DIP
Data, Information and Process Integration with Semantic Web Services

FP6 – 507483

Document

DIP Interface Description Ontology

Michael Stollberg (UIBK)
Stefania Galizia (OU)
Jacek Kopecky (UIBK)
Matthew Moran (NUIG)
James Scicluna (UIBK)
Axel Polleres (UIBK)
Jens Lemcke (SAP)
Laurent Henoque (ILOG)
Barry Norton (OU)
Edward Kilgarriff (NUIG)
Mathias Kleiner (ILOG)

January 20, 2006
SUMMARY

This document specifies the common description ontology and languages for choreography interfaces and orchestration in DIP. It accompanies the respective DIP deliverables D3.4 “An Orchestration and Business Process Ontology” and D3.5 “An Ontology for Web Service Choreography”.

The contributions of this document are:

- explication of the approach for defining a common description ontology and languages for choreography interfaces and orchestration, summarized as behavior interface descriptions
- definition of the meta-layer ontology for the structure of behavior interface descriptions
- detailed specification of the description language elements
  1. The user language based on UML2 Activity Diagrams
  2. Ontologized Abstract State Machines as the formal model
  3. translation from the user language to the formal model
- specification of the grounding of behavior interface to WSDL
- discussion and position of the approach within related work.

Disclaimer: The DIP Consortium is proprietary. There is no warranty for the accuracy or completeness of the information, text, graphics, links or other items contained within this material. This document represents the common view of the consortium and does not necessarily reflect the view of the individual partners.
DIP Interface Description Ontology

**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>
</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>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>EU Project Officer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kai Tullius</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Deliverable Number</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>DIP Interface Description Ontology</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Work Package Number</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Service Ontologies and Service Description</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Status</th>
<th>Nature</th>
<th>Dissemination Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>version 0.1</td>
<td>prototype □ report □ dissemination □ ontology □</td>
<td>public □ consortium □</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Authors (Partner)</th>
<th>Resp. Author</th>
<th>E-mail</th>
</tr>
</thead>
<tbody>
<tr>
<td>see title page</td>
<td>Michael Stollberg</td>
<td><a href="mailto:michael.stollberg@deri.org">michael.stollberg@deri.org</a></td>
</tr>
<tr>
<td>Partner</td>
<td>UIBK</td>
<td></td>
</tr>
<tr>
<td>Phone</td>
<td>+43 (0)512-507-6479</td>
<td></td>
</tr>
</tbody>
</table>

**Abstract (for dissemination)**
The objective of work-package 3 will be to employ the ontology and Semantic Web infrastructure by adding semantics to the web service description. This deliverable D3.4 addresses orchestration and business process ontology that is an important component for the Semantic Web and Web Services usage and as consequence, used for several purposes in DIP. In this context, the D3.4 aims to define the formalisms and build an orchestration and business process ontology that will provide the basis for advanced querying, reasoning and constraints implementation to be used in web services composition.

**Keywords**
Web Service Interface, Choreography, Orchestration Description, Languages
### Version Log

<table>
<thead>
<tr>
<th>Issue Date</th>
<th>Rev No.</th>
<th>Author</th>
<th>Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>09-11-05</td>
<td>1</td>
<td>Michael Stollberg</td>
<td>document creation and first editing</td>
</tr>
<tr>
<td>17-11-05</td>
<td>2</td>
<td>Michael Stollberg</td>
<td>integrated partners' input</td>
</tr>
<tr>
<td>23-11-05</td>
<td>3</td>
<td>Michael Stollberg</td>
<td>integrated changes from Vienna meeting</td>
</tr>
<tr>
<td>27-11-05</td>
<td>4</td>
<td>Stefania Galizia</td>
<td>added related work about ws technologies; added dio.bib</td>
</tr>
<tr>
<td>27-11-05</td>
<td>5</td>
<td>Michael Stollberg</td>
<td>corrected meta-model layer definition</td>
</tr>
<tr>
<td>28-11-05</td>
<td>6</td>
<td>Stefania Galizia</td>
<td>added Formalisms for Process and Dynamics Representation</td>
</tr>
<tr>
<td>05-12-05</td>
<td>7</td>
<td>Laurent Henocque</td>
<td>improved corrected section uml2ad and partially corrected bib file</td>
</tr>
<tr>
<td>05-12-05</td>
<td>8</td>
<td>Stefania Galizia</td>
<td>correct summary and some correction in section related work</td>
</tr>
<tr>
<td>05-12-05</td>
<td>9</td>
<td>Michael Stollberg</td>
<td>final editing for submission for internal review</td>
</tr>
<tr>
<td>15-12-05</td>
<td>10</td>
<td>Laurent Henocque</td>
<td>applied improvement suggested by James Sicluna as point 2 in introdution</td>
</tr>
<tr>
<td>28-12-05</td>
<td>11</td>
<td>Michael Stollberg</td>
<td>editing wrt reviewers' comments and partner input</td>
</tr>
<tr>
<td>17-01-06</td>
<td>12</td>
<td>Barry Norton</td>
<td>corrected errors and rewrote Future Work</td>
</tr>
<tr>
<td>19-01-06</td>
<td>13</td>
<td>Stefania Galizia</td>
<td>Updated template</td>
</tr>
</tbody>
</table>

### Reviewers

<table>
<thead>
<tr>
<th>Reviewer</th>
<th>E-mail</th>
<th>Phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thomas Haselwanter</td>
<td><a href="mailto:thomas.haselwanter@deri.org">thomas.haselwanter@deri.org</a></td>
<td></td>
</tr>
<tr>
<td>Partner UIBK</td>
<td></td>
<td>+43 (512) 507-6450</td>
</tr>
<tr>
<td>James Sicluna</td>
<td><a href="mailto:james.sicluna@deri.org">james.sicluna@deri.org</a></td>
<td></td>
</tr>
<tr>
<td>Partner UIBK</td>
<td></td>
<td>+43 (512) 507-6485</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,  
  PO Box 11, County Hall, Duke Street,  
  Chelmsford, Essex, CM1 1LX,  
  United Kingdom.  
  Email: maryr@essexcc.gov.uk  
  Tel: +44 (0)1245 436524 |
| Forschungszentrum Informatik | FZI | Andreas Abecker  
  Forschungszentrum Informatik  
  Haid-und-Neu Strasse 10-14  
  76131 Karlsruhe  
  Germany  
  E-mail: abecker@fzi.de  
  Tel: +49 721 96540 |
<table>
<thead>
<tr>
<th>Company</th>
<th>Contact Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Institut für Informatik,</td>
<td>UIBK</td>
</tr>
<tr>
<td>Leopold-Franzens Universität</td>
<td>Prof. Dieter Fensel</td>
</tr>
<tr>
<td>Innsbruck</td>
<td>Institute of computer science</td>
</tr>
<tr>
<td></td>
<td>University of Innsbruck</td>
</tr>
<tr>
<td></td>
<td>Technikerstr. 25</td>
</tr>
<tr>
<td></td>
<td>A-6020 Innsbruck, Austria</td>
</tr>
<tr>
<td></td>
<td>Email: <a href="mailto:dieter.fensel@deri.org">dieter.fensel@deri.org</a></td>
</tr>
<tr>
<td></td>
<td>Tel: +43 512 5076485</td>
</tr>
<tr>
<td>ILOG SA</td>
<td>ILOG</td>
</tr>
<tr>
<td></td>
<td>Christian de Sainte Marie</td>
</tr>
<tr>
<td></td>
<td>9 Rue de Verdun, 94253, Gentilly, France</td>
</tr>
<tr>
<td></td>
<td>E-mail: <a href="mailto:csma@ilog.fr">csma@ilog.fr</a></td>
</tr>
<tr>
<td></td>
<td>Tel: +33 1 49082981</td>
</tr>
<tr>
<td>inubit AG</td>
<td>inubit</td>
</tr>
<tr>
<td></td>
<td>Torsten Schmale, inubit AG,</td>
</tr>
<tr>
<td></td>
<td>Lützowstraße 105-106 D-10785 Berlin, Germany</td>
</tr>
<tr>
<td></td>
<td>E-mail: <a href="mailto:ts@inubit.com">ts@inubit.com</a></td>
</tr>
<tr>
<td></td>
<td>Tel: +49 30726112 0</td>
</tr>
<tr>
<td>Intelligent Software Components</td>
<td>iSOCO</td>
</tr>
<tr>
<td>S.A.</td>
<td>Dr. V. Richard Benjamins, Director R&amp;D</td>
</tr>
<tr>
<td></td>
<td>Intelligent Software Components, S.A.</td>
</tr>
<tr>
<td></td>
<td>Pedro de Valdivia 10</td>
</tr>
<tr>
<td></td>
<td>28006 Madrid, Spain</td>
</tr>
<tr>
<td></td>
<td>E-mail: <a href="mailto:rbenjamins@isoco.com">rbenjamins@isoco.com</a></td>
</tr>
<tr>
<td>NIIA WEB Solutions</td>
<td>NIWA</td>
</tr>
<tr>
<td></td>
<td>Alexander Wahler</td>
</tr>
<tr>
<td></td>
<td>NIWA WEB Solutions</td>
</tr>
<tr>
<td></td>
<td>Niederacher &amp; Wahler OEG,</td>
</tr>
<tr>
<td></td>
<td>Kirchengasse 13/1a</td>
</tr>
<tr>
<td></td>
<td>A-1070 Wien</td>
</tr>
<tr>
<td></td>
<td>E-mail: <a href="mailto:wahler@niwa.at">wahler@niwa.at</a></td>
</tr>
<tr>
<td></td>
<td>Tel.: +43 131 95843 11</td>
</tr>
<tr>
<td>The Open University</td>
<td>OU</td>
</tr>
<tr>
<td></td>
<td>Dr. John Domingue</td>
</tr>
<tr>
<td></td>
<td>Knowledge Media Institute,</td>
</tr>
<tr>
<td></td>
<td>The Open University, Walton Hall,</td>
</tr>
<tr>
<td></td>
<td>Milton Keynes, MK7 6AA, UK</td>
</tr>
<tr>
<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>Tel.: +44 1908 655014</td>
</tr>
<tr>
<td>SAP AG</td>
<td>SAP</td>
</tr>
<tr>
<td></td>
<td>Dr. Elmar Dorner</td>
</tr>
<tr>
<td></td>
<td>SAP Research, CEC Karlsruhe</td>
</tr>
<tr>
<td></td>
<td>SAP AG</td>
</tr>
<tr>
<td></td>
<td>Vincenz-Priessnitz-Str. 1</td>
</tr>
<tr>
<td></td>
<td>76131 Karlsruhe, Germany</td>
</tr>
<tr>
<td></td>
<td>E-mail: <a href="mailto:elmar.dorner@sap.com">elmar.dorner@sap.com</a></td>
</tr>
<tr>
<td></td>
<td>Tel: +49 721 6902 31</td>
</tr>
<tr>
<td>Sirma AI Ltd.</td>
<td>Sirma</td>
</tr>
<tr>
<td></td>
<td>Atanas Kiryakov</td>
</tr>
<tr>
<td></td>
<td>Ontotext Lab, - Sirma AI EAD,</td>
</tr>
<tr>
<td></td>
<td>Office Express IT Centre, 3rd Floor</td>
</tr>
<tr>
<td></td>
<td>135 Tzarigradsko Chausse, Sofia 1784, Bulgaria</td>
</tr>
<tr>
<td></td>
<td>E-mail: <a href="mailto:atanas.kiryakov@sirma.bg">atanas.kiryakov@sirma.bg</a></td>
</tr>
<tr>
<td></td>
<td>Tel.: +359 2 9768 303</td>
</tr>
</tbody>
</table>
| Unicorn Solution Ltd. | Unicorn Solutions Ltd,  
| Malcha Technology Park 1  
| Jerusalem 96951,  
| Israel  
| E-mail: Jeff.Eisenberg@unicorn.com  
| Tel.: +972 2 6491111 |
|---|---|
| Vrije Universiteit Brussel | Pieter De Leenheer,  
| Starlab- VUB  
| Vrije Universiteit Brussel  
| Pleinlaan 2, G-10  
| 1050 Brussel, Belgium  
| E-mail: Pieter.De.Leenheer@vub.ac.be  
| Tel.: +32 (0) 2 629 3749 |
# Table of Contents

1 Introduction 1

2 Interface Description Ontology Structure 3
   2.1 Choreography and Orchestration as Behavior Interfaces 3
   2.2 Interface Descriptions Requirements and Approach 5
      2.2.1 Requirements for Behavior Interface Description 5
      2.2.2 Approach for Interface Description Ontology 6
   2.3 Meta-Model Layer Description Ontology 7
      2.3.1 MOF Model and Language 7
      2.3.2 Web Service Description Structure 8
      2.3.3 DIP Interface Description Structure 10

3 UML2 Activity Diagrams 12
   3.1 Z preliminary 14
   3.2 Activity Groups 14
      3.2.1 Classes 14
      3.2.2 Semantics 15
      3.2.3 Relations and roles 15
      3.2.4 Constraints 15
   3.3 Activity Nodes and Edges 16
      3.3.1 Classes 16
      3.3.2 Semantics 16
      3.3.3 Attributes 17
      3.3.4 Constraints 17
   3.4 Action and Object Nodes 18
      3.4.1 Classes 18
      3.4.2 Semantics 18
      3.4.3 Relations 19
      3.4.4 Constraints 19
   3.5 Control Nodes 19
      3.5.1 Classes 19
      3.5.2 Semantics 20
      3.5.3 Constraints 20
   3.6 Graphical Representation 21
   3.7 Usage and Tool Support 21

4 Ontologized Abstract State Machines 23
   4.1 Aim and Approach 23
      4.1.1 A Bird’s Perspective 23
      4.1.2 Purpose and Application 25
   4.2 Abstract State Machines 27
      4.2.1 Classical ASMs Overview 27
      4.2.2 Single-Agent ASMs 28
      4.2.3 Multi-Agent ASMs 30
   4.3 Language Constructs and Definitions 31
1.3.1 Meta-Model Structure ........................................... 31
1.3.2 Definitions ...................................................... 36
1.3.3 Formal Execution Semantics ..................................... 38

5 Translation User Language to Formal Model ................................. 40
5.1 Overview ............................................................. 40
5.1.1 Execution in WSML/ASM ........................................... 40
5.1.2 Expressing WSML/UML2AD Execution ............................ 41
5.2 Prerequisites ......................................................... 44
5.2.1 Ontology .......................................................... 44
5.2.2 Definitions ....................................................... 44
5.2.3 Normal Form ...................................................... 46
5.3 Translation ............................................................ 47
5.3.1 Global Structure .................................................. 48
5.3.2 Concepts ........................................................ 48
5.3.3 Transition Rules .................................................. 50
5.4 Conclusion ............................................................ 53

6 Grounding ............................................................... 54
6.1 Grounding WSMO Ontologies to XML .................................. 54
6.1.1 Background and Related Work ................................... 54
6.1.2 Approaches to Grounding WSMO Ontologies to XML .......... 58
6.1.3 Grounding by Creating Mappings at the Conceptual Level .... 58
6.1.4 Definition of Mapping from XML Schema to WSMO ......... 60
6.1.5 Summary .......................................................... 63
6.2 Grounding Ontologized ASMs to WSDL ............................... 63
6.2.1 WSDL Overview .................................................. 64
6.2.2 Grounding to existing WSDL descriptions ..................... 67
6.2.3 Generating WSDL from WSMO choreography .................. 74

7 Related Work .......................................................... 80
7.1 Web Service Technologies and Languages ............................. 80
7.1.1 WSDL based proposals .......................................... 80
7.2 Semantic based approaches .......................................... 81
7.2.1 WSMO ............................................................ 81
7.2.2 IRS-III ........................................................... 82
7.2.3 METEOR-S ....................................................... 82
7.2.4 OWL-S ........................................................... 83
7.2.5 SWS Framework ................................................ 84
7.3 Formalisms for Process and Dynamics Representation .................. 84
7.3.1 Process Algebras ................................................. 85
7.3.2 Situation Calculus ............................................... 86
7.3.3 Petri-Nets ........................................................ 86
7.3.4 Concurrent Transaction Logic .................................... 88
7.4 Existing work on business process and protocol languages ............ 90
7.4.1 Definition of Business Process .................................. 90
7.4.2 Definition of Business Protocol .................................. 90
7.4.3 Definition of Business Process and Protocol Ontology ....... 91
7.4.4 WS-BPEL ................................................................. 91
7.4.5 BPML/WSCI ............................................................ 92

8 Conclusions and Future Work ......................................... 95
  8.1 Summary .................................................................. 95
  8.2 Future Work ............................................................ 95
1 Introduction

This document specifies the meta-model layer ontology and languages for describing choreography interfaces and orchestrations of Semantic Web services in the DIP project. Containing the detailed specification of the description languages and their interrelation, this document accompanies the DIP deliverables D3.5 “An Ontology for Web Service Choreography” that explains the approach for choreography related descriptions and D3.4 “An Orchestration and Business Process Ontology” that is concerned with orchestration.

Choreography is concerned with the interactions of Web services with their users in order to consume a Web service, and orchestration is concerned with how a Web service uses and aggregates other Web services in order to achieve its functionality. In our approach, both choreography and orchestration are considered as interfaces that describe the external visible behavior of Web services and requesters for interaction. This document provides the specification of the common description language for both types of interfaces, referred to as the DIP Interface Description Ontology. The reason is that although choreography and orchestration are concerned with different aspects from a conceptual and functional perspective, the same description language can be applied for them. Hence, we define the structure and languages for describing behavioral interfaces in this document while the respective DIP deliverables D3.5 and D3.4 explain the requirements and intended usage of choreography and orchestration descriptions.

The DIP Interface Description Ontology specified in this document is an extension of the conceptual approach of choreography and orchestration interfaces undertaken in the Web Service Modeling Ontology WSMO [53]. Following conceptual implications for comprehensive meta-model description ontologies for Semantic Web services, end-user requests, and taking technical requirements for related DIP technologies into account, the requirements for choreography and orchestration descriptions can be summarized as follows: (a) support for ontologies as the underlying data model for the information interchanged, (b) appropriate representation of the communication process excepted and supported by a Web service for consuming its functionality, (c) an appropriate formal model with clearly defined semantics in order to allow reasoning on choreography related descriptions of Semantic Web services, (d) grounding to executable Web service technologies in order to allow invocation and consumption of Web services, and (e) a suitable graphical representation of choreography related descriptions of Semantic Web services.

With regard to this, the DIP Interface Description Ontology defines a layered description language that consists of a formal model and a higher-level user language along with a translation between them. The low-level formal model are ‘ontologized Abstract State Machines’, which is the description model for Web Service interfaces defined within the Web Service Modeling Ontology WSMO [54]. Abstract State Machines (short: ASMs) are used as the underlying formalism for specifying the dynamics of choreography interfaces, which provide a rich and flexible formalism with minimal ontological commitment. WSMO ontologies are inherently supported as the data model for information that are interchanged. Besides, the formal model serves as the input format of execution engines for client-service as well as service-service interaction that are under development. UML2 Activity Diagrams are used as the language for higher-level process construct definition along with a graphical representation.
main motivation for having chosen such a higher level language is that several tools, as for instance the composer, must reason about the static properties of choreographies and orchestrations, which cannot be deciphered from an ASM based operational semantics alone. More details are provided concerning this choice in Chapter 3 "UML2 Activity Diagrams" in this document. The operational semantics for these activity diagrams are defined by translating them to the formal model of ontologized ASMs. The intended usage scenario is that a service provider or requester models the choreography interface by creating UML2 Activity Diagram description supported by a graphical tool. Then, this definition is translated into the 'ontologized Abstract State Machines' representation. In consequence, both the UML2 as well as the ASM model can be used as interchange formats for choreography related descriptions in DIP.

Following the methodology for defining description models of OMG’s Meta Object facility that is also used in WSMO, the DIP Interface Description Ontology represents a meta-model layer ontology. Apart from specifying the overall ontology structure and the languages in detail, we discuss and position our approach within related work. We will see that current Web service technologies related to choreography have deficiencies in several aspects that we consider to be crucial, so that our model presents a novel approach for semantically describing choreographies and related aspects in Web services. For future versions of the DIP Interface Description Ontology, this model will be extended with Cashew, a comprehensive process model for Semantic Web services that supports richer process constructs as well as advanced reasoning support.

The document is structured as follows: Section 2 recalls the approach for choreography and orchestration descriptions as interfaces chosen in DIP, and specifies the general structure of the DIP Interface description ontology. The subsequent sections specify the ontology elements in detail, namely: Section 3 specifies the subset of UML2 Activity Diagrams used; Section 4 explains ontologized Abstract State Machines, the WSMO approach that provides the formal model; Section 5 defines the translation from the user language to the formal model; Section 6 specifies the Grounding towards WSDL as an executable, standardized Web service technology. Section 7 discusses related work and positions our approach therein, and Section 8 concludes the document and points out plans for the future development of the DIP Interface Description Ontology.
2 Interface Description Ontology Structure

This section introduces the approach taken for specifying the meta-model layer ontology for choreography and orchestration descriptions in DIP. We first recall the underlying understanding of choreography and orchestration descriptions as behavioral interfaces, and then explain the overall structure of the DIP Interface Description Ontology that results from the approach.

2.1 Choreography and Orchestration as Behavior Interfaces

The following recaptures the underlying understanding of choreography and orchestration to be behavior interfaces of Web services and requesters as exposed in the respective DIP deliverables D3.4 and D3.5.

The notions of choreography and orchestration are concerned with the behavior of Web services for consuming their functionality, respectively for achieving the functionality of a Web service by using and aggregating other Web services. The W3C defines Choreography and Orchestration as follows [34]:

Choreography

- A choreography defines the sequence and conditions under which multiple cooperating independent agents exchange messages in order to perform a task to achieve a goal state.

- Web Services Choreography concerns the interactions of services with their users. Any user of a Web service, automated or otherwise, is a client of that service. These users may, in turn, may be other Web Services, applications or human beings.

Orchestration

- An orchestration defines the sequence and conditions in which one Web Service invokes other Web Services in order to realize some useful function. That is, an orchestration is the pattern of interactions that a Web Service agent must follow in order to achieve its goal.

Refining these general definitions, we understand the respective interfaces of Web services as the most important aspects. An interface of a Web service describes how the service can interact with its environment with respect to the ordering of message exchange expected for consumption of the service functionality, respectively the information interchange and coordination of other Web services used for achieving the functionality of a Web service. The rationale for this conceptual model is that Web service usage and interaction is happening in a peer-2-peer manner, meaning that information interchange and cooperation is controlled by the Web services as the peers without any central control unit. A main implication of our conceptual model is that we do not need to describe client-service, respectively service-service interaction from a global perspective. Instead, we envision advanced conformance checks for determining the feasibility of interaction execution [39].

In consequence, we follow the conceptual model of WSMO [53] by defining two types of interfaces for describing Web services. The so-called Choreography Interface that describes the behavioral interface of a Web service for interaction with a
client that wants to consume the Web service functionality, and the **Orchestration** that defines the behavioral interface of a Web service for achieving its functionality by using and aggregating other Web services. Figure 2.1 shows the overall picture of these notions and their intended usage within Semantic Web service technology. Discussed in the respective DIP deliverables in more detail, the most important aspects of this conceptual model are:

![Figure 2.1: Choreography and Orchestration Interfaces of a Web Service](image)

- information interchange for consumption and cooperation of Web services happens in a peer-2-peer manner wherein Web services denote the peers that control the interaction instead of a central control unit.

- Hence, in correspondence to WSMO, we define two types of Web service interfaces: The choreography interface that describes the interaction behavior of a Web service for consuming its functionality, and the orchestration that describes how a Web service aggregates other Web services in order to achieve its functionality.

- a choreography interface description is mandatory for a Web service description in order to allow service consumption by clients; an orchestration is optional in dependence on how the service provider decides to realize the functionality of a Web service.

- the compatibility of the behavioral interfaces of clients and services that are supposed to interact is determined by respective conformance tests; this requires a sound formal model for choreography and orchestration interface descriptions.

- all client-service interaction for consuming a Web Service happens via the choreography interface of the service.

- when a Web Service A uses another Web Service B in its Orchestration, A consumes B through the Choreography of B.

- when a Web Service A uses another Web Service B in its Orchestration, A appears as a ‘normal’ client to B (meaning that B is not aware of itself being used by the orchestration of another Web Service).
2.2 Interface Descriptions Requirements and Approach

While choreography and orchestration descriptions as behavioral interfaces have different, interrelated application purposes as outlined above, they can be described by the same languages. The reason is that both types of behavior interfaces need to define the external visible communication behavior for interacting with clients or other services.

2.2.1 Requirements for Behavior Interface Description

The following summarizes the common requirements for describing choreography interfaces as well as orchestrations for Semantic Web service. These result from the overall description model developed within DIP for semantically describing Web services, from end-user requirements gathered, as well as attained from development plans for execution and reasoning technologies that deal with behavior interfaces and interactions between clients and Web services, respectively among Web services.

1. Ontologies as Data Model
   With respect to the overall design of Semantic Web services, ontologies are to be used as the underlying data model. This means that all resource descriptions are to be based on ontologies, and all data elements interchanged between client and Web service as well as between Web services are to be ontology instances. Thereby, support for the Semantic Web is assured inherently, and the basis for semantic interoperability as well as advanced information processing is given.

2. Sound Formal Model for Representation of Dynamics
   A major aim of Semantic Web services is to enable advanced, inference-based mechanisms for automating the Web service usage process. Regarding both choreography and orchestration within our approach, one central reasoning task commonly referred to as conformance testing is to determine whether for service consumption the interaction of a client and a Web service can be achieved successfully, respectively the interaction of Web services for achieving a functionality via an orchestration [59]. Another reasoning task is concerned with resolving process level mismatches that hamper successful interaction [16].
   Such techniques provide the actual benefit of semantic behavioral descriptions of Web services, as syntactic service descriptions do not support such reasoning tasks. The pre-requisite therefore is a sound formal model with unambiguous semantics for describing the dynamics within choreography interfaces and orchestrations. This formal model should be appropriate with respect to the dynamics that can occur in Web service interface descriptions, and it should integrate ontologies as the data model for the information that are to be interchanged.

3. Support for higher-level Process Constructs
   In order to allow definition of more complex dynamic constructs of the communication structure in choreography interface descriptions and to support advanced handling of these, higher-level process constructs should be supported. Especially general workflow patterns as defined in [62] should be taken under consideration therefore.
4. **Grounding to Executable Communication Technology**

In order to enable execution of Web service consumption by clients, choreography interface descriptions need to be aligned with executable Web service technologies. Commonly referred to as grounding, it is desirable to not restrict this to only certain executable communication technologies. In order to allow usage of existing Web services, respective standards shall be supported.

5. **Graphical User Language**

In order to provide sophisticated support for definition of interfaces, a suitable graphical representation should be provided. Editing, browsing, and maintenance of interface definitions in this graphical user language should be supported by respective Web service management tools.

### 2.2.2 Approach for Interface Description Ontology

As stated above, these requirements on suitable description languages are common for both choreography and orchestrations descriptions. In consequence, we develop a common description language for behavioral interfaces whereby some constructs are only used in orchestration descriptions. With respect to the five requirements determined above, our approach consists of the elements depicted in Figure 2.2 along with their interrelations.

![Figure 2.2: 2-Layered Description Languages for Choreography and Orchestration Interfaces in DIP](image)

Starting from the top, we have chosen UML2 Activity Diagrams as the graphical user language that is to be supported by respective tools for editing and managing Web service interface descriptions. Ontologies are used in the UML2 Activity Diagrams as the data model for the information that is to be sent or received within an interface definition. Apart from being a suitable graphical user language, UML2 Activity Diagrams support higher-level process constructs as required.

