Web Services –
DIAgnosability, MONitoring
and Diagnosis

WS-DIAMOND

IST-516933

Deliverable D1.1

Requirements, application scenarios, overall architecture, and test/validation specification, common working environment and standards at Milestone M1

Version: M1.0

Report Preparation Date:	01.03.2006
Classification:	Public
Contract Start Date:	1.9.2005
Duration:	30 months
Project Co-ordinator:	University of Torino (I)
Partners:	Vrije Universiteit Amsterdam (NL)
	Politecnico of Milano (I)
	Université of Paris 13 (F)
	University of Klagenfurt (A)
	Laboratoire d'Analyse et d'Architecture des Systèmes - National Center for Scientific Research (F)
	Université de Rennes 1 (IRISA) (F)
	Universtity of Vienna (A)

Executive Summary

The aim of this report is to give an overview on the approaches, techniques and tool in the domain of web services that currently exist in scientific community and how to apply them for diagnosis and repair tasks. These issues are the base for our future research: each diagnosis and repair techniques must be compatible with existing standards in this domain, and append them.

Application scenarios provide detailed description of example situations for real life, where discussed problems may happen. They include models of workflows, created according to the existing standard and languages that are used for web-services workflow modelling, and heuristic descriptions of failure situations within workflow and repair actions that have to be provided. These descriptions show how diagnosis and repair processes have to be done, and which results are expected.

Current standards and test cases give us initial requirements for diagnosis/repair solution. Four main groups of requirements were considered: requirements for web services composition and execution for self-healing environments, for model-based diagnosis and repair, for design for diagnosability and repairability, and for semi-automatic acquisition of semantic mark-up for web services.

Finally, the preliminary architecture, in chapter 5, show possible structures of diagnosis, repair and monitoring solution modules and their place in the common WSDIAMOND-enabled Web Services execution environment. Two main approaches are considered – centralised and decentralised.

Table of Contents

1 INTRODUCTION 8

2 WORKING ENVIRONMENT 10

2.1 STANDARDS 10

2.1.1 WEB SERVICES STANDARDS 10

2.1.2 WEB SERVICES COMPOSITION, ORCHESTRATION, AND EXECUTION 13

2.1.3 CONCLUSIONS 42

2.2 SOFTWARE PLATFORMS 43

2.2.1 SELECTION CRITERIA 44

2.2.2 OVERVIEW 45

2.2.3 COMMON WORKING ENVIRONMENT 59

3 APPLICATION SCENARIOS 61

3.1 TEST CASE: FOOD SHOP 61

3.1.1 WORKFLOW 61

3.1.2 EXCEPTIONS 67

3.1.3 PRELIMINARY MODEL OF THE PROCESS 68

3.1.4 DIAGNOSIS PROCESS 78

3.1.5 REPAIR STAGE 79

3.1.6 COMPARISON 84

3.2 TEST CASE: COOPERATIVE REVIEW 85

3.2.1 WORKFLOW 85

3.2.2 ARCHITECTURE AND WORKFLOW 86

3.2.3 EXCEPTIONS 103

3.2.4 DIAGNOSIS PROCESS 109

3.2.5 REPAIR STAGE 112

3.2.6 EVALUATION AND CONCLUSIONS 113

3.3 TEST CASE: TRAVEL SERVICES 117

3.3.1 WORKFLOW 118

3.3.2 EXCEPTIONS 125

3.3.3 DIAGNOSIS AND REPAIR STAGE 129

3.3.4 COMPARISON 129

4 GENERAL REQUIREMENTS FOR THE PROPOSED SOLUTION 132

4.1 REQUIREMENTS FOR WEB SERVICE COMPOSITION AND EXECUTION FOR SELF-HEALING ENVIRONMENTS 132

4.1.1 GENERAL SERVICE ASPECTS 132

4.1.2 SEMANTIC ANNOTATIONS 136

4.2 REQUIREMENTS FOR MODEL-BASED DIAGNOSIS AND REPAIR OF COOPERATIVE WEB SERVICES. 139

4.2.1 CURRENT TRENDS IN MODEL-BASED DIAGNOSIS AND REPAIR. 139
4.2.2 MODEL-BASED DIAGNOSIS AND REPAIR FACED TO WEB SERVICES 140
4.2.3 REQUIREMENTS 140
4.3 REQUIREMENTS FOR DESIGN FOR DIAGNOSABILITY AND REPAIRABILITY 144
4.3.1 MODELS FOR DIAGNOSABILITY AND REPAIRABILITY 144
4.3.2 ARCHITECTURE OF THE SUPERVISION SYSTEM, IMPACTS ON DIAGNOSABILITY AND REPAIRABILITY 148
4.3.3 ON-LINE VERSUS OFF-LINE SUPERVISION ACTIVITIES FOR WEB SERVICES 148
4.3.4 DESIGN REQUIREMENTS 149
4.4 REQUIREMENTS FOR SEMI-AUTOMATIC ACQUISITION OF SEMANTIC MARKUP FOR WEB SERVICES150
4.4.1 GATHERING FUNCTIONAL PROPERTIES FROM STATIC INFORMATION 151
4.4.2 GATHERING FUNCTIONAL PROPERTIES FROM EXECUTING A SINGLE SERVICE 151
4.4.3 GATHERING DATA FROM EXECUTING A COMPOSITION OF SERVICES 151

5 PRELIMINARY ARCHITECTURE 153
5.1 DIAGNOSIS ARCHITECTURES 154
5.1.1 THE DIAGNOSER IS A WEB SERVICE 155
5.1.2 THE DIAGNOSIS ARCHITECTURE 156
5.2 CENTRALISED DIAGNOSIS AND RECOVERY ARCHITECTURE 156
5.3 DECENTRALISED/SUPERVISED DIAGNOSIS AND CENTRALISED RECOVERY ARCHITECTURE 157
5.4 DISTRIBUTED DIAGNOSIS AND DECENTRALISED RECOVERY ARCHITECTURE 157

6 SUMMARY AND OUTLOOK 164

7 GLOSSARY OF TERMS 165

APPENDIX A. FOODSHOP EXAMPLE BPEL CODE 167

REFERENCES 172

List of Tables

Table 1: Web Service orchestration vs. choreography ... 21

Table 2. Software tools evaluation table ... 60

Table 3: Levels of faults occurrence, type of fault, and examples ... 81

Table 4 : QoS parameters associated with conferences and reviewers. ... 86

Table 5 : Correspondences between diagnosability levels and repair actions ... 147

Table 6 : Consistent situations for diagnosability and repairability ... 149

List of figures

Figure 1: Service Oriented Architecture. 11

Figure 2 : WSDL elements 12

Figure 3 : A contextualized view on currently used terminology; the two main nomenclatures concerning respectively internal and external perspective on Web Services can further be specialized by actor and execution time 14

Figure 4 : Web Service composition-oriented protocol stack of vendor-specific and standardized protocols and languages. Within the composition layer, we propose BPML in on top of WSCI as they share a common process model. However, other executable BPM languages could be adopted as well [DP06] 15

Figure 5 : Emergence and evolution of today's principal standards and languages concerning WS composition. The figure tries to reflect the official release or publication dates of the specifications (at the best of the authors' knowledge), first appearance of or discussions about them could differ from the proposed dates. XLANG and WSFL are not treated in this paper; they heavily contributed to BPEL and are reported for the sake of completeness [DP06] 17

Figure 6 : Ordered message exchange between a Web Service and its client. 18

Figure 7 : Interaction involving multiple Web Services; messages depend semantically and chronologically from one another. 19

Figure 8: Orchestration refers to an executable process, choreography tracks the message sequences between parties and sources [Pel03] 22

Figure 9 : Vehicle-process as activity diagram 25

Figure 10: Vehicle -process by means of a statechart 26

Figure 11: Vehicle-process specified by means of a Petri net 27

Figure 12: vehicle-process as activity hierarchies 29

Figure 13: Choreography described with UML sequence diagrams 30

Figure 14: Explicit data flow approach 38

Figure 15 : Axis server architecture 46

Figure 16: Axis client architecture 47

Figure 17: MAIS registry architecture 48

Figure 18: Service ontology structure 49

Figure 19: Engine Architecture 50

Figure 20: ActiveWebflow snapshot 51

Figure 21 : the BPWS Runtime architecture 52

Figure 22: LAMP Application Server. 54

Figure 23: Oracle BPEL Process Manager. 56

Figure 24: BPEL Maestro tool. 58

Figure 25: JBOSS architecture 59

Figure 26: FoodShopping example actors 61

Figure 27: CUSTOMER workflow 63

Figure 28: SHOP workflow 64

Figure 29: WAREHOUSE workflow 66

Figure 30: SUPPLIER workflow 67

Figure 31: Definition of Supplier service *69*
Figure 32: Definition of Shop Service *70*
Figure 33: Definition of Customer Service *71*
Figure 34: Definition of Warehouse Service *72*
Figure 35: BPEL process model /1 .... *74*
Figure 36: BPEL process model /2 .... *75*
Figure 37: BPEL process model /3 .... *76*
Figure 38: BPEL process model /4 .... *77*
Figure 39: General architecture of the cooperative reviewing system. *87*
Figure 40: Looking for conferences. *88*
Figure 41: Sequence diagram for conference search activity. *89*
Figure 42: Looking for reviewers. *90*
Figure 43: Sequence diagram for reviewers search activity. *91*
Figure 44: Sequence diagram for author inscription activity. *92*
Figure 45: Paper submission *93*
Figure 46: Sequence diagram for paper submission activity. *94*
Figure 47: Paper assignment *95*
Figure 48: Sequence diagram for paper assignment activity. *96*
Figure 49: Getting reviewers' reports *97*
Figure 50: Sequence diagram for report transmission activity. *98*
Figure 51: Getting approval decision *98*
Figure 52: Sequence diagram for author notification activity. *100*
Figure 53: Getting final papers *100*
Figure 54: Sequence diagram for final paper submission activity. *101*
Figure 55: General activity diagram *102*
Figure 56: Classification of Considered Mismatches *103*
Figure 57 QoS parameter classification *104*
Figure 58 :QAC contract parameters *108*
Figure 59 : Cooperative review Mismatches classification *108*
Figure 60 : General architecture for diagnosis and reparation *110*
Figure 61 : Proposed architecture *111*
Figure 62 : Sequence Diagram for diagnosis and repair *112*
Figure 63 : Diagnosis and Recovery module *113*
Figure 64 :The architecture of the travel agent Example *117*
Figure 65 : Graphical representation of a WSDL operation *118*
Figure 66 : Cutomer WSDL interface *119*
Figure 67 : Travel agent WSDL interface *120*
Figure 68 : Hotel/Flight WSDL interfaces *121*

Figure 69 : Notation UML vs BPEL constructors *122*
Figure 70 : The Customers BPEL process *124*
Figure 71 : The travel agent process behavior *126*
Figure 72 :Abstract and instantiated services *133*
Figure 73 : Schema of services environment *133*
Figure 74 : A Travel-Service example, its Management Interface, process flow, and annotation for Quality *134*
Figure 75 : Negotiation Handlers *135*
Figure 76 : A Virtual Travel Agency Example *138*
Figure 77 : A Multi-Channel Constrained Example *139*
Figure 78 : The logical organization of the diagnosis and repair task *141*
Figure 79 : Petri net, automaton, and process algebra models of a simple system *145*
Figure 80 : WS-Diamond environment *153*
Figure 81 : Cooperation of modules *154*
Figure 82 : Self-healing Web services framework model *155*
Figure 83 : Centralised diagnosis and centralised recovery *156*
Figure 84 : Decentralised diagnosis and centralised recovery *157*
Figure 85 : Distributed diagnosis and decentralised recovery *158*
Figure 86 : Generic Web services execution environment *160*
Figure 87 : Cooperation of the modules inside a node and with the execution environment *161*
Figure 88 : Sequence diagram of modules interaction *162*

1 Introduction

This document reports on the activities carried on in the first phase of the project, leading to Milestone 1 at month 6. The document is organized in four main parts:

1. Definition of the working environment
2. Application scenarios
3. Requirements
4. Preliminary architecture

The first three of such item corresponds to the first three results in Milestone M1 (the fourth and last result of Milestone M1 is outside the scope of this document, being the subject of D1.2). The last item defines a preliminary architecture for the surveillance platform that will be developed in the project.

The definition of the working environment implied a thorough analysis of the state of the art in Web Services languages and standards. The report summarizes the current situation as regards the standards and the evolution of languages for service execution, composition and orchestration. It then introduces the choices we made in this first stage of the project motivating them (Section 2.1). It must be noticed, at this point, that we are aware that standards (and languages) will evolve in the next months and we are prepared to revise the choices accordingly if needed. As a consequence of the choices we made as regards standards and languages, we made an analysis of the software platforms that are currently available. After an overview we discuss the criteria that we took into account to define the software platform for the project and for testing the diagnostic engines that will be developed. The resulting choices are reported in Section 2.2.3.

The second part of the report (Section 3) focuses on the application scenarios that will be used as test beds during the development of the surveillance platform. The application scenarios have been selected starting from realistic applications after the initial suggestions coming from the project Industrial Advisor. The application scenarios are thus realistic models of real-world applications, containing all relevant features to be tackled in the design of the surveillance platform. We report the three application scenarios on which we worked and, for each one of them, we describe the main characteristics as regards the workflow, the exceptions (and failures) that may arise, the goals that a diagnostic and repair process should achieve. Moreover, a model of the process is also reported. A comparison of the scenarios points out the aspects that they will allow to tackle during the design and test of the surveillance platform.

Section 4 introduces the general requirements that the surveillance platform that will be developed in the project will have to meet. We considered four main groups of requirements: (1) the requirements concerning the web service composition and execution languages and, specifically, the features that have to be introduces for supporting self healing services (Section 4.1); (2) the requirements for diagnosis and repair, that is those that the surveillance platform will have to meet (Section 4.2); (3) the requirements concerning the design process and, in particular, those concerning design for diagnosability and repairability that the project will have to meet; (4) the requirements for the semi-automatic acquisition (learning) of the models of the services that are needed to support the diagnostic process (Section 4.4). The set of requirements reported in this document are inputs to the Workpackages 3,4,5 and 2 respectively and indeed have been used as the first pint during the kick-off of these workpackages (2, 3 and 4; Workpackage 5 will start at a later stage).

The definition of requirements was considered as a necessary but not sufficient step to start effectively the work in these workpackages, especially as regards the design of the software modules. For such a reason we laid down a preliminary architecture for the surveillance platform.

This is, in our view a very important point to b used as a reference in the future work. Obviously the architecture is still preliminary as several aspects have to be sorted out before making a final choice. Section 5 reports these preliminary considerations, briefly discussing also the alternatives that are currently under examination for the definition of the diagnostic process.

A final important contribution of these reports is the glossary as we discovered that the Web Service and Model-based diagnosis community use a different terminology and that in some cases the same terms have a very different meaning in the two communities. The glossary can thus be read in two different ways. On the one hand it introduces and explains the various terms; on the other hand it introduces a standard terminology that will be used as a reference throughout the process.

2 Working Environment

2.1 Standards

2.1.1 Web services standards

During the nineties, several enterprises recognized, even in those early stages, how the Internet would affect the way they communicate with their customers. With the Internet the potential audience grows and, consequently, new commercial relationships can be built more easily. In this respect, the proliferation of such concepts as e-commerce, e-business, e-government, and e-procurement has been the result of efforts to integrate, both vertically and horizontally, existing information systems both within the same enterprise and between different enterprises. As early as the eighties, in fact, when the number of information systems implemented, often on different platforms, was growing, enterprises already needed to integrate them. Integration affects not only different enterprises, but may also the same enterprise, where, for historical and organizational reasons, different branches adopted different solutions when setting up their information system. Nowadays, the Internet provides a network that enterprises are exploiting to automate supply chains and to create virtual marketplaces. Such integration requires a great effort both from an organizational and a technological standpoint, and solutions have to strike a balance between the need for the various actors involved to exchange information, and the need to leave them with a certain degree of autonomy. From an organizational standpoint, an enterprise plays the role both of a service consumer and a service producer. The provided services, which represent the functionality that the enterprise intends to export, define the borderline between what is public and what is private for an enterprise. On the other hand, from a technological standpoint, in order to achieve the required level of interoperability, an enterprise must agree on a set of standard languages with which to describe services. In this way all the actors involved can understand what a system exports and the characteristics of the services provided. Mechanisms to retrieve the available services and to exploit the Internet as a communication infrastructure are also required. Although organizational aspects are very important in order to understand how an enterprise decides whether to export a service or keep it private, in this report we concentrate on the technological aspects, and more specifically on Web Services as a solution for *information system integration*. In fact Web Services not only provide a solution for system interoperability but can also be used to provide new services. As a first step to describe a Web Service, as happens in component based programming, it is important to separate the *presentation logic* from the *application logic*. Web Services, in fact, are only related to application logic, therefore whoever is going to use the service will be responsible for creating the presentation logic according to their needs [PP04].

Before discussing technical aspects, and in order to better define our scope, we will explain exactly what a Web Service is. In doing this, we want to avoid limiting our definition to the commonly held view that a Web Service is made up of just three main specifications: Web Service Description Language (WSDL) [CCMW01], Simple Object Access Protocol (SOAP), and Universal Description Discovery and Integration (UDDI). For this reason, our starting point will be the Service Oriented Architecture (SOA) where it is possible to define the goals, the usage, and the future development of Web Services. This architecture fits in perfectly with what we the topics presentee here above, especially in the business-to-business environment [PP04] Web Services are driven by the paradigm of the SOA, which describes the relationships that exist among service providers, consumers and service brokers, and thereby provides an abstract execution environment for Web Services. Accordingly, current research addressing service composition is based on technologies and solutions from the area of Service-Oriented Computing (SOC). From their first appearance, SOA and SOC have emerged as key conceptual frameworks for the world of Web Services. Focusing on service definition, the provider and the user are the main actors involved,

as is also the case in a typical client-server interaction. In an SOA, a new actor, called a Service Directory or Service Broker, is introduced in the provider/user relationship, as shown in Figure 1.

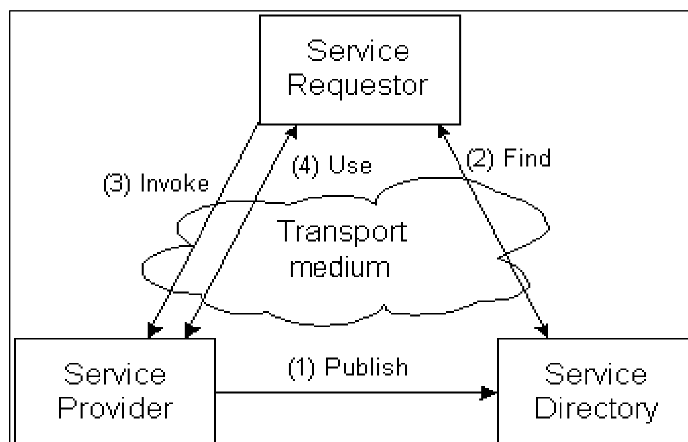


Figure 1: Service Oriented Architecture.

The roles of the actors in this configuration are as follows:

• *Service Provider*: is responsible for building a service and making it available. A service is advertised via the *publish* operation which stores a document summarizing the service features in a public registry. Once the service is published, the Service Provider awaits for users interested in the service.

• *Service Directory or Service Broker*: this component is responsible for maintaining the public registry in which the description of services are stored, allowing users to find the services that best meet their needs. The Service Directory can also define a set of access policies to limit user accessibility for security or privacy reasons. In this report, we consider the registry as fully accessible.

• *Service Requestor*: represents a potential user for published services. By means of the *find* operation, users interact with the Service Directory to find the services that best meet their needs. Once the service is identified, the Service Requestor communicates directly with the related Service Provider (bind) and he starts to interact with the service (use).

These three actors involved can be distributed and can rely on different platforms, but during interaction each uses the same *communication channel*, which is one of the parameters of the architecture. Thus a SOA can be used for example in mobile systems, or on the Web, as well as in e-mail systems, allowing the creation of multi-channel systems where the same service can be used through different devices.

On the basis of these considerations, we define an *e-Service* as an instance of an SOA in which an electronic channel identifies the communication channel, whereas for Web Service the communication channel is represented by the Web.

A typical instantiation of the SOA uses SOAP as a protocol for the transport medium, service described as Web Services in WSDL and UDDI for service retrieval. SOAP is an XML based protocol capable of defining an interaction pattern among remote components on the Web. Although one of the main purposes of SOAP is to support RPC (Remote Procedure Call) on the Web, this protocol can support asynchronous or message based communications as well. In SOAP, the way a remote operation is invoked is specified through an XML document, while HTTP represents the transport protocol. By using HTTP, SOAP solves the problem encountered by

typical invocation protocols such as COM+, Java RMI, and CORBA, where firewalls often block interaction messages.

WSDL (Web Service Description Language) allows one to formalize the service features according to a schema very similar to a typical API definition. WSDL is an XML based language able to specify the service feature we have just described in text form. The first element comprising a WSDL specification is service, which identifies *a set* of services, each specified by a port. It should be noted that a port only represents the physical address where the service operates and the protocols the user should adopt to communicate with it, with no description of the provided functionalities. This aspect is defined by the portType, directly associated with the port, which is responsible for defining the available operations. Hence, portType defines what the service does, whereas port defines where the service is (Figure 2).

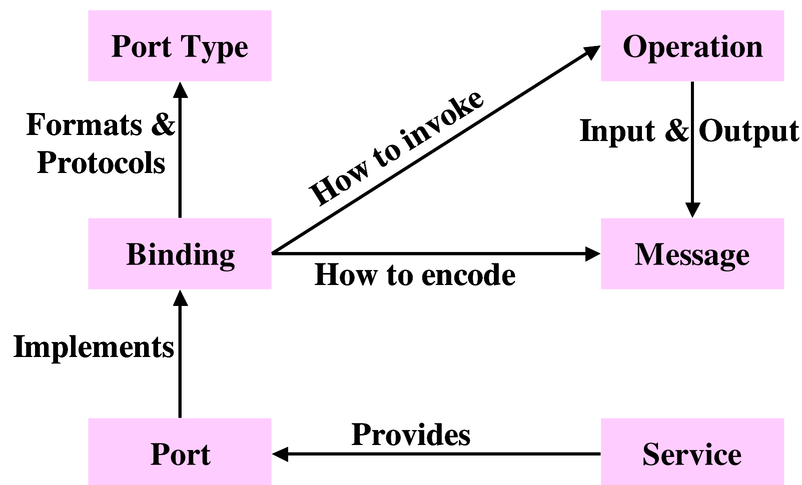


Figure 2 : WSDL elements

The binding element is responsible for defining this specialization by mapping the operations specified by the portType to a port, according to a particular protocol such as SOAP, HTTP, or SMTP. For example, we can have a single service called LoanApproval indicated by the port tag. This service is accessible at http://tempu ri.org/services/approve and the customer can invoke it using the SOAP protocol. In more detail, a portType is composed of a set of *operations* which reflect the functionalities characterizing the service available to the user. An operation must refer to one of the following four predefined patterns:

• One_way. The operation is composed of only one incoming message for the service provider.

• Request_response. Upon a request from the customer, the service responds.

• Solicit_Response. Here the provider starts the communication and waits for a response from the customer.

• Notification. Composed of a single outgoing message from the service.

