Deliverable

WP 5: Service Mediation
D5.3a
Business Process-level Mediation Module Specification

Emilia Cimpian
Jens Lemcke
Adrian Mocan
Michael Schumacher

June 29th, 2005
EXECUTIVE SUMMARY

The Business Process Mediation Module Specification deliverable addresses the process heterogeneity problem and proposes solutions in overcoming this problem. The processes addressed in this deliverable are the public processes of Semantic Web services, which define the behaviour of a participant in a conversation.

For solving the heterogeneity problem, we followed the approach initially proposed in DIP deliverable 5.1 (Report on the State-of-the-Art and Requirements Analysis) [8], and we took full advantage of the assumption that both the requestor and the provider of the service use ontologies modelled conforming to the Web Service Modeling Ontology (WSMO) specifications; additionally, we consider that the two participants in the conversation use choreographies for specifying their public behaviours (both WSMO ontologies and choreographies are described in DIP deliverable 3.2 Service Description Framework [4]; more up-to-date information can be found in [20] and [21]).

The document proposes a way to solve the heterogeneity problem, which addresses discrepancies in the interaction between the requestor and the provider of a service. The mediation of behaviours may be needed when a requester of a service attempts to invoke and execute the corresponding Web service. The communication pattern of the Web service may be different from the one the requester expects, in which case one of them has to adjust to the other’s communication pattern – meaning that it has to change its process execution in order to match the other parties’ specifications. The adjustment of the different patterns in order to make them match is called process mediation, and this adjustment happens in neither of the involved parties’ sides, but in the Process Mediator. That is, each time a message is sent by one of the two parties involved in the conversation, the Process Mediator has to determine if the message is expected by the other party. The mediator also has to consider situations when only part of a message or a combination of this message with a previously received one is expected, which is done by analyzing the choreographies of the parties. The deliverable presents an overview of the addressed mismatches, and the way the Process Mediator is able to automatically solve these problems, strictly by analyzing the choreographies of the two involved parties. The interfaces that the Process Mediator should implement, as well as the interfaces of the Process Mediator’s subcomponents are also presented.

This work took place as a joint effort of DIP project and Lion2 project; the authors of this document are involved in the Lion project, and this work was also provided as part of Lion D5.1b Process Mediation – Concepts, Architecture and Implementation in WSMX [10].

This deliverable will represent the starting point in the development of a Process Mediator prototype, which will be part of the DIP deliverable 5.5 Mediation Module Prototype. The interfaces for this prototype already exists as part of the overall DIP architecture (the focus of WP6 in DIP), which is a Service Oriented Architecture for

---


2 DERI-Lion is a joint effort of DERI and of Hewlett-Packard Galway; the main objective of this project is to develop a scalable semantics-based Web Service environment where business partners can seamlessly share services across dynamic business communities; more information about Lion project are available at: http://lion.deri.ie/.
allowing the dynamic discovery, selection, mediation and invocation of semantic Web services. Process mediation is a must for allowing interoperability of two Web services, or of a Web service with a requestor of a service.

The document is addressed to researchers working in the field of Semantic Web services integration and interoperation, both from the design and from the implementation point of view.

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.
Document Information

<table>
<thead>
<tr>
<th>IST Project Number</th>
<th>FP6 – 507483</th>
<th>Acronym</th>
<th>DIP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full title</td>
<td>Data, Information, and Process Integration with Semantic Web Services</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Project URL</td>
<td><a href="http://dip.semanticweb.org">http://dip.semanticweb.org</a></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Document URL</td>
<td>Kai Tullius</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Deliverable</th>
<th>Number</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>5.3</td>
<td>Business Process-level Mediation Module Specification</td>
</tr>
<tr>
<td>Work package</td>
<td>Number</td>
<td>Title</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>Service Mediation</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Date of delivery</th>
<th>Contractual</th>
<th>M 18</th>
<th>Actual</th>
<th>30-Jun-05</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status</td>
<td>version. 0.11</td>
<td>final</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nature</td>
<td>Prototype [ ]</td>
<td>Report [ ]</td>
<td>Dissemination [ ]</td>
<td></td>
</tr>
<tr>
<td>Dissemination Level</td>
<td>Public [ ]</td>
<td>Consortium [ ]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Authors (Partner) Emilia Cimpian (NUIG), Jens Lemcke (SAP), Adrian Mocan (NUIG), Michael Schumacher (EPFL)

<table>
<thead>
<tr>
<th>Responsible Author</th>
<th>Email</th>
</tr>
</thead>
<tbody>
<tr>
<td>Emilia Cimpian</td>
<td><a href="mailto:emilia.cimpian@deri.org">emilia.cimpian@deri.org</a></td>
</tr>
<tr>
<td>Partner NUIG</td>
<td>Phone +353-91-495113</td>
</tr>
</tbody>
</table>

Abstract (for dissemination) One of the most difficult obstacles Web services have to overcome in the attempt to exploit the true potential of the World Wide Web is heterogeneity. Due to the nature of the Web itself, heterogeneity problems occur both at the data level and at behavioural level of business logic, message exchange protocols and Web service invocation.

Process mediation is one of the crucial points on the road towards establishing new, ad-hoc cooperation on the Web between various business partners. If semantically enhanced data enables dynamic solutions for coping with data heterogeneity, semantically enhanced Web services can do the same for behavioural heterogeneity. Based on Web Service Modeling Ontology (WSMO) specifications that offers support in semantically describing Web services, we propose a solution that acts on these semantic descriptions and offers the means for defining of what we call a Process Mediator. Such a mediator acts on the public processes...
(represented as WSMO choreographies) of the parties involved in a communication and adjust the bi-directional flow of messages to suit the requested/expected behaviour of each party.

**Keywords**  Process, choreography, behaviour, mediation, composition

<table>
<thead>
<tr>
<th>Version Log</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Issue Date</strong></td>
</tr>
<tr>
<td>&lt;dd-mmm-yy&gt;</td>
</tr>
<tr>
<td>26-01-05</td>
</tr>
<tr>
<td>18-05-05</td>
</tr>
<tr>
<td>25-05-05</td>
</tr>
<tr>
<td>01-06-05</td>
</tr>
<tr>
<td>03-06-05</td>
</tr>
<tr>
<td>05-06-05</td>
</tr>
<tr>
<td>06-06-05</td>
</tr>
<tr>
<td>08-06-05</td>
</tr>
<tr>
<td>10-06-05</td>
</tr>
<tr>
<td>11-06-05</td>
</tr>
<tr>
<td>17-06-05</td>
</tr>
<tr>
<td>26-06-05</td>
</tr>
</tbody>
</table>
# Project Consortium Information

<table>
<thead>
<tr>
<th>Partner</th>
<th>Acronym</th>
<th>Contact</th>
</tr>
</thead>
</table>
| National University of Ireland Galway | NUIG | Prof. Dr. Christoph Bussler  
Digital Enterprise Research Institute (DERI)  
National University of Ireland, Galway  
Galway  
Ireland  
Email: chris.bussler@deri.org  
Tel: +353 91 512460 |
| Fundacion De La Innovacion BANKINTER | Bankinter | Monica Martinez Montes  
Fundacion de la Innovacion. BankINTER  
Paseo Castellana, 29  
28046 Madrid,  
Spain  
Email: mmtnez@bankinter.es  
Tel: 916234238 |
| Berlecon Research GmbH | Berlecon | Dr. Thorsten Wichmann  
Berlecon Research GmbH  
Oranienburger Str. 32  
10117 Berlin,  
Germany  
Email: tw@berlecon.de  
Tel: +49 30 2852960 |
| British Telecommunications Plc. | BT | Dr John Davies  
BT Exact (Orion Floor 5 pp12)  
Adastral Park Martlesham  
Ipswich IP5 3RE,  
United Kingdom  
Email: john.nl.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  
Email: Karl.Aberer@epfl.ch  
Tel: +41 21 693 4679 |
| Essex County Council | Essex | Mary Rowlett,  
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 Abeckert  
Forschungszentrum Informatik  
Haid-und-Neu Strasse 10-14  
76131 Karlsruhe  
Germany  
Email: abeckert@fzi.de  
Tel: +49 721 9654 0 |

---

**Deliverable 5.3a**  
6/29/2005
<table>
<thead>
<tr>
<th>Institution</th>
<th>Contact Person</th>
<th>Address</th>
<th>Email/Phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>Institut für Informatik, Leopold-Franzens University Innsbruck</td>
<td>Prof. Dieter Fensel</td>
<td>Institute of computer science University of Innsbruck Technikerstr. 25 A-6020 Innsbruck, Austria Email: <a href="mailto:dieter.fensel@deri.org">dieter.fensel@deri.org</a> Tel: +43 512 5076485</td>
<td></td>
</tr>
<tr>
<td>ILOG SA</td>
<td>Christian de Sainte Marie</td>
<td>9 Rue de Verdun, 94253 Gentilly, France Email: <a href="mailto:csma@ilog.fr">csma@ilog.fr</a> Tel: +33 1 49082981</td>
<td></td>
</tr>
<tr>
<td>inubit AG</td>
<td>Torsten Schmale</td>
<td>inubit AG Lützowstraße 105-106 D-10785 Berlin Germany Email: <a href="mailto:ts@inubit.com">ts@inubit.com</a> Tel: +49 30726112 0</td>
<td></td>
</tr>
<tr>
<td>Intelligent Software Components, S.A.</td>
<td>Dr. V. Richard Benjamins, Director R&amp;D</td>
<td>Intelligent Software Components, S.A. Pedro de Valdivia 10 28006 Madrid, Spain Email: <a href="mailto:rbenjamins@isoco.com">rbenjamins@isoco.com</a> Tel: +34 913 349 797</td>
<td></td>
</tr>
<tr>
<td>NIWA WEB Solutions</td>
<td>Alexander Wahler</td>
<td>NIWA WEB Solutions Niederacher &amp; Wahler OEG Kirchengasse 13/1a A-1070 Wien Email: <a href="mailto:wahler@niwa.at">wahler@niwa.at</a> Tel:+43(0)1 3195843-11</td>
<td></td>
</tr>
<tr>
<td>The Open University</td>
<td>Dr. John Domingue</td>
<td>Knowledge Media Institute The Open University, Walton Hall Milton Keynes, MK7 6AA United Kingdom Email: <a href="mailto:j.b.domingue@open.ac.uk">j.b.domingue@open.ac.uk</a> Tel.: +44 1908 655014</td>
<td></td>
</tr>
<tr>
<td>SAP AG</td>
<td>Dr. Elmar Domer</td>
<td>SAP Research, CEC Karlsruhe SAP AG Vincenz-Priessnitz-Str. 1 76131 Karlsruhe, Germany Email: <a href="mailto:elmar.domer@sap.com">elmar.domer@sap.com</a> Tel: +49 721 6902 31</td>
<td></td>
</tr>
<tr>
<td>Business Process-level Mediation Module Specification</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-------------------------------------------------------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Sirma Al Ltd.</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sirma</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Atanas Kiriakov,</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ontotext Lab, ~Sirma AI EAD</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Office Express IT Centre, 3rd Floor</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>135 Tzarigradsko Chausse</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sofia 1784, Bulgaria</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Email: <a href="mailto:atanas.kiriakov@sirma.bg">atanas.kiriakov@sirma.bg</a></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tel.: +359 2 9768 303</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Unicorn Solution Ltd.</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unicorn</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Jeff Eisenberg</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unicorn Solutions Ltd.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Malcha Technology Park 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Jerusalem 96951</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Israel</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Email: <a href="mailto:Jeff.Eisenberg@unicorn.com">Jeff.Eisenberg@unicorn.com</a></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tel.: +972 2 6491111</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Vrije Universiteit Brussel</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VUB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Carlo Wouters</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Starlab- VUB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vrije Universiteit Brussel</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pleinlaan 2, G-10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1050 Brussel, Belgium</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Email: <a href="mailto:carlo.wouters@vub.ac.be">carlo.wouters@vub.ac.be</a></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tel.: +32 (0) 2 629 3719</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
LIST OF KEY WORDS/ABBREVIATIONS

