from candidate services, which are returned by the discovery engine. The Execution environment performs actual late binding of services returned by the Constraint Analyzer and converts abstract BPEL to executable BPEL.

Quality of service parameters and cost-measures for the execution of services composed as workflows are addressed by some of the previously mentioned approaches. Still, in the majority of cases, the procedure for establishing these costs requires some heavy coupling between the requesting entity and the sites that could possibly execute the composed application.

http://i3.cs.berkeley.edu/
Figure 1.3: The producer-shipper composed workflow - packed
Figure 1.4: The producer-shipper composed workflow - unpacked
Figure 1.5: Simplified meta-model for workflows activities
2 Problem statement and requirements

2.1 Prototype objectives

The prototype specified here is meant to be a helper application for use in the WSMO Studio [20, 21]. The program allows a user to design a composite web service from elementary web services by defining an abstract composition goal from more elementary sub-goals. The composer then automatically produces a valid workflow interleaving the execution of participant web services that match goals. The resulting composite WS can then be exported to WSMO.

The prototype composer should hence demonstrate the feasibility of "press-button" SWS composition in DIP, under rather unconstrained assumptions concerning SWS choreographies and orchestrations. Very significantly, the composer will have the possibility to design valid orchestrations involving Semantic Web services having non-trivial choreographies. For instance, the case when SWS choreographies involve multiple operation protocols, in the presence of internal or external choice, will be supported. Within the boundaries set by the chosen UML2AD subset, the program limitations are combinatorial.

2.2 Requirements

The prototype depends upon a number of DIP related elements

- The DIP Interface Description Ontology. This document annexed to the DIP D3.4 and D3.5 deliverables details all issues relevant to the core choreography and orchestration languages, as well as the WSMO workflow ontology. More precisely, the current prototype depends upon:
  - the UML2AD subset chosen to describe choreographies
  - the UML2AD subset chosen to describe orchestrations (this is the same sub-language)
  - the WSMO ontology based upon the previous language
  - the WSMO4J [19] tool for accessing WSMO ontologies
  - the DIP discovery interface
  - DIP publishing, and notably the automatic translation from UML2AD diagrams to ASMs coding for orchestration and choreographies.
  - the Hotblu directory (optional)
  - DIP data mediation

2.3 Restrictions

The prototype functionalities obey a number of restrictions:
• The graphical interface for editing composition goals will remain simple.

• There will be no specific interface for editing UML2AD choreographies and orchestrations as part of the tool. All existing UML editors provide means to export specifications in the XMI\(^1\) format. A separate WSMO Studio plugin could load such xmi files to populate WSMO choreographies and orchestrations using WSMO4J.

• The prototype composer will operate using a complete search algorithm. Considering the combinatorial complexity inherent to this problem, the scalability issues raised by the growing number of potential participant choreographies may be addressed solely by incomplete (local) search methods. This issue may be tackled in future evolution of the prototype but remain beyond the current program objectives.

2.4 Operating conditions

The prototype composer will run as a Java based WSMO Studio plugin, and will exploit ILOG JConfigurator constraint based configuration Java libraries.

\(^{1}\text{XMI: http://www.uml.org}\)
3 Prototype Validation

An evaluation of the composer prototype requires a repository of services. The services in the repository may cover the shipper-producer example presented before. An exemplary composition problem should involve 3 goals, and for each of them there should be at least 5 different matching services. Thus, in total, at least 15 services are needed in the repository. Populating the directory will be achieved by manually defining service advertisements that fit into the producer-shipper scenario.

Details concerning the producer-shipper scenario can be found in the following chapters:

- A possible composition goal, which is the client request, is presented in Chapter 7.
- SWS choreography diagrams were presented in the introduction before.
- Composite SWS orchestration and choreography are discussed in Chapter 8.

By default, the composer will retrieve the available service advertisements from the DIP directory. Alternatively, if an HotBlu directory is available, the composer may incrementally retrieve only relevant (matching) service advertisements from the HotBlu directory. The latter approach is beneficial only in large-scale settings. As in the absence of large-scale service repositories we can evaluate the composer prototype only within a small-scale application scenario, incremental retrieval of service advertisements from the HotBlu directory is not mandatory.
4 Composer operation

This chapter lays out the basic process of the composer, and presents all phases executed in the operation. The phases are explained in detail in their respective chapters.

4.1 Use case diagrams

Figure 4.1 illustrates the use case diagram for the prototype composer. The diagram involves four distinct actors:

1. a (human) user,
2. a goal repository
   This is a place where the building blocks for orchestrations are stored
3. the DIP repository or, optionally, the Hotblu directory
   Where Web services are registered
4. WSMX
   The WSML execution engine

Figure 4.2 illustrates the different phases involved in composition as well as the precise relation between them and the DIP components.

4.1.1 Phase 1: Editing of a composition goal

The composition goal is the request for the composer. It contains all necessary elements for composition and is defined in a precise way as an instance of the metamodel presented in Chapter 7. Those elements include: atomic goals to be achieved, relations between them, constraints/policies on their inputs/outputs and/or on their non-functional properties, information the user can provide, information/effects the user wants to be achieved. This phase is optional as the user could already have and thus provide a composition goal. A full description of these composition goals is given in Chapter 7.

4.1.2 Phase 2: Semantical discovery and matching candidate SWS from subgoals present in the composition directive

Using the composition goal defined in Phase 1, the composer will propagate all constraints on atomic subgoals and request the discovery component for each one in order to get a list of matching Web services. It will then retrieve the semantic description of those WS (capabilities and choreography).

The prototype will be designed to work with the DIP discovery component. Optionally, a Hotblu directory may be supported as well. Those two discovery modules have a sensible different approach: Hotblu requests a list of inputs/outputs of the Web service to be discovered, while DIP discovery requires a capability (which can be seen as a restriction to outputs/effects only). The composition prototype may support both types of requests: any of them can be easily extracted from the composition goal.

A description of the Hotblu directory is given in Chapter 5 and in the annex of this document. A full description of the DIP discovery process is given in D4a.8.
4.1.3 Phase 3: Automatic generation of an orchestration

Once Web services have been discovered to fulfill atomic goals, the composer will try to find a solution to the whole problem using as input:

- the constraints defined in the composition goal (relations between SWS, user possible inputs, atomic goals to be achieved, policies)
- the choreographies of the available SWS
- the constraints defined by the UML2AD meta-model
- the choice among several composition modes

The composer will build a complete valid workflow adding necessary elements such as mediators and control flow constructs, which guarantees that at least one path of execution fulfills all atomic goals (success path). Different composition modes can be applied at this level; for example, one could ask that non-success paths imply that no atomic goal is achieved. Those additional modes will not be covered in the prototype though.

From this complete workflow, the composer will then extract all information relevant to
the orchestration. In order to do this, all data flows targeting or coming from a SWS will now be expressed with SendEvent and AcceptEvent actions, in replacement of the choreographies. Those special actions contain a partnerID allowing to know which SWS is concerned, and a LinkedEvent specifying which event of the SWS choreography it refers to. The same process is applied to the user interactions which is simply seen as being another partner, although its choreography might be created on-the-fly by the composer (under predefined constraints).

### 4.1.4 Phase 4: Publishing

To be published, the SWS needs:

- a choreography. This is extracted from the complete workflow by isolating the communications with only one partner, which will most commonly be the user. It will then be exported using WSMO4J.

- a capability. This information is contained in the composition goal.

- non-functional properties. Some of them could be extracted from the orchestration, but this is out of the prototype’s scope. Therefore they will have to be added manually.
5 Indexing Service Descriptions

If there is a large number of available service descriptions, loading them all into the composer before starting reasoning would be inefficient. For performance reasons, it is better to maintain a dedicated directory for service composition that provides an index of the available service. The composer can dynamically query this specialized directory in order to retrieve only those service descriptions that are relevant for a particular composition problem.

Depending on the composition algorithm, the composer may either (1) query the directory once in the beginning of the composition, or (2) iteratively query the directory whenever a composition problem cannot be solved with the services discovered so far. As the composition prototype developed in D4.15 will be demonstrated using a fairly small repository of service descriptions, the prototype will follow the first approach, obtaining the necessary service descriptions only in the beginning. Therefore, the indexing directory described in this document, which we call Hotblur, is an optional component. While it is not required for the prototype, it is an important component for future, large-scale settings.

In this chapter we discuss how to represent services in order to index them in an efficient directory. As the indexing directory is an optional component, we present its features and implementation issues in the annex of this document.