Regardless of the type of pattern, the interaction is composed of a set of messages specified in WSDL with the message tag specifying the format of the message. Each message is composed of a set of parts which refer to data types. A type can be an XML predefined one (e.g., int, string), or a custom type defined in the types section of the WSDL specification (in the example, the type section is not considered since all messages are specified according to the basic types). In the relationship between port and portType within a WSDL specification, the same portType, and therefore service, can be accessed through different protocols, depending on the binding set

specified. Currently, the protocols that can be used for binding are SOAP, HTTP, and SMTP and it is only possible to specify a Web Service over these protocols. Therefore, a good service definition contains a set of WSDL documents: a WSDL Interface document where the type, the messages and the portType are specified, and a WSDL Implementation document for each kind of binding, where the related port is defined. In fact, even if the WSDL is very general purpose, the specification is enough to describe a large number of types of service. The four communication patterns allow both asynchronous and synchronous services to be specified. In the first case only one-way and notification patterns are used, whereas in the second case the communication takes place according to the request-response and the solicit-response patterns.

In Jan. 2006, W3C has published a candidate recommendation for WSDL-2, which modifies the previous specification of WSDL. In particular, *fault propagation rules* may be defined, and new communication patterns are introduced to enable the specification of mandatory and optional input and output parameters.

WSDL only focuses on the static description of the service. Behaviour specifies how the service works and what the available operations are depending on the service status. For this purpose, several languages have been proposed. WSCL (Web Service Conversation Language) is an XML based specification, which describes how a user can converse with a service by defining the service as a state-finite machine, in which operations represent the states and the transition between such states are also defined. BPEL4WS (Business Process Execution Language for Web Services), which defines a service composition language that can also be used to specify the behaviour of a single service.

2.1.2 Web services Composition, Orchestration, and Execution

Web services are one of the most promising approaches for the integration of heterogeneous systems and web-based communication between business partners. As many enterprises started to implement their own web services, composition of such services comes with the actual surplus.

Because the most of business process definition languages do not directly support the web services standards, short-term solutions like individual web service composition protocols can sidestep this gap[P03a]. Web service orchestration and choreography are long-term solutions and based on open standards and facilitate the maintenance of web service composition.

Web service compositions are workflows based on web services. Systematic execution of business processes is the primary task of a web service composition management system (WSCMS). A business process is a group of manual or automatic activities undertaken by an organization in pursuit of a commercial or organizational goal. These activities may be extended beyond the own organizational scope and integrate the activities of consumers, suppliers and other partner organizations. An activity is a unit of work within a process which specifies the actors (persons, machines and applications) and resources (tools and machines) assigned to activities and temporal dependencies between activities (order and duration of execution, etc). In web service compositions, activities are either web services or processes.

As standards and technologies still have to reach stable definitions, also authors writing about service composition are far from using a commonly agreed on terminology. For Web Services, *choreography* "…tracks the message sequences among multiple parties and sources — typically the public message exchanges that occur between Web Services — rather than a specific business process that a single party executes…" [P03a].

[ACKM04] prefer the terms *coordination* (protocol) and *composition,* rather than choreography and orchestration. Literally, they clarify "…we will use the term *conversation* to refer to the sequences of operations (i.e., message exchanges) that could occur between a client and a service

as part of the invocation of a Web Service. We will use the term *coordination protocol* to refer to the specification of the set of correct and accepted conversations…" And: "…we refer to a service implemented by combining the functionality provided by other Web Services as a *composite service*, and the process of developing a composite Web Service as *service composition*…"

The W3C's Web Services Choreography Working Group defines *choreography* as the specification of the sequences and conditions under which multiple cooperating independent agents exchange messages in order to perform a task to achieve a goal state. Web Services choreography concerns the interactions of services with their users. Any user of a Web Service, automated or otherwise, is a *client* of that service. These users may, in turn, be other Web Services, applications, or human beings. An *orchestration* defines the sequence and conditions in which one Web Service invokes other Web Services in order to realize some useful function, i.e., an orchestration is the pattern of interactions that a Web Service agent must follow in order to achieve its goal (W3C, n.d.).

		Perspective	
		internal	external
Actor	Execution Engine (runtime)	Orchestration	Choreography
	Composition Designer (design time)	Composition	Coordination

Figure 3 : A contextualized view on currently used terminology; the two main nomenclatures concerning respectively internal and external perspective on Web Services can further be specialized by actor and execution time

As this terminological comparison outlines, different authors prefer different names and thus emphasize different aspects even within the same Web Service domain. Figure 3 attempts to characterize and aggregate the currently used terminology through contextualizing the most commonly used terms [DP06]. For this purpose, it distinguishes two main dimensions: the perspective of the observer and the kind of observer along with its observation time. According to a common approach, the perspective is divided into *internal* and *external*, with respect to the observer's view, whereas the novel aspect of Figure 3 is represented by the dimension *actor*, which allows distinguishing between composition designers and execution engines. An *execution engine* executes a composite service (runtime orchestration: the engine is already provided with the set of component services, the *orchestra*) that has previously been defined by a composite service designer (design time composition: the orchestra is *composed* by selecting the right services). A *service designer* thus composes new services driven by a final goal and by taking into account the restrictions imposed by the coordination protocols of the component services (design time coordination: once selected, he *coordinates* the services). At the composite level and at runtime, externally visible coordination effects can be interpreted as choreography with respect to the orchestra of compound services.

The taxonomy of Figure 3 provides a coarse contextualization of the most used terms and has orientation purposes (it should not be considered a widely acknowledged categorization).

2.1.2.1 A possible Protocol Stack

Figure 4 shows a possible Web Service protocol stack that concentrates on service coordination and composition. Besides traditional transport protocols such as HTTP, SMTP, or IIOP, SOAP has become widely acknowledged as basic messaging protocol (nevertheless, other protocols could be used).

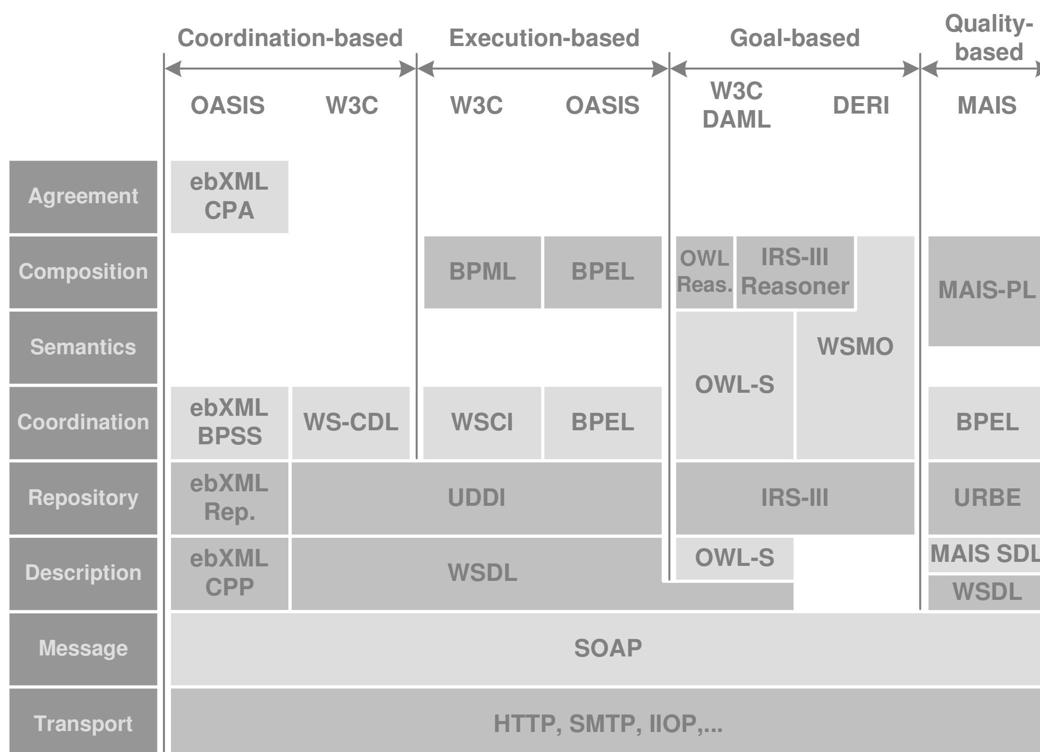


Figure 4 : Web Service composition-oriented protocol stack of vendor-specific and standardized protocols and languages. Within the composition layer, we propose BPML in on top of WSCI as they share a common process model. However, other executable BPM languages could be adopted as well [DP06]

Web Service description is achieved by means of WSDL, but when coming to service coordination and composition, a wide range of different protocols and languages are proposed by different vendors or organizations. The main are the following.

- **ebXML (*Electronic Business using eXtensible Markup Language*);** UN/CEFACT, OASIS [EN01].This is a (vertical) suite of specifications of how electronic commerce exchanges should be specified, documented, and conducted, and can be subdivided into three different protocols:
    - **CPP (*Collaboration Protocol Profile*);** A CPP is similar to a UDDI registry entry and includes interface and message descriptions as well as business data and data exchange capabilities of a particular trading partner.
    - **BPSS (*Business Process Specification Schema*);** The BPSS protocol can define both the choreography and communications between services. The definition of a proper business process execution language is explicitly outside the scope of ebXML.

    - **CPA (*Collaboration Protocol Agreement*);** A CPA contains the business agreement among cooperating partners. It is derived from the intersection of the CPPs of the cooperating trading partners.

- **WSCI (*Web Services Choreography Interface*);** initially Sun, SAP, BEA and Intalio; now W3C Note [AAFJ02] It is an XML-based interface description language that describes the flow of messages exchanged by a Web Service participating in choreographed interactions with other services. WSCI is a coordination protocol, in that it does not address the definition and the implementation of the internal processes that actually drive the message exchange;

- **WSDL-S** [AK05]; WSDL-S is a straightforward extension to WSDL. It makes use of the extensibility elements to embed pointers to (external) domain ontologies. The idea behind WSDL-S is that it should be as easy as possible for software engineers to create semantic annotations for web services by embedding them directly into the WSDL descriptions. WSDL-S is agnostic with respect to the ontology language used for annotation;

- **WS-CDL (*Web Services Choreography Definition Language*);** W3C Working Draft WS-CDL is an XML-based language that describes peer-to-peer collaborations of parties by defining, from a global viewpoint, their common and complementary observable behaviour, where ordered message exchanges aim at accomplishing a common business goal. It is neither an "executable business process description language" nor an implementation language.

- **BPML (*Business Process Management Language*);** Business Process Management Initiative (BPMI.org, 2002). BPML is a language for the modelling of business processes and was designed to support processes that a business process management system could execute. BPML and WSCI share the same underlying process execution model. Therefore, developers can use WSCI to describe public interactions among business processes and reserve, for example, BPML for developing private implementations. However, other coordination protocols than WSCI can be adopted.

- **BPEL (also BPEL4WS, *Business Process Execution Language for Web Services* or WS-BPEL);** initially Microsoft, IBM, Siebel Systems, BEA, and SAP; now OASIS (*Web Services Business Process Execution Language*) [WF02]. It provides an XML-based grammar for describing the control logic required to coordinate Web Services participating in a process flow. BPEL can act both as coordination protocol and proper composition language. BPEL orchestration engines can execute this grammar, coordinate activities, and compensate activities when errors occur.

- **OWL-S (*Ontology Web Language for Web Services*);** DAML.org [M03a]. OWL-S is an ontology-based description language that supplies Web Service providers with a set of markup language constructs for describing the properties and capabilities of their Web Services at a semantic level and in an unambiguous, computer-interpretable form. It allows the definition of semantic descriptions as well as coordination rules. Previous releases of this language were built upon DAML+OIL, known as DAML-S. Theoretically, OWL-S is not limited to one specific grounding, but its current version provides a predefined grounding for WSDL that maps OWL-S elements to a WSDL interface [PL05]. On top of OWL-S, proper *OWL reasoners* will allow automatic service composition and execution.

- **WSMO (*Web Service Modeling Ontology*);** DERI [RLK04]. Based on the conceptual model provided by the WSMF *(Web Service Modeling Framework*) [FB02], WSMO serves the purpose of describing various aspects of semantic Web Services, ranging from coordination constraints over semantics to composition issues, and aims at solving existing integration problems. The vision of WSMO is that of an automated, goal-driven service

composition that builds on pre- and post-conditions associated to component services. In its current version, WSMO does not define any grounding of services, but DERI is planning to allow multiple groundings for their service descriptions.

- **IRS** (***Internet Reasoning Service***) [CDM04]**;** IRS is KMi's Semantic Web Services framework, for semantically describing and executing Web Services. The IRS supports the provision of semantic reasoning services within the context of the Semantic Web. The primary goal is to support the discovery and retrieval of knowledge components (i.e., services) from libraries over the Internet and to semi-automatically compose them according to specified goals. It is based on problem solving methods, using task descriptions in terms of input roles, output roles, pre-conditions, assumptions, and goals and ontologies.
- **MAIS** (***Multichannel Adaptive Information Systems***) [MMMP04] [CMPP04]; the Italian MAIS research project proposes a quality-based approach to service description, selection, and composition. Web Services, described through MAIS-SDL (Service Description Language) based on WSDL and annotated with quality properties defined in WSOL [TPP02], are dynamically composed in context variable process executions. Web Services are selected from URBE, a UDDI-compatible registry with a service ontology and service quality information, according to an abstract process description, formulated associating to BPEL local and global quality constraints and on the basis of information available in the current context of execution (using the MAIS-PL MAIS Process Language).

As the above list and Figure 4 show, composite service designers currently face a huge amount of partly mutually exclusive, partly dependent specifications that all serve similar purposes. They are supposed to know and master all the above specifications together with their peculiarities in order to be able to choose the right combination for their particular composition problem.

2.1.2.2 Evolution of today's Standards

The high number of candidate standards is mainly due to two reasons: first, vendor-related political and strategic aspects (each supports its specification as a common standard); second, the relatively young age of the Web Service technologies. Unavoidably, this results in a lack of stability when one comes to choose reference specifications.

Figure 5 : Emergence and evolution of today's principal standards and languages concerning WS composition. The figure tries to reflect the official release or publication dates of the specifications (at the best of the authors' knowledge), first appearance of or discussions about them could differ from the proposed dates. XLANG and WSFL are not treated in this paper; they heavily contributed to BPEL and are reported for the sake of completeness [DP06]

Figure 5 graphically depicts the emergence of the above listed standards and/or specifications. Along the diagram's diagonal, a trend towards high-level and semantically enriched specifications can be derived, which enables designers to comfortably specify or to automatically derive executable service compositions. Hopefully, at the end of this ongoing evolution, the different approaches and languages will contribute to or converge into a stable SOP (Service Oriented Programming) framework.

2.1.2.3 The need for coordination protocols

As introduced earlier, coordination and choreography describe the external message exchange that occur between a Web Service and its client or among several collaborating Web Services. The main concerns that have to be addressed within the coordination layer are: Can messages be sent and received in any order? Which rules govern message sequences? Is there a relationship among incoming and outgoing messages? Is it possible to undo (parts of) already executed sequences? The following sections will try to provide answers and details by discussing the conceptual backgrounds and core ideas of the most representative coordination approaches.

2.1.2.3.1 Conversation between Service and Client

WSDL as interface description language already provides a limited set of constructs that aim at specifying how to correctly interact with a particular Web Service. Several extensions have been investigated that tried to extend the basic WSDL description with concepts for better describing conversation-related aspects

Figure 6, for example, graphically depicts the problem of ordering of exchanged messages.

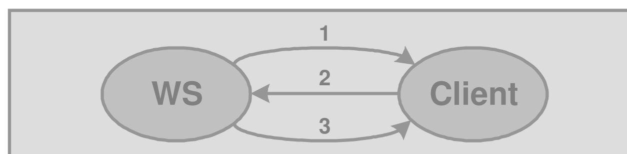


Figure 6 : Ordered message exchange between a Web Service and its client.

WSDL extensions such as WSCL (Hewlett-Packard Company, 2002) only had limited success, probably since the underlying client-server conversation model does not really fit into the service-oriented architecture of Web Services. Graphically, the functionality of WSCL could best be described by a state machine model, whose expressive power allows describing conditions and ordered messages, but does not distinguish between involved actors.

2.1.2.3.2 Multi-service Conversations

Figure 7, for example, depicts a conversation scenario that cannot be adequately described by means of client-server protocols. The main novelty with respect to Figure 6 here is, that now support for an arbitrary number of interacting services is required.

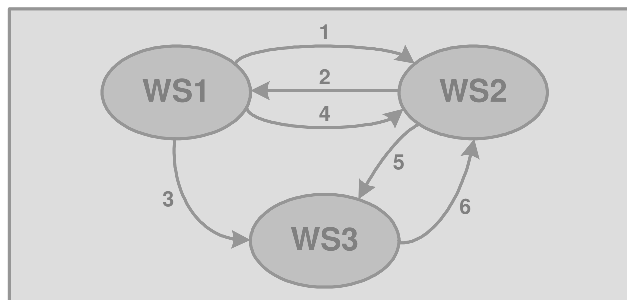


Figure 7 : Interaction involving multiple Web Services; messages depend semantically and chronologically from one another.

Each of them plays a different role within the overall conversation; roles are usually labelled with names like *supplier*, *purchaser*, or *broker*. Graphically, such roles and the conversation itself could best be described by UML activity diagrams where each role has its own “swim lane” in an overall state chart diagram or by sequence diagrams.

As first representative, WSCI goes one step further in its support for long lasting, choreographed and stateful message exchanges with respect to WSCL. In particular, it supports order, rules and boundaries of messages, correlation, transactions and compensation as well as exception handling. Through its concept of *interface*, WSCI goes beyond simple client-server interface descriptions and supports interaction contexts with different external services, despite lacking an overall global view of the conversations a service is involved in. A WSCI interface only describes one partner’s participation in a message exchange and, therefore, a WSCI choreography must include a set of WSCI interfaces, one for each partner constituting an interaction. The sample scenario in Figure 7 would thus require three different WSCI interface descriptions.

WS-CDL, the latest choreography protocol proposal, finally provides a global view over multiparty coordination through explicitly modelling all the involved roles [KBRF04]. Its purpose can be considered as twofold: on the one hand, it provides syntactical primitives for describing involved roles and the messages exchanged during interaction, on the other hand it can be interpreted as well as binding interaction agreement between business partners that intend cooperating and require a language for formalizing their cooperation.

2.1.2.3.3 Other Protocols and Specifications

There also exists a set of proprietary vertical protocols, such as RosettaNet, or xCBL (*XML Common Business Library*), which provide conversation description mechanisms for specific domains. RosettaNet, for example, aims at facilitating dynamic and flexible trading relationships between business partners in the context of IT supply chains. xCBL, in the context of order management, combines an XML version of EDI (*Electronic Data Interchange*) with predefined business protocols.

Along a somewhat orthogonal dimension of the composition problem, there further exist specifications such as *WS-Coordination* or *WS-Transactions* that can be considered as meta-specifications providing a framework for the definition of proper coordination protocols with particular characteristics. For example, WS-Coordination proposes some solutions for the problem of message correlation within conversations involving several different partners. For this purpose, it defines a reference data-structure called *coordination context*, to be added to the exchanged

SOAP headers, that serves the purpose of passing a unique identifier between interacting Web Services.

Vinoski (2004) – in a quite critical way and with no claim for completeness – further provides an impressive list of WS-* specifications, each concerned with the support for particular functionalities:

- WS-Addressing
- WS-Attachments
- WS-BusinessActivity
- WS-Coordination
- WS-Discovery
- WS-Enumeration
- WS-Eventing
- WS-Federation
- WS-Inspection
- WS-Manageability
- WS-MetadataExchange
- WS-Notification
- WS-PolicyFramework
- WS-Provisioning
- WS-ReliableMessaging
- WS-Resource
- WS-Security
- WS-Topics
- WS-Transactions
- WS-Transfer

As can be derived from the names of the single specifications, all WS-* efforts are re-inventing a distributed computing platform on top of standard Web technologies. Comparable to the number of APIs available to .Net or Java/J2EE developers, the amount of WS-* specifications is continuously growing in order to provide suitable APIs and wire protocols for satisfying emerging novel interoperability requirements. The first steps towards commonly agreed on, proper programming libraries for the envisioned SOP infrastructure are being made.

2.1.2.3.4 Coordination Middleware

The coordination protocol specifications described in the last subsections are all so-called description languages. They are not executable languages that actively coordinate conversations among different Web Services. Therefore, the necessary runtime logic must be implemented either by the services themselves or by higher-level process management languages.

[ACKM04] in order to actively support service coordination, suggest an additional middleware layer on top of the coordination layer, containing so-called *conversation controllers* with message routing and protocol compliance verification capabilities. Such conversation controllers could

address the message dispatching problem arising when it comes to one Web Service being engaged in several concurrent conversations. For this purpose, the *coordination context* as described by WS-Coordination could be exploited for messages correlation purposes.

2.1.2.4 Types of web service compositions

In the following two standards will be discussed.

Table 1 shows the important differences.

2.1.2.4.1 Web Service Orchestration

Web Service orchestration refers to an executable process that can communicate with the web services inside as well as those outside of an organization. It contains information about massage exchange between web services and definition of business logic and execution order of activities. These activities can be associated to applications or organizations. The outcome is a long-term, transactional process. In the orchestration, the process is always controlled by one business partner.

2.1.2.4.2 Web service choreography

Web Service Choreography deals with the cooperation between web services. All involved partners specify the communication with the process. Choreography defines the message exchange between partners and the process and is associated with the communication between web services. The outcome of choreography is a descriptive, non-executable process definition, an abstract process.

Table 1: Web Service orchestration vs. choreography

Orchestration	**Choreography**
Executable process	Non- Executable process (abstract)
Control and data flow	Only visible (public) message exchange
Process itself will be published as web service	Only components for message exchange
Composition engine (Server) is required	No composition engine is required

Orchestration differs from choreography in specification of process flow between activities. No partner in choreography has control over composition and solely message exchange is defined. (Figure 8)

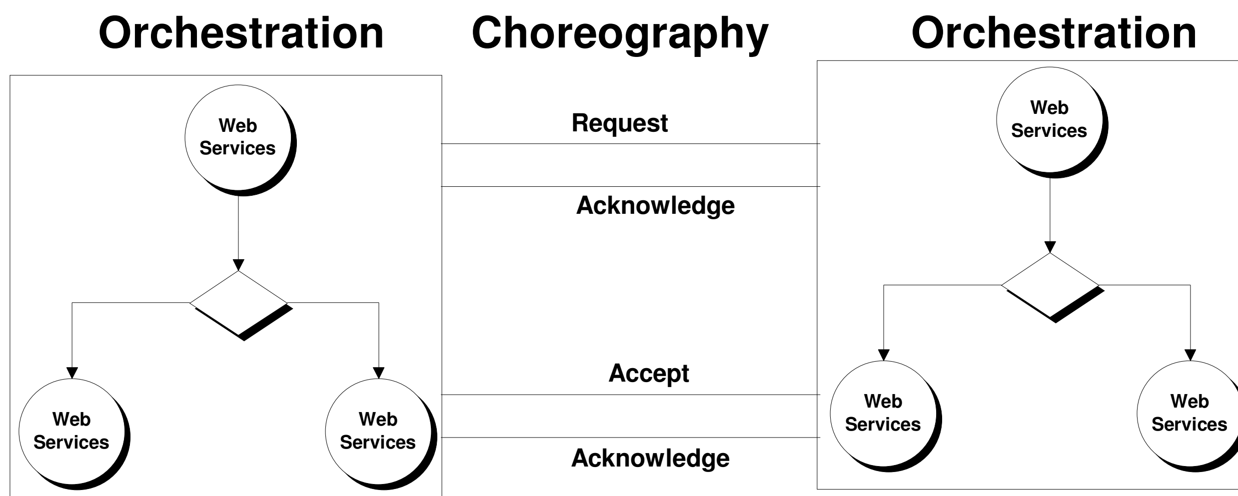


Figure 8: *Orchestration* refers to an executable process, *choreography* tracks the message sequences between parties and sources [Pel03].