Process
Choreography
Behaviour
Mediation
WSMO – Web Service Modeling Ontology
WSML – Web Service Modeling Language
WSMX – Web Service Execution Environment
TABLE OF CONTENTS

EXECUTIVE SUMMARY .................................................................................................................. 1
LIST OF KEY WORDS/ABBREVIATIONS ........................................................................ VIII
TABLE OF CONTENTS .............................................................................................................. IX

1 INTRODUCTION ...................................................................................................................... 1

2 PROBLEM DEFINITION ........................................................................................................... 3
  2.1 Process .............................................................................................................................. 3
    2.1.1 Public Processes ........................................................................................................ 4
    2.1.2 Private Processes ..................................................................................................... 5
  2.2 Process Compatibility ....................................................................................................... 5
    2.2.1 Resolvable Message Mismatches .......................................................................... 6
    2.2.2 Unresolvable Message Mismatches ..................................................................... 8

3 PROCESS REPRESENTATION ................................................................................................. 10
  3.1 State ............................................................................................................................... 11
  3.2 Guarded Transitions ....................................................................................................... 12

4 PROCESSES MEDIATION ...................................................................................................... 14
  4.1 Process Mediator – Component of the DIP Architecture ............................................... 14
  4.2 Process Mediation Approach ........................................................................................ 17
    4.2.2 Interaction Diagram for the Process Mediator Components .................................. 18

5 CONCLUSIONS AND FURTHER DEVELOPMENT ............................................................... 21

REFERENCES ............................................................................................................................ 22

ANNEX1: EXAMPLE .................................................................................................................. 24
  Virtual Travel Agency Service ............................................................................................... 24
  Virtual Travel Agency Requestor ........................................................................................ 29
  Mediation Between the Service and the Requestor ............................................................. 34

ANNEX2: EXECUTION SEMANTICS FOR INTERACTION MISMATCHES ................................. 37
  Pattern a) .............................................................................................................................. 40
  Pattern b) .............................................................................................................................. 41
  Pattern c) .............................................................................................................................. 42
  Pattern d) .............................................................................................................................. 42
  Pattern e) .............................................................................................................................. 42
LIST OF FIGURES
Figure 1: Graphical representation of several processes ........................................... 3
Figure 2: Example of a public process................................................................. 5
Figure 3: Resolvable message mismatches ....................................................... 7
Figure 4: Unresolvable message mismatches ................................................. 9
Figure 5: Overview of Web service execution .............................................. 15
Figure 6: The Process Mediator’s interactions .............................................. 16
Figure 7: Interaction Diagram for the Process Mediator’s Components .......... 19
Figure 8: Communication patterns of the service requestor and service provider 35

LIST OF TABLES
Table 1: Concepts in the internal ontologies .................................................. 4
Table 2: WSMX Process representation ......................................................... 10
Table 3: Concepts in the requestor’s choreography ..................................... 11
Table 4: Transition Rule for the requestor .................................................... 12
Table 5: Process Mediator Interface ............................................................. 16
Table 6: Choreography Parser Component Interface .................................... 19
Table 7: WSML Reasoner Component Interface ........................................ 20
Table 8: VTA Service Ontology ................................................................. 24
Table 9: VTA Service Choreography .......................................................... 26
Table 10: VTA Service Choreography Transition Rules .............................. 26
Table 11: VTA Requestor Ontology ............................................................. 29
Table 12: VTA Requestor Choreography ..................................................... 31
Table 13: VTA Requestor Choreography Transition Rules ............................ 33
1 INTRODUCTION

The advantages offered by the huge amount of information available and by the continually increasing number of services deployed on the Web are hampered by the inherent heterogeneity issues existing between all these resources. The information is represented using different languages and different conceptualizations of the same domain. Similarly Web services describe their functionalities and behaviours in different ways and expect the clients to align with various interaction patterns in order to consume their functionalities.

Numerous approaches propose various solutions to cope with data heterogeneity by adding semantic meaning to data, to make it machine-understandable. Even if this area is well explored and many semi-automatic solutions are available for this problem there is one more gap to fill: how the semantic enriched data is interchanged by machines. That is, even if the machine can understand the data they receive, they also have to understand the communication process it is part of. As a consequence, a coherent mode of describing the expected interaction patterns, together with means of mediating between heterogeneous patterns, is necessary.

This deliverable tackles the process mediation problem from the behaviour mismatches point of view. The addressed processes are the public ones, formalised by using WSMO choreography [4] [20] [21].

Based on the DIP deliverable 5.1 Report on the State-of-the-Art and Requirements Analysis [8], which defines the requirements for such a mediator, as well as on D3.1 [4], which defines the Service Description Framework, we propose a solution for coping with the differences in the way a requester wants to consume the functionality of a Web service and the way this functionality is made available by the Web service to the requester. We use Web Service Modeling Ontology (WSMO)\(^3\) choreography to describe the expected/requested behaviour of the two parties, which is in fact a formalization of their public business processes. Using these descriptions and the services of a data mediator (to solve data heterogeneity problems) we introduce the Process Mediator, a system able to adjust the two parties' behaviour and to enable their communication.

This work took place as a joint effort of the DIP and Lion\(^4\) projects. The authors of this document are involved in the Lion project, and this work was also provided as part of Lion D5.1b Process Mediation – Concepts, Architecture and Implementation in WSMX [10].

This document is structured as follows: the next section (Section 2) contains a description of what we mean by a process and process compatibility in the context of Semantic Web services; Section 3 presents details about the representation of the processes, while Section 4 describes our approach in process mediation, constituting the starting point in the development of a Process Mediator prototype. Section 5 presents

\(^3\) WSMO – Web Service Modeling Ontology Working Group (www.wsmo.org) develops an ontology for describing various aspect related to Semantic Web Services.

\(^4\) DERI-Lion is a joint effort of DERI and of Hewlett-Packard Galway; the main objective of this project is to develop a scalable semantics-based Web Service environment where business partners can seamlessly share services across dynamic business communities; more information about Lion project are available at: http://lion.deri.ie/.
the conclusions and future work. There are also two annexes with this deliverable; the first one presents an example of how the process mediation should work, considering the choreographies of a requestor of a service and of the Semantic Web service that offers it; the second annex presents execution semantics for the identified interaction mismatches.
2 PROBLEM DEFINITION

Process mediation is concerned with determining how two public processes can be matched in order to provide certain functionality. In other words, how two business partners can communicate, considering their public processes.

Process mediation is a complex task, and this document does not address all the problems and all the mediation scenarios that may appear in a business context, but only a small subset of them; this subset will be extended as our work progresses.

In the first subsection of this chapter we describe what we mean by a business process and illustrate the distinction between public and private processes. This section also provides a rationale for only addressing public process heterogeneity. The following subsection describes what process compatibility means in the context of Semantic Web services, providing a list of heterogeneity mismatches that our Process Mediator prototype should be able to automatically overcome, and also a list of mismatches that no mediation prototype, no matter how advanced, can address.

2.1 Process

A process is a collection of activities designed to produce a specific output for a particular customer, based on a specific input; an activity is a function, or a task that occurs over time and has recognizable results [6].

Depending on the level of granularity, each process can be seen as being composed of different, multiple processes. The smallest process possible consists of only one activity. Figure 1 presents a graphical representation of several processes.

![Graphical representation of several processes](image)

**Figure 1: Graphical representation of several processes**

One can distinguish between two types of processes: private processes, which are carried out internally by an entity, and usually are not visible to any other entity, and public processes, which define the behaviour of one entity in collaboration with other entities [12]. As the objective of the Process Mediator is to accommodate the
behavioural mismatches of two participants in a conversation, it is concerned only with the public processes defined by a certain entity, and not with the internal, private ones.

2.1.1 Public Processes

Public processes (called abstract processes or business protocol in Business Process Execution Language for Web Services [22]) define the behaviour of an entity (endpoint) in collaboration with another endpoint, which is expressed by the exchange of messages. To establish communication, each endpoint has to match the behaviour of the other one. The matching means that the two endpoints have to have symmetric behaviour, for example when one of them is sending a message, the other one has to be ready to receive it.

Consider the following example: a Virtual Travel Agency (VTA) service offers the possibility of on-line ticket booking for certain routes, and a requestor of such a service attempts to invoke it. In their internal ontologies, both of them have the following concepts⁵:

<table>
<thead>
<tr>
<th>Table 1: Concepts in the internal ontologies</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>concept</strong></td>
</tr>
<tr>
<td><strong>nonFunctionalProperties</strong></td>
</tr>
<tr>
<td>dc⁶:description hasValue &quot;concept of station, containing the code of the station, and two attributes showing whether this station is the start or end point of a trip&quot;</td>
</tr>
<tr>
<td><strong>endNonFunctionalProperties</strong></td>
</tr>
<tr>
<td>code <strong>ofType</strong> xsd⁷:String</td>
</tr>
<tr>
<td>startPoint <strong>ofType</strong> xsd:Boolean</td>
</tr>
<tr>
<td>endPoint <strong>ofType</strong> xsd:Boolean</td>
</tr>
<tr>
<td><strong>concept</strong></td>
</tr>
<tr>
<td><strong>nonFunctionalProperties</strong></td>
</tr>
<tr>
<td>dc:description hasValue &quot;concept of route, having two attributes of type station which show the start and the end point of the route&quot;</td>
</tr>
<tr>
<td><strong>endNonFunctionalProperties</strong></td>
</tr>
<tr>
<td>sourceLocation <strong>ofType</strong> station</td>
</tr>
<tr>
<td>destinationLocation <strong>ofType</strong> station</td>
</tr>
</tbody>
</table>

⁵ All the concepts and axioms from this document are described using Web Service Modeling Language (http://www.w3.org/wsml) version 0.1; all the tools developed so far in WSMX are using this version of WSML, the authors intend to follow the newer releases of WSML as soon as tool support will be provided for it

⁶ dc: The Dublin Core Element Set v1.1, available at: http://purl.org/dc/elements/1.1#

The assumption that both the requestor and the provider of the service understand the same concepts was made only for the sake of simplicity. If they use different conceptualizations of the same domain, an external Data Mediator [17] [8] should be invoked to solve the data heterogeneity problem.