5.1 Representing Service Descriptions for Indexing

A service description specifies aspects strictly related to the functionality available from a service provider or requested by a service consumer. We represent service advertisements and service requests through variables and constraints on these variables. Variables refer to required or provided service parameters or to aspects related to the state of the world before or after the invocation of the service. Constraints allow the representation of restrictions on the possible combinations of values that different variables can take.

In our formalism each variable is defined by two elements:

- A description specifying the actual semantics of the information that the variable is holding and its role in the current service description (e.g., in a travel domain the description of a parameter could be DepartureInput or ArrivalInput). Usually, the description is defined by the name of the variable itself.

- A type defining the way data of the variable is represented and the set of values that the variable can take (e.g., possible values for DepartureInput and ArrivalInput could be represented by the sets \{Geneva, Basel\} and \{Barcelona, Nice\} where all four city values could be of type Location).

We presume that both variable descriptions and types can be defined using a class/ontological language like OWL [69]. Primitive data-types used for specifying the variable type can be defined using a language like XSD [71].\(^1\) Moreover, WSML can be used to describe both concepts and XML datatypes [72].

\(^1\)At the implementation level both primitive data-types and classes are represented as sets of numeric intervals [15].
Since variable definitions (e.g., description/type) are usually shared between several service descriptions, they are linked in such descriptions by reference. We take an approach similar to OWL-S [46] where variables can be used either for describing service parameters or for specifying aspects related to the state of the world as represented by the execution engine. For defining parameters, the description of a variable can be specialized to either an \textit{input} parameter or an \textit{output} parameter. For specifying the world state, the description of a variable can be specialized to be either a \textit{prior} state or a \textit{post} parameter. \textit{Prior} variables refer to aspects of the world established prior to the invocation of the service; \textit{post} variables refer to aspects of the world affected by the invocation of the service.

Constraints on variables can specify either \textit{preconditions} (set of possible parameter and world state values required to be true prior to the invocation of the service) or \textit{effects} (how parameters and world states are affected by the execution of the service). Constraints are specified in the form of sets of possible variable assignments. Each assignment represents a set of variable/value pairs. Constraints are identified by keywords (e.g., \textit{PRE} for preconditions, respectively \textit{EFF} for effects).

To summarize, a service description has the following structure:

\begin{verbatim}
Variable Map
  Variable -> Set of Values

Constraint (Preconditions)
  Set of Assignment Map
    Variable -> Value

Constraint (Effects)
  Set of Assignment Map
    Variable -> Value
\end{verbatim}

\textbf{Figure 5.1: Structure of service descriptions}

In service advertisements variables and constraints describing parameter and world aspects, have the following semantics:

- In order for the service to be invokable, a value must be known by the service execution engine for each of the \textit{input} or \textit{prior} variables and it has to be consistent with the respective semantic \textit{description} and syntactic \textit{type} of the variable. The value provided as parameter \textit{input} or world state \textit{prior} has to be semantically more specific than what the service is able to accept. Regarding the variable 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 type of the variable.

- Upon successful invocation, the service returns a value for each of the \textit{output} parameters and the execution engine assigns a value to each of the \textit{post} variables representing a change in the world state. Each of these values is consistent with the respective \textit{description} and \textit{type} of the variable.
Regarding preconditions, in order for the service to be invokable, at least one assignment set in the constraint has to be satisfied by the current values of variables defining parameters and states of the world.

Effect constraints represent guarantees on the possible combinations of values for variables describing output parameters and post world states as well as how prior world states are maintained after the invocation of the service.

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

- The service request input variables represent available parameters (e.g., provided by the user or by another service). Prior variables describe aspects of the world specifying an initial state of facts. Each of these variables has attached a semantic description and either some description of its type or a concrete value.

- The service request outputs represent parameters that a compatible (composed) service must provide. Request posts specify aspects of the state of the world that have to be influenced by the execution of the service. The variable role defines the actual semantics of the required information and the variable type defines what ranges of values can be handled by the requester. A compatible (matching) service must be able to provide a value for each of the variables in the output and post of the service request, semantically more specific than the requested variable description, and having values in the range defined by the requested parameter type.

- Preconditions in a request represent restrictions on the combinations of possible values for available parameters in input or possible initial world states described by prior variables.

- Effects represent restrictions on the combinations of possible values for variables describing required output parameters or required final world state (posts) that the current request description is willing to accept.

For manipulating service advertisements or requests regarding their variables we introduce the following functions:

- \( \text{vars}(S) \) – returns the set of variables for an advertisement or request \( S \). For service descriptions introduced here we assume that the parameter role of any variable has been restricted to one of \( \text{IN}, \text{OUT}, \text{PRIOR}, \text{POST} \). We assume variables to be described as concepts using a language similar to [69] and accordingly the normal semantics for the DL operators \( \equiv, \subseteq, \cap, \cup, \bot, \top \).

- \( \text{type}(V, S) \) – returns the type of the variable named \( V \) in the frame of an advertisement or request \( S \) as the set of possible values that \( V \) can take. The \( \equiv, \subseteq \) and \( \cap \neq \emptyset \) operators in conjunction with this function can be used to determine if two value sets are equivalent, subsume each other or are overlapping. These set inclusion operators might possibly require subsumption tests (\( \subseteq \)) between set values when those are described in a high-level language like [69].

- \( \text{constraint}(\text{PRE} \mid \text{EFF}, S) \) – returns the set of possible variable assignments (variable/value pairs) for the precondition or effect constraint in the service description \( S \).
As in [46] we assume that any kind of service description ($S$), either advertisement or request, is well formed in that the description cannot have the same variables restricted to more than one parameter type. The rationale behind this assumption is that if a description had an overlap between variables of different types, it would only lead to two equally undesirable cases: either the two variables would have the same role in which case one of them would be redundant or they would have different roles in which case the service description would be inconsistent.
6 An UML2AD subset for expressing choreographies and orchestration

The subset of the UML2AD notation [30] chosen for expressing choreographies and composing orchestrations is fully described in the DIO annex of the D3.4/D3.5 deliverables (choreography/orchestration ontology respectively). There, a complete list of arguments detailing the rationale for the current choice of UML2AD is proposed. Briefly sketched however:

The rationale for having a static description of choreographies and orchestrations is that we are using a finite model search algorithm for composing. Choreographies and orchestrations are modeled as collections of interconnected objects, that are further interconnected by the composition mechanism, which also has the possibility to introduce required elements as for instance

- fork/join constructs to distribute/merge data to several participants,
- decision/merge nodes to account for choices,
- mediators from a large typology to adapt data by aggregating, extracting, transforming it.

The whole process can be achieved on the basis of explicit workflow data, and would have been impossible from raw ASM specification as available in “low level” DIP orchestrations and choreographies.

The rationale for using UML2AD as a language is its wide market acceptance and the fact that it almost fully supports the whole list of workflow patterns. We are aware (and have struggled with) a number of difficulties raised by ambiguities in the UML2 specifications. The current subset chosen solves these both at the denotational level, and the operational level.

At the denotational level, we are using a subset of the Z language [58, 33, 3] to document the diagram restrictions that we have chosen to enforce. Although these restrictions do not impair expressivity, they significantly enhance the rigor of the associated diagrams. The choice of Z for static constraints here echoes the choice of ASMs for dynamics. In fact, both languages are closely related, and Z allows for a fully formal and unquestionable specification of metamodel constraints, that replaces equivalent - but less readable - OCL statements [23].

At the operational level, the semantics of the selected UML2AD subset are defined by the translation to ASMs. An early version of this translation is specified in the D3.4 and D3.5 DIO annex as well, but will not be sketched here, since the current document solely relates to workflow construction issues, not emulation.
7 Goal composer input

7.1 Context

SWS compositions will be based on designer requirements which can be seen as constraints that the composed SWS has to fulfill. DIP SWS descriptions are defined using:

- Information on what the SWS performs (post-conditions and output messages) and what is required to achieve it (pre-conditions and input messages). This information is called a capability. Post/pre-conditions and input/output messages are concepts taken from ontologies referred to by the SWS.

- Information on how a user can interact with the SWS (protocol) which is called choreography.

Interacting with a WS does not imply a post-condition is always achieved, as complex interactions can lead to many different states. When such an ambiguity can occur, the capability needs to specify precisely when post-conditions are realized in the choreography, by linking them to the presence (and possibly the “state”) of tokens in an edge.

When such an information is omitted or missing, it is assumed by the composer that any interaction with the WS leads to the validity of all its post-conditions.