The followings are prerequisite for both process definition language and the underlying infrastructure [P03a]:

- Flexibility

Flexibility, offered through the language, is one of the most important aspects. It intends to a clear separation between process logic and the communication with web services. This can be achieved through a control instance that monitors the whole process. With it, removal or modification of services is facilitated for organizations.

- Basic and structures activities

A language has two tasks: on the one hand it must offer activities in order to assure the communication with web services and on the other hand it must be able to execute the workflow semantic (business logic). Basic activities are components that allow a conversation with internal or external web services. In the contrary, structured activities control the conversation. They declare which basic activities and in which order have to be executed.

- Recursive composition

A single business process can interact with several web services and in turn be published as a web service. Thereby, it is possible to compose higher level processes.

- Persistence and correlation

The ability of monitoring process state is an important requirement, especially in asynchronous web service conversation. The language and infrastructure ought to support a mechanism which is in the position to assure data persistency and correlate request and response of web services enabling complex conversations.

- Failure handling and transaction

A web service orchestration shall possess exception handling mechanisms and support transactions. The most transactions are long-running and failures can lead to compensation problems. For example no resource may be locked during a long period of transaction (locking problem).

2.1.2.5 Main elements of web service compositions

Web service composition defines how web services can be strung together and executed in a given sequence. The web service composition middleware consists of three main elements [ACKM04]:

- Composition model and language

It comprises specification of web services, which will be combined, their order of execution (normally based on conditions that are evaluated at run time) and requests and responses for the message exchange. Specification of a composite web service like a “workflow process description language” includes business logic of a composite web service (control and data flow). This can be described by a language, i.e. BPEL4WS, and is referred to as “composition schema”.

- Development environment

Usually characterized by a graphical user interface. The GUI possesses several functionalities in order to include web service in the composition schema. The graphical illustration of all activities helps users better understand process definitions. Web services can be linked with edges, which are tagged with conditions. The resulted process graph will be mapped to a textual specification (composition schema) in subsequent steps.

- Runtime environment

The runtime environment is often called the composition engine. It implements the business logic of composite web services. The order of execution of web services is defined in composition schema and each implementation of a composite web service is referred to as “composition instance”.

2.1.2.6 Dimensions of a web service composition model

Six different dimensions of a composite web service are considered as component model, orchestration and choreography models, data and data transfer model, service selection, transactions and exception handling [ACKM04]. Below these dimensions will be discussed in details.

2.1.2.6.1 Component model

Component model specifies the component type (HTTP, SOAP, WSDL, etc.) of each web service and states which component types are supported. Limitation of heterogeneity simplifies the composition of web services. The leading composition language, BPEL4WS, is confined to components (web services) described by a WSDL specification. On the other hand, a composition model can make only few demands on used components e.g. components exchange XML messages. The advantage is a more general model and the disadvantage is a more costly and complex composition. A quick fix would be support of several models and additional implementation of not modeled components. This, in turn, may result in multiple models and too complex languages.

2.1.2.6.2 Orchestration and Choreography Models

Orchestration Models

An orchestration model defines the needed process definition language in order to specify the control flow of related web services. The orchestration model allows the description of the internal structure of composite web services (e.g. internal control flow like sequences, parallel execution etc.) and their execution conditions. Most of the models are based on few basic models like: activity diagrams, state charts, Petri nets, activity hierarchies and rule based orchestration.

State charts are formalisms based on an extended variant of state machines to enable the modeling of performed activities while entering, exiting or within a state. In addition, it is possible to define events and conditions. There exist as well other variants including composite transitions, parallel states and synchronization after the execution of parallel composite states. Next figure models the orchestration of vehicle -Process by the means of a statechart. Activities are almost hidden and the concentration is on states. Through assignment of meaningful names to states, it is possible to get useful information on the progress of process. In contrary, activity oriented models allow identification of activities, which are in execution and not the associated state. That is the reason why statecharts are better tools to monitor and track information.

- Petri Nets

Petri nets can be seen as a graphical modeling language with a strong and well understand formalism behind it. Orchestration models based on Petri nets combine activity oriented models (like activity diagrams) with the definitions of process definitions (like statecharts). Figure 11 makes these combinations visible. Each circle defines a state within the process execution (e.g. local system accessed). The availability of vehicle on stock triggers and activity and the next state is reached. If vehicle is not on stock, in the next step "*orderVehicleExtern*" is executed and "READY FOR INFO" is reached.

Many static and dynamic properties of Petri nets can be mathematically proven. Detection of deadlocks and other potentially erroneous condition, enabled through availability of many existing automated tools, and a well formed semantic are important advantages of Petri nets. The basic Petri net model has many extensions, which are more process oriented. Some of them can be found in[VTKB03].

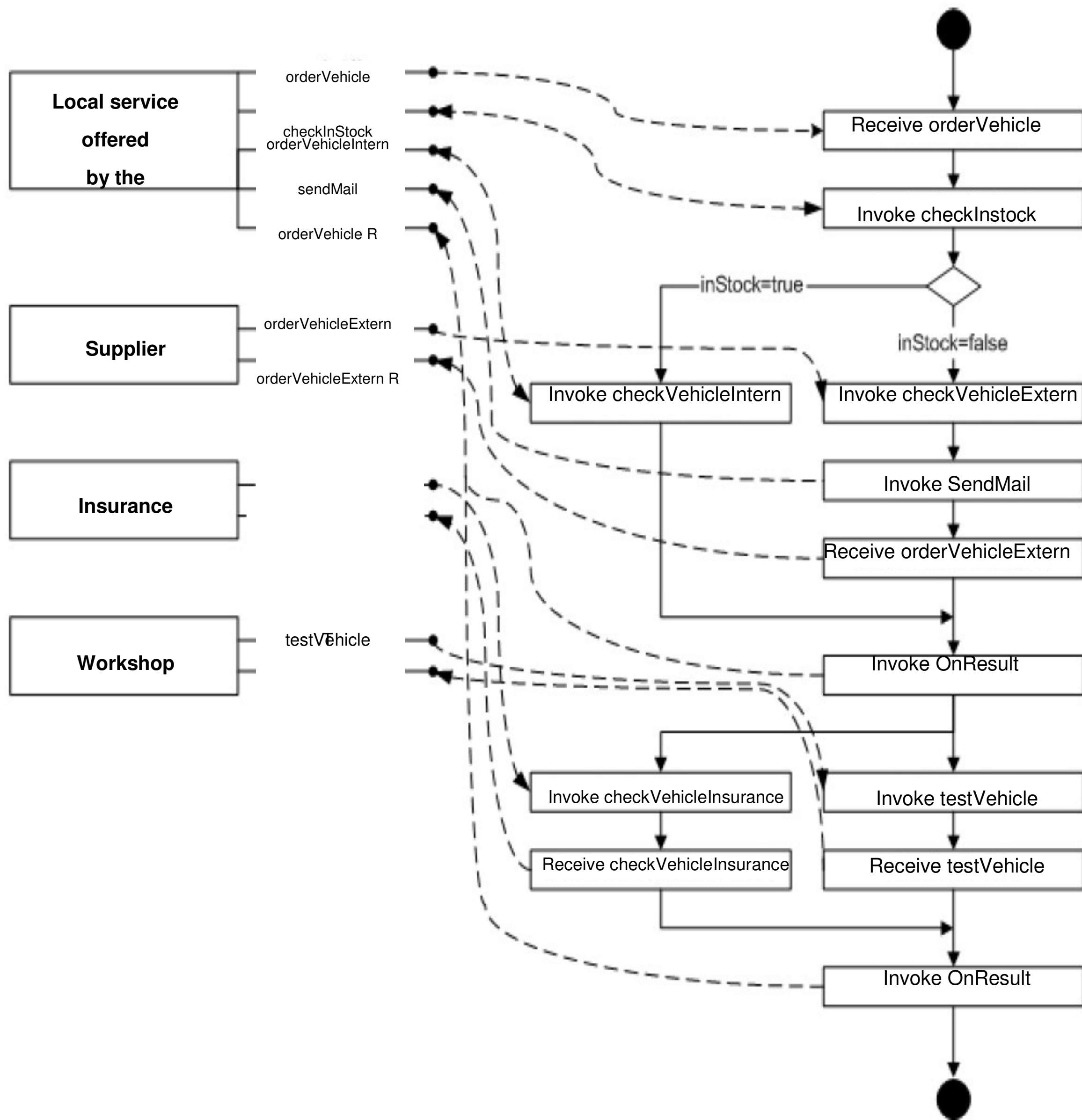


Figure 9 : Vehicle-process as activity diagram

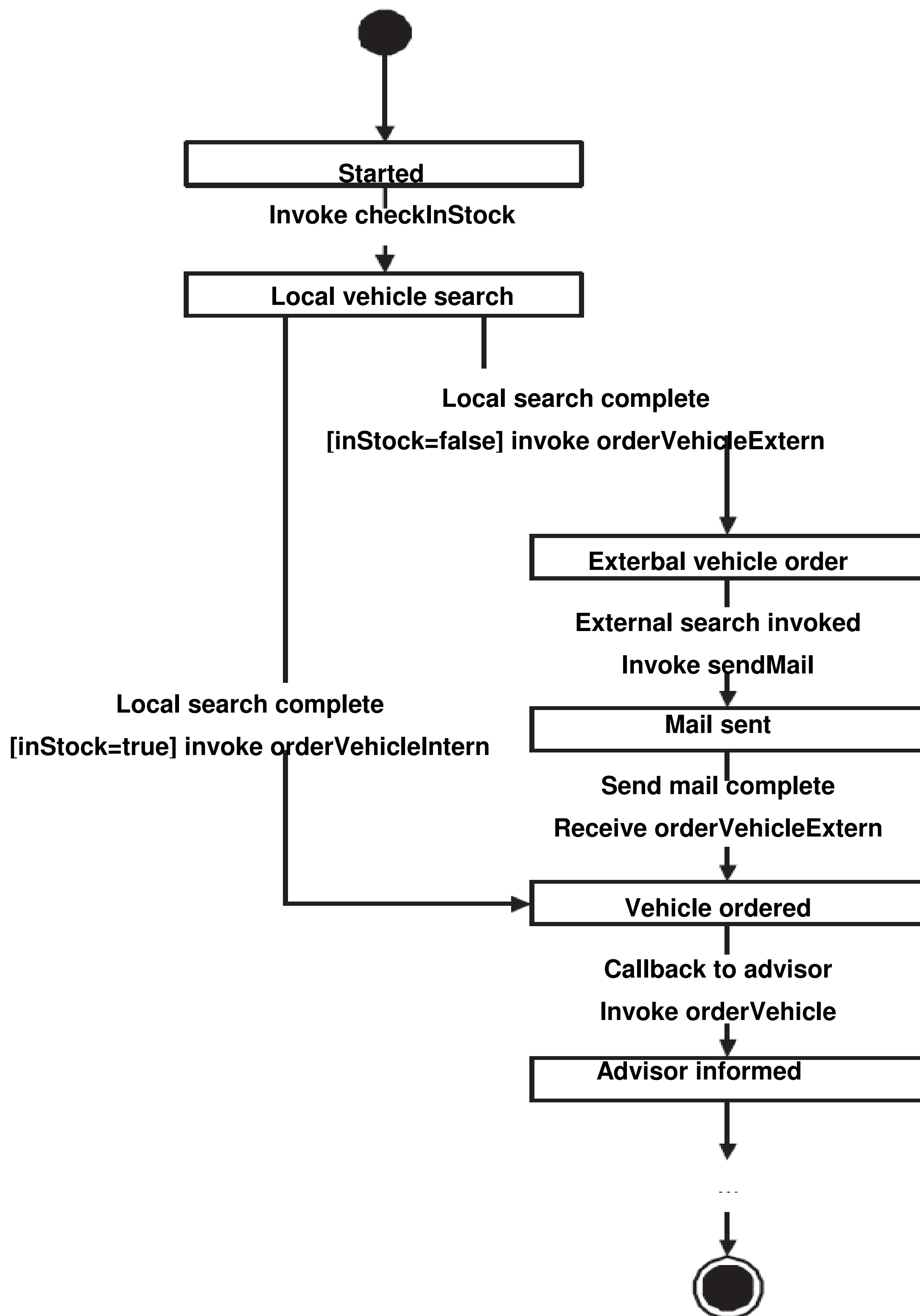


Figure 10: Vehicle -process by means of a statechart

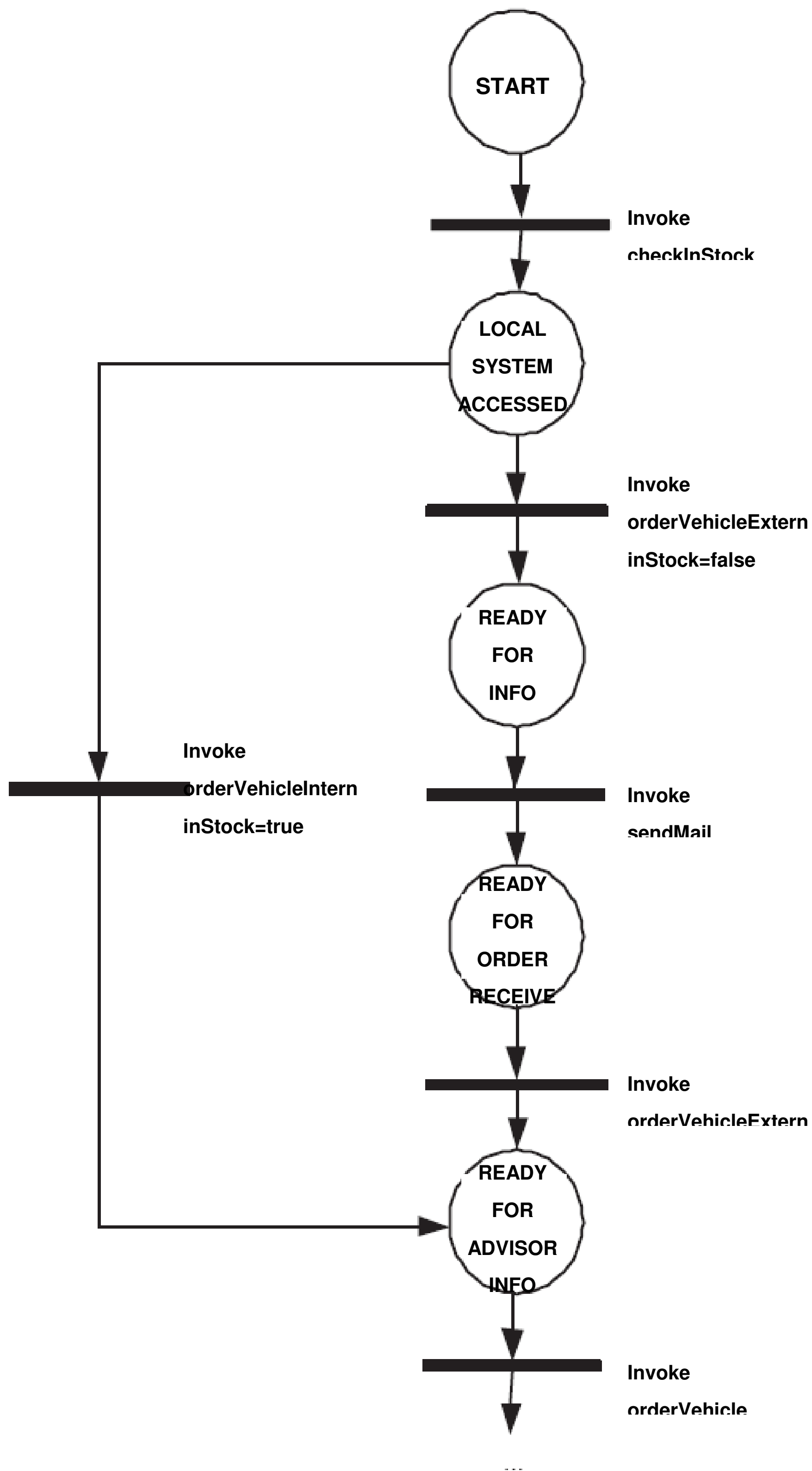


Figure 11: Vehicle-process specified by means of a Petri net

- Π-calculus

Π-calculus is a process algebra and is an attempt at developing a formal theory for process models. It provides a precise and well-studied formalism for describing and verifying processes. Π-calculus has its origins in Communication sequential processes (CSP)[CH85], algebra of communicating processes with abstractions (ACP) [JBK85] and calculus of current systems (CCS) [MIL89]. The existing languages for web service composition (XLANG [ST01], BPEL) admit to be inspired by Π-calculus.

The sequential, parallel and conditional process execution can be described by using the following constructs:

A.B: activity A happens before activity B

A|B: activity A and activity B occur in parallel

A+B: either activity A or activity B is executed

A sample Π-calculus specification of the vehicle -process is presented in the following Listing:

A = receiveOrderVehicle . invokeCheckInStock

B = invokeOrderVehicleIntern

C = invokeOrderVehicleExtern . invokeSendMail . receiveOrderVehicleExtern

D = invokeSendConfirmation

E = invokeCheckVehicleInsurance . receiveCheckVehicleInsurance | invokeTestVehicle .receiveTestVehicle

F = invokeOrderVehicle

VehicleOrder = A. (([inStock = t rue]B +

([inStock = f a l s e]C)) .D.E.F

Listing 2.1: Vehicle-process in Π-Calculus Notation

- Activity Hierarchies

Activity hierarchies try to specify a process in a hierarchically constructed activity tree. Leaf nodes represent the activities to be executed and the intermediate nodes set the ordering constraints. The advantage of this approach is the different levels of abstraction. Higher abstractions are toward the root of the tree while more details will be toward the leaf nodes. Figure 12 shows the vehicle -process modeled by means of activity hierarchies. An example of orchestration model based on activity hierarchy is Little-JIL [CAM00]

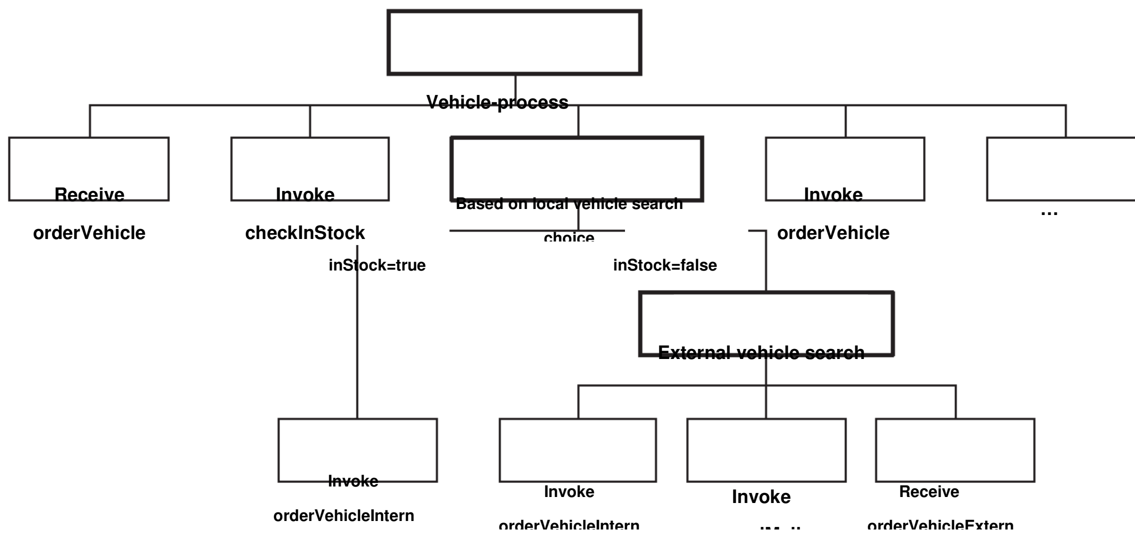


Figure 12**:** vehicle-process as activity hierarchies

- Rule-based Orchestration

An orchestration model can be specified by means of a set of rules. These rules are based on events. This means if an event has occurred, a particular action is executed. Furthermore, conditions can be added to the event- controlled actions in order to provide more complex orchestration schemas. If a rule-based language allows the specification of conditions, the rule model is said to follow the ECA (event-condition-action) paradigm[CIY00]. In the context of web service composition, the composition engine can be viewed as a reactive system. Actually, the composition engine executes specific rules with consideration of their conditions when responses from clients or other services are received or it reacts to requests. Listing 2.2 specifies a sequence of rules with conditions. Obviously, this model is well suited for compositions that have few constraints and the entire schema can be specified using few rules. Otherwise the schema will be complex and hardly understandable.

ON r e c e i v e orderVehicle

IF t rue THEN invoke checkInStock

ON complete (checkInStock)

IF (inStock == true) THEN invoke orderVehicleIntern

ON complete (orderVehicleIntern)

IF t rue THEN invoke invoke orderVehicle

ON complete (checkInStock)

IF (inStock == f a l s e) THEN invoke orderVehicleExtern

ON complete (orderVehicleExtern)

IF t rue THEN invoke sendMail

ON r e c e iv e orderVehicleExtern

IF true THEN invoke orderVehicle

. .

Listing 2.2: Vehicle-process by means of rule-based orchestration

Choreography Models

The choreography model describes collaboration between a collection of web services in order to achieve a common goal. It captures the interactions in which the participants engage to achieve this goal. It also describes the dependencies between these interactions, e.g. control flow dependencies, data flow dependencies, message correlations etc. The interactions are captured from a global perspective, i.e. all participating services are treated equally. Moreover, the choreography does not describe any internal action that occurs within a participating service and does not directly result in an externally visible effect.

- Sequence Diagrams

Sequence diagrams provide a view on interactions of multiple partners. The interactions are modelled as messages exchanged between partners. The messages can be passed synchronously or asynchronously. Individual participants are represented by so called lifelines. The sequence diagrams in UML 2 allows to define within a sequence the areas with different behaviour specified by an operator. Such areas are called combined fragments. An additional guard condition checks which part of a combined fragment should be executed.

A sample choreography described with UML sequence diagrams is presented in Figure 13:

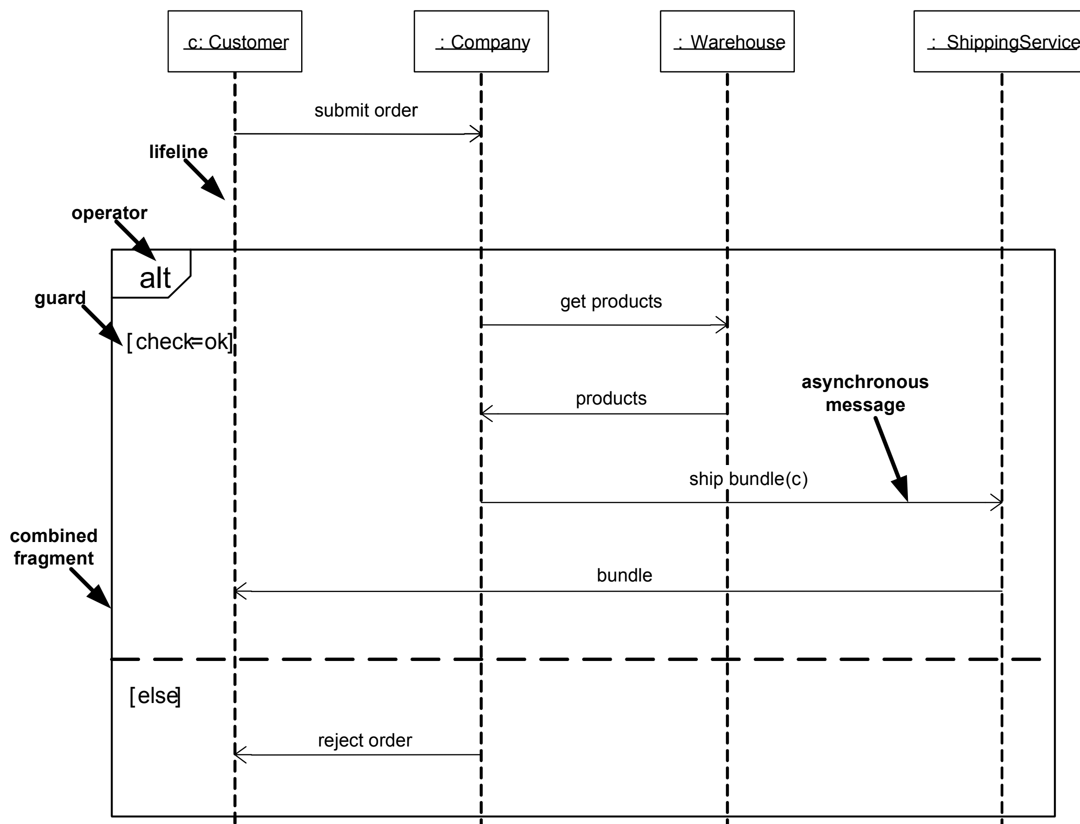


Figure 13: Choreography described with UML sequence diagrams

Other Choreography Models

Another model for describing choreography is Message Sequence Charts (MSC). MSC is a graphical and textual language for the description and specification of the interactions between system components. The main area of application for Message Sequence Charts is as an overview specification of the communication behavior of real-time systems, in particular telecommunication

switching systems. Message Sequence Charts may be used for requirement specification, simulation and validation, test-case specification and documentation of real-time systems.