A public process for the requestor could be that after sending two instances of station, it expects to receive an instance of route. The equivalent public process from the requestor side could be that it expects two instances of station in order to generate an instance of route. The relations between the two instances of station and the instance of route are not specified in the public interface of either of the two endpoints.

Figure 2 is a graphical representation of the requestor’s public processes.

![Figure 2: Example of a public process](image)

### 2.1.2 Private Processes

Private processes (called executable business processes in BPEL4WS [22]) model the actual behaviour of an endpoint involved in an interaction, that is, the internal processing of events.

For example, in the previously described scenario, the computations performed by the VTA in order to generate the instance of route are not visible to the outside world. The private process of generating this instance could check, for example, whether the two instances of station exist in the particular ontology of the service (otherwise it is impossible for it to have a route between them) and whether the requestor specified which station is the start and which the end point of the trip.

### 2.2 Process Compatibility

In this context, we understand process compatibility to mean full matching of the communication patterns between the source and the target of the communication; that is, when one of them is sending a message, the other one is able to receive it. We are not addressing the private process compatibility because the private process are not visible to the outsiders, and there is no way of mediating between them.
Since a business communication usually consists of more than one exchange message, finding equivalences in the message exchange patterns of the two (or more) parties is not at all a trivial task. Intuitively, the easiest way of doing this is to first determine the matches and mismatches, and then search for a way to eliminate them. Fensel and Bussler ([12]) identify three possible cases that may appear during a message exchange:

**Precise match:** The two partners have exactly the same pattern in realizing the business process, which means that each of them sends the messages in exactly the order that the other one requests them. In this ideal case the communication can take place without using a Process Mediator. However, this does not mean that the services of a Data Mediator may not be required.

**Resolvable message mismatch:** Here, the two partners use different exchange patterns, and several transformations have to be performed in order to resolve the mismatches (for example when one partner sends more than one concept in a single message, but the other one expects them separately. In this case the mediator can “break” the initial message, and send the concepts one by one.)

**Unresolvable message mismatch:** One of the partners expects data that the other one does not intend to send (considering the example from Section 2.1.1, if the provider of the service expects to receive an intermediate station, and the requestor does not intend to send it). Unless the mediator can provide this message, the communication reaches a dead-end.

In order to communicate, two participants have to define compatible processes, or use an external mediation system as part of the communication process. The role of the mediator system will be to transform the client’s messages and/or the Web service’s messages, in order to obtain a sequence of compatible processes.

In the following subsections we describe the resolvable message mismatches that our Process Mediator is intended to address, and the unresolvable message mismatches that a Process Mediator cannot address. The ideal case of precise match does not raise any problems from the communication pattern point of view, so we ignore it in our discussion.

### 2.2.1 Resolvable Message Mismatches

Some of the communication mismatches can be addressed by means of a mediation system. In this section we provide a list of resolvable mismatches, identified in [9] that our mediator is intended to automatically address.

a) **Stopping an unexpected message** (Figure 3. a) – If one of the partners sends a message that the other one does not want to receive, the mediator should just retain and store it. This message (or a subset of the data contained in this message) can be sent later, if needed, or it can just be deleted after the communication ends.

b) **Inverting the order of messages** (Figure 3. b) – If one of the partners sends the messages in a different order than the other partner expects, the messages that are not yet expected will be stored and sent when needed. This case involves the previous one, the first message being actually stored when received, and not immediately sent to the targeted partner.
c) **Splitting a message** (Figure 3. c) – If one of the partners sends in a single message pieces of information that the other one expects to receive in different messages, the information can be split and sent in a sequence of separate messages. The splitting of a certain instance in several different instances, that may appear during the data mediation process, is only a subset of the message splitting, since a message can consist of multiple instances.

d) **Combining messages** (Figure 3. d) – If one of the partners expects a single message, containing information sent by the other one in multiple messages, the information can be combined into a single message. This combined message may also contain data that was already transmitted. Similarly with the previously described set, combining messages may also involve combining several instances in a single one during the data mediation process.

e) **Sending a dummy acknowledgement** (Figure 3. e) – If one of the partners expects an acknowledgement for a certain message, and the other partner does not intend to send it, even if it receives the message, an acknowledgment can be automatically generated and sent to the partner which requires it.

![Diagram](image-url)

**Figure 3: Resolvable message mismatches**

In addition to this informal, verbal representation, [2] adds a formal representation to the description of the given patterns. This is done by defining a mathematical set-based representation of the message exchanges, and a stateful execution semantics operating on this mathematical structure. The formal representation therefore makes the following implicit assumptions (amongst others) explicit:
- ** Capitals describe sets of data, whereas arrows depict messages.** This differentiation is needed, since in Pattern c messages may transport multiple data sets as well as single ones.

- ** The order of messages.** An arrow above an other arrow means that the message represented by the upper arrow is sent before the one represented by the lower arrow. This is needed, since Pattern b refers to converting message order, whereas Pattern a could be interpreted as “concurrently receiving two messages and discarding one of those”, too. However, no account is made on conditional branches (depending on the current state, one entity may performed one action or another), since the order of messages may change depending upon what previously occurred.

- ** The mode of data transmission is reliable.** This assumption is covered in Pattern e. Only if PM can be sure that Business Partner 2 actually receives the message containing data set A (via a reliable connection), it can fake the acknowledgement that it sends back to Business Partner 1.

As detailed in [3], a formal execution semantics can be used to prove certain properties of interactions, e.g. the compatibility of two communication patterns with respect to a requested interface. For the sake of completeness, we provide a simplification of the formal representation for the resolvable interaction mismatches in Annex2: Execution Semantics for Interaction Mismatches.

This list of resolvable message mismatches contains only the initial subset of mismatches that our Process Mediator is intended to address. More complex patterns can be derived by combining the initial ones. Additionally, in future work, the Process Mediator will be further extended, in order to address more possibly resolvable mismatches.

### 2.2.2 Unresolvable Message Mismatches

There are several communication mismatches that cannot be addressed by a Process Mediator.

a) **Generating a message** (Figure 4. a) – If one of the partners expects a message containing information that the other partner did not previously send, the Process Mediator is not able to generate this missing information.

b) **Sending a dummy acknowledgement** (Figure 4. b) – If one of the partners expects an acknowledgement for a certain message, but the other partner does not want to receive the message, the Process Mediator cannot send a dummy acknowledgement. Although this may seem similar with case e) from the previous section, the different is that in this case Business Partner 2 did not received the message; by sending the acknowledgement the Process Mediator could alter the entire communication between the two parties.
Like in the previous section this list could also be extended as more complex scenario are identified by the DIP use-cases (Work Packages 8, 9 and 10).
3 PROCESS REPRESENTATION

The process representation used in DIP project was initially developed in the Web Service Execution Environment (WSMX®) working group. As a consequence, in the rest of this chapter we will sometimes refer to the public processes as “WSMX processes”.

As in WSMO Choreography and Orchestration [21] the representation of a business process is based on the Abstract State Machine [13] methodology. ASMs have been chosen as the underlying model of choreography and orchestration for the following three reasons:

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

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

3. **Formality**: ASMs provide a rigid mathematical framework for expressing dynamics.

For a detailed explanation of ASMs we refer the reader to [5].

In order for this deliverable to be self-contained, we describe in the next paragraphs the main features of WSMX processes, i.e. the main features of WSMO choreography, as described in [21].

Taking the ASMs methodology as a starting point, a WSMX process is state-based and consists of two types of elements: states and guarded transitions.

<table>
<thead>
<tr>
<th>Table 2: WSMX Process representation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Process Representation</strong></td>
</tr>
<tr>
<td><strong>Class</strong> wsmxProcess</td>
</tr>
<tr>
<td><em>hasState</em> type ontology</td>
</tr>
<tr>
<td><em>hasGuardedTransitions</em> type guardedTransition</td>
</tr>
</tbody>
</table>

All the wsmxProcess elements are defined similar with WSMO choreography [21], but for consistency reasons we will provide them here as well.

**State**

A state is described by an ontology as defined in [20].

**Guarded transitions**

Transition rules that express changes of states by changing the set of instances.

---

The WSMX Working Group’s objective is to create an execution environment for the dynamic discovery, selection, mediation, invocation and inter-operation of the Semantic Web Services; most of the WSMX working group members are also involved in DIP project.
3.1 State

A state is described by a set of explicitly defined instances and values of their attributes or through a link to an instance store.

In extension to a standard WSMO ontology, an ontology that is used to describe states in a WSMX process introduces a new non-functional property, the mode attribute. When a concept, relation or function in a process is defined as being part of the choreography, the attribute mode needs to be defined as a new non functional property. It can take one of the following values:

- **static** - meaning that the extension of the concept, relation, or function cannot be changed (not even by its owner). If not explicitly defined, the attribute mode takes this value by default. The second participant in the conversation does not even have read access over this element, but this does not mean that its instances can not be encapsulated in information received by it (for example, as attribute values).

- **controlled** - meaning that the extension of the concept, relation, or function can only be changed by its owner. As in the previous case, the second participant in the conversation can not have direct read access over instances of this element, but can receive them as part of other elements.

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

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

- **out** - meaning that the extension of the concept, relation, or function can only be changed by its owner. A grounding mechanism that implements read access for the environment for this item must be provided.

For more details on grounding we refer the reader to [16].

Considering the previous example described in Section 2.1.1, the requestor of the service has the attribute mode set to out for the concept station, and set to in for the concept route:

<table>
<thead>
<tr>
<th>Table 3: Concepts in the requestor’s choreography</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>concepts</strong> <strong>stationInChoreography</strong> <strong>subConceptOf</strong> <strong>station</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
The signature of the states is defined by WSMO identifiers, concepts, relations, functions, and axioms. This signature is the same for all states. The elements that can change and that are used to express different states of a choreography, are the instances (and their attribute values) of concepts, functions, and relations that are not defined as being static. In conclusion, a specific state is described by a set of explicitly defined instances and values of their attributes or through a link to an instance store.