As the formal model for semantically describing choreography interfaces, we use so-called *ontologized Abstract State Machines* as the approach developed within WSMO for semantically describing the dynamics of Web service interface definitions [54].
While explaining the structure and semantics of this language in the Section 4 in detail, we denote the following benefits of this approach: (1) ontologies are inherently supported as the data model, (2) the approach is inherently integrated with other aspects for semantically describing Web services as defined in DIP, and (3) the language serves as a basis for respective execution technologies for client-service as well as service-service interaction.

This approach of layered languages appears to be appropriate as it satisfies all requirements for behavior interface descriptions and ensures compatibility and integration with respective DIP technologies.

- the approach of “ontologized ASMs” defined in WSMO provides a low-level semantic description. It is based on an highly flexible and expressive formalism, inherently supports ontologies as the data model for the information to be interchanged, and serves as a basis for grounding to executable Web service technologies

- UML2 Activity Diagrams provide a suitable language for graphically representing choreography and orchestration interface descriptions; these can be translated to the formal model of “ontologized ASMs” in order to precisely define their semantics

- The translation from the user language to the formal model integrates both layers and ensures unambiguous semantics of the layered languages.

2.3 Meta-Model Layer Description Ontology

On basis of the preceding examinations, the following specifies the overall structure of the DIP Interface Description Ontology. In terms of the OMG’s Meta Object Facility (MOF, a classification framework for system modelling [30], we understand the model for semantically describing interfaces of Web service for choreography and orchestrations as meta-model layer ontology. We thereby follow the approach for defining the Web Service Modeling Ontology WSMO, wherefore the MOF model appears to be an appropriate means for clearly classifying ontological definitions and designating their application purpose [52].

2.3.1 MOF Model and Language

MOF distinguishes 4 layers: the M-0 “information layer” is concerned with real data interchange, which in our setting refers to the interchange of concrete data between a client and a Web service during service consumption, respectively between web services during cooperation; the M-1 “model layer” is concerned with the description of a resource, which in our setting refers to a behavior interface description of a concrete Web service; the M-2 “meta-model layer” is concerned with the definition of how resources are described, which refers to the definition of description models and languages for behavioral interfaces of Semantic Web service - i.e. what we provide within this document. Finally, the M-3 “meta-meta-model layer” defines means for describing M-2 layer specifications. Figure 2.3 shows this model, and reveals the correlation between the DIP Interface Description Ontology and the WSMO ontology that itself is understood as a meta-model layer ontology for describing Semantic Web services.
As the M-3 layer language, we adopt the respective language developed in WSMO. As the most frequently used MOF meta-modeling constructs, this contains the Class construct, together with Attributes, the type of the Attributes and their multiplicity specifications, and the class-generalization construct sub-Class. The following rules are defined for M-2 layer modeling:

- the default multiplicity of attributes is multi-valued; single-valued multiplicity of attributes is explicitly stated
- for defining an attribute value type to be a union of several types, a new Class as super-Class of all the types required in the definition of the attribute (that represents the union of the single types), with the Constraint that each instance of this new Class is an instance of at least one of the types which are used in the union. This is defined by curly brackets with an enumeration of the Classes that represent the required types.

2.3.2 Web Service Description Structure

DIP Web service descriptions follow exactly the WSMO approach that defines a description model that is intended to encompass the information needed for automatically determining the usability of a Web service. As shown in Figure 2.4, a WSMO Web service description is comprised of three main elements: (1) non-functional properties, (2) a capability as the functional description of the service; (3) and any number of interfaces. In turn, any interface is composed by (i) a choreography that describes the interface for service consumption by a client, and (ii) an orchestration that describes how the functionality of the service is achieved by aggregating other Web services. These notion describe the functionality and behavior of a Web service, while its internal implementation is not of interest.

The following listing recalls the M-2 layer definition of a WSMO Web Service description as defined in [53] with further explanation below.

```plaintext
Class service
  hasNonFunctionalProperties type nonFunctionalProperties
  importsOntology type ontology
  usesMediator type {ooMediator, wwMediator}
  hasCapability type capability
  hasInterface type interface
```
As outlined above, a Web service description consists of non-functional properties, one or more capabilities as the functional description, and possibly several interfaces. Following the structure of WSMO element definitions, an interface imports terminology definitions as ontologies or via OO Mediators, and can have one or more choreography and orchestration definitions. The reasons for allowing several capabilities and interfaces for one Web Service are (a) the service might provide multiple functionalities that are defined in separate descriptions, and (b) that different groundings can be supported by the service.

WSMO defines choreography and orchestration descriptions as behavioral interfaces of a Web service that specify its interaction behavior for service consumption, respectively interaction with aggregated Web services. These are described by ontologized Abstract State Machines [54], as explained in Section 4 of this document.

1non-functional properties encompass information that is not related to the semantic description but provides information relevant for management, usage, and selection. We refer to the WSMO specification for the definition of non-functional properties [55].
2.3.3 DIP Interface Description Structure

With respect to the five requirements for DIP interface descriptions discussed above, we extend the WSMO specification for service interfaces while the overall Web service description structure remains the same. In a nutshell, we add UML2 Activity Diagrams as a higher-level process description language, use the approach developed in WSMO as the formal model, and define an algorithm for automated translation from UML2 Activity Diagrams to the formal representation of the dynamics in Web service interfaces.

The resulting M-2 layer definition for DIP interface descriptions is the following. The Class \texttt{interface} replaces the one of the WSMO definition discussed above, defining that the choreography and orchestration description of an interface are of type \texttt{DIPChoreography}, respectively \texttt{DIPOrchestration}. These classes themselves are subclasses of \texttt{DIPInterface} as the common description structure for behavior interface descriptions in DIP. We explain the distinct description elements of the M2-layer class \texttt{DIPinterface} below in more detail.

\begin{verbatim}
Class interface
  hasNonFunctionalProperties type nonFunctionalProperties
  importsOntology type ontology
  usesMediator type \{ooMediator, wgMediator, wwMediator\}
  hasChoreography type DIPChoreography
  hasOrchestration type DIPOrchestration

Class DIPinterface
  hasNonFunctionalProperties type nonFunctionalProperties
  hasStateSignature type stateSignature
  hasUMLDescription type ontology
  hasFormalDescription type transitionRules

Class DIPChoreography sub-Class DIPInterface

Class DIPOrchestration sub-Class DIPInterface
\end{verbatim}

\textbf{Non-Functional Properties} Following the structure of WSMO element definitions [53], the non-functional properties than can be defined for behavior interface descriptions are (in alphabetical order): Accuracy, Contributor, Coverage, Creator, Date, Description, Financial, Format, Identifier, Language, Network-related QoS, Owner, Performance, Publisher, Relation, Reliability, Rights, Robustness, Scalability, Security, Source, Subject, Title, Transactional, Trust, Type, Version.

\textbf{Import Ontology} defines the ontologies that are used as terminology definitions within an interface description.

\textbf{Used Mediator} in case data level heterogeneities need to be resolved between the ontologies used, OO Mediators are used. WG and WW Mediators can be used in orchestration descriptions for binding in- and outputs between the interfaces of Web services.
State Signature defines the information space of an interface description. This consists of the domain ontologies used as terminology by import (directly or via OO Mediators in case of need for data level mismatch resolution), and information roles that denote how ontology instances are used for information interchange at interaction via an interface. For the later, five different roles are defined as discussed in Section 4 in more detail.

- **static** - meaning that the extension of the concept cannot be changed. This is the default for all concepts and relations imported by the signature

- **controlled** - meaning that the extension of the concept can only be changed by the owner of the interface description

- **in** - meaning that the extension of the concept or relation can only be changed by the environment.

- **out** - meaning that the extension of the concept or relation can only be changed by the interface execution.

- **shared** - meaning that the extension of the concept or relation can be changed by the interface owner and the environment.

UML Description provides the interface description as an UML2 Activity Diagram. The type declaration ontology denotes that UML2 Activity Diagram definitions are stored as an WSML ontology. This allows to store, manage, and interchange UML2 descriptions of interfaces via ontology management infrastructure, as we explain in Section 3 in more detail.

Transition Rules provide the formal description of an interface as an ontologized Abstract State Machine. As specified in Section 4 in detail, this consists of a state signature and transition rules that represent the dynamics of the interaction behavior defined in an interface. While the state signature is the same as defined in a DIPInterface (see above), the transition rules represent another description element that is derived from UML2 Activity Diagrams by automated translation. Section 5 of this document explains the algorithm therefore.
3 UML2 Activity Diagrams

We choose a comprehensive subset of UML2AD as a language for expressing choreographies and orchestrations. This choice is a pragmatic compromise meant to ensure sufficient coverage of workflow patterns and business usage compliance. This decision is also backed by several significant moves made by the Business Process Modeling community recently, including the merger of BPMI and OMG.

Usage of Static Workflow Descriptions

One important push for having a static description of choreographies and orchestrations attached to Semantic Web Services is that one approach to automatic SWS composition in DIP (as detailed in deliverable D4.12) is based on using a finite model search algorithm. There, 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.

UML 2 Activity Diagrams as Language

The rationale for using UML2AD\(^1\) as a language is its wide market acceptance and the fact that it almost fully supports the whole list of workflow patterns\(^62\). We are aware of (and have struggled with) a number of difficulties raised by ambiguities in the UML2 specifications. The current subset chosen solves these issues both at the denotational level, and the operational level.

At the denotational level, we are using a subset of the Z language\(^57\) to document the diagram restrictions that we have chosen to enforce. Although these restrictions do not impair expressiveness, they significantly enhance the rigor of the associated diagrams. The choice of Z for static constraints here echoes the choice of ASMs for dynamicity. 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.

At the operational level, the semantics of the selected UML2AD subset are defined by the translation to ASMs, as further detailed in the current DIO deliverable annex.

Among other choices are mostly YAWL\(^65\), certainly the best choice wrt. workflow pattern coverage, but still lacking widespread editor support. YAWL diagrams also end up non being easily readable to untrained eyes which introduces an extra difficulty:

\(^1\)The UML documentation is available at http://www.uml.org/, and more specifically at http://www.omg.org/technology/documents/modeling_spec_catalog.htm#UML.
users would have to learn YAWL in addition to what they already know. YAWL authors themselves acknowledge the fact that UML, although with some ambiguities, provides good support for all workflow patterns.

**Z for Meta-Model Constraint Specification**

This section describes the precise UML2AD subset used in both choreography and orchestration. This description is presented using a subset of the UML metamodel, involving class diagrams as usual to introduce the concepts. The specification below only slightly differs from the corresponding subset of the official UML specification [29]. It restricts it in some places, and introduces a limited number of extra classes.

In order to produce an unquestionable specification, we have chosen not to use the UML constraint language OCL (as in [1]), but instead a fragment of the Z language, as shown in [2, 35]. This has several advantages:

- the limitations brought by the exclusive use of the dotted notation in OCL are overcome using Z, a language with extremely rich expressiveness
- all workflow well formedness rules can be presented unambiguously
- the Z specification listed in the latex source for this document has been entirely type checked using the *fuzz* type checker. It is hence free of lexical and type errors
- the constraints as listed in Z receive a direct translation to configuration rules in ILOG JConfigurator
- Z is extensible: it allows the declaration of user defined operators that complement the syntax. We use this feature to introduce the largely accepted dotted notation. As often as required and possible, Javascript or OCL like dotted statements will be used.

**Presentation Options**

The current section presents the workflow metamodel using a combination of class diagrams, plus the associated Z specification, instead of UML plus OCL as common. Each diagram introduces a number of classes, their relations, and attributes. Together with the diagram are the following sections:

- Classes: the formal declaration of diagram classes
- Attributes: the formal declaration of attributes as Z functions
- Relations and roles: the formal declaration of relations (as Z relations) and roles as Z functions
- Semantics: the operational semantics of the workflow constructs, which impact the ASM translation but are not presented other than textually here
- Constraints: the structural (well-formedness) constraints. They are presented as Z axioms using the previously declared classes and relations.

We begin by a short preliminary describing Z constructs needed in the sequel.
3.1 Z preliminary

Before entering the metalanguage description, we present the logical constructs used for the Z specification. For simplicity and readability, the present document will not introduce the detailed specification of a complete constrained object system in Z, but rather focus on the elements that complement the native UML class diagram semantics. Thus, a limited number of definitions are 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 $P WFObject$.

\[
WFClass \equiv P WFObject
\]

We also introduce the possibility to use UML’s OCL like dotted notation for dereferencing relation roles. This is achieved in several steps. First declare two latex commands \texttt{\callfun} and \texttt{\callrel} that both display as a dot ".", using:

\[
\newcommand{\callfun}{.}
\newcommand{\callrel}{.}
\]

Then inform the \texttt{fuzz} type checker that they are both infix operators (with identical priority "6") as:

\[
\textbf{\%\%inop} \ \callfun \ 6
\]

\[
\textbf{\%\%inop} \ \callrel \ 6
\]

Finally axiomatically define \texttt{\callfun} and \texttt{\callrel} as follows (where both display as the dot "cdot"):

\[
\begin{align*}
\callfun & : (WFObject \times (WFObject \rightarrow WFObject)) \rightarrow WFObject \\
\callrel & : (WFObjec\t \times (WFObjec\t \rightarrow WFClass)) \rightarrow WFClass
\end{align*}
\]

\[
\forall e_1 : WFObject; e_2 : (WFObject \rightarrow WFObject) \bullet e_1.e_2 = e_2(e_1)
\]

\[
\forall e_1 : WFObject; e_2 : (WFObject \rightarrow WFClass) \bullet e_1.e_2 = e_2(e_1)
\]

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

\[
| f : WFObject \rightarrow WFObject
\]

and an element \( o \) of $WFObject$, the notations \( f(o) \) and \( o.f \) are equivalent.

3.2 Activity Groups

3.2.1 Classes

\[
\begin{align*}
\text{ActivityGroup} & : WFClass \\
\text{InterruptibleActivityRegion} & : WFClass \\
\text{ActivityNode} & : WFClass \\
\text{ActivityEdge} & : WFClass
\end{align*}
\]
3.2.2 Semantics

- Groups: no special semantic, it just enables to group together a part of the activity. Web services can be represented as a group.

- InterruptibleRegions: used to model external choices, see the shipper choreography for an example. Whenever a token traverses an interrupting edge, all other tokens of the region are consumed.

3.2.3 Relations and roles

<table>
<thead>
<tr>
<th>relation</th>
<th>role type</th>
</tr>
</thead>
<tbody>
<tr>
<td>interrupts</td>
<td>ActivityEdge → InterruptibleActivityRegion</td>
</tr>
<tr>
<td>isInputOf</td>
<td>ActivityEdge → ActivityNode</td>
</tr>
<tr>
<td>isOutputOf</td>
<td>ActivityEdge → ActivityNode</td>
</tr>
<tr>
<td>incomingEdges</td>
<td>ActivityNode → P ActivityEdge</td>
</tr>
<tr>
<td>outgoingEdges</td>
<td>ActivityNode → P ActivityEdge</td>
</tr>
<tr>
<td>immediatelyContainedGroups</td>
<td>ActivityGroup → P ActivityGroup</td>
</tr>
<tr>
<td>immediatelyContainedNodes</td>
<td>ActivityGroup → P ActivityNode</td>
</tr>
<tr>
<td>nodeGroup</td>
<td>ActivityNode → ActivityGroup</td>
</tr>
</tbody>
</table>

\[
\forall n : \text{ActivityNode}; g : \text{ActivityGroup} \quad g = \text{nodeGroup}(n) \iff n \in \text{immediatelyContainedNodes}(g)
\]

\[
\forall n : \text{ActivityNode} \quad \text{incomingEdges}(n) = \{ e : \text{ActivityEdge} \mid e.\text{isInputOf} = n \}
\]

\[
\forall n : \text{ActivityNode} \quad \text{outgoingEdges}(n) = \{ e : \text{ActivityEdge} \mid e.\text{isOutputOf} = n \}
\]

3.2.4 Constraints

- InterruptibleRegions: Interrupting edges have source in the region and target outside the region
∀ x : ActivityEdge; y : InterruptibleActivityRegion | y = x.interrupts •
  x.isOutputOf.nodeGroup = y ∧
  x.isInputOf.nodeGroup ≠ y

3.3 Activity Nodes and Edges

Figure 3.2: Activity Nodes

Figure 3.3: Activity Edges

3.3.1 Classes

| ActionNode : WFClass |
| ObjectNode : WFClass |
| ControlNode : WFClass |
| DecisionNode : WFClass |
| MergeNode : WFClass |
| ForkNode : WFClass |
| JoinNode : WFClass |

| ObjectFlow : WFClass |
| ControlFlow : WFClass |

3.3.2 Semantics

The operational semantics of object and control flows are described in the UML as "traverse-to-completion" semantics. The aim of these semantics is to allow workflow not to enter undue self blocking states, that could be caused for instance by tokens mistakenly sent to an alternative outgoing path, and thus missing for a synchronization to occur via an other outgoing path.
The currently presented subset of UML2AD diagrams overcomes most difficulties by disallowing random alternative routes outgoing actions. In other words, when a token is produced by an action, it is presented to an output pin that has no more than one edge connected.

- Object Flows: carry data tokens
- Control Flows: carry control tokens.
- Guards: conditions expressing which decision node’s outgoing edge will receive a token.

### 3.3.3 Attributes

We define *Guard* as an uninterpreted set

\[
[\text{Guard}]
\]

and *else* a particular member of *Guard*:

\[
\text{else : Guard}
\]

We now specify the *guard* attribute as a partial function from *ActivityEdge* to *Guard*:

\[
\text{guard : ActivityEdge} \mapsto \text{Guard}
\]

### 3.3.4 Constraints

- **ActivityEdge:**
  - Only edges outgoing from a decision node can have a guard. Decision nodes are visually and formally presented with the other control nodes later in the document in Figure 3.5

\[
\forall e : \text{ActivityEdge}; g : \text{Guard} \mid g = \text{guard}(e) \bullet e.\text{isOutputOf} \in \text{DecisionNode}
\]

- Only one edge outgoing from the same decision node can have an *else* condition as the guard.

\[
\forall n : \text{DecisionNode} \bullet \\
\#\{e : \text{ActivityEdge} \mid n = \text{isOutputOf}(e) \land \text{else} = \text{guard}(e)\} = 1
\]

- **Control Flow:**
  Control flows may not have object nodes at either end

\[
\forall e : \text{ControlFlow} \bullet \\
e.\text{isInputOf} \notin \text{ObjectNode} \land e.\text{isOutputOf} \notin \text{ObjectNode}
\]
3.4 Action and Object Nodes

3.4.1 Classes

- **Ontology**: WFClass
- **Pin**: WFClass
- **InputPin**: WFClass
- **OutputPin**: WFClass
- **Event**: WFClass
- **AcceptEvent**: WFClass
- **SendEvent**: WFClass
- **OOMediator**: WFClass
- **Concept**: WFClass
- **Atomic**: WFClass
- **Composite**: WFClass
- **Bag**: WFClass
- **Set**: WFClass
- **List**: WFClass

3.4.2 Semantics

- **ActionNode**:
  - Denotes that a local action is realized at this node
  - Pins are used to receive and send data tokens
  - The inputs are synchronized (all incoming edges and input pins have to carry a token for the action to start)

- **OOMediators**: have no side effects. This is an additional construct from UML2AD specification

- **AbstractEvent**: this is an additional construct from UML2AD specification. Not executable, i.e any AbstractEvent has to be specialized
3.4.3 Relations

\[ \text{concept} : \text{ObjectNode} \rightarrow \text{Concept} \]
\[ \text{ontology} : \text{Concept} \rightarrow \text{Ontology} \]
\[ \text{node} : \text{Pin} \rightarrow \text{ActionNode} \]
\[ \text{inputPins} : \text{ActionNode} \rightarrow \text{P InputPin} \]
\[ \text{outputPins} : \text{ActionNode} \rightarrow \text{P OutputPin} \]

\[ \forall y : \text{ActionNode} \bullet \text{inputPins}(y) \cap \text{outputPins}(y) = \emptyset \]
\[ \forall x : \text{Pin}; y : \text{ActionNode} \bullet \text{node}(x) = y \Leftrightarrow x \in \text{inputPins}(y) \cup \text{outputPins}(y) \]

3.4.4 Constraints

- ObjectFlow:
  - Object Flow connects exclusively an output pin to an input pin (with the exception of decision and merge control nodes)

\[ \forall n : \text{ActionNode}; f : \text{ObjectFlow} | f.\text{isOutputOf} = n \bullet \\
\text{f.\text{isOutputOf}} \in n.\text{outputPins} \]
\[ \forall n : \text{ActionNode}; f : \text{ObjectFlow} | f.\text{isInputOf} = n \bullet \\
\text{f.\text{isInputOf}} \in n.\text{inputPins} \]

- The downstream object node type must be the same of the upstream object node type

\[ \forall f : \text{ObjectFlow}; s, t : \text{Pin} | \\
s = \text{isOutputOf}(f) \land t = \text{isInputOf}(f) \bullet \\
s.\text{ontology} = t.\text{ontology} \]

- AcceptEvent:
  No incoming activity edge

\[ \forall e : \text{ActivityEdge} \bullet e.\text{isInputOf} \notin \text{AcceptEvent} \]

- SendEvent:
  No outgoing activity edge

\[ \forall e : \text{ActivityEdge} \bullet e.\text{isOutputOf} \notin \text{SendEvent} \]

3.5 Control Nodes

3.5.1 Classes

AbstractSplit : WFClass
AbstractJoin : WFClass
InitialNode : WFClass
FinalNode : WFClass
FlowFinal : WFClass
ActivityFinal : WFClass
3.5.2 Semantics

- **AbstractSplit**: this is an additional construct from UML2AD specification. Not executable: any AbstractSplit has to be specialized.

- **AbstractJoin**: this is an additional construct from UML2AD specification. Not executable: any AbstractJoin has to be specialized.

- **MergeNode**: any token offered on any incoming edge is offered to the outgoing edge.

- **DecisionNode**: each token arriving can traverse to only one outgoing edge.

- **ForkNode**: incoming token duplicated to outgoing edges.

- **JoinNode**: when all incoming edges have tokens, one is created on outgoing edge. Only one incoming edge can be an object flow. Outgoing edge can be an object flow only if there is an object flow among the incoming edges (in this case, the incoming data token is sent to the outgoing edge).

- **Flow Final**: consumes one token.

- **Activity Final**: all tokens in the activity are consumed.

3.5.3 Constraints

- **AbstractSplit**: 1 incoming edge only.
  \[ \forall x : \text{AbstractSplit} \implies \#(x.\text{incomingEdges}) = 1 \]

- **AbstractJoin**: 1 outgoing edge only.
  \[ \forall x : \text{AbstractJoin} \implies \#(x.\text{outgoingEdges}) = 1 \]

- **JoinNode**: Only one incoming edge is an object flow.
  \[ \forall x : \text{JoinNode} \implies \#((x.\text{incomingEdges}) \cap \text{ObjectFlow}) \leq 1 \]
• InitialNode: no incoming edge

\[ \forall x : \text{InitialNode} \implies x.\text{incomingEdges} = \emptyset \]

• FinalNode: no outgoing edge

\[ \forall x : \text{FinalNode} \implies x.\text{outgoingEdges} = \emptyset \]

• DecisionNode: the edges coming into and out of a decision node must be either all object flows or all control flows

• MergeNode: the edges coming into and out of a decision node must be either all object flows or all control flows

\[ \forall x : \text{ActivityNode} \mid x \in \text{DecisionNode} \cup \text{MergeNode} \implies (x.\text{incomingEdges} \cup x.\text{outgoingEdges}) \subseteq \text{ObjectFlow} \lor (x.\text{incomingEdges} \cup x.\text{outgoingEdges}) \subseteq \text{ControlFlow} \]

3.6 Graphical Representation

The graphical representation of all the workflow constructs presented so far is sketched in Figure 3.6. These diagrams are from the UML 2 superstructure specification\(^2\).

3.7 Usage and Tool Support

Concluding this section, we outline the tolling support for UML2 Activity Diagrams as the user language for behavior interface descriptions.

At first, a facility for editing, browsing, and maintaining choreography and orchestration descriptions as UML2 Activity Diagrams is intended to be integrated within Web service editing and management environments, like WSMO Studio as a part of the DIP technology (see DIP Deliverable 4b.11). Secondly, UML2 Activity Diagrams serve as the specification language for Web service interfaces within the DIP composition engine for automatically creating Web service orchestrations (see DIP Deliverable 4a.12). In order to allow usage of existing infrastructure for managing choreography and orchestration descriptions as UML2 Activity Diagrams, these are stored as WSML ontologies. Hence, UML2 descriptions can be stored, retrieved, and interchanged by ontology infrastructures as developed in DIP work packages 2 and 4. The WSML ontology schema definition for this purpose is provided in Appendix A of this document.

\(^2\)http://www.uml.org/
<table>
<thead>
<tr>
<th>Construct</th>
<th>Notation</th>
<th>Construct</th>
<th>Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>AcceptEventAction</td>
<td><img src="image1" alt="Image" /></td>
<td>InitialNode</td>
<td><img src="image2" alt="Image" /></td>
</tr>
<tr>
<td></td>
<td></td>
<td>JoinNode</td>
<td><img src="image3" alt="Image" /></td>
</tr>
<tr>
<td></td>
<td><img src="image4" alt="Image" /></td>
<td>MergeNode</td>
<td><img src="image5" alt="Image" /></td>
</tr>
<tr>
<td>Action</td>
<td><img src="image6" alt="Image" /></td>
<td>SendEventAction</td>
<td><img src="image7" alt="Image" /></td>
</tr>
<tr>
<td>ActivityFinal</td>
<td><img src="image8" alt="Image" /></td>
<td>ControlFlow</td>
<td><img src="image9" alt="Image" /></td>
</tr>
<tr>
<td>DecisionNode</td>
<td><img src="image10" alt="Image" /></td>
<td>ObjectFlow</td>
<td><img src="image11" alt="Image" /></td>
</tr>
<tr>
<td>FlowFinal</td>
<td><img src="image12" alt="Image" /></td>
<td>Shortcut notation for</td>
<td><img src="image13" alt="Image" /></td>
</tr>
<tr>
<td></td>
<td></td>
<td>AcceptEventAction</td>
<td><img src="image14" alt="Image" /></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Shortcut notation for</td>
<td><img src="image15" alt="Image" /></td>
</tr>
<tr>
<td></td>
<td></td>
<td>SendEventAction</td>
<td><img src="image16" alt="Image" /></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Shortcut notation for</td>
<td><img src="image17" alt="Image" /></td>
</tr>
<tr>
<td></td>
<td></td>
<td>same partner send and</td>
<td><img src="image18" alt="Image" /></td>
</tr>
<tr>
<td></td>
<td></td>
<td>receive events</td>
<td><img src="image19" alt="Image" /></td>
</tr>
</tbody>
</table>

Figure 3.6: Graphical Representation for all constructs
4 Ontologized Abstract State Machines

The formal model of the DIP Interface Description Ontology are so-called ontologized Abstract State Machines as the approach for description behavior interfaces developed within the Web Service Modeling Ontology WSMO [54]. This approach adopts Abstract State Machines as the underlying formalism for describing the dynamics of behavior interfaces and integrate them with ontologies as semantic terminology definitions for the information that is to be interchanged. The following rationalizes the approach, and specifies the language constructs and their application for describing choreography interfaces and orchestrations.