The choreography can be also described using the activity diagrams as presented in [BAM05].

How orchestration depends on choreography

After this overview of Web Service choreography and orchestration and the main concerns they address, in this section we highlight to what extent the two aspects depend from on another. To this aim, we distinguish three dimensions: structural, functional, and resource dependencies.

- Structural Dependencies

Structural dependencies are those driving the overall structure or organization of a process definition, and thus concern involved activities, conditions, ramifications within the process flow, an so on.

[ACKM04] well explain the dependencies between coordination protocols and composition schemas by stepwise refining the portion of a process definition relative to only one of the participating services. Starting from an overall activity diagram, the authors first extract the role-specific view of the process and then refine it in order to reach a granularity level where the single activities of the remaining diagram reflects the single service invocations required for achieving the specific functionality. This so-called process skeleton on the one hand describes the role-specific view of the process, on the other hand provides a proper protocol description of that participant's public interactions. In this way, the authors show how the definition of the executable process intrinsically must match the constraints imposed by the underlying coordination protocol.

- Functional Dependencies

Functional dependencies concern mainly functionalities or capabilities like transaction support, security, reliability, or correlation; mainly those provided by the wealth of WS-* specifications are considered. Dependencies arise whenever the functionalities they provide are used within a process specification and the composition language "delegates" the relative competencies to the underlying coordination protocols.

As already exemplified earlier, coordination can be achieved either explicitly at process level or implicitly at coordination level. For example, once the choice of adopting the WS-Coordination framework has been made, the process definition does not require further explicit coordination constructs. The same considerations also hold in case of transaction support, reliable messaging, or the like.

- Resource Dependencies

Most of the process definition languages have inherited their modelling approaches from the field of workflow management. At process or composition design time, however, service composition presents some methodological differences that are rooted in the dependencies that exist between coordination and composition.

WfMSs allow for a straightforward top-down structure of the process model, describing, e.g., an administrative workflow. Resources executing a specific work item are provided with the exact amount of data that is required for the correct execution of that task. For executing one task, there is no need to know about possible other tasks before or after that specific task within the same process flow. Possible task constellations are subject only to the constraints imposed by the final goal of the underlying business process. Involved resources do not have a task-surviving behaviour with constraints affecting the overall process definition. Rearranging tasks (i.e., putting some of them in parallel), when specifying process definitions, is a common practice for improving process efficiency.

When defining the logic that constitutes a composite Web Service, a strict top-down approach does not guarantee that the resulting process definition is still always executable. As already outlined earlier when dealing with the need for coordination protocols, a Web Service may by subject to certain conversation rules in order to be executed correctly. For example, before accepting a user's credit card number for payment, the service must be provided with an appropriate list of goods the user wants to buy. This externally visible behavior of Web Services distinguishes the *Web Service* resource from those we have in WfMS. Single tasks cannot anymore be rearranged arbitrarily without loosing functionality.

Composite service designers must know about the coordination requirements of the services they use and take them into account when defining composite services. Thus, starting from an initial process idea (top-down), designers select the services providing the right functionality, and then refine their initial idea (by rearranging initially presumed invocations or adding new ones) in order to conform with the coordination requirements the selected services impose (bottom-up). Therefore, the resulting process definition combines the advantages of both a coarse-grained top-down approach and a fine-grained bottom-up method.

2.1.2.6.3 From Coordination to Composition

Despite the intrinsic passive behaviour of description languages or protocols, they have proven to have enough expressive power in the context of service coordination, which indeed does not require any executable logic. However, coming to orchestration, things change and active support for the execution of process or flow definitions is required. Furthermore, process execution implies the need for dedicated execution environments, so-called execution or process engines able to interpret process definitions and to carry out the specified activities.

There are several different interpretations of what orchestration actually should be. Some authors refer to it as to proper programming languages, others tend to prefer a more general and evolutionary interpretation: "…these systems are often labelled the second generation *Workflow Management Systems* (WfMSs) because they provide much richer integration capabilities than traditional WfMSs…" (BPMI.org, n.d.). This second interpretation is probably too simplistic and puts too much emphasis on the business perspective of the problem. Nevertheless, current orchestration approaches definitely inherit their core modelling concepts from research in the field of WfMSs. For instance, various structured process models have been proposed using traditional workflow constructs as a basis. A classification of typical workflow constructs, originating from a structured programming language approach to workflow definition, has been proposed [VTKB03]. The following subsections provide insight into composition approaches and issues in the context of Web Services.

Model-based Composition

Model-based service composition approaches concentrate on the explicit definition of the possible process flow that governs a composite Web Service. Such process definitions are fed into a process or execution engine that manages the overall execution of the compound activities and thus actively orchestrates the composite service. Commercial composition tools usually provide intuitive high-level visual modelling tools that aid designers in the predominantly explicit definition of processes, such as Microsoft's BizTalk *Orchestration Designer* (Microsoft Corporation, n.d.). Internally, these models are then translated into low-level process models for execution purposes. Several approaches for internal process structures have been proposed. These approaches like Petri Nets, Statecharts, Π-Calculus have been introduces in the previous subsections.

Two representatives of structured process models: BPEL(4WS) vs. BPML

BPEL is an XML-based Web Service composition language that is rooted in both Microsoft's XLANG and IBM's WSFL. In BPEL, a composite service is named a *process*; processes export and import functionality by using Web Service interfaces exclusively. Two main kinds of processes are distinguished: *executable processes* model the actual behaviour of participants in a business interaction (service composition), *abstract processes* describe business protocols, specifying the mutually visible message exchange behaviour of each of the parties involved (coordination). According to this twofold applicability, BPEL is located both in the *Coordination* and *Composition* layers within the protocol stack depicted in 2.11. Besides processes, participating services are called *partners*, and message exchanges or intermediate result transformations are called *activities*. BPEL distinguishes between basic and structured activities. *Basic activities* represent synchronous and asynchronous calls (<invoke>, <invoke>…<receive>), *structured activities* manage the overall process flow (<flow> to denote parallelism, <switch> for alternatives...).

BPEL is designed primarily as a composition language, but developers can use the same formalism for both service composition and conversation definition. As such, it lacks many of the necessary and, from a discovery and binding perspective, particularly useful properties needed for defining conversations (e.g., for activation and compensation). Furthermore, the structure of BPEL is flat, i.e., sub-processes cannot be defined.

BPML, with respect to BPEL, provides similar modelling capabilities, but also supports some additional constructs, making it more flexible in general, such as sub-processes, dynamic partners, etc. In particular, the BPML specification provides an abstract model and an XML syntax for expressing executable business processes. But, BPML itself does not define any application semantics, but rather defines an abstract model and a grammar for expressing generic processes. This allows BPML to be used for a variety of purposes that include, but are not limited to, the definition of enterprise business processes, the definition of complex Web Services, and the definition of multi-party collaborations. BPML is conceived as block-structured programming language. Recursive block structures play a significant role in scoping issues that are relevant for declarations, definitions and process execution.

Both BPEL and BPML provide support for long-running business transactions and robust exception handling facilities. BPML does not provide constructs for the definition of message coordination protocols as BPEL does, but developers easily can use WSCI for this purpose, which shares the same underlying process execution model. This apparent shortcoming of BPML, on the other hand, allows for a more flexible use of BPML and WSCI when it comes to defining conversations, due to the good separation of concerns. Currently, there is, however, less industry support for BPML in comparison to BPEL.

Ontology-driven Composition

Besides explicit process modelling approaches, the Semantic Web and service ontologies offer alternative ways for the composition and execution of compound services. This kind of approach, rather than concentrating on an explicit definition of the flow logic, aims at providing suitable frameworks for the automatic derivation and execution of composite services, defined in an implicit manner by means of goals as well as pre- and post-conditions over service inputs and outputs.

For example, [AAZM04] propose an ontology-driven Web Services composition platform where the requirements of the composite services are specified by users as inputs and expected outputs. The described approach allows the automatic generation and execution of a composite service that produces the expected outputs by combining existing individual services, using their semantic descriptions. A human-assisted and an automatic composition mechanism are outlined.

Lightweight Web Service Semantics: WSDL-S

A recent alternative to the heavy-weight semantic web service ontologies OWL-S and WSMO is WSDL-S, a simple extension to WSDL proposed by IBM and LSDIS Lab published as a W3C technical note.

The motivation for WSDL-S was to lower the threshold for annotating semantic web services. As opposed to complex ontologies about web services, WSDL-S allows for pointers from (XML Schema defined) elements in the web service description to an external ontology. These external ontologies can be defined in any ontology language. WSDL-S is agnostic towards it. Datatype mappings can be either on the element level (one-to-one) or on the level of complex types by specifying a more complex schema mapping for example in XSLT. WSDL-S is, however, also agnostic towards the mapping language. WSDL-S also allows specifying preconditions and effects of an operation in the same way by pointing to an external ontology. Furthermore, a category can be assigned to the service as a whole.

OWL-S vs. WSDL-S: A Comparison

A web service description in OWL-S usually consists of several parts: First, the *profile* specifies a categorization of the service as a whole by placing it into a profile hierarchy or taxonomy. The profile specifies the semantics of input and output parameters of the individual operations of a service as well as its preconditions and effects. Further, the OWL-S profile also specifies some non-functional properties such as the name of the provider of the service. The OWL-S *process model* specifies the internal workflow of a web service. A web service is modeled as a process. An operation in a web service corresponds to an *atomic process*. To model workflows and business processes, OWL-S offers the construct of *composite processes*. For modeling the workflow within a composite process OWL-S offers control constructs similar to those offered by BPEL. OWL-S relies on WSDL to describe the syntax of a web service interface. The OWL-S *grounding* makes the connection between the concepts specified in the OWL-S ontology and the service as described by the standard WSDL description. To ensure compatibility with “legacy” (non-semantic) web services, the grounding supports XSLT transformations to map the XML used by the web service to an RDF representation. An explicit, separate grounding is not necessary in WSDL-S, since the semantic annotations are just extensions to the standard WSDL description, so the connection is clear. OWL-S and WSDL-S follow diametrical philosophies here. In OWL-S, syntactic and semantic descriptions are kept completely separate. The WSDL description remains unaltered; a semantic description can separately be added on top. In contrast, WSDL-S is an extension to WSDL and thus provides syntactic and semantic description within the same document. However, in both cases the actual domain ontology is usually declared separately.

WSDL-S offers the same functionality as the OWL-S profile and grounding, including XSLT transformations for schema mapping. In contrast to OWL-S, WSDL-S does not offer any support for workflow or process descriptions. To specify composed web services as in OWL-S, BPEL is needed in addition to WSDL-S. However, this does not affect the suitability of WSDL-S with respect to (semi-automated) ontology driven composition of web services, where the requirements are rather that matching web services can be discovered. WSDL-S is suited for semi-automated discovery.

Comparing the two approaches, the advantage of WSDL-S is that it is much easier to handle. To create an OWL-S description, a developer has to be familiar with an ontology that is quite complex. In contrast to that, WSDL-S is a straightforward and backward-compatible extension to WSDL using extensibility elements.

Two emerging standards: OWL-S vs. WSMO

OWL-S allows providers of Web Services to describe properties, capabilities, and behaviours of their services by means of ontologies, and provides proper language primitives for their semantic description. The final goal of OWL-S is to provide a machine-interpretable description of services,

in addition to the human-understandable descriptions already provided by WSDL, and thus to support automatic service discovery, execution, and composition. The core of OWL-S, the ontology-driven description approach, builds on the *Ontology Web Language* (OWL) [M03a], which provides the necessary constructs for explicitly representing the meaning of terms and the relationships existing among them within a specific domain. OWL and OWL-S are evolutions of DAML+OIL, a semantic markup language for Web resources.

OWL-S ontologies are structured into three main parts: i) a *service profile* serves the purpose of advertising and discovering services published by service providers and contains a semantically enriched and machine-interpretable service description. ii) a *process model* describes how a service operates (by means of proper control constructs and conversation descriptions) and comprises inputs, outputs, preconditions, results and effects of the service. According to their complexity, *atomic*, *simple,* and *composite* processes are distinguished, being composite the most complex ones. iii) the service *grounding* provides the necessary details for accessing a specific service, i.e., protocols and message formats. Whereas *profile* and *model* provide rather abstract representations, *grounding* refers to the concrete specification. The semantics- and ontology-based approach adopted by OWL-S is particularly suited for advanced service and conversation description.

WSMO aims as well at describing relevant aspects of semantic Web Services. Within the *Web Service Modeling Framework* (WSMF), WSMO provides an (open source) executable solution for goal-driven service composition through extensive use of ontologies, semantic service descriptions and pre- and post-conditions for service description. Besides ontologies, goals and service descriptions, so-called mediators should bypass interoperability problems. Interoperability is one of the main issues WSMO tries to solve, and this aspect differentiates it from OWL-S.

Just as for OWL-S, *ontologies* provide the formal semantics that allows for automatic information processing and for human- and computer-understandable goal definitions. A *goal* specification expresses the final objective a client may have when interacting with a service and consists primarily of constraints over post-conditions after service execution. *Mediators* provide the necessary support for integrating heterogeneous elements when combining several component services. They define mappings and transformations between connected elements. Four types of mediators exist, according to the elements they link: goal-goal mediators, ontology-ontology mediators, Web-Service-goal mediators, and service-service mediators. Finally, Web Services are described by means of their non-functional properties, the mediators they use, their capabilities, and their interfaces and groundings.

Parallel to WSMO, DERI is working on an execution environment for WSMO-based Web Services, called *Web Services Execution Environment* (WSMX) [H05]. The goal of WSMX is that of providing an environment for dynamic inter-operation of Web Services, including automatic discovery, selection, mediation and invocation mechanisms.

Other Composition Approaches

Besides proper language or protocol standardization efforts, several academic research works go one step further in service composition and also investigate the value of additional aspects of the composition problem, such as QoS, personalization, or context. Along a somehow orthogonal dimension with respect to the previous approaches,[MMY05], for example, extend their state-chart-based service composition model with an agent-based and context-oriented approach to composite service execution. The authors define three kinds of software agents (*composite-service-agent*, *master-service-agent*, and *service-agent*) and their execution contexts (*C-context*, *W-context*, and *I-context* respectively), where the term *context* reflects the point of view of services rather than to the one of users. More precisely, C/W/I-contexts and their respective agents refer to three different abstraction levels of the composition problem, namely to the ones of *composition*, *Web Service* (intended as resource) and *instance* (of services). At runtime, agents are engaged in conversations with their peers on behalf of the user to agree on the actual Web Services to

participate in the process, according to the runtime context conditions and the global composition model.

[BBG03] finally, provide a valuable approach to Web Service composition within the initially mentioned workflow domain and with special focus on enterprise workflow interconnection. The process interconnection model presented by the authors builds on Web Service-based workflow integration and allows for heterogeneous workflow systems coexisting in a so-called "workflow of workflows". The main contribution of the work consists in the introduction of a certain level of dynamism, proper of the Web Services area, into workflow definitions; more precisely, the authors postpone the selection of nested sub-processes from build-time to runtime, by introducing proper discovery, negotiation and wrapping mechanisms for so-called process services.

2.1.2.6.4 Data and Data Transfer Model

Data and the data transfer model specify how data are defined and how they can be passed between involved components:

Data types

The literature classifies several categories of data [ACKM04].

- Control data

The web service composition management system (composition engine) monitors the execution of composition using the control data. These data include the identification of actual state of processes or activity instances as well as other internal state information. The control data are not visible to process instances but they are passed between composition engines.

- Control flow relevant data (workflow relevant data)

Control flow relevant data are used by the service composition management system to determine the state transitions of a process instance. They are used by the composition engine in order to identify the next activity to be executed in conditional branches or pre- and postconditions are called control-flow or workflow relevant data, e.g. the control flow in Fig. 2 depends on a value of a variable *inStock*. Control flow relevant data must be available to the composition engine and therefore they are usually stored in process variables.

Control flow relevant data, in contrast to application data, are hardly structured. These data have mostly primitive data types such as String, Integer or Real data type and serve for specification of execution order of involved web services in composition. There exist models that allow more complex data types like arrays.

- Application data

Application data are passed between components (web services, activities) involved in a composition, e.g. *orderVehicleIntern* sends and receives application data as request and response messages.

The application data can be stored in process variables, similarly to the control flow relevant data. In this case the composition engine has direct access to these data and their content. In an alternative solution the application data are handled as a black box. In this approach process variables store only URLs or other kind of pointers to the actual localization of data. Therefore data are hidden in composition and only references are forwarded between activities. The activities must have their own mechanisms of accessing the referenced data. The advantage of this approach is that complex data structures can be ignored while exchanging messages. Complex messages as a part of process may lead to a system overload. This approach presents a good alternative when the used documents are extensive but minimally changed by participating activities. In this way data is not a part of the process but available through the reference.

Data Transfer

Data transfer concerns with a transfer or a data flow of application data from one component to another. There are following approaches:

- Implicit data flow

Implicit data flow is realized via shared data elements (e.g. process variables) which may be passed to and from activities. This resembles techniques used in many programming languages. A shared element may be passed to an activity as an input parameter and an activity output may be passed back to the same or another shared element. The variables can be transformed so that they comply with the required data format.

- Explicit data flow

The explicit data flow approach is based on making data flow among activities an explicit part of the composition. In doing so, data flow connectors are required. Data flow connectors provide the request of an activity with passed response of a previously executed activity. In the example illustrated in Fig. 6 a data flow is represented by dashed arrows, e.g. activity C uses response set of activity A as request.

In the context of web services, the WSFL specification [LRT03] uses explicit data flow. Also some workflow management systems like MQSeries (IBM, 1999) and BioOpera [BAP03] apply this approach.

Generally speaking, explicit data flow approach is more flexible and extensive than implicit data flow approach. Its design is more complex but comprises additional control mechanisms. In the above example, activity A must be completed for activity C to start. Data flow approaches may cause malfunction if the same input can be provided by different data flows. In this case we talk about *race condition.*

2.1.2.6.5 Service Selection

One of the main novelties introduced by research efforts, as well as by the ontology- or semantics-driven composition approaches, consists in the dynamic selection of the services to be composed, besides the dynamic service composition itself.

Service selection is probably the point where current orchestration approaches definitely lack flexibility with respect to traditional WfMSs, which usually include a (centralized) resource manager that at runtime decides to which resource instance, respecting a precise role definition, a specific task should be assigned (WfMC, n.d.). The question, hence, is whether component services should be selected at process *definition time* or at r*untime* during process execution. Some authors even distinguish between service selection at *design time* and *deploy time*. The overall purpose of dynamic service selection is mostly that of guaranteeing the availability of a composite service, being the Web a highly variable and fast changing environment.

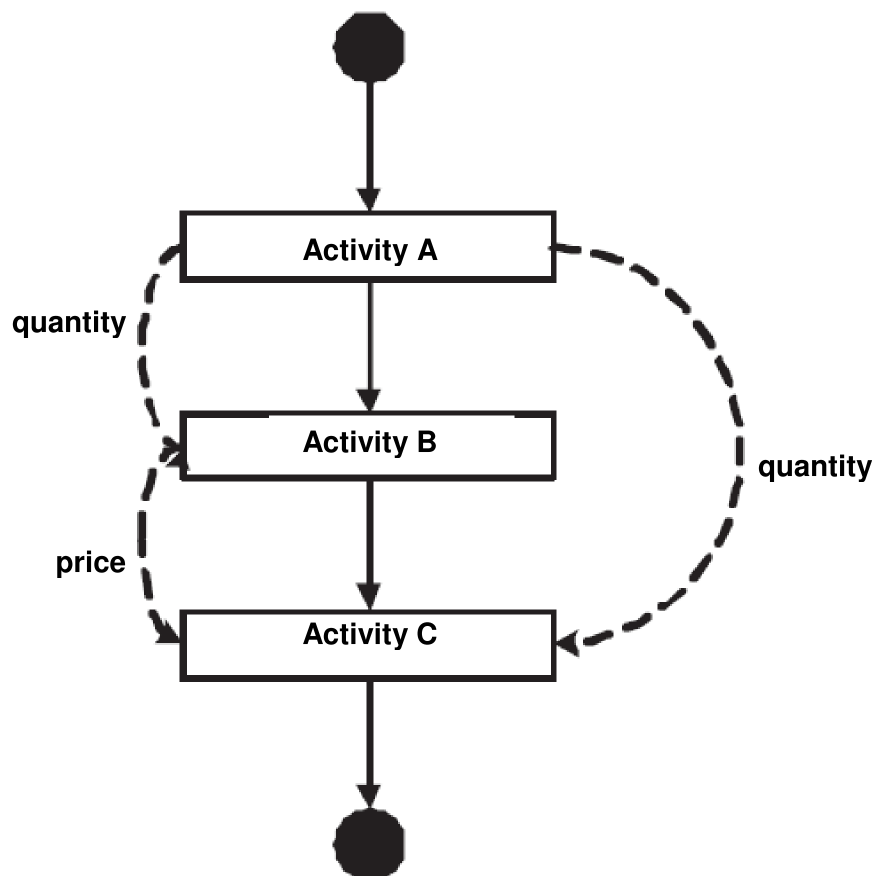


Figure 14: Explicit data flow approach

Selection decisions not only are influenced by the selection time, but – and even at a higher degree – by the selection algorithm itself. As the ontology-driven approach shows, semantic and goal-driven considerations could drive the selection algorithm [AAZM04], as well as context-based or QoS-driven ones. Also, syntactical similarities or abstract services as representatives for a specific class of equivalent services could constitute the decision domain.

Recent proposals have emerged to support WSMO and OWL-S service selection using IRS [CDM04], using the IRS discovery and retrieval mechanisms, mapping semantic service descriptions provided by those two approaches to the knowledge representation language OCML[HDMC04].

In the URBE registry developed for MAIS, services are selected from the registry according to their functional characteristics, organized according to a service model), their quality characteristics, the invocation context, and application or user requirements [BD05]. Similarity functions are provided to assess the functional suitability of a service, according to given functional requirements, in conjunction with a lightweight ontology model.

Static or dynamic bindings affect the selection of a component (web service) by an activity. The target of a request (URI) is an abstract part of a composition schema, typically as *PortType*. Composition engine must resolve the *PortType* so that it can look the end point of web service up. This happens at run time. In other words, a composition must bind to specific services through resolving of *PortTypes* at run time. This can be done in four ways: static binding, dynamic binding by reference, dynamic binding by LookUp, and dynamic binding selection.