3.2 Guarded Transitions

Guarded Transitions are used to express changes of states by means of rules, expressible in the following form:

if Cond then Updates.

Cond is an arbitrary WSML axiom, formulated in the given signature of the state.
The Updates consist of arbitrary ontology instances.

In the previously described example, the guarded transition that states that in the requestor’s choreography an instance of route should exist (internally created or received from the service) only after at least one instance of station exists is the following:

<table>
<thead>
<tr>
<th>Table 4: Transition Rule for the requestor(^9)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Transition Rule for the requestor</strong></td>
</tr>
</tbody>
</table>
| \( ?x \text{ memberOf routeInChoreography}[^] \)
| \( \quad \text{sourceLocation hasValue ?startLocation_}, \)
| \( \quad \text{destinationLocation hasValue ?endLocation_} \)
| \( \quad \leftarrow \)
| \( \quad ?\text{startLocation_ memberOf stationInChoreography and} \)
| \( \quad ?\text{endLocation_ memberOf stationInChoreography}. \) |

Please note that there is no assumption made in this rule about the relation between the two instances of `stationInChoreography`; as much as this rule states, there can even be only one instance. The additional conditions imposed on the two instances of `stationInChoreography`, and on their relation with the instance of `routeInChoreography` can be modelled in the internal ontology of the requestor. How much information is made public using choreography, and how much remains private in the internal ontology is strictly a modelling choice. The ideal case is for the

\(^9\) The syntax used in this listing is equivalent with the classical if-then-else syntax.
choreography to make public exactly the amount of information needed to allow external users to interact without errors, but every participant may choose to publish more or less than is actually needed.
4 PROCESSES MEDIATION

This chapter presents our approach for process mediation. It is structured in two main parts: the first describes the Process Mediator component as part of the overall DIP architecture; the second provides the interface that our Process Mediator will implement and presents the Process Mediator sub-components as well as their interfaces.

4.1 Process Mediator – Component of the DIP Architecture

When the DIP system\textsuperscript{10} receives a message [25], either from the requestor of a service or from a Web service, it has to check if it is the first message in a conversation. If it is the first, the system creates copies (instances) of both the sender and the targeted business partner choreographies, and stores these instances in a repository. If it is not the first message of a conversation, the DIP system has to determine the two choreography instances corresponding to the conversation (their IDs), and the id of the conversation. These computations performed on the message are done by other two components, Receiver and Choreography Engine (described in DIP deliverables 6.5 [25]). After the IDs of the two choreography instances are obtained, the Process Mediator (PM) receives them, together with the message, consisting of instances of concepts from the sender’s ontology. Based on the IDs, the PM loads the two choreography instances from the Repository, by invoking the Resource Manager. All the transformations performed by the PM will be done on these instances. If different ontologies have been used for modelling the two choreographies, the PM has to invoke an external Data Mediator to transform the message into the terms of the target ontology (Figure 5).

\textsuperscript{10} By the DIP system we are referring to the implementation of the DIP architecture, described in [25].
Figure 5: Overview of Web service execution\textsuperscript{11}

After various internal computations [9] described in the following section, the PM determines whether, based on the incoming message, it can generate any message expected by either one of the partners. The generation of any message determines a transformation in the choreography instance of the party that receives that message. After sending the message, the Process Mediator re-evaluates all the rules, until no further updates are possible.

The interactions between the Process Mediator and other components are represented in Figure 6.

\textsuperscript{11} Source [26]; The tool used is, CPNTools [19], which make it possible to model so-called high-level Petri-nets [1], extending classical Petri nets with hierarchy, colour and time.
The Process Mediator should implement the following interface:

**Table 5: Process Mediator Interface**\(^{12}\)

<table>
<thead>
<tr>
<th>Process Mediator Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>public interface ProcessMediator {</td>
</tr>
<tr>
<td>public Map&lt;Identifier, List&lt;Identifiable&gt;&gt; generate(Identifier source, Identifier target, Set&lt;Identifiable&gt; data)</td>
</tr>
<tr>
<td>throws ComponentException, UnsupportedOperationException;</td>
</tr>
<tr>
<td>public Map&lt;Identifier, List&lt;Identifiable&gt;&gt; generate(Ontology source, Ontology target, Set&lt;Identifiable&gt; data)</td>
</tr>
<tr>
<td>throws ComponentException, UnsupportedOperationException;</td>
</tr>
<tr>
<td>public Map&lt;Identifier, List&lt;Identifiable&gt;&gt; generate(Choreography source, Choreography target, Set&lt;Identifiable&gt; data)</td>
</tr>
<tr>
<td>throws ComponentException, UnsupportedOperationException;</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

\(^{12}\) The interface is described in Java 1.5;

The classes used in describing the methods are defined in WSMO4J, an API and a reference implementation for building Semantic Web Services applications compliant with the Web Service Modeling Ontology; available at: http://wsmo4j.sourceforge.net/index.html
When the Process Mediator is invoked, it should receive information about the two participants in the conversation in terms of what ontologies or choreographies they use, as well as the actual data sent by one of the partners. The information about the two participants in the conversation can be transmitted as Identifier objects of an ontology or of a choreography (the first method), as Ontology objects (the second method) or as a Choreography objects (the last method). The data sent by one of the partners is already parsed [25] and decomposed in ontology elements; all the Process Mediator needs to receive is a set of Identifiable objects, each Identifiable uniquely identifying an element from an ontology.

The returned Map object, for all three methods, contains two pairs. The first element of a pair is the identifier of the targeted partner (the result of receiving a message from one participant can be sending a message to its partner or to the initial sender – for example in the case of sending an acknowledgement); the second element is a list containing the actual information to be sent.

All three methods have the same name for consistency reasons. The distinction between them is made by the different types of parameters.

### 4.2 Process Mediation Approach

The Process Mediator is triggered when it receives a message, and the two choreography instances’ IDs. The message contains instances of concepts in terms of the sender's ontology.

After being invoked, the PM performs the following steps:

1. Loads the two choreography instances from the repository.
2. Mediates the incoming instances in terms of the targeted partner ontology, and checks whether the targeted partner is expecting them at any phase of the communication process. This is done by checking the value of the mode attribute, for the mediated instances' owners. If the attribute mode for a certain concept is set to in or shared, than this concept's instances may be needed at some point in time. The instances that are expected by the targeted partner are stored in an internal repository. Each instance will be stored together with the identifiers of the choreographies’ instances of the two participants, in order to uniquely identify the conversation it is part of.
3. For all the instances from the repository, the PM has to check whether they are expected at this phase of the communication, which is done by evaluating the transition rules of the target partner choreography [21]. The evaluation of a rule will return the first condition that cannot be fulfilled, that is, the next expected instance for that rule. This means that an instance is expected if it can trigger an action (not necessarily to change a state, but to eliminate one condition for changing a state).

The possibility that various instances from this repository can be combined in order to obtain a single instance, expected by the targeted business partner, is also considered.
4. Each time the PM determines that one instance is expected, it sends it, deletes it from the repository, updates the targeted partner choreography instance, and restarts the evaluation process (step 4). When a transition rule can be executed, it is marked as such and not re-evaluated at further iterations, unless one of the partners generates an error (for example it rejects one of the received instances because it does not fulfil its internal conditions).

The PM only checks whether a transition rule can be executed, and does not execute it, since it cannot update any of the two choreography instances without receiving input from one of the communication partners. By evaluating a rule, the PM determines that one of the business partners can execute it, without expecting any other inputs.

This process stops when, after performing these checks for all the instances from the repository, no new message is generated.

5. For each instance forwarded to the targeted partner, the PM has to check whether the sender is expecting an acknowledgement. If the sender expects an acknowledgement, but the targeted partner does not intend to send it, the PM generates a dummy acknowledgement and sends it. The acknowledgements are represented in the two partners choreographies as concepts having the mode set to in.

6. The PM checks all the sender's rules and marks the ones that can be executed.

7. The PM checks the requestor's rules, to see if all of them are marked; when all are marked, the communication is over and PM deletes all the instances created during this conversation (together with the choreography instances), from both its internal repository and the system's repository. This simple condition may not hold when more complicated scenarios are considered (for example loops or conditional branches). When such cases will appear, this condition will be replaced with a more complex one (for example, the PM may encapsulate the corresponding transition rule in a different rule that states the stopping condition), or more than one condition will be evaluated in order to terminate the conversation. However, even if the Process Mediator will not determine if a conversation is over, this will not affect the conversation in itself, but only our internal repositories.

The only thing that should be kept in an internal storage is the actions the Process Mediator needs to take when receiving a message. This could be useful if the same two partners are later involved in a second conversation.

This algorithm is implemented by the PM in order to solve the communication heterogeneity problem.

**4.2.2 Interaction Diagram for the Process Mediator Components**

The following figure presents the steps performed during the process mediation, as well as the involved components. The ovals are used for representing actions, while rectangles are used for representing components. The components from the bottom part of the figure are external components, not part of the Process Mediator.
The links with the Choreography Engine are not represented in the above figure. The Choreography Engine is the one that triggers the entire process, by calling the Generate method.

The Process Mediator consists of three components: Choreography Parser, WSML Reasoner, and an internal Repository.

**Choreography Parser**
The Choreography Parser has the role of determining if any instance obtained after the data mediation process is expected by the targeted partner. That is, the choreography parser will have to perform the following operations:

1. determine the owner of an certain instance (the concept that is instantiated) contained in a message;

2. determine the value of the mode attribute for the owner; if the value is set to in or shared, the instance will be stored in the internal repository for further usage.

The interface implemented by this component is depicted in the following listing:

**Table 6: Choreography Parser Component Interface**

<table>
<thead>
<tr>
<th>Choreography Parser Component Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>public interface ChoreographyParser {</td>
</tr>
<tr>
<td>public boolean required(Identifiable data);</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

As stated in the interface, the Choreography Parser does not have to return the value of the mode attribute, but only a boolean value: true if the data is required by the targeted partner at some point in time (the mode is set to in or shared) or false otherwise.
**Internal Repository**

The Internal Repository is used for storing information that will be sent to one of the partners at some point in time. It offers the methods `store`, `extract`, `delete` and `update`. There is only one Internal Repository for all the conversations in which the PM is involved, and each entry in the repository contains not only the actual information to be sent, but also the IDs of the two participants choreography instances, in order to uniquely identify the conversation.

**WSML Reasoner**

The WSML Reasoner is the most complex component of the Process Mediator. Its task is to extract one by one the instances from the repository, and to check if by sending that instance at least one condition of one transition rule can be fulfilled. The transition rules were previously obtained from the Choreography Engine, together with the concepts part of the choreographies.