4.1 Aim and Approach

The rationale for developing ontologized Abstract State Machines as a novel approach for describing the dynamics within Web services emerges from the underlying idea of behavior interfaces of Web services. As outlined introductory, we consider the interfaces of Web services as the most relevant aspect in client-service and service-service interaction. Furthermore, the corresponding requirements analysis has revealed that the same language constructs can be used for describing choreography and orchestration interfaces. The requirements for a low-level formal description model for behavioral interfaces are:

1. to provide representation means for the dynamics of the information interchange that takes place when a service is used and interacts with other services
2. to support ontologies as the underlying data model along with an appropriate communication technology for information interchange
3. to rely on a sound formal model that defines the semantics of service interface specifications in order to allow operations on them.

The aim of the ontologized Abstract State Machines is to provide a solution for this. Therefore, while Abstract State Machines (short: ASM) provide a reasonable formalism with respect to the first and the third aspect, the second aspect requires the integration of ontologies. Hence, the formal model uses ASMs for describing the dynamics of service interface along with ontologies as the data model for information interchange. Before specifying the language constructs, the following provides a high-level overview of the approach and its intended usage for describing choreography and orchestration interfaces and handle information interchange.

4.1.1 A Bird’s Perspective

As the base formalism for describing dynamics, the formal model for behavior interface descriptions developed in WSMO uses Abstract State Machines, ASM for short. ASMs are a high-level, abstract technique for validating complex systems or programs and provide a highly expressive, flexible, and formally sound means for representing dynamics [11]. The core principles of ASMs are that they are state-based, they represent a state by a formal algebra, and they model state changes by guarded transition
rules that change the values of functions and relations defined by the signature of the algebra. The reason for choosing ASMs as the underlying formalism for defining service interface definitions in WSMO is the generality and expressiveness provided on the one hand, and, on the other, they allow to overcome the ”Frame Problem” [26]. This refers to that a state in a dynamic system is defined by the current information existing at a certain point in time; at a state change, the changes on all information items have to be defined. In contrast to other formalisms, guarded transitions in an ASM fire in parallel such that each condition of the transitions is checked over the current state. Thus, the ASM model overcomes the frame problem as only information that is changed at a state transition needs to be defined within guarded transitions. We explain the aspects of Abstract State Machines relevant for behavior interface descriptions in Section [12] in more detail.

The second foundation of the approach are ontologies as semantic terminology definitions. Following the conventional AI-theory of ontologies [58], these consist of: Concepts that describe the entities of a domain that are characterized by Attributes; Relations as associations between concepts, whereby subsumption and membership relationships define the taxonomic structure of an ontology; Axioms that define constraints and complex aspects of the domain in terms of logical expressions; and Instances as concrete individuals of a concept or a relation. In order to provide the basis for semantic interoperability throughout communication and cooperation of Web services and clients, all information that is interchanged shall be based on ontologies. More precisely, each data element that is sent or received by an entity via some behavior interface is to be a valid instance of an ontology. Therefore, we need to define how instances are to be used within a behavior interface - i.e. which data element is sent or received by an entity in each state state of service consumption or interaction of several Web services. Apart from providing the basis for semantic interoperability and information processing techniques, the integration of ontologies as the data model for information interchange, our approach inherently provides support for the Semantic Web [24].

Ontologized Abstract State Machines integrate ASMs and ontologies as follows. Interaction for service consumption or cooperation is understood as a state-based process. This means, it commences in some start end, ensues an arbitrary sequence of states with respective information interchange, and terminates in some state [54]. In consequence, choreography and orchestration interfaces describe the communication behavior of an entity for participating in an interaction. Following the semantics of ASMs, states are defined in some algebra wherefore guarded transitions describe possible paths to the next states. The algebra is defined by ontologies, in particular by the ontology instances that are known or can be created by the entity. Thus, ontologized Abstract State Machines for describing behavior interfaces consist of the following three constructs:

- **a State Signature** $\Omega$ that defines the information space of a service interface as an ontology. It is defined as the ontological schema of the information interchanged in a service interface by denoting the used concepts, relations, and functions of ontologies. The communicative usage of this ontological schema information is indicated by sub-information spaces for ontology instances: $\Omega_{in}$ denotes the vocabulary of information received by the service interface; $\Omega_{out}$ the vocabulary of information that is provided by the service interface; $\Omega_{shared}$
denotes the vocabulary of information both received and provided by the service interface; \( \Omega_{\text{static}} \) defines the vocabulary of ontology notions that cannot be changed by the service interface, and \( \Omega_{\text{controlled}} \) denotes those that can only be changed by the service. Each of these sub-information spaces can have one or several Grounding for translating the formal behavior descriptions into executable Web service technology for communication and information interchange.

- **States** \( \omega(\Omega) \) that denote a status of the information space within the dynamics of a service interface that is defined by the class membership and by the attribute values of the ontology instances of \( \Omega \). A state denotes a stable status within the dynamics of a service interface that is existent as long as attribute values of instances are not changed, thus includes all communicative activities that do not change information in \( \omega(\Omega) \).

- **Guarded Transitions** \( T \) that specify the dynamics of a service interface. The general structure of \( T \) is: if \( \text{condition}(\omega) \) then \( \text{update}(\omega) \). The condition is an arbitrary logical formula on the current state \( \omega \) that guards the updates to the next state \( \omega' \). Updates define the changes on the information space performed in the transition to a subsequent state \( \omega' \). The changes are modifications on the ontology by adding, removing, or modifying ontology instances while the state signature remains unchanged (i.e. the concepts, relations, and axioms along with their respective sub-information space allocation). At a state change from \( \omega \) to \( \omega' \), all \( T \) are executed whose condition is satisfied.

The differences between choreography and orchestration descriptions occur in the Guarded Transitions. The condition refers to the current state \( \omega \) in both choreography and orchestration interfaces, whereby \( \omega \) is described from the subjective, individual perspective of the entity that communicates via the respective behavior interface. The update in choreography interfaces can only consist of updates as modifications of the ontology, i.e. adding or removing ontology instance data to be interchanged. Within design time orchestration descriptions, the update of a guarded transition can also be defined as a 'web service' or a 'goal' that is to be invoked or achieved in order to reach the subsequent state. The conceptual reason for this is explained in detail in D3.4 "An Orchestration and Business Process Ontology".

### 4.1.2 Purpose and Application

In principle, we understand this model to define an evolving ontology of the information space that progresses during service consumption or interaction. Thereby, each entity keeps and manages such an ontology for each interaction session that it is involved in via its choreography interface or via its orchestration.

In such an evolving ontology, \( \Omega \) contains the concepts, relations, and axioms as ontology schema on basis of a domain ontology. A state \( \omega(\Omega) \) is stable status of the information space of a service interface, defined by the concrete attribute values of ontology instances; the communicative activities to be performed on instance data are denoted by the sub-information spaces of \( \Omega \). The set of guarded transitions \( T \) defines state changes with regard to the evolution of the information space throughout the usage of the service interface. Figure [4.1] shows the structure of behavior interface descriptions with ontologized ASMs and the correlations of the description elements.
An interface - regardless of being a choreography or an orchestration interface - is described by a State Signature that defines the terminology on basis of domain ontologies, along with how instances of concepts and relations used for information interchange via sub-information spaces. Possible state transitions are defined by Guarded Transitions, consisting of conditions on the current state and updates as the information interchange that occurs as the transition to the next state. Each Web service or client that participates in an interaction via some behavior interface described in this way, whereby an interface defines the communication behavior subjectively from the perspective of the entity. The actual information interchange of concrete data is performed via grounding of the update activities towards a communication technology executable over the Web.

Concluding the introductory overview, this model appears to be a suitable low-level formal language as aspired in our model for semantically description behavior interfaces of Semantic Web services. The main merits of this model can be summarized as follows:

- Abstract State Machines provide a rigid formalism and a highly expressive means for representing dynamics with minimal ontological commitment (i.e. nearly all possible constructs of system dynamics can be represented without limitations or restrictions for modeling)

- ontologized ASMs have formal execution semantics (see below) that allow construction of engines for executing and controlling client-service as well as service-service interaction in a peer-2-peer manner, therewith complying to the concept of distributed computing of the Internet via Web services
• integrating ontologies as the data model for the information to be interchanged provides inherent support for the Semantic Web; besides, ontology techniques can be applied for semantically enabled information processing as well as advanced conformance tests [59]

• ontologized ASMs are not restricted to a specific communication technology, but can be grounded to any web-based information exchange and communication technology. While the current grounding as specified in Section 6 supports WSDL as the current quasi-standard, ontologized ASMs can be aligned with other communication technologies like Triple-Space Computing [25].

4.2 Abstract State Machines

Before specifying the structure and language constructs of ontologized ASMs, the following recalls the notions and properties of Abstract State Machines that are relevant for our purpose. We refer to [11] for a comprehensive overview of Abstract State Machines, their semantics and application.

4.2.1 Classical ASMs Overview

Formerly known as Evolving Algebras [33], Abstract State Machines provide means to describe systems in a precise manner using a semantically well founded mathematical notation. The core principles are the definition of ground models and the design of systems by refinements. Ground models define the requirements and operations of the system expressed in mathematical form. Refinements allow to express the classical divide and conquer methodology for system design in a precise notation which can be used for abstraction, validation and verification of the system at a given stage in the development process.

Abstract State Machines rely on a state-based model. States are represented by a formal algebra, and they model state changes by guarded transition rules. By representing ASM as rules, the sequence of operations and the message pattern instantiations are generated through the evaluation of conditions. A condition is a generic statement on the current situation, for instance, that an error has occurred. The executive part of the guarded transitions updates the state. The general form of a guarded transition is given below:

\[
if \ currentstate = s \land Cond \ then \ currentstate = s_1
\]

Shown in Figure 4.2, Abstract State Machines are divided into two main categories, namely basic or single-agent ASMs and Multi-Agent ASMs. The former express the behavior of a system within the environment. Multi-Agent ASMs allow to express the behavior of the system in terms of multiple entities that are collaborating to achieve a functionality. The latter can be further divided in two categories: Synchronous and Asynchronous Multi-Agent ASMs, both of which can be of a distributed or non-distributed nature.

As choreography and orchestration descriptions in our model represent interfaces of single agents (i.e. a Web service or a client), we use basic, single-agent ASMs for
Figure 4.2: Categories of Abstract State Machines

describing both types of behavior interfaces. This means that our interface execution model is defined in terms of finite set of transition rules which are execute in parallel. Thereby, ASMs have been chosen as the underlying model for the following three reasons:

1. **Minimality**: ASMs provide a minimal set of modeling primitives, i.e., enforce minimal ontological commitments. Therefore, they do not introduce any ad-hoc elements that would be questionable to be included into a standard proposal.

2. **Maximality**: ASMs are expressive enough to model any aspect around computation.

3. **Formality**: ASMs provide a rigid framework to express dynamics.

### 4.2.2 Single-Agent ASMs

A single-agent ASM (most commonly known as Basic ASM) is defined in terms of a finite set of transition rules which are executed in parallel. It may involve non-determinism as described below.

**Basic Transition Rules**

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

\[ f(t_1, ..., t_n) := t \]

The execution of a set of such updates is carried out by changing the value of the occurring functions \( f \) at the indicated arguments to the indicated values in parallel. Hence, the parameters \( t_i \) and \( t \) are for example evaluated to \( v_i, v \). The value of \( f(v_1, ..., v_n) \) is then updated to \( v \) which represents the value of the function \( f(v_1, ..., v_n) \) in the next state. The pair of the function name \( f \) specified by the signature and the optional arguments \( (v_1, ..., v_n) \) (which is a list of dynamic parameter values of any type), are called locations. These locations form the concept of the basic ASM object containers or memory units. The location-value pairs \( (loc, v) \) are called updates and represent a basic unit of state change in the ASM.

More complex transition rules are defined recursively, as follows. (Note that for the sake of clarity, we slightly deviate here from the original syntax used in [11].) First, transition rules can be guarded by a Condition as follows:
if Condition then Rules endIf

Here, the Condition is an arbitrary closed first order formula. Such a guarded transition rule has the semantics that the Rules in its scope are executed in parallel, whenever the condition holds in the current state. Next, basic ASMs allow some form of universally quantified parallelism by transition rules of the form:

forall Variable with Condition do Rules(Variable) endforall

Here, the Variable is a variable occurring freely in Condition with the meaning that the Rules[Variable/Value] are executed in parallel for all possible bindings of the Variable to a concrete Value such that the Condition[Variable/Value] holds in the current state. Here, Condition[Variable/Value] respectively Rules[Variable/Value] stand for the condition (or rule, resp.) where each occurrence of Variable is replaced by Value. Similarly, basic ASMs allow for non-deterministic choice by transition rules of the form:

choose Variable with Condition do Rules(Variable) endChoose

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

A single ASM execution step consists of the following four steps. These steps are repeated until no condition of any rule evaluates to true, i.e. the unfolding yields an empty update set. In case of inconsistent updates, the machine run is either terminated or an error is reported (or both).

1. Unfold the rules, according to the current state and conditions holding in that state, to a set of basic updates.
2. Execute simultaneously all the updates.
3. If the updates are consistent. I.e. no two different updates update the same location with different values, which means that there must not be a pair of updates \(((loc, v),(loc, v'))\) with \(v \neq v'\), then the result of execution yields the next state.
4. All locations which are not affected by updates, keep their values.

Function Classification

ASMs define a classification for functions that can be subject to updates or used in conditions. All functions are either static or dynamic. On the one hand, static functions never change during a run of a machine. Dynamic functions can be classified in four other categories, namely, controlled, monitored (or in), interaction (or shared) and out. Controlled functions are directly updatable by the rules of the machine M only. Thus, they can neither be read nor updated by the environment. Monitored functions can only be updated by the environment and read by machine M and hence constitute the externally controlled part of the state. Shared functions can be read
and updated by both the environment and the rules of the machine $M$. Out functions can be updated but not read by $M$, but can be read by the environment. Furthermore, ASMs define the so-called derived functions. These are functions neither updatable by the machine or the environment but which are defined in terms of other static and dynamic (and derived) functions.

As we will see later, we will base our model of the behavioral aspects of a single service on basic ASMs, operating on dynamic WSMO Ontologies, describing the state of the machine in terms of concepts, relations and their instances, where we can define the ontological axioms in terms of derived functions.

### Control State ASMs

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

```plaintext
if $ctl\_state = i$ then
  if $Cond1$ then $Rule1$
  $ctl\_state := j_1$
endIf
...
if $Cond_n$ then $Rule_n$
$ctl\_state := j_n$
endIf
endIf
```

### Modularization

In order to structure and modularize ASM descriptions, it is allowed to define modules which take the following form:

```plaintext
ModuleName$(Variable_1, ..., Variable_n) = Rules$
```

In the definition of other rules these modules can be used as (possibly recursive) "rule calls" only when the parameters are instantiated by legal values (objects, functions, rules, so that the resulting rule has a well defined semantical meaning on the basis of the (informal) explanations given in the previous subsection. See [11] for details.

Such modularization may be viewed as submachine calls which can also be recursive and nested, nut not directly as composition of different (possibly distributed) ASMs since in a scenario with multiple ASMs run by different agents one need to consider details such as synchronous vs. asynchronous invocation, different clocks, etc. which is discussed in the following subsection.

### 4.2.3 Multi-Agent ASMs

As described above, there are two types of Multi-Agent ASMs, namely, synchronous and asynchronous. A synchronous Multi-agent ASM consists of a set of basic ASMs each running their own rules and which are synchronized by an implicit global system
clock. Such ASMs are equivalent to the set of all single-agent ASMs operating in the
global state over the union of their state signatures. The global clock is considered as a
step counter. Synchronous ASMs are particularly useful for analysing the interaction
between components using precise interfaces over common locations. We consider this
model insufficient for the description of the collaboration of Web services.

Asynchronous ASMs consist of a finite number of independent agents each executing
a basic or structured ASM in its own local state. The problem which arises in such
a scenario is that moves of the different agents cannot be compared due to different
data, clocks and duration of execution. Furthermore, the global state is difficult to
define since different agents may partially share the same state(s) or may not. The
coherence condition for such ASMs is the well-definedness for a relevant portion of a
state in which an agent is supposed to perform a step, thus providing the notion of
"local" stable view of "the" state in which an agent makes a move.

4.3 Language Constructs and Definitions

On basis of the preceding examinations, the following specifies and explains the lan-
guage constructs of ontologized Abstract State Machines along with their semantics.
At first, we define the structure of interface descriptions in the formal model, then pro-
vide definitions for the specification language, and finally define the formal execution
semantics of ontologized Abstract State Machines.

4.3.1 Meta-Model Structure

The following defines the structure of behavior interface descriptions with ontologized
ASMs. We base the description of the behavior interfaces exposed by a single service
on the basic ASM model discussed above. An interface description inherits the core
principles of such kind of ASMs, which summarized are: (1) they are state-based, (2)
they represents a state by a signature, and (3) they model state changes by transition
rules that change the values of functions and relations defined by the signature of the
algebra.

As outlined above (see Section 4.1), behavior interface descriptions consist of state
signatures, states, and guarded transitions. As states occur during runtime and hence
are not explicitly modeled, the description of a behavioral interface consists of the
state signature and guarded transitions along with non-functional properties for re-
source management. Following the overall model presented in Section 2 choreography
and orchestration descriptions represent sub-classes of DIP interface descriptions and
described by the same means. The following shows the meta-model layer definition
wherefore we use the meta-model layer definition language introduced in Section 2.3.

Class wsmoIinterface
  hasNonFunctionalProperties type nonFunctionalProperties
  hasStateSignature type stateSignature
  hasTransitionRules type transitionRules

Non-Functional Properties
the non-FunctionalProperties that can be defined for the formal description of an
interface are the same as for overall DIP interface descriptions as defined in Section 2.3
State Signature
defines the state ontology used by the service together with the definition of the types of modes the concepts and relations may have. See further definition below.

Transition Rules
Transition rules that express changes of states by changing the set of instances. See further definition below.

The formal terminology definitions used are compliant with the ontologies and OO Mediators defined for the overall DIP interface description, see the definition in Section 2.3.

The integration of ontologies as the data model is achieved by defining the state signature as a WSMO ontology. This means that a state signature consists of concepts, their attributes, relations and axioms over these. Instead of dynamic changes of function values as represented by dynamic functions in ASMs we allow the dynamic modification of instances and attribute values in the state ontology. Note that an interface describes the interaction with respect to a single instance of the choreography.

The state for a given signature of an interface is defined by all legal WSMO identifiers, concepts, relations and axioms. The elements that can change and that are used to express different states of an interface are instances of concepts and relations which are used similar to locations in ASMs. These changes are expressed in terms of creation of new instances or changes of attribute values. The key extension compared with basic ASMs based above is that the machine signature is defined in terms of a WSMO ontology and the logical language used for expressing conditions is WSML.

State Signature

The signature of an interface defines the information space over which the guarded transitions are executed. Therefore, it defines domain ontologies used in the interface description together with the definition of the types of modes the concepts and relations may have. In consequence, state signatures are described by (1) importing ontologies (possibly more than one) as the domain terminologies used, and (2) a set of statements defining the modes of the concepts and a set of update functions. The following shows the meta-model layer definition of state signatures with further explanations below.

Class stateSignature

hasNonFunctionalProperties type nonFunctionalProperties
importsOntology type ontology
hasIn type in
hasOut type out
hasControlled type controlled
hasShared type shared
hasStatic type static

While the imported ontology denotes the formal terminology definitions used in the interface description, the latter five description elements denote the sub-information space that denote the information on how ontology instances are used in information interchange. In a similar way to the classification of locations and functions in ASMs, the concepts and relations of an ontology are marked to support a particular role (or mode). These roles are of five different types:
• *static* - meaning that the extension of the concept cannot be changed. This is the default for all concepts and relations imported by the signature

• *controlled* - meaning that the extension of the concept can only be changed by the owner of the interface description

• *in* - meaning that the extension of the concept or relation can only be changed by the environment. A **grounding mechanism** for this item implements *write* access for the environment

• *out* - meaning that the extension of the concept or relation can only be changed by the choreography execution. A **grounding mechanism** for this item implements *read* access for the environment

• *shared* - meaning that the extension of the concept or relation can be changed by the interface owner and the environment. A **grounding mechanism** for this item implements *read/write* access for the environment.

Since Web services deal with actual instance data, the classification inherits to instances of the respectively classified concepts and relations. That is, instances of *controlled* concepts and relations can only be created and modified by the interface, instances of *in* concepts can only be read by the interface, instances of *out* concepts can only be created by the interface but not read or further modified after its creation. Instances of *shared* concepts and relations are supposed to be read and written by both the interface and possibly the environment, i.e. can also be modified after creation. The latter role become particularly important for grounding mechanisms alternative to WSDL which do not rely on strict message passing such as semantically enabled Tuple Spaces [25].

As the default, the mode *static* for concepts of the imported ontologies not listed explicitly in the modes statements is static. Note: It is not allowed to assign one of the modes *in* or *out* to concepts which have explicitly defined instance data in the ontology imported by the state signature.

**Transition Rules**

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

• **add**(fact)

• **delete**(fact)

• **update**(fact$_{old}$ = fact$_{new}$) or simply **update**(fact$_{new}$)
Thereby, a fact can be either a membership fact (memberOf), an attribute fact (hasValue) or a combination in the form of a WSML molecule abbreviating conjunctions of membership and attribute facts (cf. [19]). The add modifier adds a new fact to the state unless it is already present. The delete modifier deletes a fact from the state, if present. The update modifier in its first form marks the combination of deleting an old fact and adding a new one; the second form of update deletes all class membership or attribute values for a particular attribute and replaces these by the new fact. See the following section for the formal definition of the fact modifiers.

More complex transition rules are defined recursively, analogous to classical ASMs by if-then, forAll-do and choose-do rules as described above:

\[
\text{if Condition then Rules endIf}
\]

\[
\text{forAll Variable with Condition do Rules(Variable) endForAll}
\]

\[
\text{choose Variable with Condition do Rules(Variable) endChoose}
\]

Compared with classical ASMs, the following restrictions apply to Conditions and Variables in ontologized ASMs:

- **Variables** are WSML variables as defined in [19]
- A (WSML Full) **Condition** is a restricted form of WSML logical expressions where all free variables which are not bound by enclosing choose or forAll constructs are interpreted as being existentially quantified
- A WSML Core **Condition** is a WSML Full logical expression which consists only of molecules built up from memberOf and hasValue atoms and the logical connectives and and or where all unbound variables are existentially quantified (i.e. a condition is a conjunctive query).

Currently, one grounding mechanism to existing web services interfaces described in WSDL is defined, see Section 6. The transition rules of ontologized ASMs can be understood as semantic descriptions of the WSDL operations, thereby grounding the semantic description of the Web Service. However, there is no one to one correspondence between rules and operations in general, since ontologized ASMs can describe much more complex message exchange patterns than supported by WSDL. Instead of grounding rules to operations, the current WSDL grounding maps concepts to messages via their roles. We refer to Section 6 for a detailed discussion.

Furthermore, also beyond WSDL, interface descriptions with ontologized ASMs allow expressing arbitrary conditions over message exchanges. Pre-conditions over the input can be expressed in the if-part of transition rules; arbitrary post-conditions can be attached via respective axioms in the signature ontology which are "triggered" by updates in the then part of a rule. Similarly, ontologized ASMs allow defining state invariants (i.e. constraints over the states during a service invocation via its choreography interface) via respective axioms in the signature ontology. Regarding the relation to functional service descriptions, ontologized ASMs interfaces together with the attached signature ontologies allow describing a more fine-grained behavioral description of the interactions with the service than the overall pre-conditions, post-conditions, assumptions and effects in the service capability.
Orchestration Descriptions

Regarding the differences between choreography and orchestration descriptions, it appears that the latter most probably require more expressive modeling means than the former. However, all language constructs required for executable types of both choreography and orchestration interface descriptions are covered by the language constructs of ontologized ASMs as presented. The only exception are abstract orchestrations that are defined at design-time and can encompass invocation of goals or services along with respective mediators as the update-part of a guarded transition (see DIP deliverable D3.4 for details on abstract design-time and executable orchestrations).

The language for orchestration descriptions is the same as for choreography interfaces. In order to link to externally called services or (sub)goals that the orchestrating Web service needs to invoke for achieving its capability, the model and language is extended as follows:

- Goals and Services can be used in place of rules, meaning that these are executed in parallel to other rules in the orchestration.
- The state signature defined in the choreography can be reused, i.e. external inputs and outputs of the service and the state of the choreography can be de-referenced also in the orchestration.
- In addition, the state signature for the orchestration interface can extend the state signature of the choreography interface with additional in/out/shared/-controlled concepts that are tied to the used services and rules by mediators.
- Respective WW or WG mediators need to be in place to map the in and out concepts defined in the orchestration to the respective out and in concepts of the choreography interfaces in the used services and goals. These mediators state which output concepts are equivalent to which input of the called service/goal and vice versa, cf. Figure 4.3 below.

![Figure 4.3: Structure of Abstract Orchestration Descriptions](image)

The respective constructs are provided in the language synopsis of ontologized ASMs defined in the next section.
4.3.2 Definitions

In addition to the above structural specification of ontologized Abstract State Machines, the following provides formal definitions for the respective language constructs. In order to provide a complete synopsis of the language constructs of ontologized ASMs, we also define the grammar for interface descriptions in a dialect of Extended BNF (Backus-Naur Form). These are extensions to the grammar of the Web Service Modeling Language WSML [19] that defines the modeling constructs for ontologies and the top-level WSMO elements.

Definition 1: Interface Header

An interface is defined by a set of non-functional properties, a signature $\Sigma$ and a set of transition rules $\text{Rule}$.