7.2 Atomic goals

Now, when acting as a user looking for an SWS, one will be expressing requirements on those elements. Such requirements are called goals in DIP. A goal is more general than a SWS, in that it may match a number of registered Web services and can hence be seen as an SWS abstraction. Defining a goal is like designing a query that can be used to retrieve concrete SWS from a repository, as e.g. the DIP or Hotblu repository. In the sequel, we call an atomic goal such an abstract SWS description. Atomic goals in our case basically involve the same elements as an SWS description:

- Requirements on what we want to realize and what we will provide. We call this input and output roles. Roles do not need to be taken from an ontology, though they can be constrained to.

- Requirements on desired user interaction specifications. This information is optional and will not be accounted for by the prototype here specified ¹. Such an interaction specification is called a client choreography in DIP.

7.3 Composition goals

The purpose of the current chapter is to define the language used by the prototype composer to describe what we call composition goals. A composition goal describes how

¹although our technical approach allows to add it rather straightforwardly
elementary atomic goals are combined to form a coherent and useful composite SWS. A composition goal altogether defines its own roles, and lists a number of atomic goals plus constraints stating how these subgoals interact. Note that in so doing, although we do not prepare for explicit client choreography restrictions, many such restrictions implicitly follow from data dependencies, for instance when some output data (from one SWS) is used as input to calculations or other SWS.

An SWS composer may propagate some composition goal constraints to its atomic subgoals. For instance, a constraint on the total price of a product naturally acts as a bound on the prices of atomic subgoals. This feature is useful to reduce the combinatorics involved in the search.

A discovery algorithm is used to find SWS that match each atomic subgoal in the composition. The list of such SWS entries is then filtered according to non-functional property constraints eventually mentioned in the composition, and the resulting list is exploited to extract their choreography descriptions. As our composition algorithm is based on the choreography description alone, SWS having isomorphic choreography descriptions can be ignored, unless specific constraints apply to non-functional properties.

7.4 A metamodel for composition goals

We now detail the language used to define composition goals. This language is defined using a metamodel involving classes, their relations and constraints, according to the model driven architecture. The metamodel is presented in Figure 7.1. An abstract goal has two forms. Being atomic, it denotes the set of Web services having similar properties (in terms of their input/output data), and being composite it denotes all the possible workflow combinations of actual SWS instances that match its requirements.

It must be emphasized that we have not opted for a recursive definition of composition goals. Indeed, the structure here presented is “flat”: a composition goal is defined using a number of “atomic goals”, but not “abstract goals”.

One originality of this language is that at the level of abstraction considered for goals, SWS choreographies are not taken into account. This information is only exploited when the composition is computed.

Also, this definition of a composition goal allows to express policies on the composition result, since any type of constraint can be defined.

7.4.1 composition goal constraints

Several types of constraints can be placed on those composition goals:

- Constraints between goals (mostly temporal constraints as eg. “capability1 must occur before capability2”)

- Property constraints on goals. These constraints restrict the values of non-functional properties of matching candidate SWS.

- UnaryConstraint constraints on roles. Those constrain the type and/or the value of the concrete message that fulfills the role in the orchestration.
• Relational constraints. These constraints apply to one or several role, while not involving polarity (participant roles are “equivalent”). Two central examples are unary constraints (they apply to a single role’s abstract attributes), and identity: two (or more) roles correspond to the same element.

• Functional constraints. These constraints are not symmetrical, involve a polarity, and can be seen as having source and target data. Examples are extraction or aggregation constraints, as well as constraints relating a computed value to its arguments.

When the composer operates, all the constraints that occur in the composition goal are mapped into appropriate constraints on the concrete orchestration. For instance, an aggregation constraint between roles implies that an aggregation mediator will be present within their dataflows in the orchestration. We cannot provide here a detailed list of all possible constraints in this document, as intensive examples analysis is necessary to find a minimal useful set. It will however be fully specified by the prototype. We give a detailed example of a composition goal in Section 7.4.

7.5 About using Z: preliminary

Aside from the class diagram presented in Figure 7.1, we must unambiguously formulate the constraints that apply to the model. For the sake of readability, considering the fact that the OCL language is unknown to most researchers, we document the constraints using a fragment of the very intuitive Z relational language.

We present the logical constructs used for the Z specification. For simplicity and readability again, the present document will not introduce the detailed specification of a complete constrained object system in Z (as detailed in [33]), but rather focus on the elements that complement the native UML class diagram semantics. A limited number of definitions are thus required.

We first declare the set $WFObject$ of workflow elements.

$[WFObject]$

Now we declare $WFClass$ as an alias for the type of sets of $WFObject$: the power set $P WFObject$.

$WFClass = \mathcal{P} WFObject$

We also introduce the possibility to use UML’s OCL like dotted notation for dereferencing relation roles. This is achieved by first declaring two latex commands \callfun and \callrel that both display as a dot “·”, then by axiomatically defining \callfun and \callrel as follows:

\[
\begin{align*}
\callfun & : (WFObject \times (WFObject \rightarrow WFObject)) \rightarrow WFObject \\
\callrel & : (WFObject \times (WFObject \rightarrow WFClass)) \rightarrow WFClass \\
\forall e_1 : WFObject; e_2 : (WFObject \rightarrow WFObject) & : e_1.e_2 = e_2(e_1) \\
\forall e_1 : WFObject; e_2 : (WFObject \rightarrow WFClass) & : e_1.e_2 = e_2(e_1)
\end{align*}
\]

The intuition of what precedes is that given a function $f$ from $WFObject$ to $WFObject$

$\left| f : WFObject \rightarrow WFObject \right|

and an element $o$ of $WFObject$, the notations $f(o)$ and $o.f$ are equivalent.
7.6 Z counterpart of the UML class diagrams

7.6.1 Classes
The composition goal metamodel, involving goals, roles and constraints, is detailed in Figure 7.1. It involves the following classes:

- AbstractGoal : WFClass
- AtomicGoal : WFClass
- Role : WFClass
- CompositionGoal : WFClass
- AbstractMessage : WFClass
- Constraint : WFClass
- GoalConstraint : WFClass
- PropertyConstraint : WFClass
- FunctionalConstraint : WFClass
- RelationalConstraint : WFClass
- UnaryConstraint : WFClass
- IdentityConstraint : WFClass

7.6.2 Relations
The composition goal metamodel also involves the following relations. Each CompositionGoal maps to a set of Atomic subgoals.

\[
\text{subgoals} : \text{CompositionGoal} \rightarrow P \text{ AtomicGoal}
\]

\[
\forall x : \text{AtomicGoal} \cdot \exists y : \text{CompositionGoal} \cdot x \in y.\text{subgoals}
\]

Note that the model is intentionally not recursive (as would have allowed a relation between CompositionGoal and AbstractGoal instead of AtomicGoal as proposed).

Then, there is a relation between goals and goal constraints:

\[
\text{goals} : \text{GoalConstraint} \rightarrow P \text{ AbstractGoal}
\]

\[
\text{constrainedBy} : \text{AbstractGoal} \rightarrow P \text{ GoalConstraint}
\]

\[
\forall g : \text{AbstractGoal}; c : \text{GoalConstraint} \cdot
g \in c.\text{goals} \iff c \in g.\text{constrainedBy}
\]

Composition goals also involve a number of related constraints:

\[
\text{constraints} : \text{CompositionGoal} \rightarrow P \text{ GoalConstraint}
\]

Among the constraints that apply to a goal are unary constraints. Such constraints typically restrict the possible values of non-functional properties (hence the function name)

\[
\text{properties} : \text{AbstractGoal} \rightarrow P \text{ PropertyConstraint}
\]

\[
\text{abstractGoal} : \text{PropertyConstraint} \rightarrow \text{AbstractGoal}
\]

\[
\forall a : \text{AbstractGoal}; u : \text{UnaryConstraint} \cdot
a = u.\text{abstractGoal} \iff u \in a.\text{properties}
\]
Furthermore, this relation is a subset of the GoalConstraint/AbstractGoal relation (or else “an AbstractGoal is constrained by its properties”):

\[ \forall g : \text{AbstractGoal} \bullet g.\text{properties} \subset g.\text{constrainedBy} \]

There is a relation between relational constraints and roles

\[ \text{roles} : \text{RelationalConstraint} \rightarrow \text{P Role} \]

and there are two relations between functional constraints and roles (# denotes set cardinality):

\[ \text{sources} : \text{FunctionalConstraint} \rightarrow \text{P Role} \]
\[ \text{targets} : \text{FunctionalConstraint} \rightarrow \text{P Role} \]
\[ \forall x : \text{FunctionalConstraint} \bullet \#(x.\text{sources}) = 1 \lor \#(x.\text{targets}) = 1 \]