- Static Binding

The easiest method to bind a web service is explicitly hardcoding the URI as part of composition schema. This is actually no selection because always the same URI is invoked. This simple approach needs the composition schema be modified when web service URI is changed. Static binding is specifically useful when prototyping and testing.

- Dynamic Binding by Reference

Some of the limitations of static binding can be avoided employing Dynamic Binding by Reference approach, where web services URIs are specified as process variables. In contrast to static binding, process variables and not composition schema need to be modified when web service URI is changed. The variables can be assigned a value through: (1) previously executed operation (2) URI information from clients invoking the composite service (3) explicitly at time of service deployment. In case (1), binding can be provided by an API operation. A web service registry can be accessed by API operation. The stored result (into a variable) can be referenced subsequently (e.g. UDDI[ABC03]).

- Dynamic Binding by LookUp

The composition middleware allows the definition of a query on some directories (registers) for each activity. Results are used for determination of invoked web service URI. The predecessor of BPEL, the composition language WSFL, allows such a query mechanism on a UDDI register.

- Dynamic Operation Selection

Dynamic models allow not only dynamic binding of web service selection but also selection of web service operations. Such service operations are called generic activities l[CAM00]. They do not explicitly specify the service operation. Analogous to the previous mentioned service selection mechanisms, the operations are selected at run time.

2.1.2.6.6 Transactions

As Web Services aim at supporting collaborations between business partners, robust transaction support is required. The classical ACID properties [GPS99] of relational databases have proven being to strict in a service-oriented environment involving several autonomous business partners, and thus, in this context, they have to be slightly relaxed. Also, compensating mechanisms must be taken into consideration, as already happened for WfMSs[GPS99].

Transactions define the transactional semantic associated with composition. Within an orchestration schema some atomic regions may be defined, if allowed by the language. An atomic region consists of a set of activities with all-or-nothing property. This means either none or all of the activities should be executed. If a failure within an atomic region occurs, committed actions must be compensated in order the original state be reestablished. To do this, rollbacks will be performed by the so-called compensation handlers. Each activity or a group of activities can implement such a compensation handler. In August 2002, IBM, Microsoft, and BEA proposed WS-Transaction, a standard protocol for long-running business transactions that builds on the framework provided by WS-Coordination. Transactions are one way to handle exceptions, but due to its compensation mechanism not in every exceptional situation transactions provide the right functionality.

2.1.2.6.7 Exception Handling

Exception handling is performed at runtime without interrupting the service. Exceptions are typically caused by the system (i.e. server is not running) or upon receipt of a fail message of an

invoked web service. There are various ways to model and catch the exception. There are three main models: flow-based, try-catch-throw and rule-based.

- Flow-based approach

This approach is used when the used language supports no exception handling mechanisms, analogous to coding application logic in third generation languages that offer no exception handling support. In general, at the end of a service invocation, the received response is tested for error and appropriate actions are taken respectively. Inclusion of conditional branches after response assures such a behavior when an unexpected response is received.

- Try-Catch-Throw approach

This technique is very similar to exception handling in Java (Sun Microsystems, 2005). The idea behind this approach is association of an exception handling logic to activities or to a group of activities. If an error condition expressed over response data is true, an activity (exception handling portion of code) is executed automatically. Such a Boolean condition can be associated to a single activity or to a specific group of activities, as well as to sub-processes. As it is possible to construct activity hierarchies, exception handlers at higher hierarchy level can handle errors occurred at lower levels of abstraction. Analogous to java (Sun Microsystems, 2005), errors can be caught by higher level methods. If there is no specific exception handling available for a particular activity or a group of activities, a suitable handler is searched by going up the hierarchy toward parents. If no exception handler is found, the process is terminated. Clear separation of service and exception handling logic is the great advantage of this approach.

- Rule-based approach

In the rule-based approaches the exception handling logic is specified by ECA-rules where events define the exception handling. The condition is a Boolean expression over the response that is sent from web service to composition instance. Receipt of no response (Time Out) causes in turn an error. Such rules are defined by a textual language and are generally applicable with only limited number of rules. Otherwise the schema becomes complex and inscrutable.

2.1.2.7 Message Correlation

Once the services that constitute the composite service have been selected, another (runtime) problem must be addressed: message correlation. As there may be several concurrent instances of the same composite service running within one and the same execution environment, these process instances and the conversations they are involved in with external Web Srvices must be uniquely identified for guaranteeing a correct overall process execution.

WS-Coordination proposes identifiers (the *coordination context*) carried by SOAP headers for uniquely associating messages to conversations. When using WSCI, designers can identify certain data items within exchanged messages that act as unique identifiers of the conversation. A possible process specification on top of these protocols must explicitly provide the necessary logic implementing the described mechanisms.

On the other hand, BPEL already proposes a solution at process level, namely so-called correlation sets that – similar as within WSCI – allow defining sets of data items as unique identifiers. By assigning the same correlation set to multiple messages, the designer can specify that messages – whenever the respective data items have the same values – belong to the same process instance or conversation.

2.1.2.8 Web Service Distributed Management (WSDM)

The objective of the Web Service Distributed Management (WSDM) standard is ambitious since it does not represent a new management protocol, but rather aims at using the Web Service technology to unify different management infrastructures to provide a vendor neutral and a platform independent management system. Using a common messaging protocol both for managed resources and consumers, WSDM enables the migration from old management infrastructure to new ones in which management components can be easily integrated into Web Service based business processes.

One of the most interesting aspects of WSDM is its resource orientation approach. In traditional management systems, consumers access resources through management agents that run on the resource side and communicate with consumers using standard protocols (e.g., SNMP and WBEM). WSDM uses a different approach in which resources are Web Services. This allows consumers to access resources directly, without using management agents, since resources can be easily integrated in generic SOA architectures that permit to search and invoke resources using standard Web Service mechanisms. Using WSDM, consumers do not need to concentrate on communication aspects. They are free to consider resources as Web Services that can be easily composed and integrated into their business processes.

The core concept of WSDM is the manageable resource that is accessible by a manageability consumer through a Web Service endpoint. Each manageable resource is described using an XML document (i.e., the resource properties document) defined accordingly to the WSRF specification.

Once that a manageable resource has been defined, manageability consumers can interact with the resource for:

- retrieving the management information about the manageable resource (e.g., retrieve the current operating status);
- affecting the state of a manageable resource (e.g., turn operating status from active to inactive);
- subscribing for receiving notifications from manageable resources (e.g., the resource notifies its operating status changes).

In detail, the interaction with a manageable resource is enabled by a set of manageable capabilities exported by the resource itself. Manageable capabilities are contained within the resource property document of the resource, and are defined as a set of properties, operations, and events, which are exposed via a Web Service interface. Using manageable capabilities, manageability consumers are able to access the properties of a resource, perform operations over a resource, and subscribe to notifications from a resource.

The WSDM standard allows developers to define their manageable capabilities, but also provides a set of standard capabilities that can be exploited.

- the Identity capability, which exposes the ResourceId of a manageable resource;
- the ManageabilityCharacteristics capability, which exposes the list of the supported capabilities by the manageable resource;
- the CorrelateProperties capability, which is useful to understand whether two different ResourceId refer to the same manageable resource;
- the Description capability, which exposes the Caption, Description, and Version of a manageable resource;
- the State capability, which exposes the state of a manageable resource. WSDM allows resources to define its own state model;

- the OperationalStatus capability, which exposes the operational status of a manageable resource. The exposed values can be Available, PartiallyAvailable, Unavailable or Unknown;
- the Metrics capability, which exposes the metric information of the performance and operations of a manageable resource. WSDM provides some metrics but allows resources to define their own metrics;
- the Configuration capability, which exposes the properties of a manageable resource that can be modified by a manageability consumer, changing the behavior of the manageable resource;
- the Relationships capability, which exposes the relationships in which a resource participates;
- the RelationshipResource capability, which exposes the properties of a manageable resource representing a relationship;
- the Advertisement capability, which exposes a mechanism to generate notifications upon the creation or the destruction of a manageable resource.

WSDM consists of two standards known as Management Using Web Services (MUWS) and Management of Web Services (MOWS). The first standard has been developed to manage any resource using Web Services and is composed of two specifications MUWS Part 1 (Vambenepe-1, 2005) and MUWS Part 2 (Vambenepe-2, 2005). The first one describes the basic capabilities of a manageable resource (i.e., identity, manageable characteristics, and correlated properties), while the second describes the remaining capabilities.

The MOWS standard consists of one document, and can be viewed as an application of the MUWS standard. It describes how to deal with Web Services, considering themselves as manageable resources.

2.1.3 Conclusions

The high number of candidate standards for Web Services composition and coordination is mainly due to two reasons: first, vendor-related political and strategic aspects (each supports its specification as a common standard); second, the relatively young age of the Web Service technologies. Unavoidably, this results in a lack of stability when one comes to choose reference specifications. Same problems were encountered in the workflow management systems, where each vendor forced its own proprietary solutions. It took many years before the Workflow Management Coalition agreed upon standards which enabled interoperability [XPDL, WfXML]. But in the Web Services the interoperability is one of the fundamental assumptions – Web Services are autonomous and loosely coupled systems which publish their interfaces and communicate using open and internet based standards. Currently XML is commonly accepted as the (meta-) data format in Web Services. Also SOAP in the messaging layer, UDDI in the discovery layer and WSDL as the interface description are generally accepted and used in the lower layers of Web Services stack. But these are standards which merely describe the basic functionality of simple Web Services. The real added value lies in the composed Web Services and this is the place where several overlapping and competing standards have been proposed like WS-BPEL, BPML, WS-CDL, WSCI, WSCL. They address some aspects of both choreography and orchestration, and sometimes the distinction between the two is not very clear in those proposals. But it is only a matter of time before one dominating standard for Web Service composition will be accepted and implemented by most of the vendors. Moreover, there is an ongoing work on a plethora of WS-* specifications, each concerned with the support for particular functionalities in order to provide suitable APIs and wire protocols for satisfying emerging novel interoperability

requirements. These are the first steps towards commonly agreed on, proper programming libraries for the envisioned SOP infrastructure.

Web Services are loosely-coupled and autonomous. From this perspective it is essential that an enterprise is able to define its own business process described as a composed Web Service and control the execution of this process or at least the execution of parts of the process, which belong to this enterprise. The important components here are the process description language together with the process modelling tools and the execution engines that can interpret this language. Currently only BPEL provides all of that in a form that can be used in practical scenarios. Moreover, there is still ongoing work on BPEL that can justify our prediction that in the near future BPEL will be the dominating standard for the Web Service composition. The BPEL has several advantages over its competitors:

- it is widely accepted de facto standard;
- there are ongoing standardization efforts lead by OASIS [WS-BPEL 2.0];
- this is the only standard proposal which has several implementations (both execution engines and modelling tools) released by big vendors (e.g. IBM, Oracle, Microsoft) and the open source community;
- there are numerous proposals for BPEL extensions, e.g. human users interactions in BPEL [BPEL4People] or subprocesses in BPEL [BPEL-SPE];
- there are several proposals of formal semantics for BPEL (e.g. based on Petri nets [HSS05] or process algebras [Fer04]) which enable verification of business processes described in BPEL;
- BPEL enables both the description of the orchestration of a particular business process (with the BPEL executable process description) and the behaviour of this process in the choreography (with the BPEL abstract process description).

A typical application scenario employing BPEL involves several partners. Each of the partners has its own BPEL engine and executes its private BPEL process, which communicate with BPEL engines and processes of other partners. One or several interfaces to a BPEL process are described with WSDL and made available to the other partners. The communication between partners is provided by exchanging the SOAP messages.

BPEL describes the orchestration and the behaviour of an orchestrated process in the overall choreography. However, it lacks the ability to describe the choreography itself. The choreography should describe the protocol of message exchange between cooperating BPEL processes. Currently the W3C works on a new proposal for describing choreographies – WS-CDL. But these efforts are not supported by the industry and there is available only one partial implementation, so it is difficult to say whether WS-CDL will be widely accepted.

Taking into account the perspectives of the BPEL and its acceptance and current support by many modelling tools and execution engines we have decided to apply it very intensively in our project. Nevertheless, we are aware of many limitations of BPEL and we will analyze them in context of other standard proposals.

2.2 Software platforms

WS-Diamond project has a set of activities for extending and analysing the BPEL processes and invocation of Web Services that has to be developed and tested in the form of software prototypes. Software configuration has to provide the common environment for developing components that release business-logics of diagnosing and executing repair strategies for composed services. It has

to extend and be based on the run-time engine, that executes BPEL process (invokes services, set values of variables according to model of BPEL-process, produces results).

2.2.1 Selection criteria

The WS-Diamond project includes a set of prototypes that are to be developed within the project. Each prototype has to show enhancements that are developed within the project, to existing standards and models of web services diagnosability and reparability.

The prototype development circle includes solving different tasks for:

- developing *mechanisms*, that realize the business-logic of enhancements to the current engines and their components;
- considering the abilities to *deploy* them within the existing business-logic components of engine;
- *testing samples* are to be developed on the base of test-beds examples, and have to be deployed to enhanced software engine;
- *monitoring* the processes, that take place within the engine, to consider how does it process the data from testing samples;
- *debugging* the enhanced engine, finding the errors;
- *analysis* of how the new features and capabilities fulfil the requirements.

Prototypes are to be developed in different groups, and that's why they have to be based and developed on the software platform, that is acceptable by each participant and provides a good environment for solving as many tasks of development process, as possible.

Selected platform has to be a workflow engine, that supports execution of BPEL processes. The most popular and known platforms, that exist on market currently are described in the chapter 2.2.2.

The evaluation criteria set out below reflect some aspects of WS-Diamond project's requirements, work separations and goals. The BPEL engine we adopt ideally should have following features:

- be opensource project – source code has to be open for future developments and testing;
- provide API for developer – has to have declared object-model and interfaces, for accessing the business-logic from third-party applications, have clear components/modules structure and be updateable within these possibilities;
- be compatible with existing standards for web services composition – must have as little limitation on data formats as possible, be compatible with BPEL1.1. specification, which means, it has to work correct with any BPEL-compatible source code project, process all types of invocations, actions, partner-links and variable assignment, be compatible with BPEL1.1. specification standards: WS-Addressing, WSDL1.1., XML Schema 1.0, XPath 1.0;
- run on one of the preferred platforms – be Java-based or run on Microsoft Windows XP platform;
- have a reasonable level of technical support available – supplier of platform has to provide accessible help services for developers and researches;
- provide IDE for designing BPEL process, WSDL descriptions of web services – provide not only run-time engine for executing BPEL process, but also a easy-to-use and flexible

editor for user, that has to design the process. This can be integrated with a BPEL engine software or stand-alone;

- provide facilities for checking consistency of BPEL-source data, check the validity of documents;
- allow to debug the process of BPEL project execution, to monitor the datasets during the process execution, follow the values of input and output data for activities, values of variables etc.;
- have the release version – not be “alpha”/”beta” version, that is on development and waits for big changes of business-logic algorithms and components structure;
- be inexpensive – the cost of the engine has to be acceptable within projects limit. The optimal configuration has to be provided with Academic License.

The selected software platform has to optimally fulfil these requirements to be selected for our future developments within the project.

The evaluation version for 30 days of this tool is available for download from Parasoft site [PBM].

2.2.2 Overview

Current market provides plenty of BPEL engines’ realizations. This section provides the brief description of their main functionality, features and characteristics. On the base of selection criteria, described in chapter 2.2.1 one of them has to be selected as one of the main parts of the overall configuration, as basis, that has to be extended by modules and components for diagnosability and executing repair strategies on composed Web Services.

2.2.2.1 Axis

The Apache Axis 1.2[1] engine (Apache Axis) is one of the most interesting project developed within the Apache Web Service initiative. Axis stands for Apache eXtensible Interaction System and essentially is a third generation SOAP engine extended with a set of features that enable users to deal with SOAP and WSDL 1.1, without worrying about the details of the specifications.

Axis is written in Java and its most interesting functionality is the capability of allowing an easy Web Service development, both on client and server side. Users can automatically generate client stubs from the WSDL description of the Web Service they want to invoke. This means that users do not have to be aware of how SOAP and WSDL works, they can interact with Web Services only by writing simple Java clients, since the parameters serialization/deserialization and the SOAP communication are transparently managed by the auto generated stubs classes.

The same facilities are provided for Web Service developers. The server version of Axis can be installed as a Web application into a Servlet container and gives to developers the ability to publish Web Services with no need to write any integration code. This means that while developers can concentrate on writing Java classes that realize the business functionalities of the service, the Axis engine provides support for both the automatic generation of the WSDL description of the service and the creation of serializer/deserializer that convert external SOAP calls into Java calls. Axis supports four different styles of Web Services:

[1] Axis 2 is now available as a beta release (Apache Axis2)

- ***RPC***, which is the default type and follows the SOAP RPC encoding rules. Using this style, SOAP messages are mapped to methods like this: *public retType method(Type1 input1, Type2 input2)*;
- **Document**, which does not use the SOAP encoding rules and sends messages using plain XML schema. Using this style, SOAP messages are mapped to methods like this: *public void method(Type element)*, where *element* is a JavaBean that can handle the structure of the XML schema contained within the SOAP message;
- **Wrapped**, which is similar to the Document style for the structure of sent messages but it is different for what concerns the mapping with Java methods. Using this style, SOAP messages are mapped to methods like this: *public void method(Type1 element1, Type2 element2)*, where *element1* and *element2* are the XML elements contained within the XML schema of the SOAP message;
- **Message**, which does not use Java objects, and let developers to deal directly with the XML documents during the execution of the published Web Services.

From an implementations point of view, one of the main advantages of Axis is that it is realized using standard specifications designed for Web Services. This approach allows developers to deal with standard API that can be easily exploited and extended to realize particular applications, different from simple Web Services (e.g., integrate Axis into application servers).

In Axis, the management of SOAP messages is done using a SAAJ 1.2 [GK03] compliant implementation, while the deployment of Web Services, the automatic generation of WSDL documents, and the serialization/deserialization of SOAP messages are performed by a compliant implementation of the JAX-RPC 1.1 [C03] specification. Axis also supports the WS-I Basic profile specification,which consists of a set of non-proprietary Web Services specifications, along with clarifications and amendments to those specifications which promote interoperability.

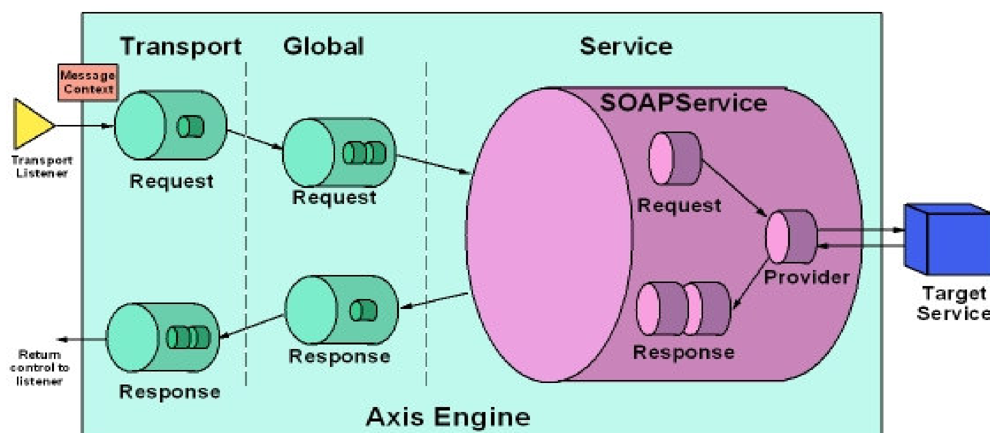


Figure 15 : Axis server architecture

Figure 15 describes the server side message path that is followed by each SOAP message sent towards an Axis engine. The engine is composed of three layers:

- **Transport** layer, which receives SOAP messages in a transport-dependent manner. In this layer, received messages are converted into *Message* objects;
- **Global** layer, which receives Message objects from the transport layers and selects the correct service-specific functionalities that have to be invoked;

- **Service** layer, which is specific to a particular service and is responsible for invoking the correct Web service instance.

Each layer is composed of a *request chain* and a *response chain*, where each chain consists in a sequence of *Handlers* that are invoked in turn. Handlers are responsible for the management of the messages that flows through the layers. The decision of which handlers execute over a particular message is done at deployment time, when developers decide which set of handlers must be associated to the published Web Service. The same approach is used on the client-side as described in Figure 16. For the deployment procedure, on the server side it is executed using *deploy.wsdd* configuration files, while on client it is performed using *client-config.wsdd* files. Both files are formatted following the Axis guidelines.

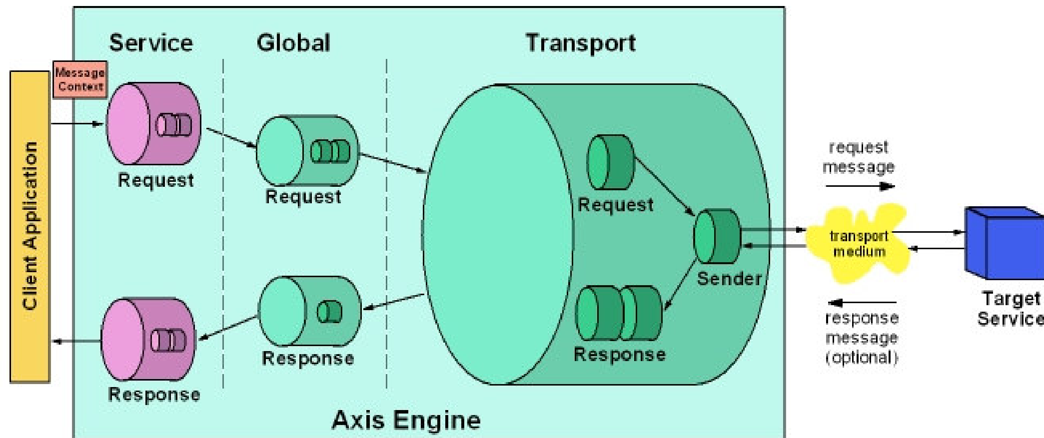


Figure 16: Axis client architecture

2.2.2.2 URBE

URBE is in an enhanced UDDI Registry in which an extended service description is used as a basis for providing service publication and retrieval facilities (see Fig.9). Service description results from the co-occurrence of several components: (i) a UDDI registry is responsible for handling offered service descriptions, (ii) a Domain Ontology provides the general knowledge about concepts of the business domain in which services are used, and (iii) a Service Ontology organizes services at different levels of abstraction. For service publication and retrieval, two matching strategies are applied: a *deductive strategy* with a reasoning procedure exploiting ontology knowledge to assess the type of match among services [BD05]; a *similarity-*

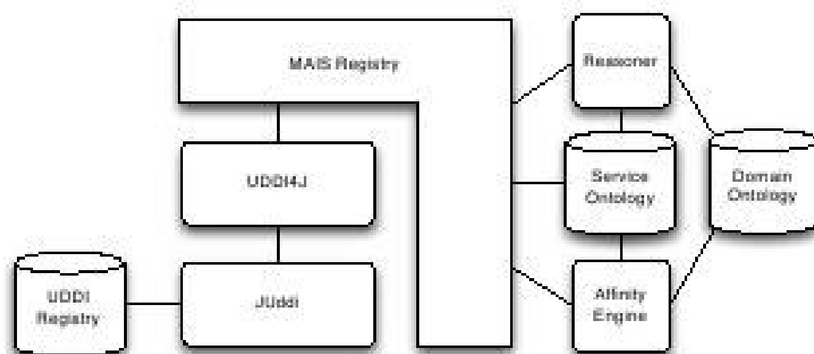


Figure 17: MAIS registry architecture

Based strategy exploiting retrieval metrics to measure the degree of match among services. The similarity approach is applied after the deductive one in order to rank selected services according to the measured matching degree.