The interface of the WSML Reasoner component is as follows:

<table>
<thead>
<tr>
<th>Table 7: WSML Reasoner Component Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>WSML Reasoner Component Interface</strong></td>
</tr>
<tr>
<td>public interface WSMLReasoner {</td>
</tr>
<tr>
<td>public Set&lt;Identifiable&gt; expected(Identifiable target);</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

The expected method has only one parameter, the `Identifiable` object corresponding to the targeted partner, and returns the set of elements expected at that particular point in the communication (there may be cases when two or more of the instances previously stored in the repository are expected at the same time).
5 CONCLUSIONS AND FURTHER DEVELOPMENT

This deliverable tackles the behavioural heterogeneity of two business partners and proposes a solution for solving this heterogeneity. Taking in consideration only the public processes of an entity, the mediation of their behaviour may be needed when a requester attempts to invoke and execute a Web service of a provider, and the communication patterns of the service requestor and the service provider do not match.

Each time a message is sent by one of the two parties involved in the conversation, the Process Mediator has to determine if the message is expected by the other party. The mediator also has to consider situations when only part of a message or a combination of this message with a previously received one is expected.

The deliverable presented an overview of the addressed mismatches, and the way the process mediation is able to automatically solve these problems. It also proposes the interface that the Process Mediator should implement, as well as interfaces for the Process Mediator’s subcomponents.

Based on this deliverable, the next step in our work will be the development of a Process Mediator prototype. The interfaces for this prototype are already defined as part of the overall DIP architecture. Also we plan to extend the addressed mismatches, with new and more complex cases raised by the use-cases.
REFERENCES


and analysing coloured petri nets. In W.M.P van der Aalst and E. Best, editors, 
Applications and Theory of Petri Nets 2003: 24th International Conference, ICATPN 
2003, volume 2679 of Lecture Notes in Computer Science, pages 450–462. Springer-
Verlag, 2003.


Orchestration of WSMO Services, WSMO deliverable D14 v0.1. available at 
http://www.wsmo.org/TR/d14/v0.1/, 2005.

[22] Specification: Business Process Execution Language for Web Services, Version 1.1, 


[25] Zaremba, Michal; Moren, M.; Haselwanter, T.; Zaremba, Maciej; Oren, E.: DIP 
deliverable 6.5 DIP Revised Architecture, 2005.

Draft v0.2, available at http://www.wsmo.org/TR/d13/d13.2/v0.2/20050425/wsmx-
ANNEX 1: EXAMPLE

In this annex we present two examples of public processes, the process defined by a Virtual Travel Agency service and the process defined by a requestor of this service.

Virtual Travel Agency Service

In this subsection, we define the Virtual Travel Agency (VTA) service's ontology, its choreography ontology and the corresponding transition rules.

The VTA service ontology, defining the concepts needed for performing on-line ticket reservation, is as follows:

Table 8: VTA Service Ontology

<table>
<thead>
<tr>
<th>VTA Service Ontology</th>
</tr>
</thead>
<tbody>
<tr>
<td>namespace</td>
</tr>
<tr>
<td>&lt;<a href="http://www.wsmo.org/TR/d13/d13.7/ontologies/VTAServiceOnt/">http://www.wsmo.org/TR/d13/d13.7/ontologies/VTAServiceOnt/</a>&gt;</td>
</tr>
<tr>
<td>dc: &lt;<a href="http://purl.org/dc/elements/1.1#">http://purl.org/dc/elements/1.1#</a>&gt;</td>
</tr>
<tr>
<td>xsd: &lt;<a href="http://www.w3.org/2001/XMLSchema#">http://www.w3.org/2001/XMLSchema#</a>&gt;</td>
</tr>
<tr>
<td>ontology</td>
</tr>
<tr>
<td>&lt;<a href="http://www.wsmo.org/TR/d13/d13.7/ontologies/VTAServiceOnt">http://www.wsmo.org/TR/d13/d13.7/ontologies/VTAServiceOnt</a>&gt;</td>
</tr>
<tr>
<td>nonFunctionalProperties</td>
</tr>
<tr>
<td>dc:title hasValue &quot;Virtual Travel Agency Service Ontology&quot;</td>
</tr>
<tr>
<td>dc:creator hasValue &quot;Emilia&quot;</td>
</tr>
<tr>
<td>dc:description hasValue &quot;an ontology for describing trip reservation service related knowledge&quot;</td>
</tr>
<tr>
<td>dc:publisher hasValue &quot;DERI International&quot;</td>
</tr>
<tr>
<td>dc:contributor hasValues &quot;Adrian&quot;</td>
</tr>
<tr>
<td>dc:type hasValue &lt;<a href="http://www.wsmo.org/2004/d2/#ontology">http://www.wsmo.org/2004/d2/#ontology</a>&gt;</td>
</tr>
<tr>
<td>dc:format hasValue &quot;text/html&quot;</td>
</tr>
<tr>
<td>dc:language hasValue &quot;en-us&quot;</td>
</tr>
<tr>
<td>dc:rights hasValue &lt;<a href="http://deri.ie/privacy.html">http://deri.ie/privacy.html</a>&gt;</td>
</tr>
<tr>
<td>version hasValue &quot;$Revision 0.1 $&quot;</td>
</tr>
<tr>
<td>endNonFunctionalProperties</td>
</tr>
<tr>
<td>concept station</td>
</tr>
<tr>
<td>nonFunctionalProperties</td>
</tr>
<tr>
<td>dc:description hasValue &quot;concept of station, containing the code of the station, and two attributes showing whether this station is the start or the end point of a trip&quot;</td>
</tr>
<tr>
<td>endNonFunctionalProperties</td>
</tr>
<tr>
<td>code ofType xsd:string</td>
</tr>
<tr>
<td>startpoint ofType xsd:boolean</td>
</tr>
</tbody>
</table>
endpoint ofType xsd:boolean

concept route

nonFunctionalProperties

dc:description hasValue "concept of route, having two attributes of type station which show the start and the end point of the route"

endNonFunctionalProperties

sourceLocation ofType station
destinationLocation ofType station

concept routeOnDate

nonFunctionalProperties

dc:description hasValue "concept of route on a certain date, containing the route, the date, the departure time and the price of a ticket"

endNonFunctionalProperties

forRoute ofType route

onDate ofType date

onTime ofType time

forPrice ofType price

concept time subConceptOf xsd:time

nonFunctionalProperties

dc:description hasValue "concept of time"

endNonFunctionalProperties

concept date subConceptOf xsd:date

nonFunctionalProperties

dc:description hasValue "concept of date"

endNonFunctionalProperties

concept price subConceptOf xsd:integer

nonFunctionalProperties

dc:description hasValue "concept of price"

endNonFunctionalProperties

concept person

nonFunctionalProperties

dc:description hasValue "concept of person, containing only the name of the person"

endNonFunctionalProperties

name ofType xsd:string

concept creditCard

nonFunctionalProperties

dc:description hasValue "concept of credit card, containing its number, owner and expiration date"
endNonFunctionalProperties

number ofType xsd:integer
owner ofType person
expirationDate ofType date
concept reservation

nonFunctionalProperties

dc:description hasValue "concept of reservation, containing its number, owner, and the route for which this reservation was made"

endNonFunctionalProperties

reservationNumber ofType xsd:integer
reservedRoute ofType routeOnDate
reservationHolder ofType person

The VTA service choreography’s ontology (any business partner express its public processes as WSMO choreography) is depicted in the following table:

Table 9: VTA Service Choreography

<table>
<thead>
<tr>
<th>VTA Service Choreography</th>
</tr>
</thead>
<tbody>
<tr>
<td>namespace</td>
</tr>
<tr>
<td>&lt;<a href="http://www.wsmo.org/TR/d13/d13.7/ontologies/VTAServiceCh">http://www.wsmo.org/TR/d13/d13.7/ontologies/VTAServiceCh</a>&gt; vtas: &lt;<a href="http://www.wsmo.org/TR/d13/d13.7/ontologies/VTAServiceOnt">http://www.wsmo.org/TR/d13/d13.7/ontologies/VTAServiceOnt</a>&gt; dc: &lt;<a href="http://purl.org/dc/elements/1.1">http://purl.org/dc/elements/1.1</a>&gt;&gt; ontology &lt;<a href="http://www.wsmo.org/TR/d13/d13.7/ontologies/VTAServiceCh">http://www.wsmo.org/TR/d13/d13.7/ontologies/VTAServiceCh</a>&gt; nonFunctionalProperties dc:title hasValue &quot;Virtual Travel Agency Service Choreography&quot; dc:creator hasValue &quot;Emilia&quot; dc:description hasValue &quot;an ontology for describing trip reservation service choreography related knowledge&quot; dc:publisher hasValue &quot;DERI International&quot; dc:contributor hasValues &quot;Adrian&quot; dc:type hasValue &lt;<a href="http://www.wsmo.org/2004/d2/#ontology">http://www.wsmo.org/2004/d2/#ontology</a>&gt; dc:format hasValue &quot;text/html&quot; dc:language hasValue &quot;en-us&quot; dc:rights hasValue &lt;<a href="http://deri.ie/privacy.html">http://deri.ie/privacy.html</a>&gt; version hasValue &quot;$Revision 0.1 $&quot; endNonFunctionalProperties</td>
</tr>
</tbody>
</table>
importedOntologies
    <http://www.wsmo.org/TR/d13/d13.7/ontologies/VTAServiceOntology>
>>

class concept station subConceptOf vtas:station
    nonFunctionalProperties
        mode hasValue in
    endNonFunctionalProperties

class concept route subConceptOf vtas:route
    nonFunctionalProperties
        mode hasValue out
    endNonFunctionalProperties

class concept routeOnDate subConceptOf vtas:routeOnDate
    nonFunctionalProperties
        mode hasValue out
    endNonFunctionalProperties

class concept time subConceptOf vtas:time

class concept date subConceptOf vtas::date
    nonFunctionalProperties
        mode hasValue in
    endNonFunctionalProperties

class concept price subConceptOf vtas:price

class concept person subConceptOf vtas:person
    nonFunctionalProperties
        mode hasValue in
    endNonFunctionalProperties

class concept creditCard subConceptOf vtas:creditCard
    nonFunctionalProperties
        mode hasValue in
    endNonFunctionalProperties

class concept reservation subConceptOf vtas:reservation
    nonFunctionalProperties
        mode hasValue out
    endNonFunctionalProperties

The behaviour of the VTA service is given by the rules defined in Table 10.