**INTERFACE HEADER**

```
interface = 'interface' choreography? orchestration?

choreography = 'choreography' id? header* state_signature* transitions*

orchestration = 'orchestration' id? header* state_signature* transitions*
```

Definition 2: State Signature

The state signature $\Sigma$ is defined by a set of WSMO ontologies $O$ and a set of $Role$ definitions. The set of ontologies define the state onto which the conditions of the transition rules are evaluated, and also onto which the updates are performed. The Role definitions mark concepts with $in$, $out$, $shared$, $controlled$ or $static$ which are “inherited” by variables and instances which are a member of the particular concept.

```
state_signature = 'stateSignature' iri? header* role+
```

The state signature description is identified by an IRI followed by a header and role declarations. The state ontology (or ontologies) is declared via the importsOntology statement as part of the header. Roles are of two main categories, namely, grounded and un-grounded. The former are of type $in$, $out$ and $shared$ and the latter are $static$ and $controlled$ as defined below.

```
state_signature = 'stateSignature' iri? header* role+

role = {grounded_role} grounded_role | {un_groundeded_role} un_groundeded_role

grounded_role = grounded_mode grounded_mode_list

un_groundeded_role = un_groundeded_mode un_groundeded_mode_list

grounded_mode = {in} ‘in’ | {out} ‘out’ | {shared} ‘shared’
```
Definition 3: Guarded Transitions

For guarded transitions, three types of rules are defined, namely: *if-then*, *forAll-do*, *choose-do* and *update-rule*

\[
\text{transitions} \quad = \quad \text{transitionRules} \; \text{iri?} \; \text{rule}+ \\
\text{rule} \quad = \quad \{\text{if}\} \; \text{‘if’} \; \text{condition} \; \text{‘then’} \; \text{rule}+ \; \text{‘endIf’} \; | \\
\quad \{\text{choose}\} \; \text{‘choose’} \; \text{variablelist} \; \text{‘with’} \; \text{condition} \; \text{‘do’} \; \text{rule}+ \; \text{‘endChoose’} \; | \\
\quad \{\text{forAll}\} \; \text{‘forAll’} \; \text{variablelist} \; \text{‘with’} \; \text{condition} \; \text{‘do’} \; \text{rule}+ \; \text{‘endForAll’} \; | \\
\quad \text{updaterule} \; | \\
\quad \text{goal} \; | \\
\quad \text{service}
\]

The rules “if”, “choose” and “forAll” allow to specify nested rules whereas update-rules are atomic. There are three kinds of atomic updates, namely: *add*, *delete* and *update* that we specify in the following. Within abstract orchestration descriptions, a rule can be comprised of the keywords “goal” or “service” that denote goals or services that are to be invoked for achieving a state transition.

\[
\text{updaterule} \quad = \quad \text{modifier} \; \text{‘(’} \; \text{fact} \; \text{‘)’}
\]

\[
\text{modifier} \quad = \quad \{\text{add}\} \; \text{‘add’} \; | \; \{\text{delete}\} \; \text{‘delete’} \; | \\
\quad \{\text{update}\} \; \text{‘update’}
\]

Facts can be of two different natures, namely membership facts (\(a \text{memberOf} b\)) and attribute facts (\(a[\text{attr hasValue} \; \text{val}]\)). It is allowed also to specify multiple facts in a single modifier. In such case, there are two ways of specifying a membership fact which includes attribute facts, namely, \(a\text{memberOf}[\text{attr}_1\text{act}_1\text{ist}]\) or \(a[\text{attr}_2\text{act}_2\text{ist}]\text{memberOf}b\), the latter being the preferred one.

\[
\text{fact} \quad = \quad \{\text{fact_preferred}\} \; \text{term} \; \text{attr_fact?} \; \text{‘memberOf’} \; \\
\quad \text{termlist} \; \text{fact_update?} \; | \\
\quad \{\text{fact_nonpreferred}\} \; \text{term} \; \text{‘memberOf’} \; \text{termlist} \; \text{fact_update?} \; \text{attr_fact} \; | \\
\quad \{\text{fact_molecule}\} \; \text{term} \; \text{attr_fact}
\]
\[ \text{fact_update} = \text{'}=>\text{' term list} \]
\[ \text{attr_fact} = \text{'}[\text{'} attr_fact_list \text{']}\]
\[ \text{attr_fact_list} = \]
\[ \{\text{attr_relation}\} \text{ term 'hasValue' term list fact_update? | attr_fact_list ',' term 'hasValue' term list fact_update?} \]

Notice that fact-update is only used for the update modifier since in such case, the old facts to be deleted and the new facts to be added can be specified.

**Definition 4: Conditions in ontologized ASMs**

Conditions in ontologized ASMs are syntactically derived form of WSML Logical Expressions as defined in [19]. More precisely, we restrict WSML Logical Expressions by disallowing the use of the use of \text{naf}, \text{!-}, and \text{:-}.

Semantically, we slightly deviate from the definition of WSML logical expressions as in Section 2.8 of [19] with respect to the scope of variables as follows: The scope of a variable in a condition is always defined by its quantification. If a variable is neither quantified inside the condition nor bound by an enclosing \text{forall} or \text{choose} rule, the variable is implicitly existentially quantified outside the condition.

The intuitive reason why we close off free variables is that conditions can be viewed as queries over the state ontology. When checking such a condition over the state we are not interested in proving that it holds for all possible bindings of a free variable, but only whether such a binding exists.

### 4.3.3 Formal Execution Semantics

As the final aspect of ontologized Abstract State Machines, the following specifies their formal executions semantics. In conjunction with the grounding specified in Section 6, this serves as the basis for development of execution engines for communication and cooperation among web services and clients via their respective behavior interfaces.

The intuitive semantics of ontologized ASMs as defined here is as follows. When beginning to interact with a service via one of its behavior interface you start with the initial state and all possible interaction sequences with the service are defined by the possible runs of the ontologized ASMs defined in the interface description.

The initial state of an interface (i.e. at the time when interaction starts) is defined by the state signature. That is, the initial state is the set \( S_0 \) consisting of all instances of all concepts and relations defined in the imported ontologies.

A run of an interface is defined analogously to the runs of basic ASMs as described above in Section 4.2. That is, a run of an interface machines is defined as sequences of possible single execution steps. Possible executions steps are defined by:

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

where \( S \) is the current state, \( U \) is a consistent update set, and \( S' \) is the resulting state of applying \( U \) in \( S \). Recall that an interface is defined by a set of transition rules \( R \).

Let \( O \) denote the imported signature ontologies. We define update sets for \( U_{\langle R,S\rangle} \) inductively as follows:
• $U_{\text{add}(\text{fact}), S} = \text{add}(\text{fact})$

• $U_{\text{delete}(\text{fact}), S} = \text{delete}(\text{fact})$

• $U_{\text{update}(\text{a memberOf } A \Rightarrow B), S} = \{\text{delete}(\text{a memberOf } A), \text{add}(\text{a memberOf } B)\}$

• $U_{\text{update}(\text{a[att hasValue } A \Rightarrow B)\}), S} = \{\text{delete}(\text{a[att hasValue } A]), \text{add}(\text{a[att hasValue } B])\}$

• $U_{R, S} = \bigcup_{r \in R} U_{r, S}$

• $U_{\text{if } \text{Cond then } R, S} =$
  - $U_{R, S}$ if $O \cup S$ entails $\text{Cond}$
  - otherwise

• $U_{\text{forall } ?\text{Var with Cond do } R \text{ endForAll}, S} = \{U_{R?, S} \mid \phi \text{ such that } \phi = \{?\text{Var} / \text{id}\} \text{ where } \text{id} \in V_I \text{ is an identifier such that } O \cup S \text{ entails } \text{Cond}\phi\}$

• $U_{\text{choose } ?\text{Var with Cond do } R \text{ endChoose}, S} =$
  - $U_{R?, S}$ where $\phi = \{?\text{Var} / \text{id}\}$ where $\text{id} \in V_I$ is a non-deterministically chosen identifier such that $O \cup S$ entails $\text{Cond}\phi$
  - $\{\}$ if $O \cup S \cup \{\exists ?\text{Var} (\text{Cond})\}$ is unsatisfiable.

An update set $U$ is consistent if it does not contain any two elements $\text{add}(\text{fact})$ and $\text{delete}(\text{fact})$ and the resulting state $S' = S \setminus \{\text{fact} \mid \text{delete}(\text{fact} \in U)\} \cup \{\text{fact} \mid \text{add}(\text{fact} \in U)\}$ is consistent with the signature ontology, i.e. $S' \cup O$ is satisfiable. A run of an ASM stops if the update set induced by the rules above is empty, i.e. no more rules can fire.
5 Translation User Language to Formal Model

This chapter specifies a translation algorithm from the user language as defined in Chap. 3 and the formal model defined in Chap. 4 in order to integrate both layers and provide a means for interchange between them.

5.1 Overview

This section gives a short repetition of the execution semantics of both WSM-L/UML2AD and WSML/ASM. Based on this elaboration, we sketch the idea of our translation algorithm which will be given in Sect. 5.3.

5.1.1 Execution in WSML/ASM

We use the WSML/ASM model with its syntax and execution semantics as specified in Chap. 4. We give a very brief and informal introduction here to clearly state the assumptions for the algorithm described later on. WSML/ASM rules are defined by guarded transition rules of the following form.

\[
\text{if } \text{Condition} \text{ then } \text{Update} \text{ endif}
\]

The description of Conditions and Updates is defined in Chap. 4 as well. Furthermore, the execution semantics interprets WSML/ASM statements as follows.

- Each time the Condition of a transition rule \( r \) becomes true, \( r \) becomes active (\( \text{active}(r) = \text{true} \)).
- Each time the Condition of a transition rule \( r \) becomes false, \( r \) becomes inactive (\( \text{active}(r) = \text{false} \)).
- A transition rule \( r \) can only be executed when it is active (\( \text{active}(r) \)).
- Each time a transition rule \( r \) is executed, it becomes inactive (\( \text{active}(r) = \text{false} \)).

There are other possibilities to express transition rules, like \texttt{forall} and \texttt{choose} as defined in Chap. 4. These constructs are used to express parallelism and nondeterminism, respectively (see [11] p. 31]). For the details of this chapter, the if-construct will however be sufficient. When we write “transition rules”, we mean rules using the if-construct from hereon.

The Conditions and Updates of WSML/ASM transition rules operate on states which are represented by the instances of an ontology. Each incoming message from the outside reflects on the ontology by the appearance of a respective instance. In the same way, instances appearing in the ontology are sent as messages when they are appropriately marked. The concepts whose instances are subject to sending and receiving respectively, are marked by being part of a special section in the interface description which denotes their mode. For further detail, see Chap. 4 and its references.

\footnote{It suffices, if the execution of an assignment makes a Condition true, that has been true before.}

\footnote{Clearly stated, it is necessary that all conjunctive blocks of a Condition must become false and then again true in order to activate a rule. (This implies that the same must hold for each block of a disjunctive Condition.)}
5.1.2 Expressing WSML/UML2AD Execution

For understanding the relations between the semantics of WSML/UML2AD and WSML/ASM, we will now give a quick overview of relevant WSML/UML2AD constructs and sketch their translation to WSML/ASM which will be detailed in Sect. 5.3.

Overview of WSML/UML2AD Diagrams

Diagrams The main constituents of WSML/UML2AD diagrams are actions. These actions are combined by directed flow edges, which can respectively express control and object flow. These flow edges can themselves be interconnected via control nodes. When object flow edges enter or leave actions, there can be pins directly attached to the action that represent the specific data types that are to be exchanged via the respective object flow edge. Another way for expressing the data type of an object flow is to enrich it by object nodes. Their purpose is the same as of pins. However, each type defined by pins or object nodes reached when following an object flow towards its target must be more general than (or equivalent to) its preceding types. A detailed description of WSML/UML2AD diagrams can be found in Chap. 3.

Token Flow For the verbal explanation of the semantics of a WSML/UML2AD diagram, the concept of “token flow” is used. The finishing of an action therefore causes tokens to be existent in every output pin and leaving flow edge. The so-called token flow semantics now is used to describe how control nodes influence the execution of the diagram. In general, an action can only be executed if all its incoming flow edges and input pins yield at least one token each. In the following, we give a description of each control node, how it influences the token flow, and how it would be described using WSML/ASM rules in an exemplified way.

Since WSML/UML2AD diagrams can be built of a variety of redundant constructs, Section 5.2.3 gives a simplification which is more verbose, but uses a smaller set of WSML/UML2AD diagram constructs than the original specification. The following elaborations will concentrate on this subset of WSML/UML2AD. In short anticipation of Sect. 5.2.3 we will express control flow as a special kind of object flow (where explicit state variables become the objects of the object flow), convert all object node object flows to the pin-only notation, and split up combined control nodes into their constituents.

Aspired Use of WSML/ASM

Transition Rules The main idea for the translation of WSML/UML2AD diagrams to WSML/ASM transition rules is to define a set of rules for each action that describes the circumstances under which the action is going to be executed. Due to the token flow semantics, this depends on the object flow that precedes its input pins. The result of an action relevant to the execution semantics is the appearance of tokens in its output pins and leaving flow edges. We reflect this in the effect of the rules.

Concepts For the description of the token flow used in the definition of WSML/UML2AD execution semantics, we use concepts and their instances in WSML/ASM. The intuitive counterpart of object nodes and pins in WSML/UML2AD therefore are
the concepts of an ontology in WSML/ASM. Tokens in WSML/UML2AD will
formally be reflected by instances in WSML/ASM. But, this mapping of aspects of
WSML/UML2AD and WSML/ASM is not of a natural homogeneity. Whereas tokens
have a more dynamic nature by locally moving around between determined actions
during the execution of a WSML/UML2AD diagram, instances in a WSML/ASM
transition rule system are of a more static nature in that they exist in the global state
space of the attached ontology and are potentially accessible by every transition rule
at every step of execution. This important difference needs to be addressed by the
translation between both paradigms.

Translation Aspects for Basic Constructs

In this section, we lay out a very intuitive description and comparison of the expression
of control nodes in WSML/UML2AD and their translation to WSML/ASM. This
section only aims at pointing out differences and commonalities of the two execution
semantics. A formal translation will later be given in the Sects. 5.2 and 5.3.

To convey the idea of the rules’ structure, we use if-then-endif-constructs con-
taining a condition and multiple updates. Conditions represent checks on the existence
of instances whereas the updates intuitively state the creation or deletion of instances
in an ontology (cp. Sect. 5.1.1).

Fork Node  At a fork node, incoming objects \( in \) are forwarded
to all leaving edges \( out_1, out_2, \ldots, out_n \). Informally, a rule would
look like the following.

\[
\text{if } in \text{ then } out_1, out_2, \ldots, out_n \text{ endif}
\]

Due to the WSML/ASM execution semantics as sketched in
Sect. 5.1.1, this rule would only fire once. We therefore do not
need to disable the event \( in \).

Join Node  Objects at a join node are passed along its leaving
edge \( out \) only if all incoming edges yield objects \( in_1, in_2, \ldots, in_n \).
A rule modeling this behaviour would look like this.

\[
\text{if } in_1 \land in_2 \land \cdots \land in_n \text{ then } out \text{ endif}
\]

Again, due to the specific WSML/ASM execution semantics, it
is not needed to delete any of the assertions stating the existence
of \( in_1, in_2, \ldots, in_n \), here.

Decision Node  At a decision node, an incoming object \( in \) will
be forwarded along one of its leaving edges \( out_1, out_2, \ldots, out_n \).
This behaviour can be understood as describing \( n \) independent
control flows of which exactly one is chosen each time an object
arrives. Therefore, we use \( n \) rules to model this behaviour.

\[
\begin{align*}
\text{if } in \text{ then } out_1, \neg in \text{ endif} \\
\text{if } in \text{ then } out_2, \neg in \text{ endif} \\
\vdots \\
\text{if } in \text{ then } out_n, \neg in \text{ endif}
\end{align*}
\]
Remarkably, we need to negate the assertion of the existence of \( in \) for each branch the object flow could follow in order to model the behaviour that only one of the branches is actually taken. This is because the fulfilment of \( in \) activates all rules in the same way. As described in Sect. 5.1.1, a rule can only be deactivated by negating its condition unless it becomes executed.

**Merge Node** A merge node directly forwards incoming objects \( in_1, in_2, \ldots, in_n \) to its leaving edge \( out \). We use \( n \) rules to model this behaviour in the following way.

\[
\begin{align*}
\text{if } in_1 & \text{ then } out \text{ endif} \\
\text{if } in_2 & \text{ then } out \text{ endif} \\
\ldots \\
\text{if } in_n & \text{ then } out \text{ endif}
\end{align*}
\]

In this case, we do not need to disable the \( in \)-assertions since they are based on different instance classes, and the WSML/ASM execution semantics prevents the same rule from firing over and over again (Sect. 5.1.1).

Important here is that we must *not* model the whole behaviour in only one rule by merging their conditions (like \( in_1 \lor in_2 \lor \cdots \lor in_n \)). Such a rule would imply that its update is only executed once, even if multiple \( in \)-assertions hold. In contrast, the semantics of WSML/UML2AD states that every token that is passed to a merge node is going to be forwarded, even when any appear at the same time.

**Translation Aspects for Complex Structures**

The descriptions given above for the translation of WSML/UML2AD control nodes to WSML/ASM transition rules work in the basic case when an object flow consists of only the one control node described. However, the translation must be able to cope with an arbitrary combination of multiple control nodes in any object flow.

**Traverse-to-Completion** For example, there is the so-called “traverse-to-completion” behaviour of the WSML/UML2AD execution semantics. This means that a token that is offered at a leaving object flow edge only traverses it at a whole when a connected action is ready to fire. This semantics was introduced in order to avoid deadlocks due to hastily distributing tokens in the network of an object flow containing decision nodes, and in the consequence reaching different actions which each rely on multiple of those different tokens.

Simply spoken, the chosen design of modeling rules out of the perspective of actions ensures capturing the “traverse-to-completion” semantics by collecting all conditions needed to be fulfilled for an action to execute in a single rule. This means, we are not modeling the single steps of the traversing of tokens, but allow a rule only to fire when all its required conditions are met. Thus, in a practical implementation for a sequential rule execution, a scheduler would choose one of the competing rules ready to fire, which would then consume all tokens at once using its update statements.
**Decision Variables** The decision node is the only control node that requires its conditions to be negated after a connected action becomes executed. In this, the translation of the decision node is irregular with respect to the modeling of the other control nodes. This anomaly makes a special treatment necessary. The modeling of all required conditions for the execution of an action in one single rule demands for the tracking of all contained decision nodes in order to disable them in the updates of this action.

Therefore, an activation status for each decision node needs to be observed. For this, we introduce new concepts in the ontology. We refer to these as “decision variables” (or “decVars”). Furthermore, multiple tokens may be eligible for passing a decision node at a time. For this reason, decision variables depend on the respective decision node and each type of token that could potentially pass it.

The special treatment of decision nodes has four implications on the translation to WSML/ASM rules. First, additional concepts have to be created in the ontology. Second, each decision variable has to be activated (instances need to be created) as soon as a token could pass it. This is realized by additional rules. Third, each rule that results from the translation of an action needs to check for the activation of all decision nodes contained in the respective incoming branch of object flow. Fourth, the execution of an action (or its respective rule) must deactivate the decision nodes that led to its execution.

**Preserving Semantics**

The outlined issues are covered by the formal translation given in the Sects. 5.2 and 5.3. In this sense, the presented translation preserves the execution semantics of WSML/UML2AD.

**5.2 Prerequisites**

**5.2.1 Ontology**

We define a WSMO ontology in WSML that will be used later on for expressing the dynamic aspects of the WSML/UML2AD in the WSML/ASM choreography representation.

**Rule 1.** For every WSML/UML2AD diagram, a local ontology localOntology will be created. Initial concepts are as follows. It will become updated during the description of the algorithm.

```plaintext
concept InternalComm
concept State subConceptOf InternalComm
concept ControlFlow subConceptOf State
concept DecVar subConceptOf InternalComm
```

**5.2.2 Definitions**

Throughout the following sections, we will use some functions to precisely express the translation of WSML/UML2AD to WSML/ASM. The signatures of the single functions are given below.
Rule 2. For every element of an object flow $\in E$, we define a set of direct successors $\text{Suc}$ and a set of direct predecessors $\text{Pre}$ which contain the appropriate set of elements $E$ with respect to their order in the WSML/UML2AD object flow.

\[
\text{Suc} : E \rightarrow 2^E \\
\text{Pre} : E \rightarrow 2^E
\]

Rule 3. For every element of an object flow $e_u, e_v \in E$, we define direct successor $\text{suc}_1, \text{suc}_2, \ldots, \text{suc}_m$ and predecessor functions $\text{pre}_1, \text{pre}_2, \ldots, \text{pre}_n$ which return an appropriate element $(e_x, e_y \in E)$ with respect to their order in the WSML/UML2AD object flow. The functions $\text{suc}$ and $\text{pre}$ are abbreviations for the case of sets of cardinality one.

\[
\text{suc}_x : E \rightarrow E \\
\text{pre}_y : E \rightarrow E \\
e_x = \text{suc}_x(e_u) \iff e_x \in \text{Suc}(e_u), \quad \bigcup e_x = \text{Suc}(e_u), \quad 1 \leq x \leq m \\
e_y = \text{pre}_y(e_v) \iff e_y \in \text{Pre}(e_v), \quad \bigcup e_y = \text{Pre}(e_v), \quad 1 \leq y \leq n \\
\text{suc} : E \rightarrow E, \quad e_v = \text{suc}(e_s) \iff e_v \in \text{Suc}(e_s), \quad \{e_v\} = \text{Suc}(e_s) \\
\text{pre} : E \rightarrow E, \quad e_w = \text{pre}(e_t) \iff e_w \in \text{Suc}(e_t), \quad \{e_w\} = \text{Suc}(e_t)
\]

Rule 4. For every pin $\in P$ and object node $\in O$, we define a string in WSML/ASM that links this entity to a concept $\in$ Concept of the ontology.

\[
\text{concept} : P \cup O \rightarrow \text{Concept}
\]

Rule 5. For every action $\in A$, we define its set of output pins $P_{\text{out}}$.

\[
\text{pin} : A \rightarrow 2^{P_{\text{out}}}
\]

The following definitions will be needed for the calculation of some specific rules necessary to remodel the token flow semantics of WSML/UML2AD in WSML/ASM.

Rule 6. The following definition specifies a multiset. Such sets can contain the same member $\in X$ multiple times $\text{mul}_X$. Due to the close similarity to sets, we use the set common operations, like $\cup$, to state operations on multisets. To explicitly denote a collection of individuals $i_1, i_2, \ldots, i_n$ as set, we write $\{i_1, i_2, \ldots, i_n\}$. For the interpretation as a multiset, we write $\{i_1, i_2, \ldots, i_n\}$. For the interpretation as a multiset, we write $\{i_1, i_2, \ldots, i_n\}$.

\[
\langle X, \text{mul}_X \rangle \ldots \text{Multiset} \quad \text{mul}_X : X \rightarrow \mathbb{N}
\]

Rule 7. The function $\text{decVar}$ links pairs of decision nodes $\in D$ and pins $\in P$ to respective concepts $\in$ Concept in the ontology. The decisions function will be used to define multisets of decisions $\{d_1, d_2, \ldots, d_m\}$ by pairs of pins $\in P$ and multisets of control flow nodes $\{e_1, e_2, \ldots, e_n\}$.

\[
\text{decVar} : D \times P \rightarrow \text{Concept} \\
\text{decisions} : P \times 2^{(E, \text{mul}_E)} \rightarrow 2^{(D, \text{mul}_D)} \quad (\text{note multisets})
\]
For the translation of WSML/UML2AD actions to WSML/ASM rules, we use the following definitions.

**Rule 8.** For every action \( a \in A \) and output signal node \( o \in O \), we will define a set of rules \( R \). For its definition, the actionFlow and objectFlow function will be used. \( C \times U \) denotes a set of pairs of conditions \( c \in C \) and updates \( u \in U \), \( n \) is an arbitrary number \( \in \mathbb{N} \). \( D \) and \( E \) are sets of decision nodes and control flow nodes as defined above.

\[
\begin{align*}
\text{rules} & : & A \cup O & \longrightarrow 2^R \\
\text{actionFlow} & : & A & \longrightarrow 2^{(2^{(C \times U)})^n} \\
\text{objectFlow} & : & P \times 2^{D \times E} \times 2^E & \longrightarrow 2^{C \times U}
\end{align*}
\]

### 5.2.3 Normal Form

In order to keep the translation algorithm of Sect. 5.3 simple, we reduce some of the redundancies entailed in WSML/UML2AD. The resulting normal form of WSML/UML2AD will be used in the following of this description. The creation of this normal form from the original WSML/UML2AD modeling is defined below.

**Remove Implications**

The interpretation of two control flow edges leading to an action is defined as an implicit join. The same holds for multiple object flow edges entering a pin and an object node, respectively. We make these interpretations explicit.

**Rule 9.** All control flow edges leading to the same \( a \) will be joined in a Join control node whose output directly leads to action \( a \).

**Rule 10.** All object flow edges leading to the same \( p \) will be joined in a Join control node whose output directly leads to pin \( p \).

**Rule 11.** All object flow edges leading to the same \( o \) will be joined in a Join control node whose output directly leads to object node \( o \).

Multiple edges leaving an action, pin or object node have an implicit fork semantics. We also make this semantics explicit.

**Rule 12.** All control flow edges leaving the same \( a \) will leave from a Fork control node whose input directly stems from action \( a \).

**Rule 13.** All object flow edges leaving the same \( p \) will leave from a Fork control node whose input directly stems from pin \( p \).

**Rule 14.** All object flow edges leaving the same \( o \) will leave from a Fork control node whose input directly stems from object node \( o \).

In both object and control flow, so-called control nodes can be used to route these flows in certain ways. For convenience, some abbreviations were defined in WSML/UML2AD for specific pairs of those nodes. We will transform these syntactic abbreviations back into their basic control node representations.
**Rule 15.** A combined decision and merge control node will be split to a decision and a merge control node.

**Rule 16.** A combined fork and join control node will be split to a fork and a join control node.

**Express Control Flow as Object Flow**

Control flow in WSML/UML2AD can be seen as a specific kind of object flow where the object of a transformed control flow edge is an (artificial and materialised) state value. We call the type of the state values ‘state’.

**Rule 17.** Every control flow is converted to an object flow by introducing an object node $o$ of type State at an arbitrary position in this control flow.

$$\text{concept}(o) := \text{localNamespace}\#\text{ControlFlow}$$

**Transform Object Nodes to Pins**

Object flows in WSML/UML2AD can be defined in two ways. One is to use object nodes in the object flow. The second is to use pins in each source and target of an object flow. In the second case, the type of each target pin must be a superset of each source pin. Since the pin notation is more expressive than the object node notation, we transform the object node to the pin notation.

**Rule 18.** For each object flow, we add pins to its source and target actions and remove the object nodes. The type (concept) of a new pin $p_{\text{new}}$ is defined as the union of the types of all former object nodes $o_1, o_2, \ldots, o_m$ and pins $p_1, p_2, \ldots, p_n$ that are (directly) reachable through edges and control nodes from the pin $p_{\text{new}}$.

$$\text{concept}(\text{NewConcept}_x \text{ subConceptOf} \{ \text{concept}(o_1), \text{concept}(o_2), \ldots, \text{concept}(o_m), \text{concept}(p_1), \text{concept}(p_2), \ldots, \text{concept}(p_n) \}) \quad \text{where } x \in \mathbb{N}$$

$$\text{concept}(p_{\text{new}}) := \text{localNamespace}\#\text{NewConcept}_x$$

such that $\forall i, j \in \mathbb{N} : \text{concept}(p_i) = \text{concept}(p_j) \iff i = j$

and $\forall k, l \in \mathbb{N} : \text{NewConcept}_k = \text{NewConcept}_l \iff k = l$

**5.3 Translation**

Here, we give a translation of the WSML/UML2AD model to the WSML/ASM language. We therefore assume the prerequisite steps from Sect. 5.2 have been executed. Thus, simply spoken, the WSML/UML2AD model we base on only consists of object flows with only basic control nodes. When we say ‘WSML/UML2AD’ from here on, we refer to the adapted model as described above, unless otherwise stated.
5.3.1 Global Structure

Rule 19. Each choreography or orchestration specified in WSML/UML2AD will be converted to WSML/ASM according to the following scheme.

```
webService serviceName
    interface interfaceName
        type chorOrchName
        importsOntology localOntology
    stateSignature
        in { inConcept_1, inConcept_2, ..., inConcept_m }
        out { outConcept_1, outConcept_2, ..., outConcept_n }
    guardedTransition rulesetName
    allRules
```

Rule 20. We define the type of the described interface as either choreography or orchestration. This is needed as an input to the translation algorithm.

```
type : { choreography, orchestration }
```

Rule 21. All names are needed as input to the translation algorithm. Those are serviceName, interfaceName, chorOrchName, rulesetName.