There also exists two relations between goals and roles. Note that a role is either “input” or “output” of a goal but not both.

\[ \text{inputs} : \text{AbstractGoal} \rightarrow \text{P Role} \]
\[ \text{outputs} : \text{AbstractGoal} \rightarrow \text{P Role} \]
\[ \text{isInputOf} : \text{Role} \rightarrow \text{P AbstractGoal} \]
\[ \text{isOutputOf} : \text{Role} \rightarrow \text{P AbstractGoal} \]
\[ \forall r : \text{Role} \bullet \#(r.\text{isInputOf} \cup r.\text{isOutputOf}) = 1 \]
\[ \forall r : \text{Role}; g : \text{AbstractGoal} \bullet (g \in r.\text{isInputOf}) \Leftrightarrow (r \in g.\text{inputs}) \]
\[ \forall r : \text{Role}; g : \text{AbstractGoal} \bullet (g \in r.\text{isOutputOf}) \Leftrightarrow (r \in g.\text{outputs}) \]

7.7 Additional metamodel constraints

This section details all the composition goal model constraints that do not strictly pertain to relation semantics. In this category are the constraints that link elements deeper in the structure, or constraints that specialize relations properties in subclasses for instance

7.7.1 UnaryConstraint constraints

UnaryConstraint constraints have exactly one attached role

\[ \forall x : \text{UnaryConstraint} \bullet \#(x.\text{roles}) = 1 \]

If atomic goal “a” is “constrainedBy” goal constraint “c” belonging to composition goal “b” then “a” is member of b.subgoals

7.7.2 Relational constraints

The abstract goals (via isInputOf or isOutputOf) of the roles bound by a relational constraint are the abstract goals to which the relational constraint is bound via the relation goals.

\[ \forall c : \text{RelationalConstraint}; g : \text{AbstractGoal} \bullet \\
\quad g \in c.\text{goals} \Leftrightarrow g \in \bigcup (\text{isInputOf} \{c.\text{roles}\} \cup \text{isOutputOf} \{c.\text{roles}\}) \]
In this axiom, the Z construct “f[s]” denotes the relational image of the set s by the relation f (here a function).

### 7.7.3 Functional constraints

Similarly, the abstract role to which a source or target of a functional constraint is attached (via isInputOf or isOutputOf) is a goal bound to the constraint via the relation goals.

\[
\forall c : \text{FunctionalConstraint}; g : \text{AbstractGoal} \quad \bullet
\]

let \( s = c.\text{sources} \cup c.\text{targets} \) \( \bullet \)

\( g \in c.\text{goals} \Leftrightarrow g \in \bigcup(isInputOf[s] \cup isOutputOf[s]) \)

The entire above specification was type checked using \texttt{fuzz}².

### 7.8 An example of a composition goal instance

Figure 7.2 presents an example of a composition goal, which can be used to solve the producer-shipper problem. This problem is the one used as a working example in deliverables D3.4 and D3.5, and in our application scenario. Rather than giving a text instance of the metamodel, we chose to display it here as a picture which should be more understandable to the reader. A diamond \( \Diamond \) represents a constraint. The note attached to it denotes its type. The arrows coming in and out specify the role(s) (or the goal(s)) this constraint applies to. Atomic subgoals are represented in yellow, with their input/output roles as yellow rectangles around it.

---

²\texttt{Fuzz}: http://spivey.oriel.ox.ac.uk/mike/fuzz/
Figure 7.1: Goals and Roles
Figure 7.2: composition goal example for the producer-shipper problem
8 Goal Composer Output

The composer produces a workflow combining a number of participant SWS choreographies in an orchestration. Although being generated in an internal format by the tool (an instance of the metamodel as implemented in JConfigurator), the results can be exported to several useful formats. The prototype focuses on one of them: the WSML ontology for UML2AD workflows as described in the DIO annex of D3.4 and D3.5.

One issue in automatically exploiting the composer results in DIP regards the publication of a valid description of the resulting orchestration. DIP allows the publication of such SWS orchestrations. However, DIP publishing requires both the description of an orchestration (which implements the functionality), and the description of the interface, needed for other SWS to invoke the composition, basically in the form of a choreography with post/pre-conditions. In WSMO, pre-conditions mostly serve to describe required SWS inputs and post-conditions describe the SWS outputs and effects on the outer world.

In order to perform such an automatic publication, the composer henceforth requires an auxiliary module, capable of generating an orchestration out of the complete workflow, and a choreography out of the orchestration.

Finally, in order to render the whole description executable, a translation of UML2AD workflows to ASMs is necessary. This operation will be carried out by an independent module, as specified in the D3.4 and D3.5.

The current chapter covers these issues.

8.1 Automatic publication to DIP

Composer output will undergo:

- exporting to a valid UML2AD ontology instance representing an orchestration of SWS using WSMO4J
- extracting a choreography workflow from the orchestration workflow
- exporting to a valid UML2AD ontology instance representing the choreography of the corresponding composition using WSMO4J
- translation of both ontology instances to WSMX compliant ASMs by a third party program as specified in the D3.4 and D3.5

Note that some non-functional properties of the composition could be automatically generated as well. This however exceeds the scope of the present prototype and may reserved for future work.

8.2 Automatic extraction of the choreography from the orchestration

We attempt here to sketch the algorithms that will be used to generate a choreography from an orchestration. This possibility was studied with great detail in [18], in the restricted case of colored Petri nets.
Briefly stated, such a transform first requires to isolate the user inputs and outputs to the orchestration, which are identified as first class components from the inputs and outputs of the composition. Then, every group of operations that do not involve user interaction can be abstracted as atomic operations in a recursive manner. The procedure itself can be viewed as a fixpoint procedure that halts when no more rewriting of the original workflow is possible.

What essentially counts in the process is the fact that the translation preserves important protocol information: the temporal (precedence) dependencies that exist between sub-activities. This information is required so that external SWS querying the DIP repository can accurately infer the proper ways to interact with the composition.

8.3 Automatic generation of a goal representing the composition

Optionally, it should be possible to automatically generate a DIP goal allowing for the discovery of the composition, but this does not pertain to the planned realization described here.

8.4 Output Example

Figure 8.1 presents the orchestration created out of the producer-shipper problem (in the packed case), and Figure 8.2 presents the composite SWS choreography.
Figure 8.1: Orchestration example
Figure 8.2: Choreography example
9 CONCLUSION AND REFERENCES

The present deliverable detailed the features and requirements for the DIP Composer Prototype. A central component of the prototype is the composition algorithm itself. Being implemented on the basis of a constraint based configuration technique, a fundamental part of the specification is the constrained object model describing the workflow metamodel used. The main reference for this specification is in the DIO annex of the D3.4 and D3.5.

Aside from the composer module, the prototype associates a module for generating choreographies out of the orchestrations the composer produces. Moreover, for large-scale settings, this specification describes an optional component to index and selectively retrieve service advertisements in order to prevent overloading the composer with service advertisements, which otherwise might deteriorate composition efficiency.

The complete picture is that of an helper application available as a WSMO editor plugin, that can be used to edit composition requests, then automate the process of generating a valid composition from SWS available (i.e. documented) in the DIP repositories, then finally publish the result to DIP as valid standalone WSMX emulated orchestrations.
REFERENCES


10 Annex – Indexing Directory

This annex describes details of an optional directory to efficiently index service advertisements, called Hotblu. First we discuss how to design a flexible directory interface that allows a composer to dynamically retrieve relevant service advertisements. Afterwards we describe techniques for the efficient implementation of such a directory.

10.1 Flexible Directories

Our directory takes into consideration two orthogonal aspects: flexibility and efficiency. While often systems trade one for the other, the novelty of our approach is to provide our system with a transformation scheme that combines high flexibility and efficient query processing. In this chapter we also describe our approach for providing directory services with support for concurrency by using a transactional approach based on multiple versions of data objects.

In this section we will describe how directory clients can interact with our directory system such that any aspect of a published service advertisement can be taken into consideration by the discovery process.