For service description, a *service descriptor* is defined, as usually done in the software components retrieval [DF97]. A descriptor is composed of information directly extracted from the service signature expressed in the related WSDL specification. Here, the service name, the operation names, and the names of the parameters involved in those operations, are considered. Once a service provider publishes its services, for each of them a service descriptor is automatically generated starting from the service WSDL specification. The set of service descriptors are organized in a *service ontology* where they are classified according to the functionalities the services provide.

The service ontology is organized in three levels as shown in Figure 18, where each box represents a service descriptor. In the bottom level, the published services are grouped in clusters. These clusters include the services which perform the same functionalities, and can be considered compatible. For this reason, we introduce the term *compatibility classes* to define such clusters. The upper level is populated by services able to represent the compatibility classes. Whereas the services at bottom level are services which can be invoked, the services at upper level are built to represent the cluster, therefore we refer to *concrete services* and *abstract services* to respectively describe such a distinction. In particular:

- Concrete services are actual directly invocable services published by service providers. The result of the discovery phase is one or more of these services.
- Abstract services are not directly invocable services, which represent the capabilities of the concrete services belonging to the same compatibility class.

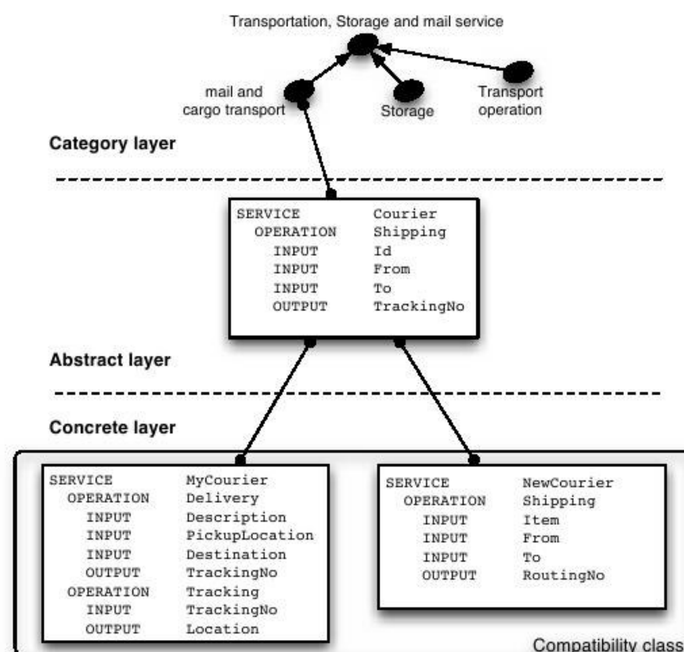


Figure 18: Service ontology structure

In the retrieval process, inference is used to classify the match between the desired service *R* (defined by a set of requested functionalities) and available abstract services *Sa*. Successively, similarity evaluation can be exploited to further refine and quantify the functional similarity between *R* and *S*.

The publication and retrieval processes have been implemented as a UDDI Registry extension. Since one of the most discussed weakness of UDDI Registry is about the limited retrieval method, with our implementation we aim at providing a new way of searching services. In particular, the new searching method allows the user to submit a WSDL expressing the desired service, in order to obtain the list of services able to perform the requested functionalities.

Figure 17 shows the architecture supporting the publication and retrieval process. Such an architecture is designed to be completely compliant with the current UDDI v.2 implementations. To this aim, the system relies on jUDDI, an open source implementation of UDDI which also exposes its functionalities according to UDDI4J API. In particular, the MAIS Registry redefines the functionalities about the service publication and introduces new functionalities which allow the user to perform the advanced retrieval functions based on the service semantics evaluation.

The Affinity Engine performs the similarity evaluation, while the Reasoner performs the deductive matching by exploiting the domain ontology and the service ontology.

During the publication phase, the MAIS Registry is able to read a user publication request and, before performing the standard publication steps required by UDDI, identifies the corresponding abstract service and updates the Service Ontology. In this way the Service Ontology can organize the published services. On the other hand, in case a typical service retrieval method is requested, the related functionality supported by jUDDI is invoked. Otherwise, if the user searches for a service according to the new retrieval method, the new functionality offered by the MAIS Registry is directly invoked. In this case the Affinity Engine, as well as the Service Ontology, are invoked in order to perform the retrieval process.

2.2.2.3 The Active EndPoints project

Active Endpoints project claims to provide a set of instruments for managing Web-based processes. It provides ActiveBPEL, that is a standard and free downloadable BPEL engine under the GNU General Public License (GPL). Another important instrument is ActiveWebFlow, that is a comprehensive environment for creating, testing and deploying BPEL processes. It is based on the Eclipse visual framework. This product is generally sold under commercial license, but it has an interesting Active Endpoints Academic Program [AES] at reduces costs always giving support and licenses according to the necessity.

The ActiveBPEL Engine

The ActiveBPEL engine [AES] is an Open Source implementation of a BPEL engine, written in Java. It reads BPEL process definitions (and other inputs such as WSDL files) and creates representations of BPEL processes. When an incoming message triggers a start activity, the engine creates a new process instance and starts the process execution. The engine takes care of persistence, queues, alarms, and many other execution issues.

The ActiveBPEL engine runs in any standard servlet container such as Tomcat. Figure 19 shows the engine architecture.

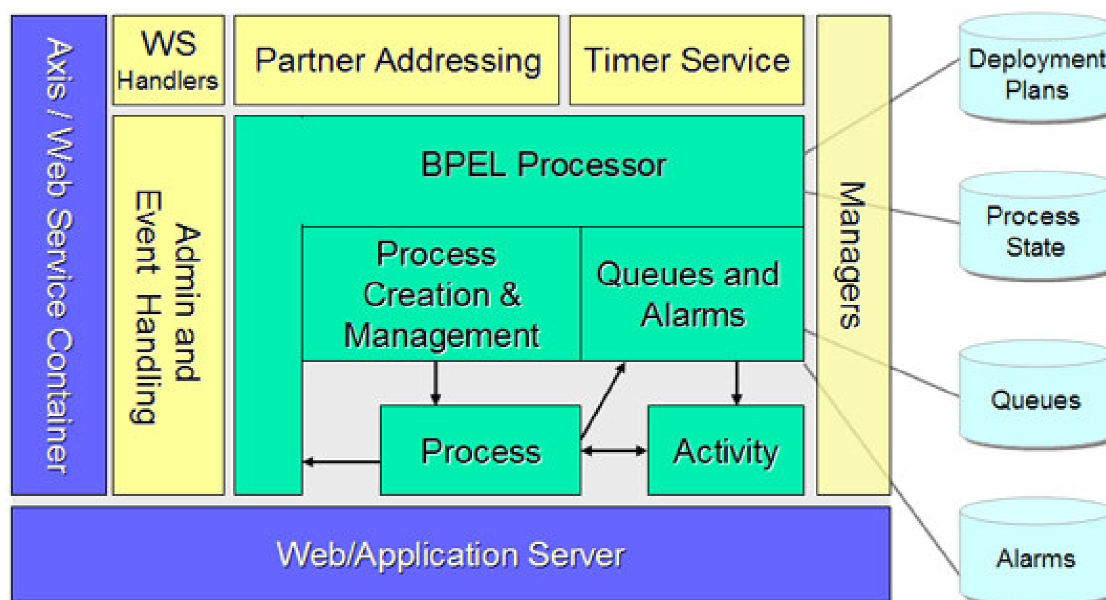


Figure 19: Engine Architecture

The ActiveBPEL engine is released under the GNU General Public License (GPL).

Beyond the free use and the contribution of users that is the core of OpenSource projects , this engine provides several benefits among which comprehensively implementations of BPEL4WS 1.1 specification and some advanced feature like process persistence, event notifications and console APIs.

It is worth notice that the OpenSource BPEL engine is also used in the commercial tools sold by Active Endpoints.

BPEL Process Designers & Tooling

ActiveWebflow Professional [AES] is a comprehensive environment for creating, testing and deploying of BPEL processes. It is based on the Eclipse visual framework. It also includes an embedded copy of the ActiveBPEL Engine.

The Editor palette includes every BPEL activities. When a user drags an activity onto his diagramming canvas, ActiveWebflow automatically generates all of the underlying BPEL code.

As the user creates his own process in diagramming view, he can always switch to code view to inspect his BPEL process definition. The process deployment is supported by a deployment wizard to package everything together and move it to ActiveWebflow server environment.

ActiveWebflow also provides for visual Web References controls for cataloging WSDL files, making it easy to bring Web services into BPEL processes.

BPEL process testing is the most complex task for SOA application development sic requires the analysis of every flow path, condition, and fault to ensure that processes are bullet-proof.

ActiveWebflow supports debugging activities by providing process simulation and debugging tools.

Figure 20 shows a snapshot of ActiveWebflow editor.

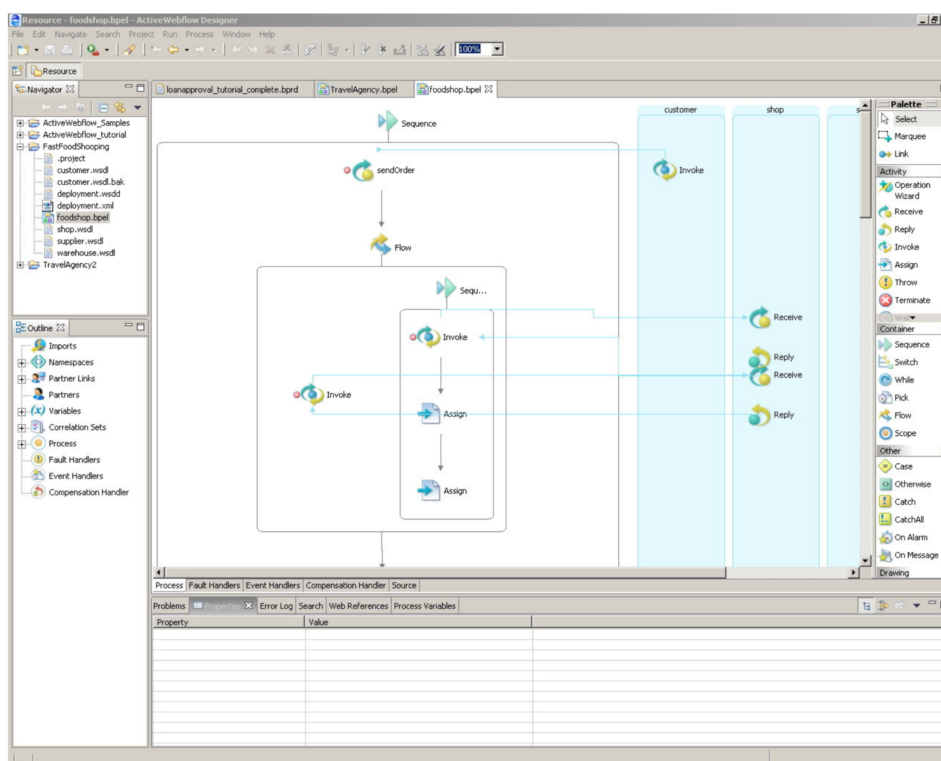


Figure 20: ActiveWebflow snapshot

An editor tool for designing BPEL process is necessary to avoid all problems related to XML writing. Even if the user need good knowledge of BPEL structure and of its relationship with WSDL, this tool is user-friendly and its output is standard BPEL 1.1, this means that it should be compatible also with BPEL engine different from ActiveBPEL.

The usefulness of testing feature is strongly related with the nature of the application the user is going to build. In any case it represents a valid instrument for a static analysis of code before it is published as Web-Service.

Finally the wizard for packing application and publishing it under ActiveBpel Engine is very useful for a quick publication of developed process.

2.2.2.4 BPWS4J

BPWS4J is an IBM execution and deployment environment implementing the BPEL4WS Standard.

BPWS4J components composed by three main components

1. **Runtime Process Model** : it correspond in a in-memory serialisation of the process specification. This model is defined in such manner that facilitate (complete information on the process definition) the processes deployment and execution for the two other component. Its produced sequentially by a parsing and writing steps.
2. **Process Container :** it provide a runtime and deployment environment for BPEL processes. It play a central rôle in the process execution. It handle in and out coming message of the processes.
3. **Interpreter** : it is an event driven flow engine which is responsible of the processes execution management. It manage the with the runtime environment

2.2.2.4.1 Runtime Architecture Overview

The Figure 21 represents the BPWS4J runtime architecture.

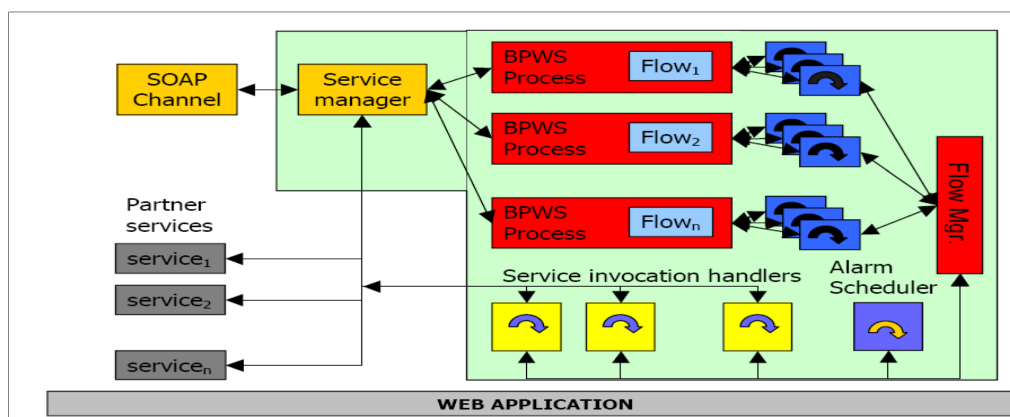


Figure 21 : the BPWS Runtime architecture

Each individual BPEL process model is deployed in the BPWS4J engine as a separate Web application All instances of a process model are thus handled by the same Web application managed by the Container ; The process container serves incoming and outgoing requests. The service manager dispatches incoming invocations to individual flow instances of deployed processes, and sends outgoing ones to process instance partners. The identification of partner services (i.e. services that are able to send messages to a process instance or received them from it) is under full control of the service manager. Flow instances, on the other hand, execute under the control of the flow manager or the Interpreter, which can request The process container.

2.2.2.4.2 Process Container

This component play a central role in the process deployment and execution management.

- Deployment process

The BPEL4WS definition specify the partner by referencing one or more of their WSDL port types. It doesn't mentioning which binding type is used during the process execution. The binding (link to a specific protocol) steps is done during the process deployment only for partner who initiate the interaction within the process it self. For that, the **Process Container** component

complete the **Process Model** information by specifying the binding information for each used partner port type. At this point PPWS4J allow only static deployment.

- Instance management

process instance are created by receiving a specific message (“startable” invocation, defined in the BPEL4WS specification by the attribute creatinstance= “true”). When receiving a message invoking such startable operation the Processes Container decide using correlation information (of existing processes) to create a new instance or to root the received message to an other instance. The new instance is created by cloning the process Model and then passed to the interpreted to manage its execution state.

- Message rooting

All the incoming and the out coming message are managed by the Process Container. For incoming Message the container, using correlation information in the process model, and their instantiation (the value of the correlation set) and the message content to root the message to the first instance which match the correlation condition. While the out coming message concerns for the most cases the invocation of partner operation. For that the Process Container take advantage of Multi-protocol support provided by the WSIF API (Web service Invocation Framework) using a dynamic binding. See next section for WSIF detail.

2.2.2.4.3 The Interpreter or Flow manager

It is the piece of the runtime responsible for one instance process execution. As noted earlier, the interpreter is not aware of the outside world, and uses the Process Container for all of its external interactions. A Process model is compiled into a runnable process Object. The structure of the Object is nearly a direct correspondence with the BPEL4WS activities hierarchy. Each Activity is implemented as Thread owned by the thread of its parent activity in the BPEL process hierarchy. Each activity thread implements the control semantics of the corresponding constructor. The execution of an instance of the process is a recursive set of control action exerted by the hierarchy of activities thread. The control is implemented using the thread status (disabled, activate enabled ruining, complete). Th event oriented constructor such as the pick and the scope activities are realised by associating event handler to such associated Thread with embedded propagation mechanism.

2.2.2.5 ActiveGrid: LAMP Application Server

ActiveGrid's LAMP Application Server is currently Open Source software distributed under the Apache Software Licence 2.0. Their website talks of a commercial version of the software with enhanced features to be released later in 2005. The server supports the latest XML standards including BPEL [AGS].

ActiveGrid's summary of the Application Server's requirements is:

- Platforms: Red Enterprise Server or Advanced Server v3.0 or higher; Novell SUSE
- Enterprise Server, Version 9.0 with SP1 or higher
- Hardware: Pentium 4/Xeon 800 Mhz or better; 1GB RAM; 10GB hard disk space [AGS]

There is an implication elsewhere on the site that the server will run with any standard LAMP stack (Linux, Apache, MySQL, PHP/Python/Perl).

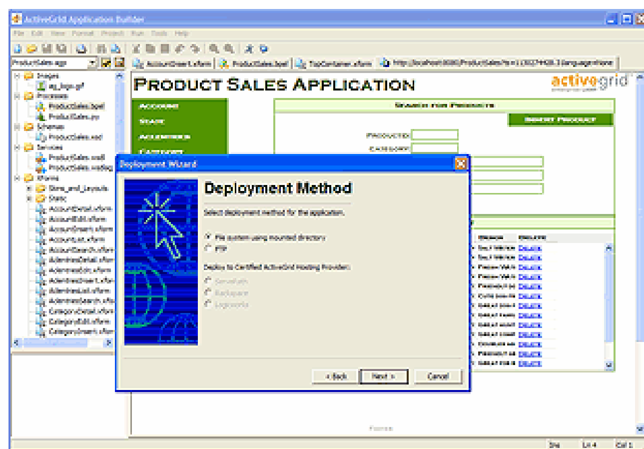


Figure 22: LAMP Application Server.

The company website seems to assume that users will use the ActiveGrid Application Builder software as their authoring tool.

Support for the server package starts at $1000 per year [AGS].

2.2.2.6 Twister

The original Twister website is still available and Twister v0.3 is still available from Sourceforge.net although nothing seems to have been added to the site or associated Blog since the announcement of the Apache adoption on 11 April 2005.

Twister is an open source product licensed under LGPL and written to the "WS-BPEL standard". It is written in Java. The software runs inside a Tomcat servlet container. The website provides extensive documentation and detailed installation instructions. However, to be able to create new processes using Twister you will need a fairly good knowledge of WS-BPEL[tos]. The Twister website invites users needing trainings to contact Smartcomps.org, the original developers. Sourceforge.net has an active Twister users' forum but the last message on it is from Matthieu Riou -responsible for the Twister website - exhorting users to switch to the users' forum on the Apache Agila site.

The software is available for free Download [TOS].

2.2.2.7 Apache Agila

The Apache Incubator Agila project has adopted the 'Twister' Web Service orchestration product so that it will now consist of two parts: Agila BPEL and Agila BPM, the latter providing "end-user oriented workflow". Little information is currently available about the Agila Project [AOS].

2.2.2.8 Cape Clear: Cape Clear 6 Enterprise Service Bus (ESB)

This product includes five components:

- Studio: an Eclipse-based design and development tool
- Server
- Data Transformer: allows the server to handle non-XML and semi-structured data
- Orchestrator: provides BPEL orchestration capability
- Manager: management and monitoring of deployed services

Cape Clear 6 (now in version 6.1) provides "comprehensive support for BPEL 1.1"[CCOS] using native BPEL technology. A comprehensive range of support manuals and tutorials is provided via the company website. Support includes a user forum.

The product runs under Windows 2000 v5 SP1 or later, or XP, or under versions of UNIX. It integrates with many J2EE/CORBA/JMS servers. Orchestration Studio, the Eclipse-based visual development and testing tool, which is included, requires Windows or LINUX.

The website, which is in other respects comprehensive, does not quote prices for the product or support.

Cape Clear Orchestrator is a new product from Cape Clear designed to simplify the design, deployment, and management of orchestrated business processes. Cape Clear Orchestrator provides a comprehensive BPEL runtime, along with extensive graphical design and management capabilities.

Key Features:

- Full BPEL 1.1 support;
- Intuitive Eclipse-based editor;
- Wizards for common workflows;
- Support for complex, long-running processes, with persistence and re-hydration;
- Support for human interactions;
- Transport-independent, sync or async.;
- Extensive logging, auditing, admin and interactive debug and test support;
- Web-based BAM console for process management and drill-down;
- Fully integrated with the Cape Clear ESB[CCOS].

2.2.2.9 Collaxa: BPEL Orchestration Server

Collaxa was bought by Oracle in the summer of 2004 and their BPEL server became the Oracle BPEL Process Manager. [COS]

2.2.2.10 Oracle: BPEL Process Manager 15

The Oracle BPEL Process Manager is a development of the Collaxa BPEL Orchestration server. At the time of writing the version number has jumped from 2.0 to 10.1.2, presumably to bring it into line with the company's Application Server 10g.

The Process Manager provides native and comprehensive BPEL support[OBS] on Oracle Application Server, WebLogic, and JBOSS. The website states that WebSphere is also supported but there did not seem to be a specific download for it as at 15 August 2005. Equally, the FAQ page indicates that the product will run on any J2EE server, but this was not obviously reflected on the downloads page.

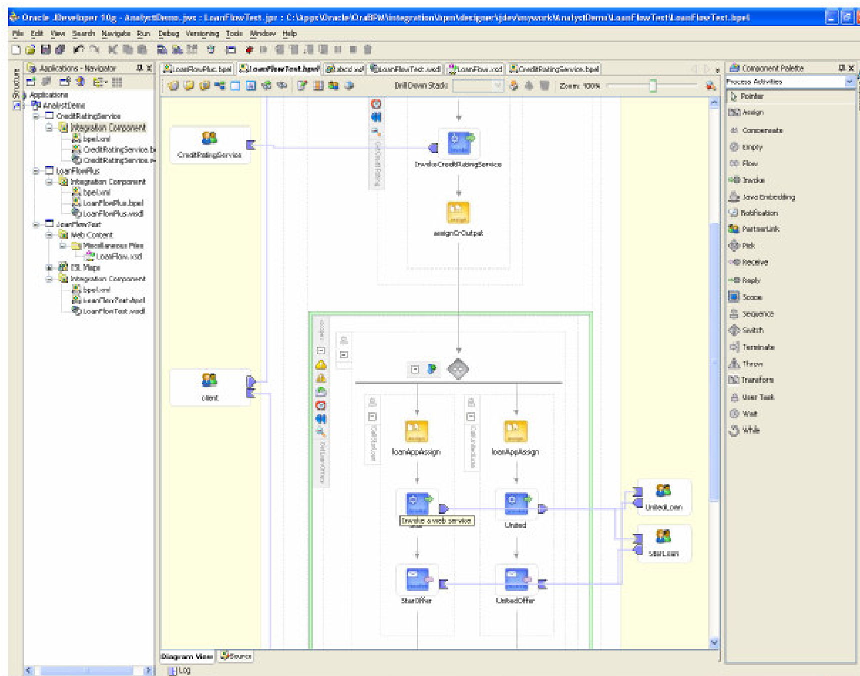


Figure 23: Oracle BPEL Process Manager.

The website provides technical information and access to support.

Oracle provide a visual BPEL Designer which works with Eclipse 3.0 and/or Oracle's JDeveloper development environment.