Rule 22. All \((m, n \in \mathbb{N})\) inputs and outputs of the WSML/UML2AD diagram given as input \((in_1, in_2, ..., in_m)\) and output signal constructs \(out_1, out_2, ..., out_n\) are converted as follows. The grounding of each input and output must be given.

```
inConcept_x := concept(in_x) withGrounding concept(grounding(in_x))
outConcept_y := concept(out_y) withGrounding concept(grounding(out_y))
\forall x, y \in \mathbb{N}, \ 1 \leq x \leq m, \ 1 \leq y \leq n
```

Rule 23. The transition rules section consists of rule sets for every action \(a \in A\), every output pin \(p \in P_{out}\), and every output signal construction \(o \in O\) of an WSML-UML2AD diagram. The unions in the following notation translate to the concatenation of its members in any order for the output of this algorithm.

```
allRules := \bigcup_{a \in A} rules(a) \cup \bigcup_{p \in P_{out}} decVarRules(p) \cup \bigcup_{o \in O} rules(o)
```

5.3.2 Concepts

State Signature

Rule 24. For every output pin \(p\) of the object flow, we define a new concept statement in the ontology in the following way. This updates the ontology initially defined in
Rule 1

\[
\text{concept } \text{newVar}_x \text{ subConceptOf State}
\]
\[
\text{state}(p) := \text{newVar}_x
\]

Decision Variables

Rule 25. Every pair of decision node \(d_1, d_2, \ldots, d_n\) and output pin \(p\) of any action or activity in the object flow graph of the WSML/UML2AD becomes a concept in the state signature in the following way. Here, the ontology as introduced in Rule 1 is being extended. In the same way for later reference, the function \(\text{decVar}\) is defined as follows. \(F\) denotes a set of only final nodes, where final nodes are only input pins of an object flow.

\[
\text{Decision Variables}
:= \{ \text{concept } \text{newVar}_x \text{ subConceptOf DecVar} \}
\[
\text{decVar}(d_x, p) := \text{newVar}_x : \ x \in \mathbb{N}, \ 1 \leq x \leq n,
\{ d_1, d_2, \ldots, d_n \} = \text{decisions}(p, F), \ \ p \in P_{\text{out}}, \ \ F \subseteq P_{\text{out}} \}
\]

Rule 26. We define the decisions function for every output pin \(p\) as follows.

\[
\text{decisions}(s, E \cup \{ \{ p \} \})
:= \text{decisions}(s, E \cup \{ \{ \text{suc}(p) \} \})
\]

Rule 27. We define the decisions function for every decision node \(d\) as follows.

\[
\text{decisions}(s, E \cup \{ \{ d \} \})
:= \{ \{ d \} \} \cup \text{decisions}(s, E \cup \{ \{ \text{suc}_1(d), \text{suc}_2(d), \ldots, \text{suc}_n(d) \} \})
\]

Rule 28. We define the decisions function for every fork node \(f\) as follows.

\[
\text{decisions}(s, E \cup \{ \{ f \} \})
:= \text{decisions}(s, E \cup \{ \{ \text{suc}_1(f), \text{suc}_2(f), \ldots, \text{suc}_n(f) \} \})
\]

Rule 29. We define the decisions function for every join node \(j\) as follows.

\[
\text{decisions}(s, E \cup \{ \{ j \} \})
:= \text{decisions}(s, E \cup \{ \{ \text{suc}(j) \} \})
\]

Rule 30. We define the decisions function for every merge node \(m\) as follows.

\[
\text{decisions}(s, E \cup \{ \{ m \} \})
:= \text{decisions}(s, E \cup \{ \{ \text{suc}(m) \} \})
\]
5.3.3 Transition Rules

Decision Variables

Due to the token flow semantics in WSML/UML2AD, we need to define a couple of rules that resemble this behavior in the WSML/ASM.

Rule 31. For every output pin p, we define a rule in the following way. The decision function is to be used as defined in Sect. 5.3.2. The output of the decision function is to be treated as multiset here.

\[
\text{decVarRules}(p) := \{ \text{if } \? \text{memberOf concept}(p) \\
\text{then add}(\? \text{memberOf decVar}(d_1, p)) \\
\text{and add}(\? \text{memberOf decVar}(d_2, p)) \\
\text{and \ldots} \\
\text{and add}(\? \text{memberOf decVar}(d_n, p)) \text{ endif} : \\
\{ d_1, d_2, \ldots, d_n \} = \text{decisions}(p, F), \quad F \subseteq P_{out} \}
\]

Object Flow

We model the object flow by defining a set of rules for every action in the WSM-L/UML2AD diagram.

Rule 32. For every action a, we define a WSML/ASM rule as follows.

\[
\text{rules}(a) := \{ \text{if } c_1 \text{ and } c_2 \text{ and } \ldots \text{ and } c_m \\
\text{then } u_1 \text{ and } u_2 \text{ and } \ldots \text{ and } u_m \\
\text{and add}(\? \text{memberOf} \{ \text{concept}(p_1), \text{state}(p_1) \}) \\
\text{and add}(\? \text{memberOf} \{ \text{concept}(p_2), \text{state}(p_2) \}) \\
\text{and \ldots} \\
\text{and add}(\? \text{memberOf} \{ \text{concept}(p_n), \text{state}(p_n) \}) \text{ endif} : \\
(\langle c_1, u_1 \rangle, \langle c_2, u_2 \rangle, \ldots, \langle c_m, u_m \rangle) \in \text{actionFlow}(a), \\
p_x = \text{pin}_x(a), \quad x \in \mathbb{N}, \quad 1 \leq x \leq n \}
\]

Rule 33. For every output signal action o, we define the translation as follows.

\[
\text{rules}(o) := \{ \text{if } c_1 \text{ and } c_2 \text{ and } \ldots \text{ and } c_m \\
\text{then } u_1 \text{ and } u_2 \text{ and } \ldots \text{ and } u_m \\
\text{and add}(\? \text{memberOf} \text{concept}(p_1)) \\
\text{and add}(\? \text{memberOf} \text{concept}(p_2)) \\
\text{and \ldots} \\
\text{and add}(\? \text{memberOf} \text{concept}(p_n)) \text{ endif} : \\
(\langle c_1, u_1 \rangle, \langle c_2, u_2 \rangle, \ldots, \langle c_m, u_m \rangle) \in \text{actionFlow}(o), \\
p_x = \text{pin}_x(o), \quad x \in \mathbb{N}, \quad 1 \leq x \leq n \}
\]
Rule 34. We define the actionFlow function for every action $a$ as follows.

$$actionFlow(a) := objectFlow(p_1, \emptyset, \{ p_1 \})$$

$$\times objectFlow(p_2, \emptyset, \{ p_2 \})$$

$$\times \ldots$$

$$\times objectFlow(p_n, \emptyset, \{ p_n \})$$

where $p_x = pin_x(a), \ x \in \mathbb{N}, \ 1 \leq x \leq n$

Rule 35. We define the objectFlow function for every input pin $p$ as follows.

$$objectFlow(t, DV \cup DV_{rel}, E \cup \{ p \}) := objectFlow(t, DV \cup \{ \langle x, pre(p) \rangle: \langle x, p \rangle \in DV_{rel} \}, E \cup \{ pre(p) \})$$

where $DV_{rel} := \{ \langle x, y \rangle: x \in D, y = p \}$

Rule 36. We define the objectFlow function for every decision node $d$ as follows.

$$objectFlow(t, DV \cup DV_{rel}, E \cup \{ d \}) := objectFlow(t,$$

$$DV \cup \{ \langle d, pre(d) \rangle \} \cup \{ \langle x, pre(d) \rangle: \langle x, d \rangle \in DV_{rel} \},$$

$$E \cup \{ pre(d) \})$$

where $DV_{rel} := \{ \langle x, y \rangle: x \in D, y = d \}$

Rule 37. We define the objectFlow function for every join node $j$ as follows.

$$objectFlow(t, DV \cup DV_{rel}, E \cup \{ j \}) := objectFlow(t,$$

$$DV \cup \{ \langle x, pre_1(j) \rangle, \langle x, pre_2(j) \rangle, \ldots, \langle x, pre_n(j) \rangle: \langle x, j \rangle \in DV_{rel} \},$$

$$E \cup \{ pre_1(j), pre_2(j), \ldots, pre_n(j) \})$$

where $DV_{rel} := \{ \langle x, y \rangle: x \in D, y = j \}$

Rule 38. We define the objectFlow function for every merge node $m$ as follows.

$$objectFlow(t, DV \cup DV_{rel}, E \cup \{ m \}) := objectFlow(t, DV \cup \{ \langle x, pre_1(m) \rangle: \langle x, m \rangle \in DV_{rel} \}, E \cup \{ pre_1(m) \})$$

$$\cup objectFlow(t, DV \cup \{ \langle x, pre_2(m) \rangle: \langle x, m \rangle \in DV_{rel} \}, E \cup \{ pre_2(m) \})$$

$$\cup \ldots$$

$$\cup objectFlow(t, DV \cup \{ \langle x, pre_n(m) \rangle: \langle x, m \rangle \in DV_{rel} \}, E \cup \{ pre_n(m) \})$$

where $DV_{rel} := \{ \langle x, y \rangle: x \in D, y = m \}$

Rule 39. We define the objectFlow function for every fork node $f$ as follows.

$$objectFlow(t, DV \cup DV_{rel}, E \cup \{ f \}) := objectFlow(t, DV \cup \{ \langle x, pre(f) \rangle: \langle x, f \rangle \in DV_{rel} \}, E \cup \{ pre(f) \})$$

where $DV_{rel} := \{ \langle x, y \rangle: x \in D, y = f \}$
Rule 40. We define the objectFlow function for a target input pin \( t \), a set of tuples \( DV \) consisting of decision nodes \( \in D \) and output pins \( \in P \), and a set of output pins \( P \) and object nodes \( O \) stemming from an input signal construct as follows.

\[
\text{objectFlow}(t, DV, P \cup O) := \{ \text{?var}_1 \text{memberOf} \{ \text{concept}(t), \text{state}(p_1) \} \\
\text{and ?var}_2 \text{memberOf} \{ \text{concept}(t), \text{state}(p_2) \} \\
\text{and \ldots} \\
\text{and ?var}_n \text{memberOf} \{ \text{concept}(t), \text{state}(p_n) \} \\
\text{and ?var}_{n+1} \text{memberOf} \text{concept}(o_1) \\
\text{and not ?var}_{n+1} \text{memberOf} \text{InternalComm} \\
\text{and ?var}_{n+2} \text{memberOf} \text{concept}(o_2) \\
\text{and not ?var}_{n+2} \text{memberOf} \text{InternalComm} \\
\text{and \ldots} \\
\text{and not \ldots} \\
\text{and ?var}_{n+m} \text{memberOf} \text{concept}(o_m) \\
\text{and not ?var}_{n+m} \text{memberOf} \text{InternalComm} \\
\text{and disjoint( \{ var_1, var_2, \ldots, var_{n+m} \} )} \\
\text{and ?_ memberOf dec Var}(dv_{d1}, dv_{p1}) \\
\text{and ?_ memberOf dec Var}(dv_{d2}, dv_{p2}) \\
\text{and \ldots} \\
\text{and ?_ memberOf dec Var}(dv_{dk}, dv_{pk}) \}
\]

where \( p_1, p_2, \ldots, p_n \in P, \quad o_1, o_2, \ldots, o_m \in O, \quad \langle dv_{d1}, dv_{p1} \rangle, \langle dv_{d2}, dv_{p2} \rangle, \ldots, \langle dv_{dk}, dv_{pk} \rangle \in DV \)

Rule 41. We define the disjoint function of a set \( \{ var_1, var_2, \ldots, var_n \} \) as follows.

\[
\text{disjoint( \{ var_1, var_2, \ldots, var_n \} )} := \text{?var}_1 \neq \text{?var}_2 \text{ and ?var}_3 \neq \text{?var}_1 \text{ and \ldots} \\
\text{and ?var}_n \neq \text{?var}_1 \text{ and \ldots} \\
\text{and ?var}_1 \neq \text{?var}_2 \text{ and ?var}_3 \neq \text{?var}_2 \text{ and \ldots} \\
\text{and ?var}_n \neq \text{?var}_3 \text{ and \ldots} \\
\text{and \ldots} \\
\text{and ?var}_{n-1} \neq \text{?var}_n
\]
5.4 Conclusion

Chapter 5 describes an algorithm that is capable of translating choreography and orchestration descriptions given in WSML/UML2AD to WSML/ASM. In this first version, the translation is defined for a subset of the constructs in WSML/UML2AD. Anyway, the parts that have been covered are translated in a way that the semantics of WSML/UML2AD is completely preserved when feeding its output to a choreography or orchestration engine as outlined in Sect. 5.1.2.

Future Versions of this specification will target at the following points.

- Implementation of further aspects of WSML/UML2AD.
- Aligning with the further evolvement of the choreography and orchestration engines and their understanding of the execution semantics of WSML/ASM.
6 GROUNDING

This section specifies the Grounding of the formal model for describing DIP behavior interfaces as defined in Section 4 towards the Web Service Description Language WSDL [15] as a W3C recommendation for executable Web service communication and information interchange technology that serves as a quasi-standard at this point in time. This grounding definition consists of two aspects: (1) grounding of WSMO ontologies to XML - the former represents the terminology definition within DIP interface descriptions, while the latter defines the terminology of WSDL descriptions, and (2) the grounding of communicative behavior descriptions specified in the formal model towards respective WSDL interface descriptions. The following addresses each aspect in detail and specifies the rules for grounding.

6.1 Grounding WSMO Ontologies to XML

Semantically, Web service inputs and outputs are described in WSMO using ontologies. Syntactically, they are described in WSDL using XML Schema. As already mentioned, we envision that semantic agents (e.g. an automatic goal fulfillment service discovery and execution engine) must be able to communicate with plain syntactic Web services (e.g. existing Amazon Web services) because industry is currently deploying heavily only syntactical Web services.

During the communication of a semantic-level client and a syntactic-level Web service, two directions of data transformations are necessary: the client’s semantic data must be written in an XML form that can be sent as a request to the service, and the response data coming back from the service must be interpreted semantically by the client. With respect to a client whose domain knowledge is represented using WSMO ontologies, this means that the request data instances must be transformed from WSMO to XML and, conversely, the response data instances must be transformed from XML back to WSMO.

The rest of this section is organized as follows. A background to XML and XML Schema as well as some details of previous work is provided in Section 2.1. Section 2.2 introduces three possible approaches for grounding the data model aspect of WSMO services. Section 2.3 describes the the preferred approach of grounding based on creating the mappings at the conceptual level. Section 2.4 describes the mappings for each of the four main XML Schema types with examples. The final section, section 2.5, provides a summary and makes some observations based on the definition of the mappings. The complete definition of the mapping is included in Appendix A.

6.1.1 Background and Related Work

In this part of the document we provide brief overviews of XML and XML Schema as background material for the later sections defining the mappings. We also examine some previous work on mapping between XML Schema and ontologies and indicate how the efforts described here are related to these earlier efforts.
XML

XML has revolutionised the way data is exchanged and represented across the Web. It provides a standard language for describing document types in any domain imaginable facilitating the sharing of data across different systems and most notably, the Internet. XML is flexible and extensible allowing users to create their own tags to match their own specific requirements. As a result many languages have been designed for use in very different fields based on XML. Some examples of these are RDF as a semantic annotation language based on triples, XHTML as a document markup language with stricter constraints then HTML, RSS for syndication of website feeds most often used by news sites and weblogs. Where two systems need to exchange data, an agreed XML document structure is often the simplest mechanism to achieve the desired results.

XML encodes data using a set of identifiable tags representing elements that may have attributes associated with them. Attributes are used to assign name-value pairs to specific elements. In XML documents, tags can be nested but may not overlap. Nested elements provide a means to define the structure of an XML document. For example, the XML snippet in Listing 1 denotes that the element with the tag ‘book’ has subelements with the tags ‘author’, ‘publisher’ and ‘yearPublished’. The element with the tag ‘book’ has one attribute denoting the book’s title. This is an example of a case where an attribute could be just as easily represented as a sub-element.

```xml
<book title="Harry Potter">
  <author> J. K. Rowling </author>
  <publisher> Random House </publisher>
  <yearPublished> 2000 </yearPublished>
</book>
```

However, despite all its advantages, XML documents are syntactic and by themselves, provide no description of the semantics of the tags used. These semantics must be agreed by the users of the XML before they exchange any messages. In reality, many applications using XML depend on a human reading the document and interpreting the tags based on their natural language meaning. Although this is effective for smaller scale applications, it does not scale to cases where the recipient of the XML may be an automated agent that does not know the semantics of the XML tags in advance.

XML Schema

XML Schema is a W3C Recommendation defining a schema language for XML. XML Schema provides a way to define constraints on the syntax and structure of an XML document. It defines the legal elements and attributes that can appear including their types. Additionally it can specify the order and number of child elements and can define default and fixed values for elements and attributes. In the sense that XML Schema can define the valid data and elements for XML documents, there is some overlap between XML Schema and WSMO ontologies. Although, principally, XML Schema addresses the problem of constraining the syntax and structure of a set of XML documents while WSMO ontologies model the semantic relationships in a particular domain.

The next paragraphs provides a description of the XML Schema components that are most interesting from the perspective of creating a mapping to WSMO. These com-
ponents are: simple type definition, complex type definitions, attribute declarations, and element declarations. Listing 2 is used to provide examples of each component.

Simple Type Definition: XML Schema provides a wide range of built-in datatypes, for example, string, integer, boolean, float, decimal etc. These are examples of one form of simple type definitions. Another form of simple type definitions can be used to provide constraints on the values of built-in types. For example, it may be necessary to restrict the allowed values of the positiveInteger datatype to a particular maximum value.

Complex Type Definition: Can be used to (i) define a datatype composed of sub-elements of other datatypes, (ii) define the allowed structure of child elements using the keywords all, sequence and choice or (iii) extend or restrict the definition of an exiting complex type. Additionally, the values of elements can be accompanied by constraints on their values.

Attribute Declarations: XML Schema attributes are are an association between a name and a simple type definition. They are used to define simple attributes for XML type definitions. Unlike element declarations, attributes can not contain other elements or other attributes. Attributes can either be defined in the scope of the entire XML Schema or inside a specific complexType definition.

Element Declarations: An element declaration is an association between a name and a type definition, either simple or complex. The scope of element declarations can be either global or in the scope of a containing complex type. Elements declarations define what element instances are valid for an XML document conforming to that XML Schema.

Listing 2 provides an example of a simple XML Schema (modified from an example in [28]).

```
01 <xsd:schema xmlns:xsd="http://www.w3.org/2000/08/XMLSchema">
02  <xsd:element name="resumes" type="resumeTypes"/>
03  <xsd:complexType name="resumesTypes">
04     <xsd:sequence>
05        <xsd:element name="applicantName" type="xsd:string"/>
06        <xsd:element name="jobsAvailable" type="jobListType"/>
07     </xsd:sequence>
08     <xsd:attribute name="applicationDate" type="xsd:date"/>
09  </xsd:complexType>
10  <xsd:complexType name="jobListType">
11     <xsd:sequence>
12        <xsd:complexType name="job" type="jobDesc">
13           <xsd:attribute name="jobid" type="xsd:string"/>
14        </xsd:complexType name="job" type="jobDesc"/>
15     </xsd:sequence>
16  </xsd:complexType>
17  <xsd:complexType name="jobDesc">
18     <xsd:element name="title" type="xsd:string"/>
19     <xsd:element name="salary">
20        <xsd:simpleType>
21           <xsd:restriction base="xsd:positiveInteger">
22              <xsd:maxExclusive value="55000"/>
23           </xsd:restriction>
24       </xsd:element>
```

Previous Work on Mapping between XML Schema and Ontologies

There has been significant work done on investigating the relationship between ontologies and XML Schema and whether or not it is relevant to compare the two. In [61], the authors demonstrate that although ontologies and XML Schema provide data description at different levels of abstraction, it is useful to compare the languages used to represent the data, and to derive some conclusions from the similarities and differences uncovered. Using the Ontology Inference Layer (OIL), defined in [37], as the ontology language for the comparison, they highlight the languages’ purposes as the core difference between them. XML Schema is concerned with defining the vocabulary and the constraints on the structure of well-formed XML documents. Ontology languages, including OIL and WSML, are concerned with providing a formal specification of a shared domain theory. Ontologies are not explicitly concerned with defining the structure of documents or even the exact representation of data items (e.g. format of a date datatype) but rather defining the meaning of the data that the documents contain. Significantly, the authors of [61] outline as the results of their comparison that both XML Schema and OIL can both be more expressive depending on the viewpoint adopted when defining the data model. The authors also propose a methodology for translating an ontology specification (in this case, OIL) into an XML Schema. Our approach for the translation between WSML and XML is intended to be based on those presented in [61]: with the difference that we are focusing on lowering ontology instances to XML instances and are not interested in obtaining an XML Schema from ontology.

Other attempts in this direction, have attempted to add semantic meta-information to instance data, to make it available for tasks involving reasoning. Such an approach proposed by [43] develops an RDF interpretation for XML documents. It considers that both structural and semantic mark-up can coexist in the same document and uses RDF to annotate existing XML documents.

Another type of approach avoids the requirement of changing or extending existing technology. In [61] the authors propose two sets of mappings, the first is used for lifting XML Schemas to OWL ontologies and the second is used to translate XML documents into RDF graphs. We use a similar approach to that used in the model for the lifting operations, with the difference that we propose the creation of only a single set of mappings (between XML Schema elements and WSMO ontology meta-model) and use it for both schema lifting and instance lifting.

The symmetric part of the problem, the lowering, was addressed in several other works investigating the relations between ontology representation languages and document structuring techniques (schemas) [61]. In [23] it is shown how an XML document type definition (DTD) is generated from a given ontology. XML instances are linked to that ontology and intelligent information integration and retrieval from the semi-structured documents (i.e. XML documents) is made possible.
6.1.2 Approaches to Grounding WSMO Ontologies to XML

The goal of this section is to describe a mechanism that will allow the author of a WSMO description of an existing service to create a mapping between the WSMO conceptual model and the XML Schema data model. Three possible approaches to this task are:

Create mappings at the conceptual (WSMO) level involving creating a WSMO ontology for the XML Schema used in the WSDL. Use XSLT to create direct mapping between XML and the XML syntax of the WSMO ontology. Use a direct mapping between the source XML data and the target WSMO ontology, using a mapping language specifically developed for this purpose. The first approach and focus of this section is to create the mappings between the two models at the conceptual (or ontological) level. A mapping between the meta model for XML Schema and the WSMO Ontology Metamodel is defined. This allows the automatic generation of an ad-hoc ontology for a specific XML Schema. Existing WSMO mediation tools can then be used to create mappings between the WSMO ontology used by the creator of the WSMO service description and the WSMO ontology created from the XML Schema. Based on these mappings, two sets of mapping rules can be created to be applied to instance data at run-time. One set of rules will be used to translate WSMO instances to XML instances. The other rule-set will be used to translate XML instances to WSMO instances. The benefit of this approach is that all mappings are created at the conceptual level taking advantage of the more expressive mapping techniques available for mapping between ontologies. It also decouples the definition of the mappings between ontologies from how these mappings will actually be executed for instance data at runtime.

A second approach is to create direct transformations between the XML used for the WSDL messages and the XML syntax of the ontology used for the WSMO description. The most prominent transformation language for this purpose is the XSL Transformations language (XSLT) which has widespread tool support. There are two main problems with this approach. One problem is that the mappings take place only on the syntactic level. There would be no possibility to use reasoning to provide a more sophisticated mapping. Another problem is that XSLT is designed to map between hierarchically structured XML documents. Representing a WSMO ontology using the WSML/XML syntax results in a flat document structure which would make the XSLT creation awkward and unnatural.

A third approach is to create a mapping language that would allow a direct translation between XML instances and WSMO instances. In this case a new language needs to be invented specifically for this transformation. A new mapping would be required between each WSDL service description (the XML part) and the ontology used by the corresponding WSMO description. There would be a limited opportunity for re-use of mappings. Additionally it would overlook the existing and ongoing work in developing data mediation tools based on semantic descriptions.

6.1.3 Grounding by Creating Mappings at the Conceptual Level

This section contains two subsections. The first of these provides an example for the first approach for grounding WSMO ontologies to XML as listed in the last section.
The second subsection describes the steps required to make the grounding a reality.

**Overview and Example**

To get an overview of what is involved, we imagine as example, a semantic service designer with the task of providing a semantic description for an Amazon Web service selling books. We consider the example in terms of grounding the data used in the messages sent and received by the service. The designer first creates an ad-hoc WSMO ontology for book sales based on the XML Schema used in the service’s WSDL description (using tool support).

In the simplest case, the designer finds that this generated ontology defines a suitable conceptual model for books and uses it when defining the messages for the service’s choreography in the WSMO service description.

A single pair of mappings is required:

- to map from instances of the generated WSMO ontology to instances of XML and to map in the opposite direction - from instances of XML to instances of the corresponding WSMO ontology.

It should be possible for these mappings to be automatically generated based on the rules used to create the ad-hoc WSMO ontology from the XML Schema.

A more complex case is where the designer wishes to use an existing ontology for books to create the WSMO service description. This may be because using an established ontology might be more beneficial in terms of advertising the WSMO service description for discovery. The data mediator is required to mediate between the ontology used in the WSMO service description and the ontology created from the XML Schema of the WSDL. The mappings created by the mediator would be created at design time using the existing data mediation tools developed for WSMO.

**Steps to ground WSMO to XML Schema**

Three distinct steps are required to ground the data part of WSMO service descriptions to the XML Schema used in WSDL:

Define a mapping between the XML Schema Conceptual Model [XMLSchema] to the WSMO Ontology metamodel. Create an executable description of the mappings in the first point to enable the automatic creation of ad-hoc WSMO ontologies from specific XML Schema. Create the bidirectional mappings rules to be used for the transformation between XML instances and WSMO instances. These mapping rules should be created at the same time as the generation of the ad-hoc WSMO ontology from an XML Schema. The creation of these mapping rules should be automatic as they should be completely derived from the actions described in the first two bullet points.

The left-most column indicates the fundamental task of creating a mapping between the XML Schema Conceptual Framework and the WSMO Ontology Metamodel. Defining this mapping allows for the creation of executable rules that can be used to automatically lift any XML Schema to an equivalent ad-hoc WSMO ontology. The central column of Figure 3 depicts this lifting. The right-most column in the figure shows the bidirectional mapping rules that will be necessary to lower instances of WSMO data to instances of XML data and conversely lift instances of XML data to
instances of WSMO data. As described earlier, lowering is required when a client using the WSMO description of a service for activities such as discovery, mediation and composition wants to make an invocation on the actual service using its WSDL interface description. Lifting is required when the service responds using instances of XML data that need to be returned to the semantics client.

The arrows labeled 1 and 2 indicate dependencies between the steps required to define WSMO grounding. Arrow 1 indicates that the mappings at the conceptual level are used to create the executable rules for lifting an XML Schema to WSMO ontology. Arrow 2 shows that during the creation of the add-hoc WSMO ontology, bidirectional rules are created for lifting XML instances to WSMO instances and for lowering WSMO instances to XML instances.

6.1.4 Definition of Mapping from XML Schema to WSMO

In this section we provide a broad description of the mappings for each of the four primary types of XML Schema elements. For some of the types, multiple nested mappings are possible which results in quite detailed mapping descriptions. For complete detailed descriptions and listings of all the mappings, the reader is referred to Appendix A. In this section we provide a representative mapping for each of the four primary XML Schema types. A mapping for a simpleType, a mapping for a complexType, a mapping for attributes and finally, a mapping for elements.

The notation $T(\ )$ is used in the following listings to denote the transformation function defined by the mappings.