10.1.1 Flexible Selection and Ranking of Service Descriptions

<table>
<thead>
<tr>
<th>Match Type</th>
<th>Service Description (SR/SA)</th>
<th>Selection Expression (SR SelExpr SA)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signature</td>
<td>Var Type Map</td>
<td>Quant1 VarSel1 Quant2 VarSel2</td>
</tr>
<tr>
<td></td>
<td>Var -&gt; SetOf Vals</td>
<td>VarIncl ValSetIncl</td>
</tr>
<tr>
<td>Specification</td>
<td>Pre Constraint</td>
<td>ConsSel AssignSetIncl</td>
</tr>
<tr>
<td></td>
<td>SetOf Assign Map</td>
<td>Quant1 VarSel1 Quant2 VarSel2</td>
</tr>
<tr>
<td></td>
<td>Var -&gt; Val</td>
<td>VarIncl ValIncl</td>
</tr>
<tr>
<td></td>
<td>Eff Constraint</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SetOf Assign Map</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Var -&gt; Val</td>
<td></td>
</tr>
</tbody>
</table>

Table 10.1: Flexible selection of service descriptions

Matching a service request SR and one or more advertisements SA is a process in which, in the first step, variable descriptions and variable types are compared (signature match), and in the second step, possible variable assignments of different constraints are compared (specification match).

The goal of our approach is to build a framework for matchmaking service descriptions that should be as complete as possible, regarding the different signature and specification match relations that can be established between requests and advertisements.

For this purpose, we introduce selection expressions that mirror best the structure of a service description, such that any aspect present in the description can be used as a decision parameter by the selection expression.
Our methodology in establishing selection expressions is to reflect the structure of the service description as follows:

- **Map → Quant Sel Quant Sel** – in the selection expression, maps are reflected by quantifiers that iterate over map keys and concept selectors that determine if a key has a given concept type. There are two quantifier/selector pairs, where each of them is applied to one map, either from the SR or from the SA, but not necessarily in that order.

- **Concept, Set, Concept Set → Inclusion Test** – since concepts and sets have quite similar semantics, we treat all three of them in the same way also syntactically, by using basic set operators to compare them.

For signature matches, the mapping above will result in the following parameterized selection expression:

\[
\text{SignatureSelection}(SR, SA, Q_1, S_1, Q_2, S_2, R, T) = \\
(\text{quant}(Q_1) \text{ var}_1 \in \text{vars(desc}(Q_1)), \text{var}_1 \subseteq S_1) \\
(\text{quant}(Q_2) \text{ var}_2 \in \text{vars(desc}(Q_2)), \text{var}_2 \subseteq S_2) \\
(\text{var}_1 R \text{var}_2) \land (\text{type}(\text{var}_1) T \text{type}(\text{var}_2))
\]

where \(Q_1, Q_2 = \{\forall_{SR}, \exists_{SR}, \forall_{SA}, \exists_{SA}\}\) represent quantifier descriptions formed by a quantifier and respectively one of the two descriptions (SR,SA) to which it could be applied. The function \(\text{quant}(Q_{SD})\) returns the actual quantifier \((Q \in \{\forall, \exists\})\) and the function \(\text{desc}(Q_{SD})\) returns the description to which the quantifier is applied \((SD \in \{SR, SA\})\). Note that \(\text{desc}(Q_1) \neq \text{desc}(Q_2)\). The function \(\text{vars}(S)\), which we introduced in Section 5.1, returns the set of variables in a service description \(S\).

![Figure 10.1: Flexible signature selection.](image)

The actual selection includes a test between variable descriptions, based on the binary operator \(R = \{\equiv, \subseteq, \supseteq, \nsubseteq, \nsubseteq, \nsubseteq, \nsubseteq, \nsubseteq, \nsubseteq, \nsubseteq, \nsubseteq, \nsubseteq, \nsubseteq\}\) and between variable types, based on the binary operator \(T = \{\equiv, \subseteq, \supseteq, \nsubseteq, \nsubseteq, \nsubseteq, \nsubseteq, \nsubseteq, \nsubseteq, \nsubseteq, \nsubseteq, \nsubseteq\}\). “\(\nsubseteq\)” and “\(\nsubseteq\)” represent tests for disjointness, respectively non-disjointness. The \(\nsubseteq\) and \(\nsubseteq\) symbols are included for completeness and represent tests that are always false, respectively always true.
For determining “how much” a SA matches a SR regarding its signature we use a modified form of the signature selection expression for which exists a match concerning the variable description and type in the request:

\[
\text{SignatureRanking}(SA, SR, S_{SA}, S_{SR}, R, T) = \\
\{ \text{vars}_{SA} \subseteq \text{vars}(SA) : (\text{vars}_{SA} \subseteq S_{SA}) (\exists \text{vars}_{SR} \in \text{vars}(SR)(\text{vars}_{SR} \subseteq S_{SR})) \\
(\text{vars}_{SA} R \text{vars}_{SR}) \land (\text{type}(\text{vars}_{SA}) T \text{type}(\text{vars}_{SR})) \} |.
\]

The functions of the selectors \(S_1, S_2\) as well as the kind of set inclusion tests that can be supplied as the \(R\) and \(T\) parameters are as above.

For determining the number of variables consistent with a given selector concept \(S\), either from a request or advertisement service description \(SD\), we use the following function:

\[
\text{CountVars}(SD, S) = | \{ v \in \text{vars}(SD) : v \subseteq S \} |.
\]

Table 10.1 gives a schematic view of selection and ranking expressions, which will be detailed in Section 10.1.3. As it results from Table 10.1 signature selection expressions test for different possible inclusions between sets of possible assignments, induced by constraints from the SA or SR. Constraints are identified by keywords (e.g., \(PRE, EFF\)). The assignment set inclusion tests are parametrized regarding the equivalence of assignment set members (maps of variables to values). This kind of equivalence is determined by an expression very similar to the signature expression above. The difference is that assignments map variables to values rather than in the previous case variables are mapped to sets of values. Formally, specification matches are represented as:

\[
\text{SpecificationSelection}(SR, SA, K_{SR}, K_{SA}, A, Q_1, S_1, Q_2, S_2, R, T) = \\
\text{constraint}(K_{SR}, SR) \text{ A constraint}(K_{SA}, SA)
\]

where the constraint keywords \(K_{SR}, K_{SA} = \{PRE, EFF\}\) and \(A = \{\equiv, \subseteq, \supseteq, \not\subseteq \not\equiv, \not\supseteq\}\). These inclusion tests use the following parameterized relation for determining equivalence between pairs of assignments in the two constraints \((K_{SR}, K_{SA})\):

\[
a_{SR} \equiv_{Q_1, S_1, Q_2, S_2, R, T} a_{SA}, a_{SR} \in \text{constraint}(K_{SR}), a_{SA} \in \text{constraint}(K_{SA}) \iff \\
\quad (\text{quant}(Q_1) \langle \text{var}_1, \text{val}_1 \rangle \in \text{assign}(Q_1), \text{var}_1 \subseteq S_1) \\
\quad (\text{quant}(Q_2) \langle \text{var}_2, \text{val}_2 \rangle \in \text{assign}(Q_2), \text{var}_2 \subseteq S_2) \\
\quad (\text{var}_1 R \text{var}_2) \land (\text{val}_1 T \text{val}_2)
\]

where \(Q_1, Q_2 = \{\forall_{a_{SR}}, \exists_{a_{SR}}, \forall_{a_{SA}}, \exists_{a_{SA}}\}\) represent quantifier descriptions formed by a quantifier and respectively one of the two assignments \((a_{SR}, a_{SA})\) to which it could be applied. The function \(\text{quant}(Q_a)\) returns the actual quantifier \((Q \in \{\forall, \exists\})\) and the function \(\text{assign}(Q_a)\) returns \(a \in \{a_{SR}, a_{SA}\}\), the description to which the quantifier should be applied. Note that \(\text{assign}(Q_1) \neq \text{assign}(Q_2)\).
For determining “how much” two constraints match, we adapt the above specification selection expression to return the cardinality of the set of assignments from SA that have a given matching relation with some assignment in SR:

\[
\text{SpecificationRanking}(SA, SR, K_{SA}, K_{SR}, Q_1, S_1, Q_2, S_2, R, T) = \left| \{ a_{SA} \in \text{constraint}(K_{SA}, SA) : (\exists a_{SR} \in \text{constraint}(K_{SR}, SR) (a_{SR} \cong Q_1, S_1, Q_2, S_2, R, T a_{SA})) \} \right|
\]

For determining the cardinality of an assignment set identified by a given keyword \( K \), in either a request or advertisement service description \( SD \), we use the following function:

\[
\text{CountAssignments}(SD, K) = | \{ a \in \text{constraint}(K, SD) \} |
\]

10.1.2 Directory Query Language Requirements

In order to have a directory that will support flexible selecting and ranking of service descriptions we would like to design a directory query language, namely \text{DirQL}, that would meet the following requirements:

- **Simplicity**: \text{DirQL} will offer only a minimal set of constructs, but will be expressive enough to express relevant signature and specification selection and ranking heuristics.