Table 10: VTA Service Choreography Transition Rules

<table>
<thead>
<tr>
<th>VTA Service Choreography Transition Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>/*an instance of route can be created only after two instances of</td>
</tr>
</tbody>
</table>

27
station exist; since the concept station has the mode set to in, these instances need to be provided by the environment; the instance of route will be sent to the requestor of the service*/
?x memberOf vtasc:route[
    sourceLocation hasValue ?startLocation_,
    destinationLocation hasValue ?endLocation_]
    ?startLocation_ memberOf vtasc:station and
    ?endLocation_ memberOf vtasc:station.
/*an instance of routeOnDate is created, assuming that instances of route, date, time and price already exist; since none of these concepts has the mode set to in, none of them is expected from the environment*/
?x memberOf vtasc:routeOnDate[
    forRoute hasValue ?forRoute_,
    onDate hasValue ?onDate_,
    onTime hasValue ?onTime_
    forPrice hasValue ?forPrice_]
    ?forRoute_ memberOf vtasc:route and
    ?onDate_ memberOf vtasc:date and
    ?onTime_ memberOf vtasc:time and
    ?forPrice memberOf vtasc:integer.
/*an instance of the reservation concept is created, assuming that instances of routeOnDate, creditCard and person already exist. The instances of creditCard and person have to be obtained from the environment*/
?x memberOf vtasc:reservation[
    reservationNumber hasValue ?reservationNumber_,
    reservationRoute hasValue ?reservationRoute_,
    reservationHolder hasValue ?reservationHolder_]
    ?reservationRoute_ memberOf vtasc:routeOnDate and
    ?creditCard_ memberOf vtasc:creditCard and
    ?reservationHolder_ memberOf vtasc:person.

Note that these rules do not reflect the actual computations done by the system in order to create certain instances, they just express the message sequencing. For example, for generating an instance of route, the system has to internally check whether the two stations are the beginning and the end of the trip, and whether it can provide a connection between these two stations. However, all these computations are private, and they are not visible to the outside world. Based on the transition rules, the order of sending/receiving messages (communication pattern) can be established.

1. Receive two instances of station (the order of receiving these two instances does not matter);
2. Send an instance of route;
3. Receive an instance of date;
4. Send an instance of routeOnDate;
5. Receive an instance of creditCard and an instance of person (the distinction between the owner of the credit card and the person that makes the reservation is made internally, and doesn’t need to be published in the choreography);

Virtual Travel Agency Requestor

In the previous example we described a service that provides on-line ticket booking facilities. In this example, we will describe the requestor of such a service.

The VTA requestor has the following ontology:

Table 11: VTA Requestor Ontology

<table>
<thead>
<tr>
<th>VTA Requestor Ontology</th>
</tr>
</thead>
<tbody>
<tr>
<td>namespace</td>
</tr>
<tr>
<td>&lt;<a href="http://www.wsmo.org/TR/d13/d13.7/ontologies/VTAResquestorOnt">http://www.wsmo.org/TR/d13/d13.7/ontologies/VTAResquestorOnt</a>&gt;</td>
</tr>
<tr>
<td>dc:</td>
</tr>
<tr>
<td>&lt;<a href="http://purl.org/dc/elements/1.1#">http://purl.org/dc/elements/1.1#</a>&gt;</td>
</tr>
<tr>
<td>xsd:</td>
</tr>
<tr>
<td>&lt;<a href="http://www.w3.org/2001/XMLSchema#">http://www.w3.org/2001/XMLSchema#</a>&gt;</td>
</tr>
<tr>
<td>ontology</td>
</tr>
<tr>
<td>&lt;<a href="http://www.wsmo.org/TR/d13/d13.7/ontologies/VTAResquestorOnt">http://www.wsmo.org/TR/d13/d13.7/ontologies/VTAResquestorOnt</a>&gt;</td>
</tr>
<tr>
<td>nonFunctionalProperties</td>
</tr>
<tr>
<td>dc:title hasValue &quot;Virtual Travel Agency Requestor Ontology&quot;</td>
</tr>
<tr>
<td>dc:creator hasValue &quot;Emilia&quot;</td>
</tr>
<tr>
<td>dc:description hasValue &quot;an ontology for describing trip reservation requestor related knowledge&quot;</td>
</tr>
<tr>
<td>dc:publisher hasValue &quot;DERI International&quot;</td>
</tr>
<tr>
<td>dc:contributor hasValues &quot;Adrian&quot;</td>
</tr>
<tr>
<td>dc:type hasValue &lt;<a href="http://www.wsmo.org/2004/d2/#ontology">http://www.wsmo.org/2004/d2/#ontology</a>&gt;</td>
</tr>
<tr>
<td>dc:format hasValue &quot;text/html&quot;</td>
</tr>
<tr>
<td>dc:language hasValue &quot;en-us&quot;</td>
</tr>
<tr>
<td>dc:rights hasValue &lt;<a href="http://deri.ie/privacy.html">http://deri.ie/privacy.html</a>&gt;</td>
</tr>
<tr>
<td>version hasValue &quot;$Revision 0.1 $&quot;</td>
</tr>
<tr>
<td>endNonFunctionalProperties</td>
</tr>
<tr>
<td>concept station</td>
</tr>
<tr>
<td>nonFunctionalProperties</td>
</tr>
<tr>
<td>dc:description hasValue &quot;concept of station, containing the code of the station, and two attributes showing whether this station is the start or the end point of a trip&quot;</td>
</tr>
<tr>
<td>endNonFunctionalProperties</td>
</tr>
</tbody>
</table>
code ofType xsd:string
startpoint ofType xsd:boolean
endpoint ofType xsd:boolean
concept date subConceptOf xsd:date
    nonFunctionalProperties
        dc:description hasValue "concept of date"
    endNonFunctionalProperties
concept myRoute
    nonFunctionalProperties
        dc:description hasValue "concept of myRoute, containing the source and the destination locations, and the date of the trip"
    endNonFunctionalProperties
sourceLocation ofType station
destinationLocation ofType station
onDate ofType date
concept person
    nonFunctionalProperties
        dc:description hasValue "concept of person, containing only the name of the person"
    endNonFunctionalProperties
name ofType xsd:string
concept time subConceptOf xsd:time
    nonFunctionalProperties
        dc:description hasValue "concept of time"
    endNonFunctionalProperties
concept price subConceptOf xsd:integer
    nonFunctionalProperties
        dc:description hasValue "concept of price"
    endNonFunctionalProperties
concept creditCard
    nonFunctionalProperties
        dc:description hasValue "concept of credit card, containing its number, owner and expiration date"
    endNonFunctionalProperties
number ofType xsd:integer
owner ofType person
expirationDate ofType xsd:date
concept creditCardAcknowledgement
    nonFunctionalProperties
dc:description hasValue "acknowledgement for receiving the
creditCard"

endNonFunctionalProperties
confirmation ofType creditCard
concept reservation

nonFunctionalProperties
dc:description hasValue "concept of reservation, containing
its number, owner, and the route for which this reservation was made"

endNonFunctionalProperties
reservationNumber ofType xsd:integer
reservedRoute ofType routeOnDate
reservationHolder ofType person

The VTA requestor Choreography Ontology is:

**Table 12: VTA Requestor Choreography**

<table>
<thead>
<tr>
<th>VTA Requestor Choreography</th>
</tr>
</thead>
<tbody>
<tr>
<td>namespace</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>&lt;&lt;<a href="http://www.wsmo.org/TR/d13/d13.7/ontologies/VTAResquestorCh">http://www.wsmo.org/TR/d13/d13.7/ontologies/VTAResquestorCh</a> &gt;&gt;</td>
</tr>
<tr>
<td>vtar:</td>
</tr>
<tr>
<td>&lt;&lt;<a href="http://www.wsmo.org/TR/d13/d13.7/ontologies/VTAResquestorOnt">http://www.wsmo.org/TR/d13/d13.7/ontologies/VTAResquestorOnt</a> &gt;&gt;</td>
</tr>
<tr>
<td>dc:</td>
</tr>
<tr>
<td>&lt;<a href="http://purl.org/dc/elements/1.1#">http://purl.org/dc/elements/1.1#</a>&gt;</td>
</tr>
<tr>
<td>ontology</td>
</tr>
<tr>
<td>&lt;&lt;<a href="http://www.wsmo.org/TR/d13/d13.7/ontologies/VTAResquestorCh">http://www.wsmo.org/TR/d13/d13.7/ontologies/VTAResquestorCh</a> &gt;&gt;</td>
</tr>
<tr>
<td>nonFunctionalProperties</td>
</tr>
</tbody>
</table>
| dc:title hasValue "Virtual Travel Agency Requestor Choreography"
| dc:creator hasValue "Emilia"
| dc:description hasValue "an ontology for describing trip
reservation requestor choreography related knowledge"
| dc:publisher hasValue "DERI International"
| dc:contributor hasValues "Adrian"
| dc:type hasValue <<http://www.wsmo.org/2004/d2/#ontology>>
| dc:format hasValue "text/html"
| dc:language hasValue "en-us"
| dc:rights hasValue <<http://deri.ie/privacy.html>>
| version hasValue "$Revision 0.1 $"

endNonFunctionalProperties
importedOntologies
     <<http://www.wsmo.org/TR/d13/d13.7/ontologies/VTAResquestorOnt >>
concept station subConceptOf vtar:station
nonFunctionalProperties
    mode hasValue controlled
endNonFunctionalProperties
concept date subConceptOf vtar:date
    nonFunctionalProperties
    mode hasValue controlled
endNonFunctionalProperties
concept myRoute subConceptOf vtar:myRoute
    nonFunctionalProperties
    mode hasValue out
endNonFunctionalProperties
concept person subConceptOf vtar:person
    nonFunctionalProperties
    mode hasValue out
endNonFunctionalProperties
concept time subConceptOf vtar:time
    nonFunctionalProperties
    mode hasValue in
endNonFunctionalProperties
concept price subConceptOf vtar:price
    nonFunctionalProperties
    mode hasValue in
endNonFunctionalProperties
concept creditCard subConceptOf vtar:creditCard
    nonFunctionalProperties
    mode hasValue out
endNonFunctionalProperties
concept creditCardAcknowledgement subConceptOf vtar:creditCardAcknowledgement
    nonFunctionalProperties
    mode hasValue in
endNonFunctionalProperties
concept reservation subConceptOf vtar:reservation
    nonFunctionalProperties
    mode hasValue in
endNonFunctionalProperties

The guarded transitions for the requestor are presented in the following table:
Table 13: VTA Requestor Choreography Transition Rules