**Simple Type Definitions**

For the XML Schema built-in types, no mapping is required as WSML supports the use of the built-in types defined by the XML Schema namespace at [XMLSchema]. Where a simple type definition is used to create a new type based on restricting one of the built-in type, the mapping results in the creation of a WSMO concept with type of the built-in type in WSML along with an axiom that defines the restriction. Where the simpleType definition is based on a list of other simpleTypes, as in listing 3, the mapping results in the creation of a concept with an attribute resulting from the transformation of the list.

$$
T(<\text{simpleType} \text{id=ID name=NCName any attributes}> \\
(\text{annotation?}, \text{list}) \\
</\text{simpleType}>)) \rightarrow \\
'\text{concept'} \ \text{NCName} \\
\ T(\text{annotation}, \text{ID}) \ ? \\
\ T(\text{list})
$$

$$
T(<\text{simpleType} \text{id=ID any attributes}> \\
(\text{annotation?}, \text{list}) \\
</\text{simpleType}>, X ) \rightarrow \\
'\text{concept'} \ X \\
\ T(\text{annotation}, \text{ID}) \ ? \\
\ T(\text{list})
$$
Complex Type Definitions

Complex type definitions can contain sub-components that are a mixture of elements, attributes and other complex type definitions. They can also contain keywords to indicate the correct structure for XML to be compliant with the type (sequence, all and choice). Complex types always map to a concept in WSML. Sub components with a simple built-in type are mapped to attributes of the concept with the same built-in type. Sub components with simple types that are not built-in are mapped to attributes with the type of the mapped simple type definition. A sub component that itself is a complex type leads first to the creation of a corresponding concept. Listing 4 shows an example of the mapping for a complexType.

```
T(<complexType
    id=ID ?
    name=NCName ?
    abstract=true|false ?
    mixed=true|false ?
    block=#all | list of (extension | restriction)) ?
    final=#all | list of (extension | restriction)) ?
    any attributes >

(annotation?,
 (simpleContent | complexContent |
  (group | all | choice | sequence)?,
  ((attribute|attributeGroup)*,anyAttribute?)
 )
 )

</complexType >) ->

'concept' NCName

'nfp'

'xmlType' 'hasValue 'complexType'

'abstract' 'hasValue ('true' | 'false') ?

'mixed' 'hasValue ('true' | 'false') ?

'block' 'hasValue ('#all' |
  { getID(T(extension1)), getID(T(extensionn)) } |
  { getID(T(restriction1)), getID(T(restrictionm)) } ) ?

'final' 'hasValue ('#all' |
  { getID(T(extension1)), getID(T(extensionn)) } |
  { getID(T(restriction1)), getID(T(restrictionm)) } ) ?

'endnfp'

( 'simpleContentAttribute' 'ofType' getID(T(simpleContent) ) | 'complexTypeAttribute' 'ofType' getID(T(complexContent) )) ) |

( 'groupAttribute' 'ofType' getID(T(group) ) | 'allAttribute' 'ofType' getID(T(all)) ) |

'choiceAttribute' 'ofType' getID(T(choice) )) |

'sequenceAttribute' 'ofType' getID(T(sequence) )) ) ?

getQName(attribute1) 'ofType' getID( T(attribute1) ) ?

...

getQName(attributeN) 'ofType' getID( T(attributeN) ) ?

getQName(attributeGroup1) 'ofType' getID( T(attributeGroup1) ) ?

...

getQName(attributeGroupN) 'ofType' getID( T(attributeGroupN) ) ?
```
Attributes

Attribute elements in XML Schema can have as parent, the element element, the schema itself or a complexType elements. When the parent is the schema itself, the attributes have schema-wide scope. We map attributes to WSMO concept definitions as shown in Listing 5. XML Schema attribute.

```xml
T(<attribute
    id=ID name=NCName default=string fixed=string form=qualified | unqualified
    use=optional | prohibited | required any attributes
  >
  (annotation?, (simpleType) )
</attribute >) ->
  'concept' NCName
  'nfp'
    'xmlType' 'hasValue' 'attribute'
    'default' 'hasValue' string ?
    'fixed' 'hasValue' string ?
    'form' 'hasValue' 'qualified' | 'unqualified' ?
    'use' 'hasValue' 'optional' | 'prohibited' | 'required' ?
  'endnfp'
    'attributeSimpleType' 'ofType' getID(T(simpleType ) )
```

Elements

Elements can also have schema-wide scope and, like XML Schema attributes, are mapped to WSMO concepts. Listing 6 gives an example.

```xml
T(<element id=ID name=NCName substitutionGroup=QName default=string
  fixed=string form=qualified | unqualified ?
  maxOccurs = nonNegativeInteger |
  unbounded minOccurs = nonNegativeInteger nillable = true |
  false ?
  abstract = true | false any attributes >
  (annotation?, ((simpleType | complexType)?, (unique | key | keyref )* ))
</element >) ->
  'concept' NCName
  'nfp'
    'xmlType' 'hasValue' 'element'
    'substitutionGroup' 'hasValue QName ?
    'default' 'hasValue' string ?
    'fixed' 'hasValue' string ?
    'form' 'hasValue' 'qualified' | 'unqualified' ?
    'maxOccurs' 'hasValue' (nonNegativeInteger |
      'unbounded') ?
    'minOccurs' 'hasValue' nonNegativeInteger ?
    'nillable' 'hasValue' ('true' | 'false') ?
    'abstract' 'hasValue' ('true' | 'false') ?
  'endnfp'
```
6.1.5 Summary

As stated in [61] there are many differences between ontologies and XML Schema which from one viewpoint suggests that it is not a good idea to identify a mapping between them. Ontologies in many ways provide a higher level of abstraction than XML Schema and a more expressive way to deal with the specification of a conceptual model including representing relationships between concepts and the formal axioms constraining on how the concepts can be instantiated. On the other hand XML Schema provide a way to define both the vocabulary and structure that a compliant XML document must use. WSML is less verbose than XML but it depends on the reader’s perspective which of the two languages is more human-readable. What is certain is that XML is a purely syntactic language and although the use of XSLT to transform XML documents to other formats is quite powerful, it is still rooted in syntax and takes no account of the semantics of the underlying data. Using a combination of XML and XSLT to transform between different schema definitions at the data instance level provides little scope for reuse. We believe that developing a mechanism to lift XML instances to equivalent WSMO instances based on mapping between the XML Schema and the WSMO Ontology offers a more flexible approach. We also believe that the potential for reuse of data mediation tools is high and that much of this process can be automated.

One of the challenges is how to avoid losing structural information about the XML document during translation to WSMO. For example, when mapping an instance of XML conforming to the XML Schema to an instance of a WSMO ontology fragment conforming to the created WSMO ontology, the information regarding which WSMO concepts corresponded to XML Schema elements and which to XML Schema attributes must not be lost. The mappings described in Appendix A maintain the XML Schema structural information using the non-functional properties mechanism of WSMO.

6.2 Grounding Ontologized ASMs to WSDL

In the grounding of a WSMO service description to WSDL, we aim to provide all information necessary for the clients to be able to communicate with the service using Web service technologies (for example using SOAP over HTTP).

As indicated in the introduction, WSMO choreography describes the interface between the Web service and its clients. A choreography is a state machine whose states are made up of concept instances and relations between them. For the purpose of marking inputs and outputs, certain concepts may be assigned one of the roles "in", "out" or "shared". The client may write (create or change) instances of concepts with roles "in" and "shared", and similarly it may read instances of concepts with roles "out" and "shared". WSMO choreography also allows roles "controlled" and "static", but such concepts are not accessible from outside the choreography and as such they are not involved in grounding.
In the WSDL view of Web services, the client can send messages to a Web service and the service can send messages back to the client. The main purpose of this section is to describe how the client generates concrete messages represent writing instances (roles "in" and "shared") in the choreography state, and how existing instances (with roles "out" and "shared") within the state can be sent as the appropriate messages to the client, in effect making the client read them.

The XML contents of the messages and the networking details necessary for their transmission are captured in WSDL descriptions. Section 1.1 below presents a short overview of the relevant parts of the Web Services Description Language. In case there is an existing WSDL description for a Web service, the appropriate mappings from WSMO can be established by following the rules in section 1.2. Alternatively, when no WSDL description exists for a WSMO Web service, section 1.3 specifies how one can be generated, so that WSDL-based tooling can be used to access the Web service.

### 6.2.1 WSDL Overview

Web Services Description Language describes Web services in two levels; an XML-based reusable abstract interface and the concrete details regarding how and where this interface can be accessed. All descriptions in WSDL are centered on the Web service and all terminology follows the service’s point of view, for example input messages are messages coming into the service from the network and output messages are messages generated by the service and sent to the network. The first three subsection below talk about various aspects of WSDL descriptions, based on WSDL version 2.0, namely about abstract Web service interfaces, binding them to concrete wire protocols and endpoints and finally about the overall organization of WSDL documents. The fourth subsection details the relevant differences in the older version, WSDL 1.1.

**Web Service Interface**

On the abstract level, a Web service interface is described in terms of data schemas and simple message exchanges. In particular, WSDL models *interfaces* as sets of related *operations*, each consisting of one or more messages. For example an *interface* of a ticket booking Web service can have operations for querying for a trip price and for the actual ticket booking:

1. interface name="BookTicketInterface"
2. operation name="queryPrice" pattern=".../in-out"
3. input element="ns:TripSpecification"/
4. output element="ns:PriceQuote"/
5. outfault ref="TripNotPossible" /
6. /operation
7. operation name="bookTicket" pattern=".../in-out"
8. input element="ns:BookingRequest"/
9. output element="ns:Reservation"/
10. outfault ref="CreditCardNotValid" /
11. outfault ref="TripNotPossible" /
12. /operation
13. fault name="TripNotPossible" element="ns:TripFailureDetail"/
14. fault name="CreditCardNotValid" element="ns:CreditCardInvalidityDetail"/
15. /interface

In WSDL, an operation represents a simple exchange of messages that follows a specific message exchange pattern (MEP). The simplest of MEPs, "In-Only", allows a single application message to be sent to the service, and "Out-Only" symmetrically allows a single message to be sent by the service to its client. Somewhat more useful is the "Robust-In-Only" MEP, that also allows a single incoming application message but in case there is a problem with it, the service may reply with a fault message. Perhaps the most common MEP is "In-Out", which allows an incoming application message followed either by an outgoing application message or an outgoing fault message. Finally, an interesting MEP commonly used in messaging systems is "In-Optional-Out" where a single incoming application message may (but need not) be followed either by a fault outgoing message or by a normal outgoing message, which in turn may be followed by an incoming fault message (i.e. the client may indicate to the service a problem with its reply).

Particular messages (incoming, outgoing) in an operation reference XML Schema element declarations to describe the content. Fault messages, however, reference faults defined on the interface level (see above the `outfault` element), with the intention that semantically equivalent faults can be shared by different operations. Additionally, there may be multiple fault references for the same MEP fault message; in effect WSDL faults are typed and one operation can declare that it can result in any number of alternative faults (apart from the single success message).

**Web Service Endpoints, Bindings**

In order to communicate with a Web service described by an abstract interface, a client must know how the XML messages are serialized on the network and where exactly they should be sent. In WSDL, on-the-wire message serialization is described in a `binding` and then a `service` construct enumerates a number of concrete endpoint addresses.

A binding generally follows the structure of an interface and specifies the necessary serialization details. The WSDL specification contains two predefined binding specifications, one for SOAP (over HTTP) and one for plain HTTP. These bindings specify how an abstract XML message is embedded inside a SOAP message envelope or in an HTTP message, and how the message exchange patterns are realized in SOAP or HTTP. Due to extensive use of defaults, simple bindings only need to specify very few parameters, as in the example below. A notable exception to defaulting in binding are faults, as in SOAP every fault must have a so called fault code with two main options, Sender or Receiver, indicating who seems to have a problem. There is no reasonable default possible for the fault code.

Bindings seldom need to contain details specific to a single actual physical service, therefore in many cases they can be as reusable as interfaces, and equivalent services by different providers only need to specify the different endpoints, sharing the interface and binding descriptions.

The `service` construct in WSDL represents a single physical Web service that implements a single interface. The Web service can be accessible at multiple endpoints,
each potentially with a different binding, for example one endpoint using an optimized messaging protocol with no data encryption for the secure environment of an intranet and a second endpoint using SOAP over HTTPS for access from the Internet.

1. binding
2. name="SOAPTicketBooking"
3. interface="BookTicketInterface"
4. type="http://www.w1.org/2004/08/wsd1/soap12"
5. wsoap:protocol="http://www.w1.org/2003/05/soap/bindings/HTTP/"
6. fault ref="TripNotPossible" wsoap:code="soap:Receiver"/
7. fault ref="CreditCardNotValid" wsoap:code="soap:Sender"/
8. /binding
9.
10. service
11. name="DERITicketBooking"
12. interface="BookTicketInterface"
13. endpoint
14. name="normal"
15. binding="SOAPTicketBooking"
16. address="http://deri.example.org/tickets" /
17. /service

**WSDL Documents**

Apart from the interfaces, bindings and services described above, WSDL documents can contain further elements, enclosed in the root `description` element.

In order to facilitate true reuse of interfaces or bindings, WSDL documents can be modularized by using include and import mechanisms. When a WSDL document is parsed, imports and includes are resolved so the resulting model is not aware that some pieces may have come from different actual files.

As a container for data type information, WSDL documents have a section called `types`. Actual schemas can either be embedded directly in this section or referred to using the appropriate import statements, for example external XML Schema documents can be imported by putting the `xs:import` element directly in the `types` section. By default, WSDL uses XML Schema to describe data, but WSDL extensibility allows other data type systems to be used instead.

Finally, every element in a WSDL document can be annotated with documentation elements or it can contain extensibility elements or attributes.
Note on the differences between WSDL 2.0 and WSDL 1.1

This note details the differences between WSDL version 1.1 [15], a specification authored by several companies and submitted to the W3C as the basis for standardization work, and WSDL version 2, the resulting draft standard. While this document uses the cleaner version 2 of WSDL, actual deployment prefers WSDL 1.1 because WSDL 2 is not yet finished and implemented. This note aims to limit any confusion stemming from the situation that some readers may only be familiar with WSDL 1.1.

The first notable difference is that several constructs from WSDL 1.1 were renamed in WSDL 2. In particular, portType in WSDL 1.1 is known as interface in WSDL 2 and port in WSDL 1.1 (occurring within a service) is now known as endpoint. Also, the WSDL document root element is called definitions in WSDL 1.1 and description in WSDL 2. Importantly, the intention of all these renamed constructs is unchanged between the two WSDL versions.

A larger difference is that while WSDL 2 uses XML Schema element declarations to describe messages, WSDL 1.1 had a special construct, message, that contained potentially several parts, each referencing a single XML Schema element or type declaration. However, the use of multiple parts in a single message is usually translatable to a single element containing a sequence of elements (one for each part), making the different approaches in WSDL 1.1 and in WSDL 2 equivalent for all practical purposes.

6.2.2 Grounding to existing WSDL descriptions

In the state signature of a WSMO choreography, certain concepts can be marked with role “in”, “out” or “shared”, and the instances of these concepts can then be accessed by the client. In terms of Web service message exchanges this means that messages coming from the client can result in the creation or updates of instances of concepts with the role ”in” or ”shared”, and conversely messages from the service represent instances of concepts with the role ”out” or ”shared”. This section describes how we can establish a concrete mapping between the accessible concepts and the appropriate WSDL interface messages.

Aside from mapping the choreography to a WSDL interface, we need to provide information on how the XML messages described in the interface are serialized on the wire and where they need to be sent. All this information is present in a WSDL service, therefore this section also describes how the appropriate WSDL service can be referenced from a WSMO Web service.

Correspondences between WSDL and WSMO

First, let us clarify what is meant by ”appropriate” WSDL messages and services, as used in the text above. In a WSMO choreography, data is represented as instances of application-specific ontological concepts and relations. In Web service messages, data is represented as XML trees conforming to application-specific schemas. The creator of the grounding mappings must know which parts of the XML trees correspond to which ontological instances. Section 2 specifies how the correspondence can be described so that the machine can transform between the two forms automatically, but usually a human is required to establish and/or verify that some WSDL message actually corresponds to some ontological data. This can be done, for example, by reading
the definitions and documentation of the WSDL messages and ontology concepts and deciding if they match.

A similar process must be used already when deciding that a particular WSMO Web service’s choreography should be grounded to a particular WSDL interface. The remainder of this document assumes that a suitable WSDL interface has been selected for grounding and that the pairs of corresponding XML types and ontological concepts are known.

It is important to emphasize that a single input or output concept in WSMO choreography does not necessarily correspond directly to a single message in WSDL. Web services tend to use a small number of operations with relatively large and complex messages [10], and in fact coarse-grained messages are suggested by many proponents of Web services as best practice that reduces networking overhead and simplifies the interactions. On the other hand, it has not been investigated yet whether WSMO choreography designers should make their state ontologies coarse-grained or fine-grained. Ontologies tend to be fine grained due to the principle of separation of concern - if a concept serves two orthogonal purposes, it will likely be split into two related concepts, with the hope that this might facilitate reuse.

Due to these granularity differences, we expect that a WSDL interface operation can generally correspond to one or more transition rules. We have identified several important correspondence patterns, described in the subsections below, which might be useful for designers creating a WSMO choreography and grounding it to a WSDL.

**Single rule for one request/response operation**

When the granularity of the WSMO choreography description of a Web service matches that of the WSDL interface for this service, each request/response operation will correspond to a single transition rule. The \textit{in} concept used in the rule’s condition will be grounded to the input message of the operation, and the \textit{out} concept whose instance is created in the rule’s update statement.

For example, the operation \textit{getStockQuote}, which takes a stock symbol as its input and returns the current price as the output, would correspond to a transition rule that matches on stock quote request instances (containing the stock symbol) and results in the creation of response instances with the appropriate prices. The concept \textit{StockQuoteRequest} would be grounded to the input message of \textit{getStockQuote}, and the concept \textit{StockPrice} would be grounded to the output message.

Note, however, that in defining an orchestration, a single request/response interaction cannot be tied to a single rule. One rule will populate the \textit{out} concept and subsequent rules will be guarded on the population of the possible \textit{in} concepts that stand for the possible responses.

**Multiple rules for an aggregate operation**

In order to achieve coarser granularity, an operation in a WSDL interface may aggregate a number of logically separate actions, which could otherwise be modeled as separate operations and executed in sequence in a finer-grained interface. The choreography may model these actions separately, to achieve better clarity of intent, in an example of granularity mismatch between the WSDL interface and the choreography.

The aggregate operation will carry a number of logically separate pieces of data in the input message, and likely an equal number of responses in the output message.
These pairs of input data and responses will each be handled by a different transition rule in the choreography. In this example all the in concepts used in the conditions of these transition rules will be grounded to the same single input message of the WSDL operation, and likewise the out concepts created in the update statements of the separate rules will all be grounded to the one output message of the operation.

As an example situation where this pattern would occur we can take the operation SetResourceProperties (from Web Services Resource Framework - WSRF), which aggregates in a single operation any number of property inserts, updates or deletions. The WSMO choreography could be modeled with three separate transition rules, each handling one of the three property change operators, and all the concepts InsertResourceProperty, UpdateResourceProperty and DeleteResourceProperty would be grounded to the single input message of the operation SetResourceProperties.

Grounding one concept to multiple operations

This pattern does not deal directly with the correspondence between transition rules and operations, instead it describes a special case that can occur in the previous patterns, of which a grounding designer should be aware.

In certain situations, detailed in the following paragraphs, multiple operations in a WSDL interface can transfer the same in or out concept, and each such concept has to be grounded to all the appropriate messages.

For in concepts, it is possible that two or more operations do in fact have the same input. For example if an interface contains an aggregate operation (like the SetResourceProperties above) together with the constituent operations (in WSRF those would be InsertResourceProperties, UpdateResourceProperties and DeleteResourceProperties, each only allowing one particular change operator). In this case, the concept InsertResourceProperty would be grounded to both the input message of the operation InsertResourceProperties and to the input message of SetResourceProperties.

For out concepts, we have a situation of multiple operations with different inputs, but with the same output. For example the Amazon Web Services interface has methods ItemSearch and ItemLookup, which both return a list of matching items. The input message of ItemSearch specifies the search criteria, the input message of ItemLookup carries a list of particular item IDs, but the output messages of these two operations are exactly the same - an ItemList. The two operations would be modeled as two different transition rules in the choreography (both according to section 1.2.1.1) but the concept ItemList would have to be grounded to the output messages of both of the operations at the same time.

Grounding values

As already mentioned, a WSMO choreography assigns externally accessible concepts to specific roles, which are either "in", "out" or "shared". Additionally, WSMO choreography requires that all accessible concepts also provide a grounding mechanism by pointing to it with its URI after the keyword withGrounding. There can be multiple grounding specifications that will describe the allowed values for this property and what they mean. In this specification, we describe what values the property can have to ground the concepts to WSDL messages, using the following rules:
• grounding may be specified using multiple URI values (cardinality issues are discussed in the following rules)

• grounding of a concept with role "in" or "shared" must contain a URI identifying an appropriate WSDL input or input fault message

• grounding of a concept with role "in" MUST NOT contain a URI identifying a WSDL output or output fault message

• grounding of a concept with role "out" or "shared" must contain a URI identifying an appropriate WSDL output or output fault message

• grounding of a concept with mode "out" MUST NOT contain a URI identifying a WSDL input or input fault message

• one concept can be grounded to multiple WSDL messages with the same direction because one piece of data can in fact be transmitted alternatively in different messages, as shown above in section 1.2.1.3

• multiple concepts can be grounded to a single WSDL message because Web service messages should be coarse-grained and transfer as much data as available (as discussed above in section 1.2.1, especially section 1.2.1.2), and the data grounding can map multiple instances to a single XML tree

Note that the rules above treat WSDL fault messages as equivalent with normal application messages. This is the case because fault messages can also carry useful information and so they can also be received and translate into "in" or "shared" concept instances or they can be emitted from "out" or "shared" concept instances.

Finally, in order to provide the on-the-wire serialization and endpoint information, we need to link from a WSMO Web service to a WSDL service. We propose a new endpointDescription non-functional property, which contains a URI identifying the appropriate WSDL service.

**URIs for identifying WSDL components**

As described in Appendices A.2 and C of [15], the URIs for WSDL 2 components are created by combining the namespace URI with fragment identifiers that unambiguously identify components on any level of granularity within a WSDL file. For our purposes, we need URI references for Interface Message References and Interface Fault References (both owned by operation components), and for Services.

The fragment identifier for Interface Message References is defined as

\texttt{wsdl.interfaceMessageReference(interface/operation/message)}

with the three parts in parentheses replaced with the following:

• \textit{message} is the message label property of the Interface Message Reference component (see below),

• \textit{operation} is the local name of the operation that contains the message,

• \textit{interface} is the local name of the interface owning the operation.
The message label property is defined in the message exchange pattern used by the operation and indirectly identifies the direction of the message. In all the MEPs predefined by WSDL 2, the message labels are either "In" or "Out" (note the capitalization) and the corresponding messages have directions matching the labels.

Similarly, the fragment identifier for Interface Fault References is defined as \texttt{wsdl.interfaceFaultReference(interface/operation/message/fault)} and the four parts in parentheses are replaced as follows:

- \texttt{fault} is the local name of the interface fault referenced by the fault message
- \texttt{message} is the message label property of the Interface Fault Reference component (see below for an important difference from message label on Interface Message References),
- \texttt{operation} is the local name of the operation that contains the fault message,
- \texttt{interface} is the local name of the interface owning the operation.

Note: the message label property does not directly point to a fault message in the message exchange pattern, instead it points to the message to which this fault is related. The kind of relation depends on the faulting rules used by the message exchange pattern. In particular, in some MEPs a fault can replace any message after the first one (in the usual input/output MEP, a fault can replace the output message), thus ending the message exchange; the message label is then the label of the replaced message and the direction of the fault is the same as that of the replaced message. On the other hand in other MEPs any application message may trigger a fault (the robust in-only MEP, for example, consists of a single incoming message and an optional fault message back, in case something is wrong with the incoming message) and in this case the message label is the label of the message that triggered the fault and the direction is opposite; the fault returns from the receiver of the original message. For illustration, an incoming fault reference may have message label "Out". Therefore to compute the URI for a fault reference, if the message label is not explicit in the WSDL file (it may be defaulted), the knowledge of the operation’s MEP is necessary.

Finally, the fragment identifier for WSDL Services is defined as \texttt{wsdl.service(service)} where the \texttt{service} parameter is the local name of the WSDL service.

For instance, if \textit{BookTicketInterface} from section 1.1.1 and the \textit{DERITicketBooking} service from section 1.1.2 are defined in a WSDL file with the target namespace "http://example.com/", the URIs for the two messages and two faults in operation \textit{bookTicket} and for the whole service will be the following:

\begin{verbatim}
http://example.com/wsdl.interfaceMessageReference(BookTicketInterface/bookTicket/In)
http://example.com/wsdl.interfaceFaultReference(BookTicketInterface/bookTicket/Out/TripNotPossible)
http://example.com/wsdl.service(DERITicketBooking)
\end{verbatim}
When using WSDL 1.1, because there is no such standard way of creating the reference URIs, we simply adapt the above approach, using the appropriate portType name where interface name is expected and reusing the strings "In" and "Out" in lieu of input and output message labels.

Grounding example

The following is a simple choreography for a ticket booking service (with the WSDL interface specified earlier in listing 19), including highlighted grounding information for its accessible concepts. This ontology has been adapted from [22].

```xml
namespace {"http://example.org/bookTicket#",
    dc_"http://purl.org/dc/elements/1.1#",
    tr_"http://example.org/tripReservationOntology#",
    wsml_"http://www.wsmo.org/wsml/wsml-syntax#",
    po_"http://example.org/purchaseOntology#"
}

ontology_"http://example.org/BookTicketInterfaceOntology#"
    nonFunctionalProperties
    dc#title hasValue "Book Ticket Interface Ontology"
    dc#creator hasValue "DERI Innsbruck"
    dc#description hasValue "an ontology that redefines concepts and relations from other ontologies in order to reuse them in the choreography and orchestration; two additional non-functional properties are defined for the targeted concepts and relations: mode and grounding"
    dc#publisher hasValue "DERI International"
    endNonFunctionalProperties


    concept reservationRequest subConceptOf tr#reservationRequest
    concept reservation subConceptOf tr#reservation
    concept temporaryReservation subConceptOf tr#reservation
    concept creditCard subConceptOf po#creditCard
    concept negativeAcknowledgement

    webService_"http://example.org/BookTicketService#"
    nonFunctionalProperties
    wsml#endpointDescription_"http://example.com/#wsdl.service(DERITicketBooking)"
    endNonFunctionalProperties

    interface BookTicketInterface
    choreography BookTicketChoreography
    importsOntology _"http://example.org/BookTicketInterfaceOntology#"
    stateSignature
    in
        reservationRequest withGrounding
    _"http://example.com/#wsdl.interfaceMessageReference
        (BookTicketInterface/bookTicket/In)"
    creditCard withGrounding
    _"http://example.com/#wsdl.interfaceMessageReference
        (BookTicketInterface/bookTicket/In)"
    out
```
47 reservation withGrounding
48 _"http://example.com/#wsdl.interfaceMessageReference
49 (BookTicketInterface/bookTicket/Out)"
50 negativeAcknowledgement withGrounding
51 _"http://example.com/#wsdl.interfaceFaultReference
52 (BookTicketInterface/bookTicket/Out/CreditCardNotValid)"
53 controlled
54 temporaryReservation
55 transitionRules
56 [...]}

Processing grounding information

From the point of view of WSDL, Web services are software entities that communicate by sending and receiving messages. In this model, both services and their clients must be able to choose at the appropriate times that they want to send a message or that they want to be able to receive a message. These decisions are currently usually hard-coded in the service or client implementation. By describing the choreography interface, WSMO attempts to automate the task of using previously unknown Web services.