- **Declarative**: \text{DirQL} will be a functional language and will not support destructive assignment. The absence of side-effects eases program analysis and transformations.

- **Safety**: As the directory executes user-defined code, \text{DirQL} expressions must not be able to interfere with internals of the directory. Moreover, the resource consumption (e.g., CPU, memory) needed for the execution of \text{DirQL} expressions is bounded in order to prevent denial-of-service attacks: \text{DirQL} does not support recursion, loops are of bounded size, and queries can be executed without dynamic memory allocation.

- **Efficient directory search**: \text{DirQL} has been designed to enable an efficient best-first search in the directory according to the ranking of results. For details, see Section 10.2.

10.1.3 Directory Query Language Grammar

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 [14].

By default, the order in which matching service descriptions are returned depends on the actual structure of the directory index. 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 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 [26].

A query consists of a functional description including variables defining input and output parameters resp. available and required world states, precondition and effect constraints on those variables as well as a custom matching and ranking function.

The matching and ranking function is written in the simple, high-level, functional query language DirQL (Directory Query Language). An (informal) EBNF grammar for DirQL is given in Table 10.2. The grammar includes the following non-terminal tokens, not explicitly shown in the grammar:

- **conceptID** – represents a reference to a concept definition in a high-level language like [69]; we assume the definition of a number of pre-defined variable description concepts, relevant to the scope of service descriptions: IN, OUT, PRIOR, POST representing variables for input and output parameters respectively for describing the world state before and after the invocation of the service.
- `constraintID` – represents an alphanumeric keyword of length bigger than 0 used for identifying different constraints by name; currently pre-defined keywords are: PRE and EFF, representing preconditions and effect constraints;
- `number` – represents a numeric constant (integer or decimal number)

A DirQL expression may contain only a selection expression, only a ranking expression, or both. In the former case, only advertisements matching exactly the selection expression are returned. If only a ranking is specified, all services in the directories are considered to be selected. Depending if the results are sorted in ascending (ASC) or descending (DESC) order, advertisements with lowest respectively highest ranks are returned first. When both a selection and ranking expression are specified, the result set will contain only advertisements matching the selection expression and it will be ordered according to the ranking expression.

A DirQL expression may involve some simple arithmetic. The operators + and * correspond roughly to the mathematical operators sum $\sum$ and product $\prod$. The operators `min` and `max` have the obvious meanings – they return the minimum, respectively maximum value of a number of numeric expressions. All these operators require two or more parameters.

For the division operator `/`, the second operand (divisor) has to evaluate to a constant for a given query such that before a query is executed, the directory can ensure that DirQL expressions will not cause a division by zero. The DirQL programmer may use the ‘if’ conditional – similar to the LISP `if`-function or the C or Java `?`-operator – to select numeric expressions consistent with some given boolean constraint (e.g., that an operand is non-negative).

A DirQL expression defines custom selection and ranking heuristics. The evaluation of a DirQL expression is based on different relations that can be established between the variable descriptions and types and on the assignment sets of a service request and a service advertisement. The syntax of DirQL has some similarities with LISP and the FIPA content language SL0 [25].

<table>
<thead>
<tr>
<th>DirQL Construct</th>
<th>Formal predicate or function</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sigMatch</code></td>
<td>SignatureSelection(SR, SA, Q₁, S₁, Q₂, S₂, R, T)</td>
</tr>
<tr>
<td><code>countVars</code></td>
<td>SignatureRanking(SA, SR, S_SA, S_SR, R, T)</td>
</tr>
<tr>
<td><code>countVarsSR</code></td>
<td>CountVars(SR, S₁)</td>
</tr>
<tr>
<td><code>countVarsSA</code></td>
<td>CountVars(SA, S)</td>
</tr>
<tr>
<td><code>specMatch</code></td>
<td>SpecificationSelection(SR, SA, K_SR, K_SA, A, Q₁, S₁, Q₂, S₂, R, T)</td>
</tr>
<tr>
<td><code>countAssigns</code></td>
<td>SpecificationRanking(SA, SR, K_SA, K_SR, Q₁, S₁, Q₂, S₂, R, T)</td>
</tr>
<tr>
<td><code>countAssignsSR</code></td>
<td>CountAssignments(SR, K)</td>
</tr>
<tr>
<td><code>countAssignsSA</code></td>
<td>CountAssignments(SA, K)</td>
</tr>
</tbody>
</table>

Table 10.3: Formal semantics of DirQL language constructs.

The `sigMatch` and `specMatch` expressions correspond to signature and specification selection expressions as defined previously in Section 10.1.1. As the equational relation that has to be determined when matching constraint assignments is very similar to signature matching, in order to make the grammar more compact, we define both signature and specification matching using a common production `qSelVarVal` for matching between maps of variables to values or sets of values. It has to be emphasized
that the semantics of the \texttt{qSelVarVal} expression are slightly different depending if it appears as a signature match or in a specification match (the second \texttt{relOP} operates over sets of values in the first case and over values in the second case).

The seven relation specifiers \texttt{EQUIV}, \texttt{SUBSET}, \texttt{SUPERSET}, \texttt{OVERLAP}, \texttt{DISJOINT}, \texttt{T}, and \texttt{F} correspond to the set or concept inclusion operators $\equiv$, $\subseteq$, $\supseteq$, $\cap \neq \bot$, $\cap = \bot$, respectively to operators returning always true and always false.

The semantics of the boolean expressions \texttt{sigMatch}, \texttt{specMatch}, and of the numeric functions for counting variables \texttt{countVars}, \texttt{countVarsSR}, \texttt{countVarsSA}, or assignments \texttt{countAssigns}, \texttt{countAssignsSR}, \texttt{countAssignsSA}, are those of the predicates and functions formally defined previously, where the exact mapping is illustrated in Table 10.3.

10.1.4 Example Queries

In the examples in Table 10.4 and Table 10.5 two basic selection and ranking functions are shown, the first one suited for service composition algorithms using forward chaining with partial type matches, the second one for algorithms based on backward chaining.

For forward chaining Table 10.4, 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).

The results are sorted in ascending order according to the sum of remaining outputs and remaining goals that are required by the query, but not provided by the service (services that provide more of the required outputs or goals come first).

For backward chaining Table 10.5, we expect that the service provides either at least one output or at least one goal 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.

10.2 Efficient Directories

In this section we present our approach that while making the discovery process inside the directory highly efficient will provide to the client a flexible interface, completley

| select (and |
|           (allSAsomeSR IN IN SUPERSET SUPERSET) |
|           (allSAsomeSR PRIOR PRIOR SUPERSET SUPERSET) |
|           (PRE SUBSET PRE |
|           (allSAsomeSR IN IN SUPERSET SUPERSET)) |
|           (PRE SUBSET PRE |
|           (allSAsomeSR PRIOR PRIOR SUPERSET SUPERSET)) |
|) order by asc |
| (* |
|   (- (countVarsSR OUT) (countVars OUT OUT SUPERSET SUPERSET)) |
|   (- (countVarsSR POST) (countVars POST POST SUPERSET SUPERSET)))) |

Table 10.4: Example \texttt{DirQL} expression for forward chaining service composition.
Table 10.5: Example DirQL expression for backward chaining service composition.

independent of the internal organization of the directory.

10.2.1 Multidimensional Indexing for Service Directories

In a real world environment created by numerous service providers that advertise their particular services, we assume a realistic setting in which directories will store numerous descriptions. The directory service must efficiently deal with data organization and retrieval. The need for efficient discovery and matchmaking leads to the creation of search structures and indexes for directories.

The novelty of our approach is to consider a service description as a multidimensional data record and then use in the directory techniques related to the indexing of such kind of information.

Generalized Search Tree (GiST)

As many solutions have been proposed for managing multidimensional data, work has been done for isolating the common approach that all these solutions take. Hellerstein [32] proposed the Generalized Search Tree (GiST) as an unifying framework.

The design principle of GiST starts 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. Every internal node has a series of keys and pointers to a number of child nodes (number usually depending on system and hardware constraints, e.g., filesystem page size).