The price of a perpetual 'processor' licence is $40,000. Time-limited or user limited licences are also available for considerably less[OBS].

2.2.2.11 Creative Science Systems: BizZyme BPEL Java Server

Creative Science BizZyme is part of the company's NetZyme Suite but can be used without the other family members. It is compatible with Windows XP/2000/NT/98, Linux RedHat and SuSE, Solaris, Solaris (x86), FreeBSD, Mac OS X and SGI (in fact any platform that runs Java SDK ... 1.4 or higher [CSS]). It supports any database with a JDBC driver. The company claim full implementation of the latest version of BPEL4WS[CSS]. The product uses a one-pass BPEL compiler and comes with a UML-style graphical design tool[CSS].

Considerable documentation for the product, including an Administrator Guide and a User Guide, is available for download from the website but there does not seem to be an evaluation download and there is no pricing information.

Features:

- OASIS BPEL4WS 1.1 compliant;
- Small footprint;
- Cross-vendor support;
- Sync and async messaging;
- Powerful control flows allowing arbitrary nesting;

- Uses callbacks and correlation sets to compose services;
- Atomic and long-running support using correlation sets;
- Debugging facilities;
- Exception handling;
- Compensation and fault processing;
- Process persistence[CSS].

Only demo version of this tool available, and can be sent by mail on the demo CD. [CSS]

2.2.2.12 FiveSight Technologies: PXE Process execution Engine 10

The FiveSight PXE is an open source product licensed partly under the Common Public Licence and partly under the MIT Licence. The Sourceforge.net pages describe the development status as "4 - beta" but the downloadable files indicate 1.0. The intended audience are software developers and architects. The current version is intended only for experimentation or single-server production use; larger deployment options are under development.

PXE is written in Java to run in a "minimal environment", a J2EE application server or other middleware stack. The documentation claims that it can run both BPEL4WS 1.1 and WS-BPEL 2.0 processes on a single runtime [POS]. However the FAQs page admits that they are working to a BPEL 2.0 recent draft and because the standard is still evolving, all language features are not yet supported. FiveSight plans to provide full support for the OASIS WS-BPEL specification concurrent with its approval as a standard. Currently all BPEL activities are supported. However, certain language constructs (principally BPEL event handlers) are not supported, and certain other constructs may not be fully supported. [POS]

PXE does not have a visual development tool associated, indeed its management is by command line. It should, however, accept "well-formed" BPEL from any source.

PXE runs on any operating system supporting the required Java environment; it has been successfully tested on Windows 2000 and XP, Linux, MAC OS X, Solans and AIX. Although PXE depends on common J2EE interfaces, it does not require a J2EE application server. PXE relies on a Binding API that allows PXE to be embedded in most any environment that can supply JTA facilities. PXE can be deployed into most common application servers but PXE is not an enterprise application in the J2EE sense: PXE manages its own transactions and threads. Consequently, if PXE is deployed using a WAR or EAR file, it will be in violation of a number of J2EE contracts. [POS]

PXE is available for free download, but no source code available.

2.2.2.13 IBM Websphere Business Integration Server Foundation 13

IBM's WebSphere® Business Integration Server Foundation v5.1 includes native support for "BPEL4WS". The server runs on a wide variety of platforms including versions of AIX, HP-UX, Linux, Solaris and Windows.

The area of IBM's website devoted to the server provides access to extensive documentation and support, including a number of user forums and newsgroups.

Use of BPEL on the server envisages use of IBM's WebSphere Studio Application Developer Integration Edition v5.1, their tool for building, testing, integrating and deploying J2EE applications, Web services and business processes[IBI].

Prices for the server start at approximately $49,000 including one year's software maintenance [IBI].

2.2.2.14 Parasoft: BPEL Maestro 16

Parasoft's BPEL Maestro provides native support for BPEL standard [PBM]. Versions are provided for Windows 2000/XP, Linux and Solaris to run in a J2EE sen/let container.

The company website provides information about the product, a user forum and technical support. The software is available for evaluation but no price is provided for the product by itself.

BPEL Maestro includes an Eclipse-based toolkit for developing, reviewing, updating, managing, deploying and debugging BPEL processes. Toolkit provides very easy-to-use interface for designing the BPEL model, using “drag-and-drop” approach. It’s very easy to create new project, add to it several WSDL descriptions of Web Services, and manage the BPEL code. Designers toolkit provide all the features, required on the design stage : graphical representation of BPEL and WSDL codes, editor of properties of each element, intuitive interface, checking the validity of all XML-based code, including issues on importing namespaces. Because of BPEL Maestro is Eclipse-based tool, it is possible within the same IDE to develop java-classes for realisation of business-logic of web services.

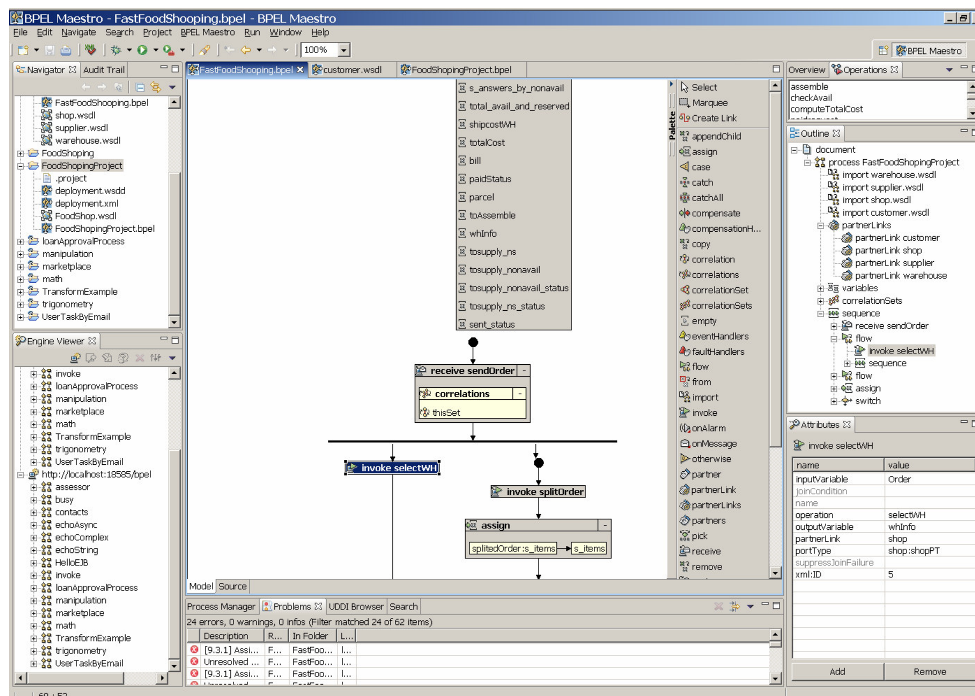


Figure 24: BPEL Maestro tool.

The same tool allows to compile, assemble and deploy BPEL process to the build-in BPEL container, and to debug and test it by sending SOAP-messages to the container engine.

2.2.2.15 JBOSS: jBPM

BPM (business process management) offers a programmatic structure for designing transactions

and executing them using automated decisions, tasks and sequence flows. For example, an

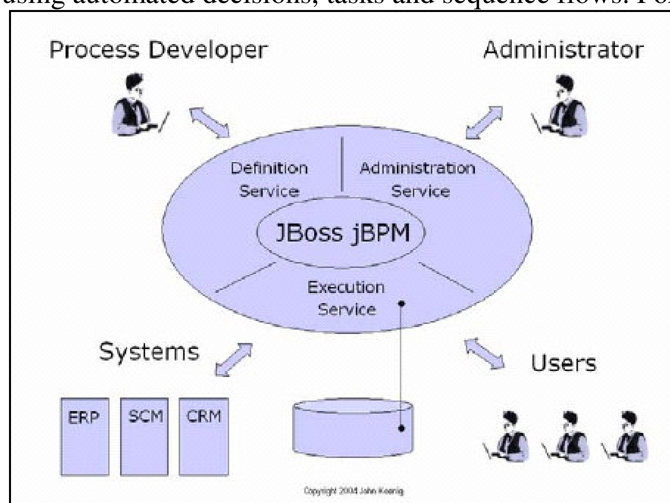


Figure 25: JBOSS architecture

insurance company can use BPM to automate the steps involved in processing insurance claims. BPM solutions typically include three components: an engine that executes process definitions, services that allow the engine to interact with the outside world, and tools that aid process development and monitoring.
Although the notion of “workflow” and BPM have promised enterprise application integration for a number of years, their mainstream acceptance has been delayed by the lack of real standards, and more significantly by the enterprise software architectural model.
With JBoss jBPM support of BPEL and beyond, serves to both encourage and improve the BPEL standard.
BPM can be seen as an orchestration engine that sits in the middle of enterprise applications, enabling integration and coordination between different dedicated applications.

1. Process Engine. The process engine keeps track of the states and variables of all active processes. It includes:
    - A Request Handler: this is the communication infrastructure that forwards tasks to the appropriate process, user or application.
    - Interaction Services: these are standard and custom services that expose existing applications as functions or data for use in end-to-end processes.
    - A State Manager: this module handles potentially thousands of processes including interlocking records and data, and prepares multi-table databases of record as the outcome of actions.
2. Process Monitor: this module provides visibility into the current end-to-end state of processes with which users and applications are interacting. It enables tracking of the status of users or applications that are performing a process.

Process Language: the core engine is based on a directed graph. JPDL, the current jBPM process language, is a powerful extension. On top of the directed graph core engine, can be build support for other standards like BPEL, BPELJ, BPML, ebXML's BPSS, WSCI and WfMC's XPDL.

2.2.3 Common Working environment

Final evaluation of discussed tools may be represented in the table as follows:

Table 2. Software tools evaluation table

	opensource	API	I D E	debug	source check	release	compatable	platform	support	inexpensive
Active BPEL	Y	Y	Y	Y	Y	Y	Y	Y	U	Y
BPWS4J	N	Y	P	P	U	Y	Y	Y	U	N
URBE	U	U	N	N	N	U	N	U	U	U
LAMP	Y	U	Y	Y	Y	Y	U	N	P	N
Twister	Y	Y	N	N	N	N	Y	Y	N	LGPL
Cape Clear	N	N	Y	Y	Y	Y	Y	Y	U	N
Oracle PM15	N	N	Y	Y	Y	Y	Y	N	U	N
BizZyme	N	N	Y	N	Y	Y	Y	U	U	N
PXE	Y	Y	N	N	N	U	Y	Y	P	Y
Websphere 13	N	N	U	Y	Y	Y	Y	Y	Y	N
BPEL Maestro	N	Y	Y	Y	Y	Y	Y	Y	U	N
JBOSS	Y	Y	N	P	U	Y	P	N	Y	LGPL

Y yes
P poor
N no
U unknown

This table show how each of tools fulfills selection criteria. Mark “Y” means, that tool supports features, described in criteria, “N” that not, “U” means, that it’s unclear how full does tool support requirement, and “P” means, that it supports requirement partially.

According to this table, we may select ActiveBPEL as main tool for our future research and development of prototypes. It fulfills all most required issues and has all features that are for us needed.

Because BPEL4WS is the most widely used and accepted standard for describing the orchestration of Web services the consortium decided to use this language for common prototypes. Although every research group is free to apply orchestration languages which fit their needs best, these languages should be compatible with BPEL4WS.

At the current state it is not clear which choreography language and model will fit the needs of the project best. A final decision regarding this issue will be taken in WP3. In addition a decision regarding a Web service management environment remains open.

In order to support collaboration and integration of software the consortium will use JAVA as the implementation language. In addition the consortium will encapsulate their prototypes (e.g. diagnosis or repair modules) as Web services such that various groups can reuse and integrate implemented systems.

In addition the consortium agrees to base their work on WSDL-S and URBE as a registry. Finally, we require from databases used within WS-Diamond to be JDBC compliant.

3 Application Scenarios

3.1 Test Case: Food shop

3.1.1 Workflow

The FoodShopping example is concerned with a FoodShop Company that sells and delivers food.

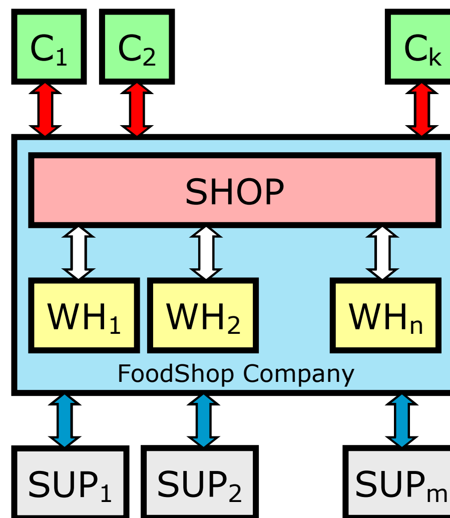


Figure 26: FoodShopping example actors

The company has an online SHOP (that does not have a physical counterpart) and several warehouses (WH_1, …, WH_n) located in different areas that are responsible for stocking unperishable goods and physically delivering items to customers, depending on the area each customer lives in.

Customers (C_1, …, C_k) interact with the FoodShop Company in order to place their orders, pay the bills and receive their goods.

In case of perishable items, that cannot be stocked, or in case of out-of-stock items, the FoodShop Company must interact with several suppliers (SUP_1, …, SUP_m).

Although most of the interactions in this example are electronic, and take place between Web Services, in some cases there are physical actions and interactions that are performed by humans (e.g. the sending of a package). These too are modeled in the context of Web Services.

The Conversation

In each conversation the following actors take part:

- one CUSTomer (represented in green);
- the online SHOP (represented in pink);
- one WAREHOUSE (represented in yellow);
- a variable number of SUPPLIERS, which could also be 0 (represented in gray).

When a CUSTomer places an order, the SHOP first selects the WAREHOUSE that is closest to the customer's address, and that will thus take part in the conversation.

Ordered items are split into two categories: perishable (cannot be stocked, so the warehouse cannot possibly have them in stock) and unperishable (the warehouse might have them).

Perishable items are handled directly by the SHOP, while unperishable items are handled by the WAREHOUSE.

The first step is to check whether the ordered items are available, either in the warehouse or from the suppliers (we have not considered items exchanges among different warehouses, in order not to make the example too complicated). If they are, they are temporarily reserved in order to avoid conflicts between several orders.

Once the SHOP receives all the answers on availability, it can decide whether to give up with the order (again, in order to keep things simple, this happens whenever there is at least one unavailable item) or to proceed. In the former case, all item reservations are canceled and the conversation ends.

If the order goes on, the SHOP computes the total cost (items + shipping) with the aid of the WAREHOUSE, that provides the shipping costs. Then it sends the bill to the CUSTomer, that can decide whether to pay or not. If the CUSTomer does not pay, all item reservations are canceled and the conversation ends here.

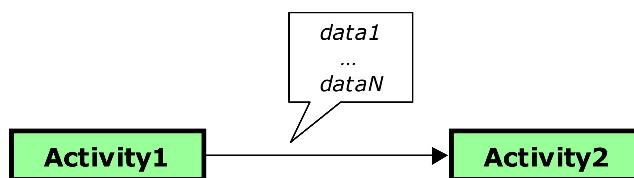
If the CUSTomer pays, then all item reservations are confirmed and all the SUPPLIERS (in case of perishable or out-of-stock items) are asked to send the goods to the WAREHOUSE. The WAREHOUSE will then assemble a package and send it to the CUSTomer.

Workflow

We describe separately the workflow of each actor, including its interactions with other actors in the same composite workflow.

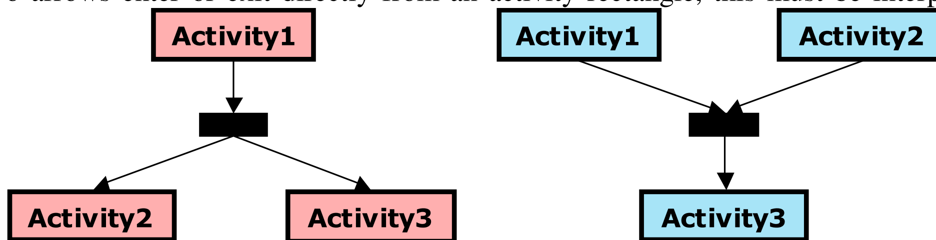
Notation

Each individual workflow is represented with an activity diagram: each activity is represented by a rectangle; flow is represented by incoming or outgoing arrows and data exchanged along the flow is mentioned in the callout boxes associated with each arrow.

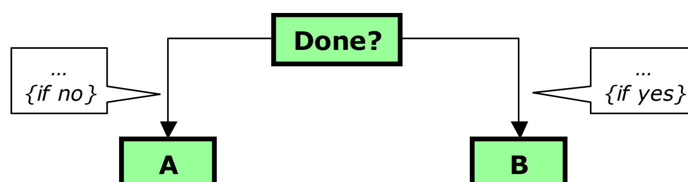


When two parallel flows are started by an activity, the two arrows depart from a black small box that has the activity rectangle in input. Analogously, if two flows must synchronize in order for an activity to take place, the two arrows enter into a black small box that has the activity rectangle in output.

When two arrows enter or exit directly from an activity rectangle, this must be interpreted as a



disjunction: only one of the two can actually happen. When this happens with output arrows, the conditions for each of the two possibility to happen is mentioned in the callout box together with



the arrows.

Black arrows denote electronic interactions; blue arrows denote physical interactions.

Thick arrows are used to represent several simultaneous interactions with different senders and/or receivers. For example, the following means that activity B can take place only after receiving several physical items as a result of several different senders performing activity A:

CUSTomer workflow

The customer workflow (Figure 27) is abstract: we represent only its interface with the other services, while we do not represent internal activities. The reason is that the customer is an external entity wrt the company, thus we cannot assume to have its detailed workflow. It seems reasonable to have in the example both detailed and abstract workflows.

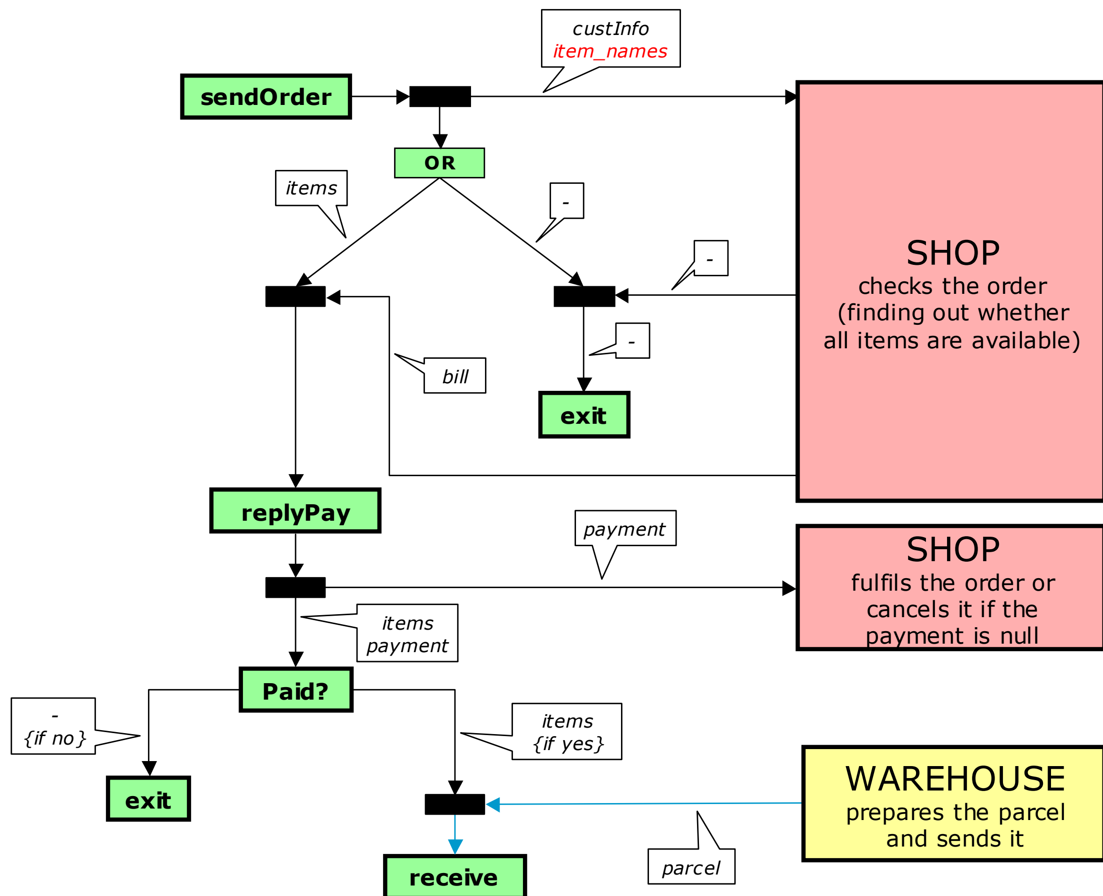


Figure 27: CUSTOMER workflow

The CUSTomer places an order (**sendOrder**) communicating the items he/she is interested in (*items*) and its personal data (*custInfo*). Then it waits for an answer from the SHOP: if some of the items are not available the conversation ends (**exit**). Otherwise the user receives the *bill* and decides whether to pay (**replyPay**) sending its *payment* to the SHOP.

If the CUSTomer decides not to pay the conversation ends (**exit**). Otherwise, he/she waits for the *parcel* sent by one of the company's WAREHOUSEs. Notice that the parcel shipment is a physical transaction, while the others are all electronic transactions.

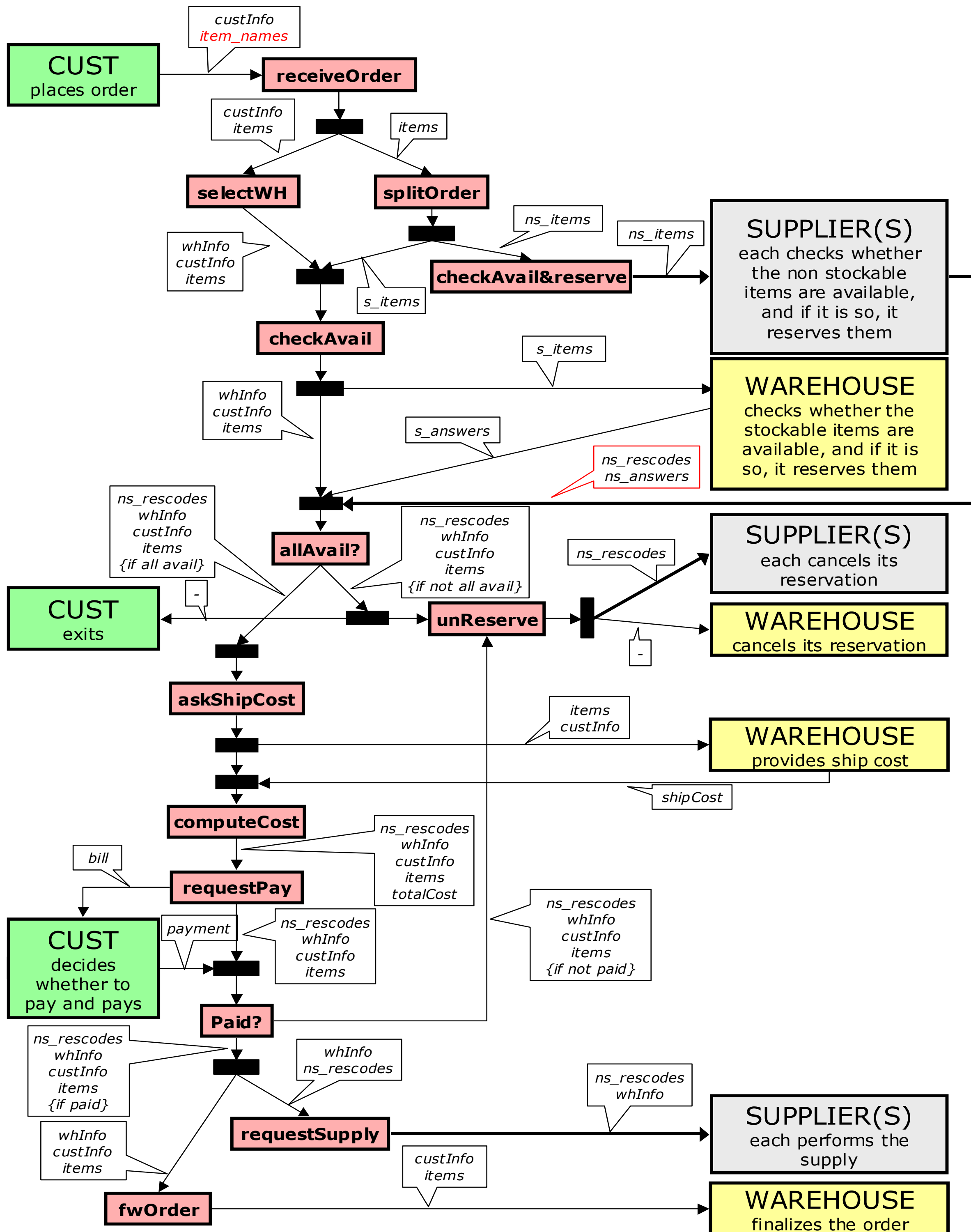


Figure 28: SHOP workflow

SHOP workflow

On the contrary, the SHOP workflow (Figure 28) is detailed, and contains several internal activities.