WSMO Choreography describes the service’s interface as a state machine that communicates with its clients using what we call “accessible” concepts in its state. The client can write instances of concepts with role "in" and "shared", and it can read instances of concepts with role "out" and "shared". For grounding in WSDL, these reads and writes must be translated to sending and receiving messages. This section specifies how a client following (executing) a service’s choreography interface makes decisions to send or to expect receiving the messages that the choreography’s accessible concepts are grounded to.

To clarify, this section does not specify how a choreography is executed, it only assumes that the client, after discovering a service, can follow the service’s choreography, for example by executing it in its (the client’s) local choreography engine.

To be able to emit messages to the service, the client must have the necessary data available to it. At least for the initial message that starts the conversation, the message data comes from the client’s previous knowledge. For example, the client may know the user’s detailed goal, it may have the results of a previously invoked Web service, or it may have the capability of directly and interactively requesting the necessary data from the user. In the remainder of this section, we simply assume the client has some data available to it that it may send to the Web service. From the point of view of the state machine, this available data is in the state.

The process followed by the client is as follows:

- The client initiates the choreography, i.e. starts its execution.

- When a rule fires having used some "in" or "shared" instances in its condition, these instances are marked internally to be sent to the service.

- Whenever the client has some instances marked to be sent, it will perform the following steps:
  - Select the WSDL messages to which the instances are grounded.
– Filter out the WSDL messages to which multiple concepts are grounded and some of them are not available to the client (see rule 7 in section 1.2.2).
– From the remaining WSDL messages, select those that the client can currently send (i.e. the client has the initiative in the MEP of the message’s operation; see below for more discussion of this point).
– If more than one message remains, one should be selected using some out-of-bounds mechanism that we do not specify.
– Send the selected message to the service, removing the ”to be sent” mark from the instances sent in the message.

• As the rules in the choreography fire, the update statements may change some instances with the mode ”out” or ”shared”. The client’s choreography engine should not attempt to perform these updates, instead the client will now expect that the service sends a message with the appropriate new data.

• From the client’s point of view, the conversation may end when the client receives the required data from the service.

Above, the rule 3c talks about messages that the client can currently send. While it would seem that the client may choose to send any message at any moment, the WSDL contract constrains this behavior: if an operation in WSDL has multiple messages in its message exchange pattern, these messages can only be sent in the temporal sequence specified by the MEP. For example, in an out-in operation (request/response from the service), the in message (the response) cannot be sent before the out message (the request). In other words, if the client could send a message but the operation mandates that another message must precede it, the client must wait until that preceding message has been sent.

It is an open question on whether the client can stop the conversation with the service when it gets the response it seeks (see point 5 above); such a situation might occur when a client only needs a part of a service’s functionality. This consideration is left for later, though, as examples of such situations are not currently clear.

### 6.2.3 Generating WSDL from WSMO choreography

The previous section has specified how a WSMO Web service description (and especially its choreography) can be grounded to a concrete existing WSDL interface. However, we can envision that Web services design can start on the semantic level, i.e. first creating a WSMO description. A Web service can even be implemented on the semantic level in frameworks like WSMX. Still, such a Web service should be able to receive and emit XML messages. This section, by providing a set of rules for creating a WSDL description for a WSMO Web service, effectively proposes a default grounding, a default communication protocol for WSMO Web services.

WSMO describes the interface between a Web service and its clients using the so called choreography. In a choreography, ontology concepts can be marked with roles ”in”, ”out” and ”shared” to indicate that they can be read or written by the environment, effectively by the client. We call these concepts and their instances ”accessible”. Since Web services communicate by sending messages, receiving a message from the
client means writing an accessible instance, and reading an accessible instance is substi-
tuted by the service sending the instance in a message to the client.

The following subsections first describe the exact rules for generating the default
WSDL description from a WSMO Web service and its choreography, then show an
example of a generated WSDL description and finally discuss the limitations of our
default binding approach.

Rules for generating WSDL

The following is a skeleton of a WSDL document generated from a WSMO Web service
and its choreography:

1. description xmlns="http://www.w1.org/2005/05/wsdl"
2. targetNamespace="Web service namespace ID"
3. xmlns:xsd="http://www.w1.org/2001/XMLSchema"
4. xmlns:wsoap="http://www.w1.org/2005/05/wsdl/soap"
5. xmlns:data="ontology ID"
6.
7. types
8. xs:import namespace="http://www.wsmo.org/wsml/wsml-syntax"
9. schemaLocation="http://www.wsmo.org/TR/d16/d16.1/v0.2/xml-syntax/wsml-xml-
syntax.xsd" /
10. xs:schema targetNamespace="ontology ID"
11. element declarations
12. /xs:schema
13. /types
14.
15. interface name="Web service local name"
16. operations
17. /interface
18.
19. binding name="DefaultSOAPBinding"
20. type="http://www.w1.org/2004/08/wsdl/soap12"
21. wsoap:protocol="http://www.w1.org/2003/05/soap/bindings/HTTP/"
22. /binding
23.
24. service name="Web service local name"
25. interface="Web service local name"
26. endpoint name="Web service local name"
27. binding="DefaultSOAPBinding"
28. address="Web service endpoint address" /
29. /service
30. /description
In the listing above, the *emphasized* items are placeholders as follows:

**Web service namespace ID**
If the IRI of the WSMO Web service can be split into a namespace/local name pair (called sQName in [19]), the *Web service namespace ID* is the namespace part. Otherwise, it is the whole Web service IRI.

**Web service local name**
If the IRI of the WSMO Web service can be split into a namespace/local name pair, the *Web service local name* is the local name part. Otherwise, it is "service".

**Web service endpoint address**
If the WSMO Web service has a non-functional property `endpointAddress`, its value becomes the *Web service endpoint address*. Otherwise this address must be provided externally when generating the WSDL.

**ontology ID**
The IRI of the state ontology used by the choreography of the WSMO Web service.

**element declarations**
XML Schema element declarations generated for all the accessible concepts as described below. **operations**
WSDL operations generated for all the accessible concepts as described below.

For every accessible concept in the choreography state signature we create an XML Schema element declaration according to the following template:

1. `xs:element name="concept name"`
2. `xs:complexType`
3. `xs:sequence`
4. `xs:element ref="wsml:instance" /`
5. `/xs:sequence`
6. `/xs:complexType`
7. `/xs:element`

In the listing above, the *concept name* is the local name of the accessible concept. The generated element declarations populate the schema in the resulting WSDL document. The content of the generated element is a WSML/XML instance element, i.e. we reuse WSML/XML instead of generating an XML Schemas for WSMO concepts, but generating such schemas would also be possible; for now we leave this as potential future work.

Next, for every "in" and "shared" concept in the choreography state signature we create an input WSDL operation according to the following template:

1. `operation name="write concept name"`
2. `pattern="http://www.w1.org/2005/05/wsdI/in-only"`
3. `input element="data: concept name"`
4. `/operation`

And finally, for every "out" and "shared" concept in the choreography state signature we create an output WSDL operation according to the following template:
1. operation name=""concept nameUpdated"
2. pattern="http://www.w1.org/2005/05/wsdll/out-only"
3. output element="data: concept name" /
4. /operation

In the two listings above, the concept name is the local name of the accessible concept, i.e. the name of the element declaration generated for this concept. For example, for a "shared" concept Vehicle we will generate an element declaration called Vehicle and two operations, readVehicle and writeVehicle. The generated operations populate the interface in the resulting WSDL document.

With the full WSDL generated, it is straightforward to annotate the WSMO description with grounding information.

Example for the default grounding

The following listing is a definition of a Web service adapted from Listing 21. The Web service here includes the highlighted endpoint address, otherwise it is unchanged.

```xml
namespace {_"http://example.org/bookTicket",
   wsml _"http://www.wsmo.org/wsml/wsml-syntax"}
webService _"http://example.org/BookTicketService"
   nonFunctionalProperties
   \textbf{wsmlendpointAddress_"http://example.org/bookTicketSOAPEndpoint"}
   endNonFunctionalProperties
interface BookTicketInterface
[...]
```

The WSDL generated for this Web service would look like this:

```xml
description xmlns="http://www.w1.org/2005/05/wsdl"
   targetNamespace="\textbf{http://example.org/}"
   xmlns:xs="http://www.w1.org/2001/XMLSchema"
   xmlns:wsoap="http://www.w1.org/2005/05/wsdl/soap"
   xmlns:data="\textbf{http://example.org/BookTicketInterfaceOntology}"
   types
   xs:import namespace="http://www.wsmo.org/wsml/wsml-syntax"
   schemaLocation="http://www.wsmo.org/TR/d16/d16.1/v0.2/xml-syntax/wsml-xml-syntax.xsd" /
   xs:schema
   targetNamespace="\textbf{http://example.org/BookTicketInterfaceOntology}"  
   xs:element name="\textbf{reservationRequest}"  
   xs:complexType  
   xs:sequence  
   /xs:sequence
   /xs:complexType
   /xs:element
   xs:element name="\textbf{reservation}"
   xs:complexType
   xs:sequence
   /xs:sequence
   /xs:complexType
```

Limitations of the default grounding

If we compare the WSDL interface in Listing 7 with the generated interface in Listing 15, it is apparent that a hand-crafted WSDL presents a more natural and usable interface than one generated from a WSMO choreography. This is the effect of the limitations of the default grounding, as detailed in this subsection. Perhaps the most visible limitation is that the default grounding cannot generate in-out (or request/response) operations. This is because we cannot easily guess a meaningful correlation between some concepts being written in the choreography state.
and some others being created, especially if there are intermediate steps, like the partial temporaryReservation in our example.

On a similar note, the default grounding cannot guess which concepts could be transmitted together in a single WSDL message, therefore distinct operations have to be generated for all accessible concepts.

Further, the default grounding cannot distinguish between normal application data and data indicating failure, which are distinguished in WSDL as normal messages and faults. The difference between normal messages and faults is in any case subjective and not unambiguous. For these reasons the generated WSDL interfaces do not contain faults.

In most WSDL interfaces, operations usually begin with a message from the client, implying that a response channel will be available for this operation, which is true, for example, for Web services using synchronous HTTP. In contrast, the default grounding generates out-only operations (sending a single message from the service to the client) for "out" concepts, so the client must be able to indicate where it can receive such messages. To do so, we recommend the use of the WS-Addressing \[40\] us:ReplyTo header. At the time of writing this, there is no agreed way of describing this behavior in the generated WSDL, but the Web Services Addressing working group at W3C plans to include a marker for this purpose.

And finally, due to the use of one-way "fire-and-forget" operations, any faults generated by the receiver of the messages will be lost. WSDL 2 currently provides robust one-way message exchange patterns (MEPs), allowing the creation of operations that only transmit a single application message, but may optionally transmit a fault back. These MEPs are not supported in the standard bindings (SOAP and HTTP), but if they get supported in the future, we may consider using them instead of the "fire-and-forget" MEPs.

To conclude, the default grounding has its uses, especially when used only as the concrete communication protocol for two WSMO-enabled nodes. For interoperation with existing syntactic-level Web services we suggest that a suitable WSDL description be created by the service provider and then the WSMO description of the service can be grounded to this WSDL using the rules of section 1.2.
7 Related Work

7.1 Web Service Technologies and Languages

The “Web services” approach is extending the Web from a support for information access to a middleware for B2B applications. In this paradigm, enterprises can externalise their businesses as Web services which are then interconnected using a stack of standards. SOAP [32], WSDL [15], and UDDI [30] are the basic technologies for Web services interactions. However, they are insufficient to manage service interactions that go beyond simple sequences of requests and responses or involve large numbers of partners.

One of the interesting concepts this technology offers is service composition; the possibility to define a new service by combining existing Web services. Standards for service composition cover mainly services choreography and orchestration.

The W3C Web Service Glossary defines choreography to be concerned with “the interactions of services with their users. Any user of a Web service, automated or otherwise, is a client of that service. These users may, in turn, be other Web Services, applications or human beings” [34]. With respect to this definition we identify two aspects that are related to choreography:

- the interface description of a Web Service (also called abstract process in BPEL, collaboration protocol in ebXML and choreography in WSMO and IRS 3) that describes the interaction behavior of the Web Service for consuming its functionality. We denote this as the choreography of a Web Service.

- the interaction protocol (also called global model in WSCI [31] and multiparty collaboration in ebXML [21]) that describes, from a global perspective, the communicative interaction of several Web Services and clients via their respective interfaces descriptions.

The W3C Glossary defines an orchestration as “the sequence and conditions in which one Web service invokes other Web services in order to realize some useful function. That is, an orchestration is the pattern of interactions that a Web service agent must follow in order to achieve its goal”. There is a strong relationship between choreography and orchestration.

A number of standardization proposals related to these concepts have been put forward over the past years (e.g., WSFL, XLang, BPML, WSCL, and WSCI), leading to two ongoing standardisation initiatives: the Business Process Execution Language for Web Services (BPEL4WS, BPEL or WS-BPEL) [12] and the Web Services Description Language (WS-CDL) [65]. In addition, one of the main purposes of the semantic Web services related approaches is facilitating automated composition. In the following we give an overview of these ongoing efforts.

7.1.1 WSDL based proposals

Web Services Description Language (WSDL) [15] describes Web services by an XML-based reusable abstract interface and the concrete details regarding how and where this interface can be accessed. In the section [?] we provide an overview on WSDL. In the rest of this section we expose the web services technologies WSDL-based.
The Web Service Choreography Description Language (WS-CDL) provides the choreography representation from a global point of view [68]. According to this vision, the choreography describes the behaviour observable from an external point of view, emphasizing the collaboration of parties, where the communication progresses only when jointly agreed ordering rules are satisfied.

A WS-CDL choreography description is contained in a package and is essentially a container for a collection of activities that may be performed by one or more of the participants. There are three types of activity in WS-CDL, namely control-flow activities, WorkUnit activities and basic activities.

There are three (types of) activities in the first category, namely Sequence, Parallel, and Choice. These activities are block-structured in the sense that they enclose a number of sub-activities. A WorkUnit activity describes the conditional and, possibly, repeated execution of an activity. The third type of WS-CDL activities, basic activities, includes Interaction, NoAction, SilentAction, Assign, and Perform.

It can be seen that the workflow and WorkUnit activities allow one to capture the basic control-flow constructs found in typical imperative programming languages. It can also be seen that these activities correspond to the Sequence, Flow, While, Switch, and Pick activities in BPEL with the exception that the mapping from Choice and WorkUnit activities to Switch an Pick activities may be non trivial.

7.2 Semantic based approaches

The syntactic-level description of Web services necessitates human intervention for services composition, thus hampering their usage in complex business contexts. Semantic Web Services overcome this problem by augmenting Web services with rich formal description of their capabilities in machine readable semantics. Mainly, we consider the following approaches: WSMO [54], IRS-III [20], OWL-S [41], SWS Framework, and METEOR-S [45].

7.2.1 WSMO

The Web Service Modelling Ontology (WSMO) semantically describes the core elements of Semantic Web Services, aiming at a framework for unambiguous formal descriptions on which inference mechanisms shall enable automated discovery, composition, execution and invocation of Web Services [54]. Choreography and Orchestration are defined as sub-classes of service interface for describing how the functionality of a Web Service can be consumed and how it is achieved by aggregating other Web Services. Therefore, a basic model for formally describing the dynamics of service interface descriptions has been defined. The formal model for WSMO service interface descriptions relies on Abstract State Machines (ASM for short). The core principles of ASMs are that they are state-based, they represent a state by a formal algebra, and they model state changes by guarded transition rules that change the values of functions and relations defined by the signature of the algebra [11].

The ASM-based model for service interfaces defined in WSMO provides the formal basis for specifying ontology data interchange within service interfaces. In accordance to the ASM framework, this model consists of three notions: (i) a vocabulary Ω that
defines the information space of a service interface on basis of ontologies, (ii) states $\omega(\Omega)$ that denote a status of the information space within the dynamics of a service interface that is defined by the attribute values of the ontology instances of $\Omega$, and (iii) Guarded Transitions $T$ that specify the dynamics of a service interface.

7.2.2 IRS-III

The IRS project has the aim of supporting the (semi) automated construction of semantically enhanced systems over the Internet. IRS-III is a framework and platform for developing semantic web services which utilizes the WSMO ontology. The overall design is based on: the use of ontologies and state, IRS-III playing the role of a broker, differentiating between communication direction and which actor has the initiative, having representations which can be executed, a formal semantics, and the ability to suspend communication.

IRS-III incorporates and extends WSMO ontology [20]. Sharing the WSMO approach, IRS-III considers choreography of single Web service. It focuses on how one Web service interacts with one another. The IRS acts as a broker for capability based invocation. A client sends a request to achieve a goal and the IRS finds, composes and invokes the appropriate web services. The choreography to the IRS is thereby fixed. Choreography descriptions are therefore written from the perspective of IRS as a client of the web service.

A choreography is described in IRS-III by a grounding declaration and a set of guarded transitions. The grounding specifies the conceptual representation of the operations involved in the invocation of a Web Service and their mapping to the implementation level. More specifically, the grounding definitions include operation-name, input-roles-soap-binding, output-role-soap-binding. The guarded transitions are the set of rules, which represent the interaction between IRS-III and the Web Service on behalf of an IRS client. They are applied when executing the choreography. This model is executed at a semantic level when IRS-III receives a request to achieve a goal. As IRS-III is also an implemented framework, we expose more in details the the its Choreography execution environment in the deliverable D3.5.

7.2.3 METEOR-S

The METEOR project at the LSDIS Lab, University of Georgia, focused on workflow management techniques for transactional workflows [45]. Its follow on project, which incorporates workflow management for semantic Web services is called METEOR-S. A key feature in this project is the usage of semantics for the complete lifecycle of semantic Web processes, which represent complex interactions between Semantic Web Services.

Different from OWL-S, WSMO and SWSO, qualified as revolutionary proposals, METEOR-S adopts an evolutionary approach by enriching the current Web services standards with semantic descriptions. With WSDL-S, one can augment the expressivity of WSDL with semantics by employing concepts similar to those in OWL-S while being agnostic to the semantic representation language. This allows Web service developers to annotate their Web services with their choice of modeling language (such as OWL, or legacy models developed in UML or other knowledge representation languages).
WSDL-S focuses on semantically annotating the abstract definition model of a service in WSDL specification to enable dynamic discovery services. In essence, it provides URI reference mechanisms via extensibility elements to the interface, operation and message constructs to point to the semantic annotations defined in the externalized domain models for services. The provided elements of extensibility are:

- an extension element, namely wssem:modelReference, to allow for one-to-one associations of WSDL input and output type schema elements to the concepts in a semantic model

- an extension attribute, namely wssem:schemaMapping, to allow for many-to-many associations of WSDL input and output type schema elements to the concepts in a semantic model - typically associated with XML schema complex types

- two new elements, namely wssem:precondition and wssem:effect, which are specified as child elements of the operation element and describe the semantics of the operation along the lines of the OWL-S approach. Preconditions and effects are primarily used in service discovery, and are not required to invoke a given service, and

- an extension attribute on interface element, namely wssem:serviceCategorization. It consists of service categorization information that could be used when publishing a service in a Web Services registry such as UDDI. It corresponds to the categorization concept proposed in OWL-S.

### 7.2.4 OWL-S

OWL-S \[11\] is an ontology designed for describing and reasoning over service descriptions of service concepts expressed in OWL-DL, a decidable description logic language. OWL-S combines the expressivity of description logics and the pragmatism found in the emerging Web Services Standards, to describe services that can be expressed semantically, and yet grounded within a well defined data typing formalism. It consists of three main upper ontologies: the Profile, Process Model and Grounding. OWL-S process models describe the composition or orchestration of one or more services in terms of their constituent processes.

The profile and process models provide semantics frameworks whereby services can be discovered and invoked, based upon conceptual descriptions defined within Semantic Web ontologies. The grounding provides a pragmatic binding between this concept space and the physical/machine/port space, thus facilitating service execution.

Although the process model of OWL-S follows the same intention of describing both the client-service interaction for consuming a Web Service and the aggregation of Web Services \[11\], the distinction between choreography and orchestration is not explicitly defined. Also, the process representation language seems not to be adequate with respect to sophisticated reasoning support and is not based on any formal model (although semantics have been provided using Petri-Nets \[50\]).
7.2.5 SWS Framework

SWS Framework aims to enable fuller, more flexible automation of service provision and use, support the construction of more powerful tools and methodologies, and promote the use of semantically well-founded reasoning about services. The Semantic Web Services Framework (SWSF) includes the The Semantic Web Services Language (SWSL) and The Semantic Web Services Ontology (SWSO).

SWSL is used to specify formal characterizations of Web service concepts and descriptions of individual services. It includes two sub-languages. **SWSL-FOL** is based on first-order logic (FOL) and is used primarily to express the formal characterization (ontology) of Web service concepts. **SWSL-Rules** is based on the logic-programming (or “rules”) paradigm and is used to support the use of the service ontology in reasoning and execution environments based on that paradigm. SWSL is a general-purpose language (that is, its features are not service-specific), but it has been designed to address the needs of Semantic Web Services. Also associated with SWSL is a simplified presentation syntax, which reduces to SWSL-FOL.

SWSO presents a conceptual model by which Web services can be described, and an axiomatization, or formal characterization, of that model. The complete axiomatization is given in first-order logic, using SWSL-FOL, with a model-theoretic semantics that specifies the precise meaning of the concepts. We call this FOL form of the ontology **FLOWS** – First-Order Logic Ontology for Web Services. In addition, the axioms from FLOWS have been systematically translated into the SWSL-Rules language (with an unavoidable weakening of some axioms). The resulting ontology, which relies on logic-programming semantics, is called ROWS – Rules Ontology for Web Services.

A key contribution of the FLOWS ontology is the development of a rich behavioural process model, based on ISO 18629 Process Specification Language (PSL) [1]. Originally designed to support interoperability among process modeling languages, PSL provides the ideal foundation for interoperability among emerging Web service process models, while supporting the realization of automation task associated with the Semantic Web Service vision. FLOWS goes beyond PSL in modeling many Web-service specific process concepts including messages, channels, inputs and outputs.

FLOWS is designed modularly. It comprises an ontology for service descriptors, somewhat akin to a domain-independent yellow-pages or OWL-S service profile, an extensive process model ontology, and a grounding that relates the process model message types to WSDL messages. The process model ontology is in turn comprised of a core ontology and a number of extensions. The statements above about FLOWS are also true of ROWS, which is derived from FLOWS.

7.3 Formalisms for Process and Dynamics Representation

This section provides an overview of the most common formal models used for representing the dynamics of processes. The models described here are Process algebra Situation Calculus, Petri-Nets, and Concurrent Transaction Logic. The Abstract State Machine also could be considered a formalism for dynamic process representation. However, an exhaustive ASM description is provided in section [1].
7.3.1 Process Algebras

As outlined above, we are mostly interested in formalisms for representing the process of interactions between parties. More precisely, we need a formalism that allows the description of the structure of the processes that parties perform when they participate in an interaction. These formalisms will serve as the formal basis for describing processes and defining mappings for the mediation of possibly heterogeneous processes. Process Algebras (PA) provide this type of process representation formalism. A PA is a formal description technique for complex computer systems that pays special attention to concurrently executing components that interact in parallel and distributed systems. The objective of PA is to allow the observation of the behaviour of a system or its components. The approach is the definition of a formal language for the constituting elements of the processes and the performance of algebraic calculations on the basis of these process descriptions [5]. Compared to the general idea of mediation facilities outlined in the introduction, PA seems to be the appropriate choice for process level mediation within Semantic Web Services. The formal description language adds formal semantics to process descriptions, and the algebraic calculations existing for process algebras can serve as a basis for the development of inference-based mediation facilities for processes. Also, currently existing approaches for formalization of processes within Choreography and Orchestration apply PAs, as investigated in more detail below. Research within PA started in the 1970s, touching many topic areas of computer science and discrete maths, including system design notations, logic, concurrency theory, specification and verification, operational semantics, algorithms, complexity theory, and, of course, algebra. The very early works on PA developed the automata theory, which identified states and state changes for modeling a process and was concerned with formally describing the execution of process, a so-called run. Soon, the notion of interaction was added, and the attention of PA research turned to behaviour observation within the interactions of process-driven systems of components [3]. The main approaches developed within PA are the Calculus of Communicating Systems (CCS) [46], Communicating Sequential Processes (CSP) [36] and the Algebra of Communicating Processes (ACP) [4].

CCS is mainly the work of Robin Milner, which developed over time. The main focus of CCS is to formalize behaviours and determine equivalence between these, which is basically the same objective as followed in process level mediation. CCS relies on a synchronization tree as the underlying model for representing processes: a node represents a process activity and an arch is a transition; arches are equipped with so-called laws that specify the conditions or validity constraints for a transition. Based on this, CCS provides an algebraic model for process equivalence. A newer approach based on CCS is \(\Pi\)-Calculus, which adds notions for handling process interactions dynamically [17]. In contrast to CSS, CSP builds on the message passing paradigm of communication as a contrary approach than describing processes individually along with the notion of process equivalence as in CSS. Besides the underlying models for process representation, CCS and CSP have developed techniques to handle process identification, failures and deadlocks, and inference-based determination of equivalences of processes. The development of CCS and CSP are very interwoven, and most modern approaches combine the concepts of both. An exhaustive comparison CSS and CSP is provided in [27].
7.3.2 Situation Calculus

The term situation calculus, initially mentioned in [42], is used for a variety of formalisms treating situations as objects, considering fluents that take values in situations, and events (including actions) that generate new situations from old. The situation calculus language mostly used as defined in [51] is a first-order logical language for reasoning about actions, based on Predicate Calculus. The aim is to represent dynamically changing worlds in which all of the changes are the direct result of named actions performed by some agent. Situations are sequences of actions, evolving from an initial distinguished situation, designated by the constant $S_0$. If $a(y)$ is an parameterized action and $s$, a situation, the result of performing $a$ in $s$ is the situation represented by the function $do(a, s)$. Functions and relations whose values vary from situation to situation, are called fluents, and are denoted by a predicate symbol taking a situation term as the last argument (for example $Own(bookName, s)$). Finally, $Poss(a, s)$ is a distinguished fluent expressing that action $a$ is possible to perform in situation $s$. The general problem within situation calculus is that there are several other problems that have to be expressed explicitly, in order to achieve the correct semantics of the clipping of the world one wants to formalize:

- **Quantification Problem**: is concerned with the executability of an action in specific situations. Usually, $Poss(a, s)$ means that activity $a$ can be executed in situation $s$. The problem is that it is nearly impossible to determine all situations in which an activity can be executed. In situation calculus, this has to be specified explicitly, narrowing the applicability of this formalism to small and closed worlds.

- **Frame Problem**: adhering to the general law of inertia, it has to be defined which information (that is the fluents in situation calculus) remains unaffected by the execution of an activity. This is called the frame problem, wherein a frame is understood as the set of information items that represent a state. Similar to the Quantification Problem, all fluents that are not affected by executing an activity have to be specified explicitly.

7.3.3 Petri-Nets

This section provides an overview of classical Petri-Nets [50] with a description of High-Level Petri-Nets as defined in [64].