Logically, predicates of inner nodes subsume predicates of all children nodes. To search records that satisfy a query predicate, only some paths of the tree are followed, those having inner predicates that can satisfy the query being processed. The tree is traversed in this manner until leaf nodes that contain data that matches the query are reached. For a given inner node, the associated predicate can be seen as an upper bound or envelope of the predicates in the leaf nodes of the subtree originated at the node.

In classical trees, predicates are constrained to specific types, such as range predicates, where keys delimit a range \([c_{\text{min}}, c_{\text{max}}]\), and a predicate is of the form \(c_{\text{min}} \leq i \leq c_{\text{max}}\) (B+-trees). But essentially a search key may be any arbitrary predicate that holds for each datum below the key.
A search tree is therefore a hierarchy of categorizations, in which each categorization holds for the data stored under it in the hierarchy. By exposing the key methods and the tree re-balancing methods to the user, arbitrary search trees may be constructed. As the name implicates, the Generalized Search Tree is a search tree that is independent of the data types it indexes, it provides basic search tree logic and supports queries which are natural to the target data. In a single piece of code, it unifies the common functionality of search trees, the user of the GiST only needs to provide the necessary extensions for the type of tree that is desired.

A large number of existing tree algorithms can be re-casted in terms of GiST: B+, R tree, R* tree (by slightly modifying the GiST insertion algorithm), extended KD trees, etc. GiST is quite well adopted by both academic (Postgresql) and industrial (Informix) DB communities for defining access methods to custom data types.

Tree Structure

A GiST is a balanced search tree of variable fanout between $kM$ and $M$, where $M$ is the maximum number of child nodes and $2/M \leq k \leq 1/2$, except for the root node, which may have fanout between 2 and $M$ [32]. Inner nodes contain $(p, ptr)$ pairs, where $p$ is a predicate that functions as a search key, and $ptr$ references another node. Leaf nodes contain the same pairs, but here $ptr$ identifies some tuple of user data. Predicates can contain any number of free variables, with the restriction that each tuple referenced by the leaves of the tree can instantiate all the variables.

Tree Properties

These are the invariant properties of the GiST:

- Every node contains between $kM$ and $M$ index entries unless it is the root.
- For each index entry $(p, ptr)$ in a leaf node, $p$ is true when instantiated with the values from the indicated tuple.
- For each index entry $(p, ptr)$ in an inner node, $p$ is true when instantiated with the values of any tuple reachable from $ptr$. 

Figure 10.2: Generalized Search Tree
• The root has at least two children unless it is a leaf.
• All leaves appear on the same level (balanced tree).

Note that, unlike e.g. with R-trees, for any entry \((p', \text{ptr}')\) reachable from \(\text{ptr}\), \(p'\) can express a predicate that is entirely orthogonal to \(p\). This aspect is interesting with respect to the classification of data in the tree.

### 10.2.2 Efficient Propositional Inference

![Diagram of formula approximations](image)

Figure 10.3: Formula approximations for fast inference.

Propositional inference is the problem of checking whether \(\Sigma \models \Gamma\) (i.e., \(\Sigma\) entails \(\Gamma\)), where \(\Sigma\) and \(\Gamma\) are logical formulas. If we use model theoretic semantics where we define as \(M(\phi)\) the set of satisfying truth assignments of the formula \(\phi\) (i.e., the set of models of the formula) then the above entailment relation is equivalent to \(M(\Sigma) \subseteq M(\Gamma)\), that is the set of models of \(\Sigma\) is included in the set of models of \(\Gamma\).

There are a large number of approaches for efficiently computing this but we are particularly interested in one initially proposed by [53] and then developed and extended by [29, 52, 10]. Their approach is based on the idea that while inference for general propositional formulae is co-NP complete, the time required in the case of Horn formulas is polynomial. As such they proposed a compilation technique where a generic formula is approximated by two Horn formulas \(\Sigma\) and \(\overline{\Sigma}\) that satisfy the following:

\[\Sigma \models \Sigma \models \overline{\Sigma}\] or equivalently \(M(\Sigma) \subseteq M(\Sigma) \subseteq M(\overline{\Sigma})\).

In the literature \(\Sigma\) is called the **Horn lower bound** or **core** of \(\Sigma\) while \(\overline{\Sigma}\) is called the **Horn upper bound** or **envelope** of \(\Sigma\). As it can be also seen in Figure 10.3, \(\Sigma\) is a **complete** approximation of \(\Sigma\), since any of the models of the core (lower bound) formula is also a model of the original formula. Conversely the envelope (upper bound) is a...
sound approximation of the original formula as any model of the original formula is also a model of the envelope.

Assignment sets of constraints can be also seen as sets of models of a propositional formula as in fact propositional logic is equivalent to 3-SAT or constraint satisfaction. Assignment sets from different constraints can be seen as different formulas (e.g. $\Sigma_1, \Sigma_2$) and can be approximated using the same technique as in the case of Horn clauses (see Figure 10.3): their union can be considered an upper bound and their intersection can be considered a lower bound. Before testing individual matching conditions between a query $\Gamma$ and existing entries $\Sigma_1$ resp. $\Sigma_2$, the upper and lower bounds can be checked first whether they satisfy a relaxed but necessary version of the original condition, also called pruning condition. If the query description does not match the bound under the relaxed condition, it will certainly not match the original description either. I.e., further tests can be pruned. If the query matches the bound then further tests may be needed, for example on more refined bound approximations.

The same kind of reasoning applies to variables and variable types and for them too, upper and lower bounds are created through union, respectively through intersection.

Formally, given the service descriptions $SD_1$ and $SD_2$ we compute the upper bound description $\overline{SD}$ as the union of constraints from the two descriptions, identified by the same keywords; variable descriptions, and their respective types are then computed in order to reflect that:

$$\forall K \in \{\text{PRE, EFF}\}, \text{constraint}(K, \overline{SD}) = \text{constraint}(K, SD_1) \cup \text{constraint}(K, SD_2)$$

$$\text{vars}(\overline{SD}) = \text{vars}(SD_1) \cup \text{vars}(SD_2)$$

$$\forall \text{var} \in \text{vars}(\overline{SD}), \text{type}(\text{var}, \overline{SD}) = \begin{cases} \text{type}(\text{var}, SD_1), \text{var} \notin \text{vars}(SD_2) \\ \text{type}(\text{var}, SD_2), \text{var} \notin \text{vars}(SD_1) \\ \text{type}(\text{var}, SD_1) \sqcup \text{type}(\text{var}, SD_1), \text{var} \in \text{vars}(SD_1) \land \text{var} \in \text{vars}(SD_2) \end{cases}$$

Figure 10.4: Computing a description envelope $\overline{SD} = SD_1 \sqcup SD_2$.

The lower bound $\underline{SD}$ of descriptions $SD_1$ and $SD_2$ is computed starting from the set of assignments that are common to constraints identified by the same keyword in both descriptions. The variables of the new description are those variables in either of the two original descriptions that appear at least once in an assignment in any of the constraints in the new description. For the selected variables, types are computed by selecting from the sets of values representing the type of the variable in the original descriptions (possibly only in one of $SD_1$ and $SD_2$) those values that appear in at least one assignment in any of the constraints in the new description.

As an example regarding pruning, assume that from the same kind of constraints (e.g., $\text{PRE, EFF}$) from several service advertisements $\Sigma_x$ ($x = 1, 2, \ldots$) bounded by $\Sigma$ and $\overline{\Sigma}$, we have to select those that satisfy the entailment $\Gamma \models \Sigma_x$, where $\Gamma$ is a constraint from a service request. As a necessary condition for $\Sigma_x$ to satisfy the entailment, $\overline{\Sigma}$ must satisfy the entailment, too. If this is the case, the individual formulas $\Sigma_x$ have to be tested for entailment. Otherwise, no further tests are necessary. I.e., the negation of the entailment, $\Gamma \not\models \overline{\Sigma}$, can be used as a pruning condition.
∀ K \in \{\text{PRE, EFF}\}, \text{constraint}(K, SD) = \text{constraint}(K, SD_1) \cap \text{constraint}(K, SD_2)

\text{vars}(SD) = \{\text{var} \in \text{vars}(SD_1) \cup \text{vars}(SD_2) |
\exists K \in \{\text{PRE, EFF}\}, \exists \langle \text{var}, \text{val} \rangle \in (\text{constraint}(K, SD_1) \cap \text{constraint}(K, SD_2))\}

∀ \text{var} \in \text{vars}(SD), \text{type}(\text{var}, SD) =
\begin{cases} 
\{\text{val} \in \text{type}(\text{var}, SD_1) | \exists K \in \{\text{PRE, EFF}\}, \exists \langle \text{var}, \text{val} \rangle \in \text{constraint}(K, SD_1), \\
\quad \text{var} \notin \text{vars}(SD_2)\} \\
\{\text{val} \in \text{type}(\text{var}, SD_2) | \exists K \in \{\text{PRE, EFF}\}, \exists \langle \text{var}, \text{val} \rangle \in \text{constraint}(K, SD_2), \\
\quad \text{var} \notin \text{vars}(SD_1)\} \\
\{\text{val} \in (\text{type}(\text{var}, SD_1) \cup \text{type}(\text{var}, SD_2)) | \\
\quad \exists K \in \{\text{PRE, EFF}\}, \exists \langle \text{var}, \text{val} \rangle \in (\text{constraint}(K, SD_1) \cap \text{constraint}(K, SD_2)), \\
\quad \text{var} \in \text{vars}(SD_1) \land \text{var} \in \text{vars}(SD_2)\}
\end{cases}

Figure 10.5: Computing a description core $SD = SD_1 \cap SD_2$.