<table>
<thead>
<tr>
<th>VTA Requestor Choreography Transition Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>/* creates an instance of my route, assuming that two instances of station and an instance of date are already created; since both station and date have the value of mode set to controlled, the requestor does not expect any input in order to create the instance of myRoute*/</td>
</tr>
<tr>
<td><code>?x memberOf myRoute{</code></td>
</tr>
<tr>
<td><code>  sourceLocation hasValue ?sourceLocation_,</code></td>
</tr>
<tr>
<td><code>  destinationLocation hasValue ?destinationLocation_,</code></td>
</tr>
<tr>
<td><code>  onDate hasValue ?onDate_ [&lt;-</code></td>
</tr>
<tr>
<td><code>    ?sourceLocation memberOf vtarc:station and</code></td>
</tr>
<tr>
<td><code>    ?endLocation memberOf vtarc:station and</code></td>
</tr>
<tr>
<td><code>    ?onDate memberOf vtarc:date.</code></td>
</tr>
<tr>
<td>/* an instance of time is expected (the departure time), after the instance of myRoute was sent to the service*/</td>
</tr>
<tr>
<td><code>?x memberOf time[] &lt;-</code></td>
</tr>
<tr>
<td><code>  ?myRoute memberOf vtarc:myRoute.</code></td>
</tr>
<tr>
<td>/* an instance of price is expected (the price of a ticket), after the instance of myRoute was sent to the service*/</td>
</tr>
<tr>
<td><code>?x memberOf price[] &lt;-</code></td>
</tr>
<tr>
<td><code>  ?myRoute memberOf vtarc:myRoute.</code></td>
</tr>
<tr>
<td>/* after receiving the time and price instances, the requestor creates an instance of the concept creditCard*/</td>
</tr>
<tr>
<td><code>?x memberOf creditCard{</code></td>
</tr>
<tr>
<td><code>  number hasValue ?number_,</code></td>
</tr>
<tr>
<td><code>  owner hasValue ?owner_,</code></td>
</tr>
<tr>
<td><code>  expirationDate hasValue ?expirationDate_ [&lt;-</code></td>
</tr>
<tr>
<td><code>    ?time_ memberOf vtarc:time and</code></td>
</tr>
<tr>
<td><code>    ?price_ memberOf vtarc:price and</code></td>
</tr>
<tr>
<td><code>    ?number_ memberOf xsd:integer and</code></td>
</tr>
<tr>
<td><code>    ?owner_ memberOf vtarc:person and</code></td>
</tr>
<tr>
<td><code>    ?expirationDate_ memberOf vtarc:date.</code></td>
</tr>
<tr>
<td>/* the requestor is expecting the confirmation of the credit card (internally it will check whether all the attributes from the confirmedCreditCard's instance have the same value as the attribute of the creditcard's instance, but these computations don't have to be public)*/</td>
</tr>
<tr>
<td><code>?x memberOf creditCardAcknowledgement{</code></td>
</tr>
<tr>
<td><code>  confirmation hasValue ?confirmation_ [&lt;-</code></td>
</tr>
<tr>
<td><code>    ?confirmation_ memberOf vtarc:creditCard.</code></td>
</tr>
</tbody>
</table>
/*after receiving the confirmation of the credit card, the requestor will send the name of the person who needs the reservation*/

?x memberOf person[
   name hasValue ?name_] <-
   ?name memberOf xsd:string and
   ?creditCardAcknowledgement_ memberOf
   vtarc:creditCardAcknowledgement.
/*an instance of reservation is expected, after instances of myRoute and person have been created*/

?x memberOf reservation[
   reservationNumber hasValue ?reservationNumber_,
   route hasValue ?route_,
   reservationHolder hasValue ?reservationHolder_] <-
   ?reservationNumber_ memberOf xsd:integer and
   ?route_ memberOf vtarc:myRoute and
   ?reservationHolder_ memberOf vtarc:person.

In this scenario, the exchange of messages is as follows:

1. Send an instance of myRoute
2. Receive an instance of price and an instance of time (the order of receiving these two messages is not important from the requestor point of view)
3. Send an instance of creditCard
4. Receive an instance of confirmedCreditCard
5. Send the details of the person
6. Receive the reservation instance

Mediation Between the Service and the Requestor

Considering the previously described choreographies, Figure 8 presents a graphical representation of the communication patterns of the two participants:
The Process Mediator has to translate any incoming message in terms of the targeted partner ontology, and to decide whether, based on this message, it can generate any message, for either one of the two partners, that could trigger an action (not necessarily to change a state, but to eliminate one condition for changing a state). In what follows, we analyze step by step the flow of messages and the internal transformations that take place on the requestor and provider side.

1. PM receives an instance of myRoute – after translating this instance in terms of the service’s ontology, the PM will obtain two instances of station and one of date. Conforming to the choreography ontology of the requestor, all these three instances are expected, but the guarded transitions show that only two of them (the instances of station) are expected at this phase. The instance of date is not yet expected since the first condition (?forRoute_ memberOf vtasc:route) of the rule that checks whether an instance of date exists is not satisfied. As a consequence, the PM will send the two instances of station and store the instance of date.

2. Internally, the provider creates the instance of route, and sends it.

3. After translating the route’s instance in terms of the requestor’s ontology, and analyzing the two choreographies, the PM discards the instance of route (nobody is expecting any information contained by that instance) and sends the previously stored instance of date, while at the same time deleting it from its internal repository.

4. The provider creates an instance of routeOnDate and sends it.

5. PM translates the routeOnDate in terms of the requestor’s ontology into two instances of station, an instance of time and one of price. Nobody is expecting any more instances of station, so these two can be deleted. The price and time instances are sent.

6. The requestor sends the details of a credit card (an instance of the creditCard concept).
7. Based on the two choreographies, the PM sends the corresponding instance of a creditCard for the provider, and creates an acknowledgement (instance of creditCardAcknowledgement) to be sent to the requestor.

8. The instance of person is send by the requestor.

9. PM sends the corresponding instance of person.

10. The service sends an instance of the reservation concept.

11. PM send the reservation instance to the requestor. Since at this moment none of the two participants is expecting anything more, the PM considers the communication over.
ANNEX2: EXECUTION SEMANTICS FOR INTERACTION MISMATCHES

In [3], the authors formally specify what we call a mediator here, as a so-called “Virtual Provider” (VP) connecting the interfaces of (at least) two business partners. The VP consists of a procedural description of a machine capable of sending and receiving messages, and executing a formally defined interaction description. Although its structure seems similar, the VP is used here to concisely define the execution semantics of the mediation patterns given in Sect. 2.2.1, rather than replicating the architecture of the PROCESS Mediator. Thus, this section points out, how the depicted interaction patterns are to be understood as message flows occurring in real systems’ interactions.

To achieve this formal description of message flow dynamics, [3] uses ASMs\(^{13}\) to define the processing of arbitrary messages in an abstract way. This high-level description is especially appropriate for the definition of generic interaction patterns, because one would not want to give a too specialized description of the patterns, rather than a high-level description being applicable to many specific messaging system implementations. Although operating on an abstract level, ASMs comprise an explicit execution semantics for the pseudo-code-like rules it consists of. For details on ASMs, see [5].

Since the VP is given as an ASM description, it also entails an explicit execution semantics through the ASM definition. It is aimed at providing a general frame for defining interaction patterns. This template is to be further specialized in order to cover certain specific interaction patterns.

For modelling generic interaction patterns of message-based systems, two aspects are needed:

1. An explicit formalization of (the endpoints of) a messaging system, and
2. A formal template definition for the conversion of messages.

These aspects are depicted in Figure 9.

![VP Architecture Diagram](image)

**Figure 9: VP Architecture**

\(^{13}\) ASM: “Abstract State Machine“, see [5].
The Virtual Provider consists of four messaging machines. RECEIVEREQ receives initial request messages from a requestor side and relates them to some internal representation of the current request. This will be detailed later on. SENDANSW is used to return a final response message for the current request to its initiator. These are the machines being active in the beginning and the ending of a mediation process. During a mediation process, the machines SENDREQ and RECEIVEANSW are used to communicate with arbitrary business partners as required for the processing of the current request. The structure of this communication is controlled by the PROCESS machine (Point 2 of the above enumeration).

The interplay of all machines is synchronized by decorated objects jointly used by multiple machines. One of those objects is the request object. After the RECEIVEREQ machine has created such a new request object, and decorated its state with a starting flag, the PROCESS machine starts executing the current mediation following a certain scheme based on a decompilation of the original request:

- A request object (currReqObj) is decompiled into a set of sequential subrequests (SeqSubReq(currReqObj)).
- A seqSubReq out of the set SeqSubReq(currReqObj) is decompiled into a set of parallel subrequests (ParSubReq(seqSubReq)).

The execution semantics for these set definitions of a decompiled request is given by the ASM code below. That is, the VP will iterate through the sequential subrequests and simultaneously send the messages generated from the parallel decompilation of the current sequential subrequest through the SENDREQ machine. After this concurrent sending, the PROCESS machine is waiting for responses and puts these into an AnswerSet of the current sequential request object. The AnswerSets of all sequential subrequests can in the end (amongst other information accessible via the current request object) be used to generate a return message to the initial requestor:

```
PROCESS(currReqObj) =
    if status(currReqObj) = started then
        INITIALIZE(seqSubReq(currReqObj))
        status(currReqObj) := subReqProcessg
    if status(currReqObj) = subReqProcessg then
        if FinishedSubReqProcessg then
            COMPILEOUTANSWMSG from currReqObj
            status(currReqObj) := deliver
        else
            StartNextRound(ITERATESUBREQPROCESSG)
    where
        INITIALIZE(seqSubReq) =
        seqSubReq := FstSubReq(SeqSubReq(currReqObj))
    FinishedSubReqProcessg =
        seqSubReq(currReqObj) = Done(SeqSubReq(currReqObj))
    COMPILEOUTANSWMSG from currReqObj =
        if AnswerToBeSent(currReqObj) then
            SentAnswerToMailer (outAnswer2Mssg(outAnswer(currReqObj))) := true
    StartNextRound(M) =
```

38
\[ \text{status(currReqObj)} := \text{initStatus(M)} \]

\textbf{ITERATE\text{SUB}	ext{RE}Q\text{PRO}\text{CESSG}} =

\textbf{if} \text{status(currReqObj)} = \text{initStatus(ITERATE\text{SUB}	ext{RE}Q\text{PRO}\text{CESSG})} \text{ then}

\textbf{FEED\text{SEND\text{REQ} with ParSubReq(seqSubReq(currReqObj))}}

\textbf{INITIALIZE(AnswerSet(seqSubReq(currReqObj)))}

\textbf{status(currReqObj)} := \text{waitingForAnswers}

\textbf{if} \text{status(currReqObj)} = \text{waitingForAnswers} \text{ then}

\textbf{PROCEED\text{TO\text{NEXT\text{SUB}\text{RE}Q}}}

\textbf{status(currReqObj)} := \text{subReqProccessg}

\textbf{where}

\textbf{AllAnswersReceived} =

\textbf{let seqSubReq = seqSubReq(currReqObj) in}

\textbf{for each req in ToBeAnswered(ParSubReq(seqSubReq))}

\textbf{there is some answ in AnswerSet(seqSubReq)}

\textbf{PROCEED\text{TO\text{NEXT\text{SUB}\text{RE}Q}} =}

\textbf{seqSubReq(currReqObj) := [NxtSubReq(SeqSubReq(currReqObj)), seqSubReq(currReqObj), AnswerSet(currReqObj)]}

As highlighted in the above ASM code for the \text{PROCESS} machine (and its submachine \text{ITERATE\text{SUB}	ext{RE}Q\text{PRO}\text{CESSG}}), the VP uses two iterator functions and a termination symbol to iterate through the sequential subrequests:

1. \text{FstSubReq} for retrieving the first iterator element from the set.
2. \text{NxtSubReq} for retrieving a following element from the set, given a current element of \text{SeqSubReq}.\textsuperscript{14}

- \text{Done} denotes a termination symbol, not element of \text{SeqSubReq}.