Petri-Nets are graphs with two types of nodes called *places* and *transitions*. Nodes are connected via directed *arcs* and connections between two nodes of the same type are not allowed. Graphically, places are characterized by circles and transitions by rectangles. Formally speaking, a Petri Net is a triple $(P, T, F)$ such that:

- $P$ is a finite set of places
- $T$ is a finite set of transitions ($P \cap T = \emptyset$)
- $F \subseteq (P \times T) \cup (T \times P)$ is a set of arcs (flow relation)

A place $p$ is an input place of a transition $t$ iff there exists a directed arc from $p$ to $t$. Similarly, a place $p$ is called an output place of transition $t$ iff there exists
a directed arc from \( t \) to \( p \). Arcs can have a weight associated with them, however, in the context of workflow patterns, a weight of 1 is always assumed since places correspond to conditions [64][66]. At any given time, a place can contain zero or more tokens (graphically denoted using black dots). The state of a Petri-Net (or marking) is defined as the distribution of tokens over places. For example, \( 2p_1 + 1p_2 + 0p_3 \) implies that there are 2 tokens in \( p_1 \), 1 in \( p_2 \) and 0 in \( p_3 \). Comparisons of states are also allowed using a partial ordering such that for any two states \( S_1 \) and \( S_2 \), \( S_1 \preceq S_2 \) iff \( \forall p \in P : S_1(p) \leq S_2(p) \). During the execution of a net, the number of tokens can thus change. The firing rules of transitions are defined as follows:

1. A transition \( t \) is said to be enabled iff each input place \( p \) of \( t \) contains at least one token

2. Transitions are allowed to fire only if they are enabled. That is, all the preconditions of the places must be fulfilled (there are enough tokens present in the input place). If transition \( t \) fires, then \( t \) consumes one token from each input place \( p \) of \( t \) and produces one token in each output place \( p \) of \( t \).

For further details about the semantics and formal definitions of classical Petri-Nets, the reader is referred to [50]. For the scope of this document, it is sufficient for the reader to have a general knowledge about such a notation. However, it worthwhile to briefly describe the extensions to classical Petri-Nets since the major work done in workflow patterns relates to such extensions [66][65].

Classical Petri-Nets allow the modelling of events, states, conditions, synchronization, parallelism, choice and iteration. However, support for modelling data and time is not provided. Several extensions of the classical Petri-Nets have been proposed to address such problems:

1. Colored Petri-Nets to allow data modelling

2. Timed Petri-Nets to allow the modelling of time constraints

3. Hierarchical Petri-Nets to allow modelling of complex structures

The tokens consumed and produced by the transitions often represent objects of the real world. For example, a booking request in the case of a holiday planning service. Furthermore, such requests may define extra attributes like departure and arrival dates, hotel preferences etc. Such attributes are hard to model using classical Petri-Nets and hence colored tokens may be used to differentiate between the different attributes. It is also possible to specify the preconditions of the transition rules that take into account the color of the token. In principle, the transition would describe the relation between the input tokens and the output tokens. Further details about Colored Petri-Nets can be found at [38]. Timed Petri-Nets can be modelled in different ways. Time constraints can be specified on the net’s different elements, that is, places, conditions and/or transitions [63]. Such types of Petri-Nets are particularly useful to model real time systems and also business processes which need some form of timing restrictions. In Hierarchical Petri-Nets, the notion of sub-nets is used [67]. Such a net is composed of other places, transitions and other sub-nets. Such an extension provides an easy way to describe large and complex systems. A Petri-Net which combines all of these extensions is called a High-Level Petri-Net.
7.3.4 Concurrent Transaction Logic

Concurrent Transaction Logic (CTR) \cite{8} is based on Transaction Logic (TR) \cite{9}. TR is an extension of classical predicate logic which provides a logical foundation for state changes in databases and logic programs. The logic programming fragment of TR provides a logical environment for programming evolutions of databases. TR comes with a reasoning system, in which one can reason about general statements and program and execute database evolutions.

Several extensions to TR have been developed; they differ in the operators for composing atomic operations into complex operations (i.e. they have different expressive power and computational complexity of reasoning tasks). Sequential Transaction Logic \cite{9} uses the usual classical operators (not, and, or, implication) and two additional operators: serial conjunction, and a modal operator for executional possibility for composing atomic formulas into complex ones. Concurrent Transaction Logic (CTR) \cite{8} additionally enables concurrent compositions of operations; it extends TR with two operators: concurrent composition, and a modal operator of isolation. Transaction Datalog (TD) \cite{7} is a reduction of the Horn-fragment of CTR, just as Datalog is a reduction of the Horn-fragment of classical logic. TD is equivalent to Horn-CTR without negation, and where all the rules are safe.

In \cite{39} and \cite{7} is shown how CTR and can be used for modelling workflows: a workflow is specified as a set of CTR formulas that describe dependencies between tasks; tasks are modelled using predicates, they can be either atomic or be defined as a subprocedure (this allows for compositional workflow modelling). Simple sequential and concurrent workflow can be defined using the basic CTR operators for sequential and concurrent composition. Choice in a workflow (or preconditions for tasks) can be modelled by sequential composition of a query and a task (if the query succeeds the task is executed). Synchronization of tasks can also be achieved by communicating through the database. Although TR and its extensions can not represent directly all the control flow patterns (actually TR and its extensions support directly only a small subset), most of all patterns, if not all, could be modelled on top of basic CTR constructs. In \cite{65} is argued that potentially other workflow language could encode (by combining their primitive constructs) the patterns, but such an approach is unacceptable from the workflow designer perspective, and the creation of a language to directly support these patterns is justifiable. However CTR is a formal language, and such a mapping of the patterns to CTR is justifiable.

Moreover, it is the verification and scheduling features of CTR that is interesting for us. In \cite{17}, an extension of CTR is presented that allows efficient reasoning about workflow executions with constraints. A procedure to compile certain constraints into the workflow specification during design-time is presented (the procedure checks whether every legal execution of a workflow satisfies a particular property, whether the workflow is consistent with the constraints, and whether the constraints are consistent with each others). The result of the compilation step is a workflow definition in which these constraints are guaranteed to be satisfied and do not need to be checked anymore during runtime. Although the compilation step is computationally expensive it can be performed before execution of the workflow, during design-time; the run-time scheduling of the workflow can then be done efficiently. After compiling the conjunctive

---

1Actually by performing such a translation, a new semantics, in terms of CTR, is defined for YAWL.
constraints into the workflow, the time complexity of scheduling is linear in the size of
the original control flow graph. Thus, the procedure facilitates verification of workflows
and optimizes run-time scheduling. CTR-based modelling and reasoning goes even
further in CCTR [55](where a logical framework for scheduling workflow under resource
allocation constraints, based on CTR and Constraint Logic Programming is given),
and in CTR-S [18](where a new formalism for modelling and reasoning about the
dynamics of service contracts is given). However, since it is unclear how to generalize
constraints to loops, the results in [17] (and also in [18]) are obtained by restricting
workflows to workflows without loops. Moreover, even with an extension that would
incorporate loops, it is unclear how CTR’s analysis techniques should be changed when
support for directly representing all the control-flow patterns is needed.

It can be noticed that CTR is a very attractive framework for modelling, verifying
and scheduling workflows, on top of which languages that implement the control flow
patterns, like YAWL, should be grounded. However, a mapping of YAWL constructs to
CTR is needed, and CTR verification and scheduling techniques need to be extended in
order to accommodate the complexity introduced by the direct support of the control-
flow patterns.
7.4 Existing work on business process and protocol languages

This section aims to provide an introduction to business process and protocol ontology, to review the existing work and to clarify several aspects and concepts that are part of it. This introduction is necessary before advancing further in building the business process and protocol ontology that is the aim of the current deliverable.

7.4.1 Definition of Business Process

A business process (from the business perspective) can be defined as a logic and successive group of activities undertaken by an organization in pursuit of achieving its business goals of profit maximising. Typical business processes include marketing services, sending and receiving request for quotations, negotiation activities, receiving orders, manufacturing activities or service providing activities, goods or services delivery, accounting activities, logistics and procurement activities and post-sales services. A business process has to rely on various support infrastructures, e.g. IT, Human Resources, logistics etc. A business process rarely operates in isolation, i.e., other business processes will depend on it and it will depend on other processes. From IT perspective (and from web services perspective in particular), a business process is a partially/totally automated and coordinated sum of business activities, carried out by automated software and independent web services. A business process itself may be designed and implemented as a web service and be used by other business processes.

Therefore a distinction has to be made between business and IT perspectives over the definition of business process. They are different due to their different approaches but they are complementary. The business perspective approaches business processes from the strategy, business execution and logistics while the IT perspective approaches business processes from the software automation point of view. However to automate business processes needs to understand in detail the business execution features and perspective.

7.4.2 Definition of Business Protocol

A business protocol is a specification (or a suite of specifications) that enables the exchange of business messages, communicating data in common terms and defining and registering business processes. ebXML Glossary [21] defines a Collaboration Protocol as the protocol that defines the following:

- The sequence, dependencies and semantics of the Documents that are exchanged between Parties in order to carry out a Collaborative Process.
- The Messaging Capabilities used when sending documents between those Parties.

In other words, a business protocol is a set of agreements facilitating the exchange of messages whose structure is specified by the protocol but whose payloads are business-process-specific.
7.4.3 Definition of Business Process and Protocol Ontology

Business process description languages like Business Process Execution Language for Web Services (WS-BPEL), Business Process Management Language (BPML) and Web Services Choreography Interface (WSCI) are languages for modeling business processes. The protocol is implied to be SOAP [32] with extensions like WS-Coordination [14] and WS-Transaction [13].

Business process ontologies are similar to business process description languages, differing only in the means of specifying the language - ontologies use ontology modeling languages to be able to specify more of the semantics of the language in a computer-understandable form. In other words, business process ontologies and business process languages express the same concepts but use different formalizations.

7.4.4 WS-BPEL

The Business Process Execution Language for Web Services (BPEL4WS) has its origins in Microsoft XLANG and IBM WSFL. BPEL4WS is the basis for OASIS WS-BPEL [49]. It represents an important initiative towards the generation of workflow specifications that try to model the behaviour of web services taking part in a business process interaction. It describes compositions where the flow of the process and the bindings between services are known a priori. WS-BPEL uses an XML-based grammar for describing the coordination logics of web services participating in a process flow. The generated logic is interpreted and executed by an orchestration engine controlled by one of the parties.

WS-BPEL presents a process model layered on top of WSDL [15], where both the process model and its partners are represented as services using WSDL. WS-BPEL differentiates among two different kinds of business processes:

- executable processes that permit to model the behaviour of a participant in an interaction, without making any separation between public and private details of the business process;

- abstract processes that are involved in a business protocol, modeling the visible behaviour of each of the involved parties based on its message exchange, without revealing its internal details; typically not executable, meant to couple web service interface definitions with behavioural specifications.

Executable processes facilitate orchestration and abstract processes facilitate choreography.

The WS-BPEL specification differentiates two types of activities - basic and structured. Basic activities represent the simplest form of interaction with a service, they describe individual operations that manipulate passing data or handle exceptions. In contrast, structured activities specify the order in which a collection of basic activities takes place. Structured activities involve the necessary programmatic conventions (sequence, condition, repetition etc.) to specify the execution flow for their sub-activities. Structured activities are in fact the underlying programming logic for WS-BPEL. Activities also represent the basic unit that can be grouped within a transaction by means of scopes. Scopes have a 1 to 1 mapping with database transactions in the sense that all activities enclosed within a concrete scope should either all complete or all fail, in
In a nutshell, WS-BPEL represents an initiative towards the generation of workflow specifications using an XML-based grammar in a web service environment, where the flow of the process and the bindings between services are known a priori. It counts with transaction and exception handling support, and uses the concept of scope as the grouping unit of activities that take part in the same transaction. It represents an alternative to the combination of BPML and WSCI, and just like them suffers of the lack of semantics, making it unsuitable for providing web service automation facilities, and in particular automatic composition.

7.4.5 BPML/WSCI

The Business Process Management Language (BPML [12]) is a meta-language that allows to model business processes in terms of control flow, data flow and event flow, for its later execution using web services.

The Web Services Choreography Interface (WSCI [31]) specification defines an XML-based language for describing web services collaboration. It describes the overall choreography in terms of messages exchanged between web services that participate in a business process. WSCI defines the public (observable) interaction in terms of messages, not paying any attention to the definition of executable business processes (control flow, data flow and event flow).

BPML and WSCI work in combination and actually are complementary efforts. While WSCI describes public interactions and choreographies between services, BPML focuses on describing the private part. Both efforts share the same execution model and their syntaxes are similar. Roughly speaking BPML takes care of the definition of the business processes that will be executed using web services, while WSCI specifies the interoperation between such services.

BPML

BPML is a meta-language for modeling collaborative and transactional business processes, based on the concept of transactional finite-state machine, which allows the definition of abstract and executable processes using a formal model. It uses XML Schema [60, 6] to describe the grammar that enables the persistence and interchange of definitions across heterogeneous systems. BPML represents business processes based on control flow, data flow, and event flow, adding orthogonal design capabilities for business rules, security roles, and transaction contexts independently of the synchronous or asynchronous nature of the transaction.

In order to define business processes BPML uses the following five elements:

- activities are components that perform specific functions, can be either simple or complex.
- processes are particular types of complex activities that define their own context.
- contexts define the environment for the execution of related activities.
- properties enable the information exchange within a context.
- signals allow the coordination of activities that are being executed in the same context.
BPML uses a modeling language to define business processes in a language independent fashion, placing such description on top of the BPML XML Schema layer. It provides primitives for sending, receiving, and invoking services, together with the typical programmatic conventions to handle conditional choices, sequential and parallel activities, joins, and looping. On top of this it provides constructs for task scheduling and XML exchanges between the participants. The language was designed to manage long-running processes, supporting persistence in a transparent manner. Finally the language supports recursive composition in order to facilitate the means to compose sub-processes into larger business processes.

Regarding transactional support and exception handling mechanisms, it is worth to mention that BPML sustains both characteristics, same for short and long running transactions, using scoping-based compensation techniques to compensate processes within a defined scope. In the case of complex transactions, BPML also supports process and transaction nesting, exception handling, and timeout constraints for activities defined within the process.

In a nutshell, BPML provides means to define business processes involving different parties and web services, with support for features such as persistence, compensation or exception handling. This description covers the definition of distributed business processes, including how the different partners and services collaborate. Nevertheless, BPML does not cover automation support, as the business process and the partners and services involved have to be specified at design time and cannot change dynamically, mainly due to the lack of explicit semantics.

WSCI
WSCI (pronounced whisky) is an XML-based interface description language that defines, using messages, the overall choreography - temporal and/or logical dependencies - of a web service taking part in an interaction. WSCI defines the public (observable) interaction in terms of messages, not paying any attention to the definition of executable business processes.

There is a direct correspondence between WSCI and WSDL. Each WSCI unit of work corresponds to a specific WSDL operation, i.e., WSCI specifies the choreography among operations.

In order to provide its functionality, WSCI uses the following concepts:

- interface models the externally observable behaviour of a web service in a choreographed, long-lasting and stateful message exchange with one or more other services

- activity is the basic unit of behaviour of choreographed Web Services; they can be either atomic or complex, composed of other activities

- process is an identifiable portion of behaviour; WSCI differentiates two types of processes, top-level processes, defined at the interface level, and nested processes defined within complex activities

- properties are used to reference values within interface definitions

- context is an environment in which a set of activities is executed; there exists a 1 to 1 mapping between sets of activities and contexts
• message correlation permits to define how the service should manage multiple conversations with the same or different partners; it describes how conversations are structured and which properties must be exchanged to retain the semantic consistency of the conversation

• exceptional behaviour defines alternative patterns of behaviour exhibited by a web service at a given point in a choreography; it associates exceptions and activities that the web service will perform in response to those exceptions

• transactional behaviour defines the operations that should be performed in a transactional way within a particular context

• global model allows describing a multi-participant view of the overall message exchange in contrast to the interface that only describes one participant

WSCI provides an answer to different issues related with the interoperation of services such as message sequencing, relation between incoming and/or outgoing messages or start and end of sequences among others. Essentially, WSCI is used to describe the dynamic interface of the web service participating in a given message exchange and models the observable behaviour of a web service, or the set of interacting web services in terms of temporal and logical dependencies among the exchanged messages. WSCI does not address the definition and implementation of the internal processes that actually drive the message exchange.
8 CONCLUSIONS AND FUTURE WORK

Concluding the specification of the description ontology and languages for choreography and orchestration interfaces in DIP, the following summarizes the document and points out directions for future development of the “DIP Interface Description Ontology”.

8.1 Summary

In this document we have in detail defined the meta-model layer ontology along with specification languages for describing behavioral interfaces of Web services in DIP, namely for choreography interfaces and orchestrations. Although these are used for different application purposes as described in the respective deliverables D3.5 “An Ontology for Web Service Choreography” and D3.4 “An Orchestration and Business Process Ontology”, they can be described by the same description language.

This common description language for behavioral interfaces of Semantic Web services has been specified in this document. The meta-model layer structure - referred to as the meta-model ontology - defines that a Web service description can have one or interfaces. An interface is comprised of an choreography description as the behavior interface for client-service interaction for service consumption, and an orchestration description that defines how a Web service interacts with other Web services in order to achieve its functionality.

Extending the specification of WSMO interfaces with respect to five common requirements identified for choreography and orchestration descriptions, we have defined a 2-layered description language for behavior interfaces in DIP. These are a higher-level user language on basis of UML2 Activity Diagrams and a lower-level formal description called ontologized Abstract State Machines. Apart from specifying and rationalizing these two languages in detail, we have presented a generic algorithm for translating down from the user language to the formal model. Furthermore, we have provided the specification of a grounding for the formal model to WSDL as the concurrent quasi-standard for communication and information interchange between Web services.

In order to clarify the rationale and novelties invented within our approach, we have exhaustively discussed related work and positioned our approach therein.

8.2 Future Work

Although the specified description ontology and languages provide a suitable basis for the intended tooling support and application requirements for behavioral interface definitions within DIP, we are aware of the following shortcomings:

- although the formal model based on Abstract State Machines allows the representation of all computable tasks, it does not support representation of workflow constructs as first class entities;

- similarly, although many workflow patterns are representable in UML Activity Diagrams, these are not unambiguously represented, nor are they necessarily cleanly nested, but may overlap leading to further ambiguity of the intended underlying workflow;
• although Abstract State Machines allow algebraic reasoning to be carried out on the data domain, only refinement-oriented reasoning is provided over behaviour;

• neither the UML nor Abstract State Machines has an equivalence/refinement theory based on the notion of observables, i.e. abstracting from internal behaviour, which is the necessary for a formal notion of conformance.

With respect to this, we plan to enhance the DIP Interface Description Ontology as specified in this document by integrating the Cashew model, an ongoing research project which aims to define a formal model for workflows that can represent both orchestrations and choreographies. For orchestrations it takes the approach of aligning the existing OWL-S Process Model [41] with Workflow Patterns [62], being complete with respect to the former except for dropping the constructor that falls within the ‘multiple instance’ workflow patterns, since it is unclear how to represent these in Abstract State Machines.

In order to deal also with choreography, Cashew also integrates the idea from Workflow Patterns of the separation between ‘internal’ and ‘external’ choice, represented as the ‘XOR’ and ‘Deferred Choice’ patterns. It is an important distinction for implementation whether an orchestration engine resolves a choice, or leaves the choice to an external party; for instance a component service. It is further claimed that it is important to formalise this distinction in order then to formalise the notion of ‘conformance’ between an orchestration and choreography, or between a service and client choreography.

Due to the representability of workflow patterns in Activity Diagrams, it is possible to translate any Cashew workflow into an Activity Diagram, for human inspection or automatic treatment in existing tools for UML. Cashew also defines a formal semantics for its workflows as Abstract State Machines via process algebra, by extension of existing work on OWL-S [48], which translation is both compositional and, via a complete axiomatisation, allows algebraic reasoning.
Figure 8.1: Future DIP Interface Description Ontology
REFERENCES


APPENDIX

The following provides the ontology schema definition of the UML2 Activity Diagram Ontology as specified in section 3 of this document. In order to allow UML2 specifications to be manageable by DIP ontology tools, this is specified in the Web Service Modeling Language WSML [19], which is the ontology specification language for DIP.

The ontology is modeled in the variant WSML Rule. Remarks on WSML modeling:

- cardinality:
  - denoted by (a b), a = min, b = max
  - (0 *) is default in WSML

- WSML attribute value definition:
  - "impliesType" means that attribute value type is inferred from the attribute definition
  - "ofType": attribute value type must be known and explicitly checked
  - for both: attribute value type is inherited to sub-concepts

- "!-" is a shortcut for "false impliedBy", used for constraint definition

```
wsmlVariant _"http://www.wsmo.org/wsml/wsml-syntax/wsml-rule"
namespace {"http://dip.semanticweb.org/ontologies/uao#",
wsml _"http://www.wsmo.org/wsml/wsml-syntax#",
dc _"http://purl.org/dc/elements/1.1#"
}
ontology _"http://dip.semanticweb.org/ontologies/uao"

nonFunctionalProperties
dc#title hasValue "DIP UML Activity Diagrams Ontology for Choreography Descriptions"
dc#description hasValue "meta-model for choreography interface description in DIP (UML2 Activity Subset in WSML)"
dc#subject hasValue {"Web service", "interface", "choreography", "description ontology"}
dc#publisher hasValue "DIP Consortium"
dc#creator hasValue {"Michael Stollberg"}
dc#contributor hasValue {"Laurent Henoque, Mathias Kleiner"}
dc#language hasValue "en-US"
dc#language hasValue "en-US"
dc#date hasValue "Date : 2005/12/28 17 : 31 : 30"
wsml#version hasValue "Revision : 1.6"
endNonFunctionalProperties

// GENERAL UML2 NOTIONS
concept object
nonFunctionalProperties
dc#description hasValue "an object"
endNonFunctionalProperties

// ACTIVITY GROUPS
concept activityGroup subConceptOf object
nonFunctionalProperties
dc#description hasValue "group of immediately contained nodes and edges"
endNonFunctionalProperties
node ofType activityNode
edge ofType activityEdge
```
concept interruptibleActivityRegion subConceptOf activityGroup
  nonFunctionalProperties
    dc#description hasValue "if interruption occurs: terminating all tokens and behaviors connected to
    activity nodes in the region"
  endNonFunctionalProperties
  interruptingEdge impliesType ActivityEdge

// ACTIVITY EDGES
concept activityEdge subConceptOf object
  nonFunctionalProperties
    dc#description hasValue "Activity Edge"
  endNonFunctionalProperties
  Guard impliesType (1 1) wsml#axiom

concept objectFlow subConceptOf activityEdge
  nonFunctionalProperties
    dc#description hasValue "connects exclusively object nodes to the exception of some control nodes (decision & merge)"
  endNonFunctionalProperties

concept controlFlow subConceptOf activityEdge
  nonFunctionalProperties
    dc#description hasValue "control flow edge"
  endNonFunctionalProperties

// ACTIVITY NODE
concept activityNode subConceptOf object
  nonFunctionalProperties
    dc#description hasValue "group of immediately contained nodes and edges"
  endNonFunctionalProperties
  incomingEdge ofType activityEdge
  outgoingEdge ofType activityEdge

// ACTION NODES
concept actionNode subConceptOf activityNode
  nonFunctionalProperties
    dc#description hasValue "denotes an action"
  endNonFunctionalProperties
  inputPins ofType inputPin
  outputPins ofType outputPin

concept wsmoOOMediator subConceptOf actionNode
  nonFunctionalProperties
    dc#description hasValue "denotes a WSMO OO Mediator"
    dc#relation hasValue wsml#oomediator
  endNonFunctionalProperties

concept event subConceptOf actionNode
  nonFunctionalProperties
    dc#description hasValue "an event / interaction with a partner"
  endNonFunctionalProperties
  Partner ofType string
  LinkedEvent ofType string

concept acceptEvent subConceptOf event
  nonFunctionalProperties
    dc#description hasValue "an event accepted by the entity"
  endNonFunctionalProperties

concept sendEvent subConceptOf event
  nonFunctionalProperties
    dc#description hasValue "an event send by the entity"
  endNonFunctionalProperties

// OBJECT NODES
concept objectNode subConceptOf activityNode
  nonFunctionalProperties
    dc#description hasValue "denotes an object"
  endNonFunctionalProperties
  hasConcept impliesType (1 1) wsml#concept
// PINS
concept pin subConceptOf objectNode
nonFunctionalProperties
dc#description hasValue "general node for inputs / outputs of an action"
endNonFunctionalProperties
node ofType (1 1) activityNode
valueRestriction impliesType wsml#axiom

concept inputPin subConceptOf pin
nonFunctionalProperties
dc#description hasValue "input of an action"
endNonFunctionalProperties

concept outputPin subConceptOf pin
nonFunctionalProperties
dc#description hasValue "output of an action"
endNonFunctionalProperties

// Ontologies
concept domainOntology subConceptOf object
nonFunctionalProperties
dc#description hasValue "domain ontology"
endNonFunctionalProperties

concept dataConcept subConceptOf object
nonFunctionalProperties
dc#description hasValue "data concept"
endNonFunctionalProperties
hasOntology impliesType (1 1) domainOntology

concept atomic subConceptOf dataConcept
nonFunctionalProperties
dc#description hasValue "Composite concept"
endNonFunctionalProperties

concept composite subConceptOf dataConcept
nonFunctionalProperties
dc#description hasValue "Composite concept"
endNonFunctionalProperties
contains ofType atomic

concept bag subConceptOf composite
nonFunctionalProperties
dc#description hasValue "multiple object in an object node organized as a bag"
endNonFunctionalProperties

concept list subConceptOf composite
nonFunctionalProperties
dc#description hasValue "multiple object in an object node organized as a bag"
endNonFunctionalProperties

concept set subConceptOf composite
nonFunctionalProperties
dc#description hasValue "multiple object in an object node organized as a set"
endNonFunctionalProperties

// CONTROL NODES
concept controlNode subConceptOf activityNode
nonFunctionalProperties
dc#description hasValue "workflow control nodes"
endNonFunctionalProperties

concept initialNode subConceptOf controlNode
nonFunctionalProperties
dc#description hasValue "where workflow starts"
endNonFunctionalProperties

concept finalNode subConceptOf controlNode
nonFunctionalProperties
dc#description hasValue "general node where workflow terminates"
endNonFunctionalProperties
concept activityFinal subConceptOf finalNode
nonFunctionalProperties
dc#description hasValue "where workflow ends"
endNonFunctionalProperties

concept flowFinal subConceptOf finalNode
nonFunctionalProperties
dc#description hasValue "where workflow ends"
endNonFunctionalProperties

concept abstractSplitNode subConceptOf controlNode
nonFunctionalProperties
dc#description hasValue "abstract node for splits"
incomingEdge ofType (1 1) activityEdge

concept decisionNode subConceptOf abstractSplitNode
nonFunctionalProperties
dc#description hasValue "node for splits by a decision"
endNonFunctionalProperties

concept forkNode subConceptOf abstractSplitNode
nonFunctionalProperties
dc#description hasValue "node for splits by a fork"
endNonFunctionalProperties

concept abstractJoinNode subConceptOf controlNode
nonFunctionalProperties
dc#description hasValue "abstract node for joins"
outgoingEdge ofType (1 1) activityEdge

concept mergeNode subConceptOf abstractJoinNode
nonFunctionalProperties
dc#description hasValue "join by merging"
endNonFunctionalProperties

concept joinNode subConceptOf abstractJoinNode
nonFunctionalProperties
dc#description hasValue "joining"
endNonFunctionalProperties