10.2.3 Two Bound Search Trees

Our approach extends the basic GiST framework by associating to nodes in the tree, not only service descriptions created as a union of underlying service descriptions but also descriptions created as intersections, descriptions that are subsumed by all values below in the tree, with each inner node. This new description can be seen as a lower bound or core (see above the constraint $\Sigma$) of the descriptions in the leaf nodes of the subtree originated at the node defining the description. Core descriptions and their respective constraints ($\Sigma$) are required for pruning conditions that include negative selection expressions ($\neg$SignatureMatch(...) or $\neg$SpecificationMatch(...)) or for lower bound of numerical expressions. In theory core descriptions can be used also for pruning conditions that include positive selection expressions but in practice this is not the case, since core descriptions are normally less informative (close to the root they can possibly become $\bot$) than envelope descriptions.

As illustrated in Figure 10.6, each node of the tree contains constraints identified by a given keyword i.e. $\Sigma_{\text{PRE}}$ for precondition and $\Sigma_{\text{EFF}}$ for effect. In inner nodes they are approximated by core and envelope constraints, noted as $\Sigma$.

As it can be seen in the right side of Figure 10.6, while the size of constraints in envelope descriptions normally increases as they are closer to the root of the tree, constraints in core descriptions become smaller or even empty ($\bot$) as they approach the root.

In our implementation, core descriptions are constructed from the intersection of the constraints in core descriptions of children nodes with their respective variables and variable types computed as above. In the example in Figure 10.6, where the function $\text{children}(\Sigma_K')$ returns $\Sigma_K'$, the constraints identified by the same keyword $K \in \{\text{PRE, EFF}\}$ in the children of the node of $\Sigma_K'$, we have:

$$\Sigma_K'' = \bigcap_{\Sigma_K' \in \text{children}(\Sigma_K')} \Sigma_K'.$$

Conversely, envelope descriptions are constructed from the union of constraints in envelopes descriptions of children nodes are constructed in a similar way. As above,
the function \( \text{children}(\Sigma''_K) \) returns \( \Sigma'_K \), the constraints identified by the same keyword in the children of the node of \( \Sigma''_K \):

\[
\Sigma'_K = \bigcup_{\Sigma'_K \in \text{children}(\Sigma''_K)} \Sigma'_K.
\]

### 10.2.4 Best-First Search

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 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 [26].

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 (see Figure 10.7). 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 priority queue or the number of retrieved results exceed a certain threshold defined by the directory service provider.

Highly relevant to our work are SS trees, the GiST extensions described in [4] regarding heuristic 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 case, search is still highly efficient by the means of a query transformation scheme that exploits the tree structure of the index, as explained below.

### 10.2.5 Query Transformation

![Diagram](image)

Figure 10.7: Processing of a directory query. While the given DirQL expression is directly applied to leaf nodes (white), it has to be transformed for inner nodes (black).

Processing a user query requires traversing the GiST structure of the directory starting from the root node. The given DirQL expression is applied to leaf nodes of the directory tree, which correspond to concrete service descriptions. The constraints present in inner node $I$ of the GiST are the envelope (or union) 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 as above 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 a previous version of the directory we required the user to explicitly define these functions for inner nodes. This approach has several disadvantages: It exposes implementation details of the directory to the client (the client has to be aware of the GiST structure) and it requires more programming. Furthermore, inconsistencies between the selection and ranking functions for inner nodes and leaf nodes are difficult to detect and may result in incorrect or incomplete selection and ranking.

Currently, we follow a different 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).

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.

### 10.2.6 Relaxation of Set Inclusions

![Figure 10.8: Inclusion relations between components of 3 sets. X corresponds to Γ. For upper bounds, p is Σ and P is Σ, for lower bounds P is Σ and p is Σ.](image)

For determining some of these transformation rules, we have started by analyzing the possible relations between three sets p, P and X. In Figure 10.8 we have considered all components that could form the three sets such that they are in the relation $p \subseteq P$ and $p$, $P$ and $X \neq \perp$. We have noted these components $a$, $b$, $c$, $d$, $e$. We have represented set $P$ by a double line, $p$ by a thicker line and $X$ by a dashed line. As it can be seen on the figure the set $p$ could be any combination of $a$ and $b$. Based on the inclusion relation, $P$ could be expressed as $p$ plus possibly a combination of the components $c$ and $d$. Finally $X$ could be expressed as any combination of the components $a - e$.

Based on the decomposition above we have inferred a number of possible inclusion implications between the three sets. We have considered five possible set relations: equivalence $\equiv$, subset $\subseteq$, superset $\supseteq$, overlapping sets $\cap \neq \perp$ and disjoint sets $\cap \neq \perp$.

In Table 10.6 we have extrapolated the relations between $p$, $P$ and $X$ and we have listed all other possible inclusion implications between a constraint in a request $\Gamma$ and a constraint in an advertisement $\Sigma$, as well as the corresponding pruning conditions for its core $\Sigma$ and envelope $\Sigma$ approximations. If no particular relation could be deduced, we used the truth symbol $\top$ (i.e., to make the implication a tautology).

The lower part in Table 10.6 applies if the selection predicate appears negated in the query formula (e.g., in the form $\neg \text{SignatureMatch}(\ldots)$ or $\neg \text{SpecificationMatch}(\ldots)$). For determining the pruning conditions for this case, we used the fact that $A \Rightarrow B$ is logically equivalent to $\neg B \Rightarrow \neg A$ and the previously determined implications of positive relations between $\Gamma$ and $\Sigma$ resp. $\Sigma$. 

57
Positive relations \((\text{SignatureMatch}(\cdots) \text{ or } \text{SpecificationMatch}(\cdots))\).

Note: we assume \(\Gamma, \Sigma, \Sigma, \Sigma \neq \bot\).

\[
\begin{array}{ll}
\Gamma \equiv \Sigma & \Rightarrow \Gamma \subseteq \Sigma \\
\Gamma \subseteq \Sigma & \Rightarrow \Gamma \subseteq \Sigma \\
\Gamma \supseteq \Sigma & \Rightarrow \Gamma \cap \Sigma \neq \bot \\
\Gamma \cap \Sigma \neq \bot & \Rightarrow \Gamma \cap \Sigma \neq \bot \\
\Gamma \cap \Sigma = \bot & \Rightarrow \top
\end{array}
\]

We apply \((A \Rightarrow B) \leftrightarrow (\neg B \Rightarrow \neg A)\) and get:

Negative relations \(\neg \text{SignatureMatch}(\cdots) \text{ or } \neg \text{ SpecificationMatch}(\cdots)\).

Note: we assume \(\Gamma, \Sigma, \Sigma, \Sigma \neq \bot\).

\[
\begin{array}{ll}
\neg(\Gamma \equiv \Sigma) & \Rightarrow \neg(\bot) \\
\neg(\Gamma \subseteq \Sigma) & \Rightarrow \neg(\bot) \\
\neg(\Gamma \supseteq \Sigma) & \Rightarrow \neg(\Gamma \equiv \Sigma) \\
\neg(\Gamma \cap \Sigma \neq \bot) & \Rightarrow \neg(\bot) \\
\neg(\Gamma \cap \Sigma = \bot) & \Rightarrow \neg(\Gamma \cap \Sigma = \bot)
\end{array}
\]

Table 10.6: Selection criteria and required pruning conditions for an advertisement constraint \(\Sigma\) (leaf node) and its core \(\Sigma\) and envelope \(\Sigma\) approximations (inner nodes).