With the above definitions, the frame for defining certain interaction patterns is set. To actually describe specific patterns, one needs to refine the abstract declarations given in the ASM code which are still left open for specialization. Those are mainly the iterator functions of the current request object, the expected set of answers for each sequential subrequest, and the generation of the response to be sent back to the initial requestor. The only remaining issue is to relate incoming request messages with the internally used request object. Thus, as deferred before, we will now have a closer look at the \text{RECEIVE\text{RE}Q} machine, which also contains the refinement spots for incorporating the specific definitions making the VP conducting a certain interaction mediation pattern.

By the definition of the \text{RECEIVE\text{RE}Q} machine, the VP is able to receive a first initial request message which immediately creates a new request object. However, it might be required, that additional incoming request messages are needed to trigger the mediation of the current request. Thus, it is left open to a refinement of the \text{RECEIVE\text{RE}Q} machine,

\textsuperscript{14} Actually, the \text{NxtSubReq} function is defined for a third argument, namely \text{AnswerSet}. Thus, more sophisticated interaction patterns can be defined whose mediation depends on the reaction of the business partners during the mediating process. However, for the elaboration given here, this modelling possibility is not needed and thus neglected for the sake of simplicity. An example using this feature is given in [3].
to decorate the current request object with a starting flag triggering the PROCESS machine. This might either be done in the

- **INITIALIZE** submachine, which is called when the first request message is received, or in the
- **REFRESHREQOBJ** submachine, which is called for each subsequently incoming request message related to the same current request object.

With these definitions, the user is able to describe stateful, blocking behaviour of a mediation pattern depending on the incoming request messages.

```
RECEIVERQ(inReqMsg) =
  if ReceivedReq(inReqMsg) then
    if NewRequest(inReqMsg) then
      CREATENEWREQOBJ(inReqMsg,ReqObj)
    else let r = prevReqObj (inReqMsg) in
      REFRESHREQOBJ(r, inReqMsg)
  where
    CREATENEWREQOBJ(m,R) =
      let r = New(R)\^15 in
        INITIALIZE(r,m)
```

The above is a brief introduction into the fundamentals of the VP that are necessary to understand the mediation pattern specifications of the following sections. For a more detailed elaboration on the concepts of the VP, the reader might want to refer to [3]. The content of the upcoming sections showing refinements of the VP concept to explicitly specify the execution semantics of the interaction patterns from Sect. 2.2.1 are entirely taken from [2]. Please note, that each section contains an own refinement of the VP not related to the code contained in the other sections (unless otherwise noted). In other words, each of the upcoming sections defines a new, more specialized (but still abstract) VP’ describing the execution semantic of the respective interaction mismatch pattern.

**Pattern a)**

The data set “A” transported by the first message from “Business Partner 1” is discarded, only data set “B” transmitted by the second incoming message is being forwarded to “Business Partner 2”. The refined VP is defined as instantiated as follows. The VP has to INITIALIZE the newReqObj by preventing its PROCESS machine from being started yet. No data need to be recorded since only the data of the second message is relevant.

```
INITIALIZE(newReqObj, inReqMsg) =
  status(newReqObj) := initStatus(PROCESS)
```

---

\(^{15}\) New is assumed to provide a sufficiently fresh element in the indicated domain at each call.
The subsequently incoming message containing data set “B” first needs to be related to the same newReqObj by prevReqObj(inReqMsg) while evaluating NewRequest(inReqMsg) to false. Second, the data set is extracted from the message and stored as the only element of SeqSubReq(prevReqObj), to be forwarded as the only parallel subrequest in ParSubReq(dataSet) to “Business Partner 2”. Then, the VP is started without having to send or to receive any answer message. For brevity, we write prevReqObj for prevReqObj(inReqMsg).

\[
\text{REFRESHREQOBJ}(\text{prevReqObj}, \text{inReqMsg}) =
\begin{align*}
&\text{let dataSet} = \text{extractDataSet(inReqMsg)} \text{ in} \\
&\quad \text{SeqSubReq}(\text{prevReqObj}) := \{ \text{dataSet} \} \\
&\quad \text{Done}( \{ \text{dataSet} \} ) := \text{nil} \\
&\quad \text{FstSubReq}( \{ \text{dataSet} \} ) := \text{dataSet} \\
&\quad \text{NxtSubReq}( \{ \text{dataSet}, \_ \_ \_ \} ) := \text{nil} \\
&\quad \text{ParSubReq}(\text{dataSet}) := \{ \text{dataSet} \} \\
&\quad \text{ToBeAnswered}( \{ \text{dataSet} \} ) := \text{empty set} \\
&\quad \text{AnswToBeSent}(\text{prevReqObj}) := \text{false} \\
&\quad \text{status(\text{prevReqObj})} := \text{started}
\end{align*}
\]

The thus instantiated VP machine starts in status \textit{started} after having received the two incoming messages and eventually stays in status \textit{deliver} after having sent the data set “B” to the mailer \texttt{SENDREQ} from where it is sent to “Business Partner 2”.

\textbf{Pattern b)}

This interaction pattern can be seen as an extension of Pattern a). In addition to just forwarding data set “B” to “Business Partner 2”, it initializes by also storing data set “A”, without triggering VP to get started\(^\text{17}\), for sending it after “B”:

\[
\text{INITIALIZE(\text{newReqObj}, \text{inReqMsg}) =}
\begin{align*}
&\text{status(\text{newReqObj})} := \text{initStatus(PRECESS)} \\
&\text{dataSetA(\text{newReqObj})} := \text{extractDataSet(inReqMsg)} \\
&\text{REFRESHREQOBJ(\text{prevReqObj}, \text{inReqMsg}) =}
\begin{align*}
&\text{let dataSetB} = \text{extractDataSet(inReqMsg)} \text{ in}
\end{align*}
\]

\[
\begin{align*}
&\quad \text{SeqSubReq}(\text{prevReqObj}) := \{ \text{dataSetB}, \text{dataSetA(\text{prevReqObj})} \} \\
&\quad \text{Done}( \{ \text{dataSetB}, \text{dataSetA(\text{prevReqObj})} \} ) := \text{nil} \\
&\quad \text{FstSubReq}( \{ \text{dataSetB}, \text{dataSetA(\text{prevReqObj})} \} ) := \text{dataSetB} \\
&\quad \text{NxtSubReq}( \{ \text{dataSetB}, \text{dataSetA(\text{prevReqObj})} \} , \text{dataSetB}_\text{\_\_\_} ) := \text{dataSetA(\text{prevReqObj})} \\
&\quad \text{NxtSubReq}( \{ \text{dataSetB}, \text{dataSetA(\text{prevReqObj})} \} , \text{dataSetA(\text{prevReqObj})}_\text{\_\_\_} ) := \text{nil}
\end{align*}
\]

\(^{\text{16}}\) We do neither specify an implementation for \text{prevReqObj(inReqMsg)} nor \text{NewRequest(inReqMsg)} here, since their behaviour depends on the real messaging system used. This aspect is an abstract parameter of the specification of this abstract interaction pattern.

\(^{\text{17}}\) Note: \text{initStatus(VP)} does not equal \textit{started}.
```
ParSubReq(d) := { d }  ToBeAnswered(d) := empty set,
for d = dataSetA(prevReqObj), dataSetB
AnswToBeSent(prevReqObj) := false
status(prevReqObj) := started

Pattern c)
In this pattern, “Business Partner 1” sends both data sets “A” and “B” in the same
message, whereas “Business Partner 2” expects to receive them in two different
messages. VP does not need to wait for any subsequent incoming messages. It therefore
initializes by saving both data sets for their later sequential sending and directly enables
the PROCESS machine getting started:

INITIALIZE(newReqObj, inReqMsg) =
    let dataSet = extractDataSet (inReqMsg) for i = A, B in
    SeqSubReq(newReqObj) := { dataSetA, dataSetB }
    Deln( { dataSetA, dataSetB } ) := nil
    FstSubReq( { dataSetA, dataSetB } ) := dataSetA
    NxtSubReq( { dataSetA, dataSetB }, dataSetA, __ ) := dataSetB
    NxtSubReq( { dataSetA, dataSetB }, dataSetB, __ ) := nil
    ParSubReq(d) := { d }
    ToBeAnswered(d) := empty set, for d = dataSetA, dataSetB
    AnswToBeSent(newReqObj) := false
    status(newReqObj) := started

Pattern d)
The specialization for this pattern can be easily adapted from Pattern b) by just
combining the received data sets into one single message to be sent to “Business
Partner 2”. We therefore indicate only the changes in the macro REFRESHREQOBJ:

```
FstSubReq( { { dataSetA(prevReqObj), dataSetB } } ) :=
    { dataSetA(prevReqObj), dataSetB }
NxtSubReq( { { dataSetA(prevReqObj), dataSetB } },
    { dataSetA(prevReqObj), dataSetB }, ) := nil
ParSubReq( { dataSetA(prevReqObj), dataSetB } ) :=
    { combine(dataSetA(prevReqObj), dataSetB) }
```

Pattern e)
In this interaction pattern, a reliable medium for transmitting messages from the VP to
“Business Partner 2” is apparently assumed. After receipt of the message carrying data
set “A”, it is saved and forwarded without waiting for further messages. After
FinishedSubReqProcessg, the VP creates an appropriate outAnswer(newReqObj)
containing data set “AckA” to be returned to “Business Partner 1”:

```
INITIALIZE(newReqObj, inReqMsg) =
    let dataSet = extractDataSet(inReqMsg) in
```

42
SeqSubReq(newReqObj) := \{ dataSet \}
Done( \{ dataSet \} ) := nil
FstSubReq( \{ dataSet \} ) := dataSet
NxtSubReq( \{ dataSet \}, __, __ ) := nil
ParSubReq(dataSet) := \{ dataSet \}
ToBeAnswered( \{ dataSet \} ) := empty set
\footnote{Realizes the assumption of a reliable transmission from VP to “Business Partner 2”.
AnswToBeSent(newReqObj) := true
status(newReqObj) := started