When the SHOP receives an order (**receiveOrder)** with the ordered *items* and the CUSTomer data (*custInfo*), it selects the WAREHOUSE that is closer to the user (**selectWH**) and splits (**splitOrder**) the ordered items into the set of perishable items (*ns_items*) and that of unperishable items (*s_items*). It then checks the availability of perishable items (**checkAvail&reserve**) with the SUPPLIERS, asking to temporarily reserve them in case they are available. The SHOP receives back the set of reserved items (*ns_resitems*), the corresponding reservation codes (*ns_answers*), and the answers on availability (*ns_answers*).

The list of unperishable items is instead sent to the WAREHOUSE (**checkAvail**), that sends back a collective answer (*s_answers*) on availability.

If any of the items is unavailable, the order is canceled. The SHOP communicates this to the CUSTomer, and cancels the reservations (**unreserved**) both with the SUPPLIERS and the WAREHOUSE.

If on the other hand all the items are available, the SHOP asks the WAREHOUSE to compute the ship cost (*shipCost*), which depends on the distance between the WAREHOUSE itself and the user address, as well as the total weight of the ordered items (for this reason, the SHOP sends to the WAREHOUSE both the list of *items* and *custInfo*).

Then the SHOP computes the *totalCost* and sends the *bill* to the CUSTomer, which sends back a *payment*. if the CUSTomer decides not to pay, the SHOP cancels all the reservations (**unreserved**) with the SUPPLIERS and the WAREHOUSE. If the payment is ok, the SHOP forwards the order to the WAREHOUSE (**fwOrder**), which from now on is responsible for it, and tells the SUPPLIERS to send the reserved items to the WAREHOUSE (**requestSupply**), providing the reservation codes (*ns_rescodes*) and the warehouse address (*whInfo*).

WAREHOUSE workflow

Again we have a detailed workflow (Figure 29).

The WAREHOUSE first receives a request from the SHOP to check the availability of some items (*s_items*) and reserve them (**reserveAvail**). If some items are out-of-stock, the WAREHOUSE contacts the SUPPLIERS in order to check for availability and to reserve them (**findSuppliers**), receiving back the set of reserved items (*s_resitems*), the corresponding reservation codes (*s_rescodes*) and the answers on availability (*s_answers*).

TheWAREHOUSE elaborates a collective answer on availability and sends it to the SHOP (**collectAnswers**). Then it waits for one of the following things to happen: either the SHOP decides to cancel the order, or to proceed.

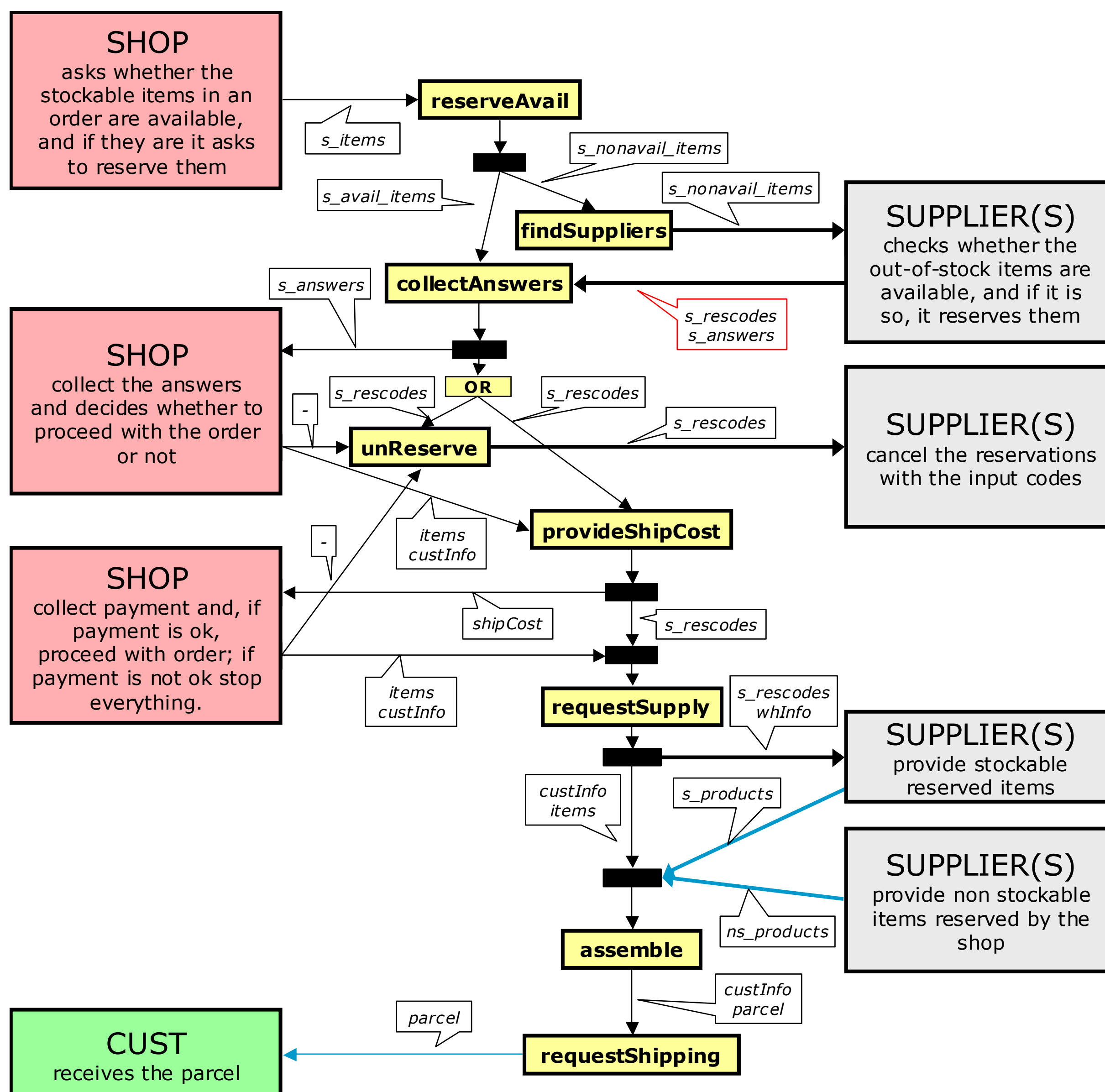


Figure 29: WAREHOUSE workflow

In the first case the WAREHOUSE has to cancel its own reservations, and, in case some SUPPLIERS were contacted, it must also cancel the reservations with the suppliers (**unreserved**).

In the second case, the WAREHOUSE is asked by the SHOP to compute the shipment cost.

Then the SHOP tells the WAREHOUSE to proceed with the order. In case of out-of-stock items, the WAREHOUSE asks the SUPPLIERS to send the reserved items (**requestSupply**), by providing the reservation codes (*s_rescodes*) and its address (*whInfo*).

At this point the WAREHOUSE must **assemble** the package. In order to do this, it must wait both for the (unperishable) items it reserved directly from the SUPPLIERS, and for the (perishable) items that were reserved by the SHOP.

Once the *parcel* is ready, the WAREHOUSE asks a shipper (**requestShipping**) to send it to the user.

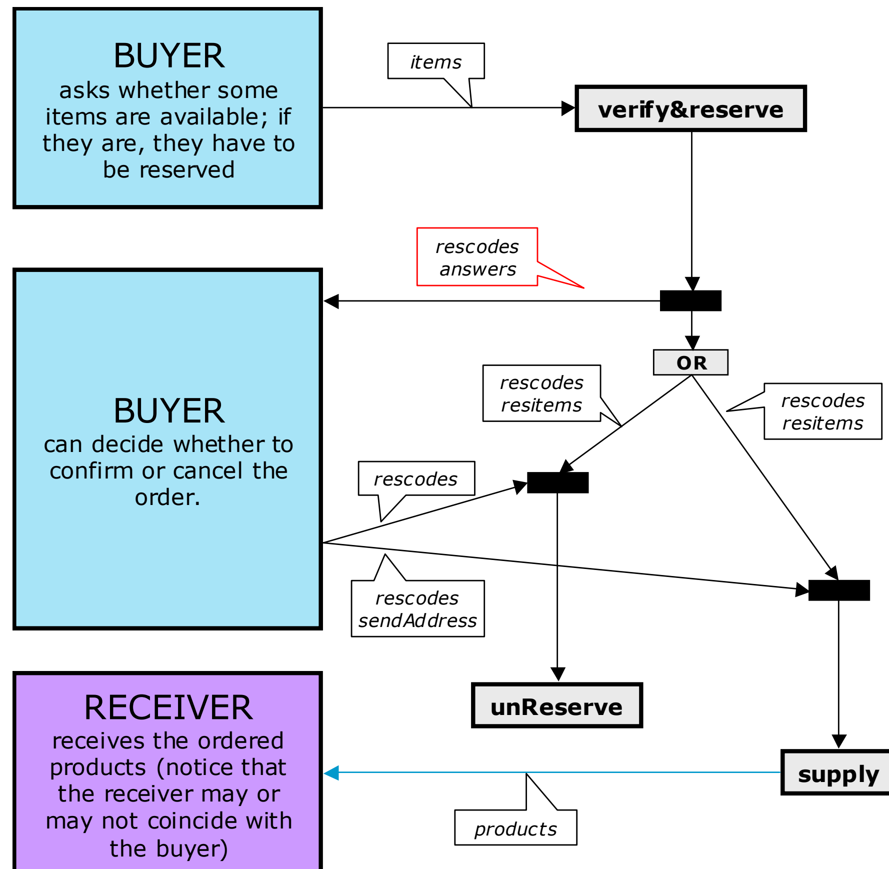


Figure 30: SUPPLIER workflow

SUPPLIER workflow

Like the CUSTomer workflow, the SUPPLIER workflow (Figure 30) is abstract since each supplier may have a different internal workflow.

Of course, it is the same workflow independently from the Web Service that contacts the SUPPLIER. For this reason, the Web Service that buys the goods is generically called BUYER, while the receiver of the products is generically called RECEIVER. It is clear that in our context the BUYER can be either the SHOP or the WAREHOUSE, while the RECEIVER is always the WAREHOUSE.

The SUPPLIER is first asked by the user to verify the availability of some *items* and reserve them (**verify&reserve**). The SUPPLIER sends back the set of reserved items (*resitems*), the corresponding reservation codes (*rescodes*) and the *answers* on availability.

Then the BUYER can either cancel the reservation (**unReserve**) or ask the SUPPLIER to send the items (**supply**) to the address (*sendAddress*) of the RECEIVER.

3.1.2 Exceptions

CUSTomer exceptions

- *WrongBillException*. CUST checks the bill and realizes that there is something wrong (missing and/or unwanted items) (just before *replyPay* activity)

- *TimeOutException*. CUST is waiting for some feedback from the shop (either an unavailability notification, or a request for payment) but none of the two takes place (just before *replyPay* activity).
- *WrongParcelException*. CUST receives a parcel with missing and/or unwanted items (upon *receive*).
- *TimeOutException*. CUST never receives the parcel (just before *receive*).

SHOP exceptions

- *WrongAnswerException*. For some items the answer from WAREHOUSE/SUPPLIER is missing, or the answer is about a different item than asked for (upon *allAvail*).
- *TimeOutException*. The SHOP never receives an answer on item availability either from the WAREHOUSE or from the SUPPLIERs. (just before *allAvail*).
- *HighShipCostException*. The shipping cost sent from the WAREHOUSE is higher than an expected threshold
- *TimeOutException*. The SHOP never receives an answer on the ship cost from the WAREHOUSE (just before *computeCost*).
- *TimeOutException*. The SHOP never receives an answer from the CUSTomer on whether he/she wants to pay or not (just before *Paid*).

WAREHOUSE exceptions

- *TimeOutException*. Some answers on item availability never arrive from the SUPPLIERs. (just before *collectAnswers*).
- *WrongAnswerException*. For some items the answer from the SUPPLIERs is missing, or the answer is about a different item than asked for (upon *collectAnswers*).
- *TimeOutException*. The WAREHOUSE never receives from the SHOP an answer on whether to cancel the reservation or to proceed computing the ship cost (after *collectAnswers*).
- *TimeOutException*. After providing the ship cost, the WAREHOUSE never receives an answer from the SHOP on whether to cancel or to proceed with the order (after *provideShipCost*).
- *WrongSupplyException*. Some items that arrive from the suppliers are wrong (upon *assemble*).
- *TimeOutException*. Some items never arrive from the SUPPLIERs (upon *assemble*).

SUPPLIER exceptions

- *WrongResCodeException*. The reservation code is not recognized by SUPPLIER (either upon *unreserved* or upon *supply*).
- *TimeOutException*. The buyer (SHOP or WAREHOUSE) never tells SUPPLIER whether to cancel the order or proceed with it (after *verify&reserve*).

3.1.3 Preliminary model of the process

Design of the BPEL process that models simplified food shopping test sample was done within the Parasoft BPEL Maestro IDE tool.

The project, which models the aspects of food shopping test bed, is organized as following:

- customer.wsdl – WSDL definition of Customer Web Service. This service wraps human activities of Customer actor;
- warehouse.wsdl – WSDL definition of Warehouse Web Service functionality;

- supplier.wsdl – WSDL definition of Supplier Web Service. Applies to a set of Web Services, each of them releases the functionality of Supplier actor, defined within the process definition;
- shop.wsdl – WSDL definition of Shop Web Service functionality;
- FastFoodShopping.bpel – BPEL model of the process;
- deployment.wsdd – common deployment data of the project.

The full source code of the project can be found in Appendix A of this Deliverable. In this chapter we will focus on the main aspects of BPEL model, which are essential for this application scenario.

Messages, which are exchanged among the partners within the process are defined with definitions of services in their namespaces. For example, *orderMsg* from "*http://wsdiamond.com/wsdl/foodshopexample/customer*" namespace defines the format of Order, that system receives from Customer and starts the whole process.

PartnerLinkType *supplierLT* defines, that the *role* "*supplier*" is defined on the operations from *portType* "*supplierPT*". There are 3 operations in this portType:

- supply : to supply items to warehouse
- veryfyAndReserve: to verify and reserve requested items, return result has type *answers*, so it containts answers and reservation codes for items;
- unreserve: to unreserved items, that previously were reserved for some warehouse.

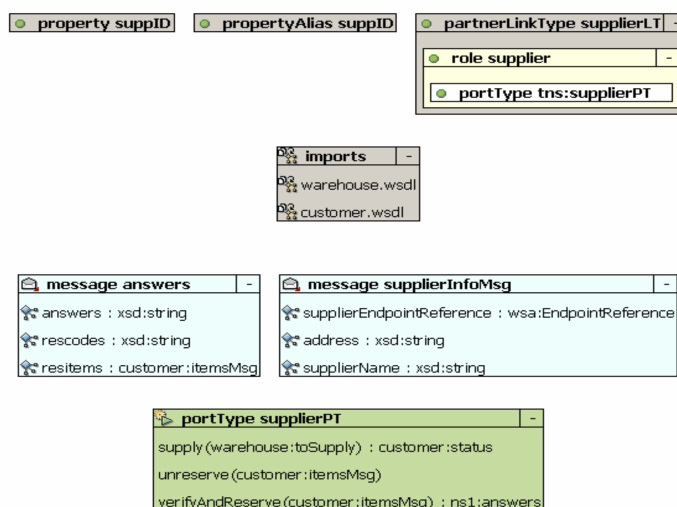


Figure 31: Definition of Supplier service

As defined within the process description, in one workflow participate exactly one customer, one shop, one warehouse and a priory unknown amount of suppliers. For the supplier identification, message *supplierInfoMsg* contains the endpoint reference of the supplier instance service. So, for identifying the instance of supplier service we use WS-Addressing type "*wsa:EndpointReference*". It is used in the part "*supplierEndpointReference*" of message "*supplierInfoMsg*". WS-Addressing types are supported by BPEL1.1 specification.

Within the BPEL processes model, this identification mechanism will be used to make the correct *correlation* during the conversation between *warehouse* and *supplier* services.

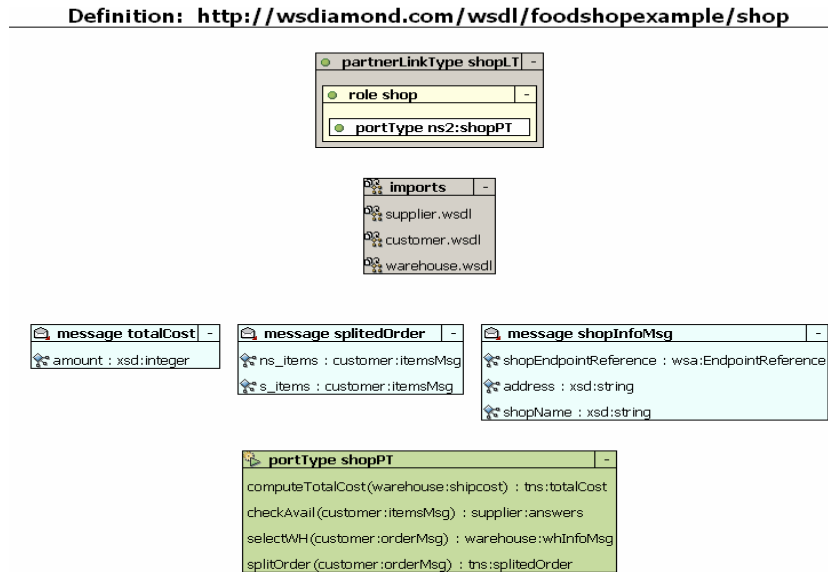


Figure 32: Definition of Shop Service

Within the Shop Services definition the operations of shop are provided. This definition has references to supplier, customer and warehouse namespaces for defining types of messages, that are used within the messages' parts and input/output variables of operations from *shop* PortType. Shop service by himself, provides definition of 3 new messageTypes:

- *totalcost*: total cost of order, that will be show to customer, contains one integer value;
- *splitedOrder*: contain two parts: set of *s_items* and *ns_items*, as they are defined within process description;
- *shopInfoMsg*: shop address, name, and ID.

PortType *shopPT* contains functions:

- *computeTotalCost*: computes total cost of assembly for customer;
- *checkAvail*: check availability of items for customer;
- *selectWH*: selects the warehouse, that is near to customer;
- *splitOrder*: splits *order* to *s_items* and *ns_items*.

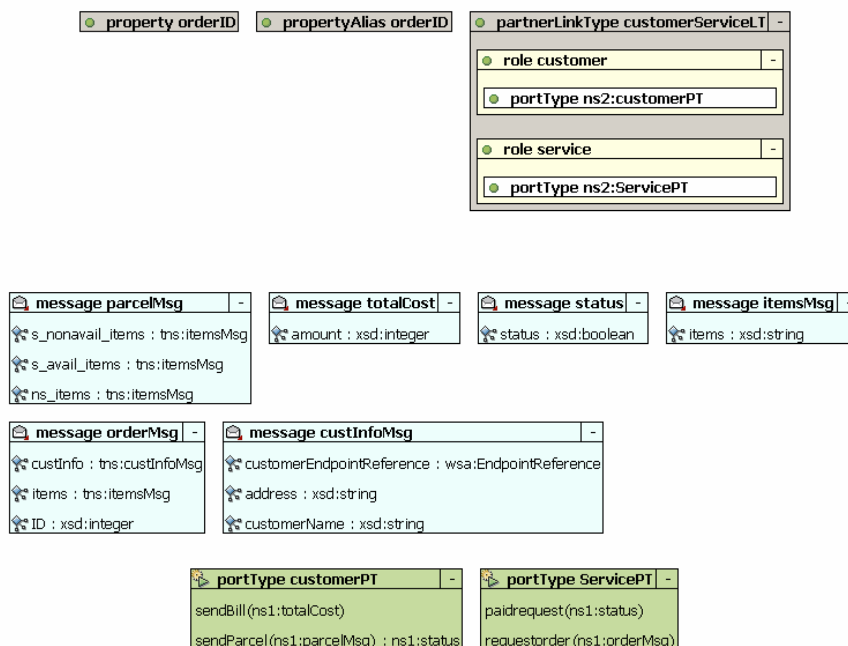


Figure 33: Definition of Customer Service

Customer Web Service wraps the human activities of customer. This service starts the processes and gets the final result, and message format of this actor is essential for all the participants of the whole process. That's why it contains not only customer-specific messages and operations (from portType "*customerPT*"), but also some data that relates on the overall process of food shopping.

These are types "*parcelMsg*","*itemsMsg*","*status*" and operations "*requestorder*" and "*paidrequest*":

- "*requestorder*" instantiates the overall process
- "*paidrequest*" is needed to provide conversation with customer.

Such operations usually appear in "*request*" statements of the BPEL code. They model the peer-to-peer conversation between process by itself and one of the actors. Such operations are exactly actor specific ("*requestorder*" is performed exactly by customer), but may not be invoked from this service.

That's why they are defined within other portType - *ServicePT*, which later within the BPEL process will be associated with the "*myrole*" property of partnerLinkType. It means, that "*requestorder*" and "*paidrequest*" are associated with BPEL engine by itself, and he will invoke them from customer service. BPEL engine acts in this case as some kind of "super actor" among all other actors in their collaboration.

Description of the Customer Service is shown on the Figure 33.

The Warehouse Service description contains main data types (messages), that are exchanged with warehouse and operations, which may be invoked from Warehouse.

- *assemble:* assembling items in one parcel, to be delivered to customer;
- *shipcost*: computes cost of shipping goods;
- *unreserve*: un-reserves the items in warehouse, that were previously reserved;

- *reserveAvail:* reserves items, that are available at warehouse and responds with message, that contains information which items are available at warehouse, and which have to be shipped.

Messages within Warehouse definition define main data formats, needed to communicate with warehouse:

- *toSupply*: which items to supply from supplier with warehouse identification;
- *shipcost*: cost of goods' shipment;
- *availItems*: sets of available and non-available items at warehouse;
- *whInfoMsg*: information about Warehouse. Needed for selecting the nearest to customer warehouse.

The only reference to customer namespace is needed for working with "*customer:itemsMsg*" message type.

The definition is shown on the Figure 34

Definition: http://wsdiamond.com/wsdl/foodshopexample/warehouse

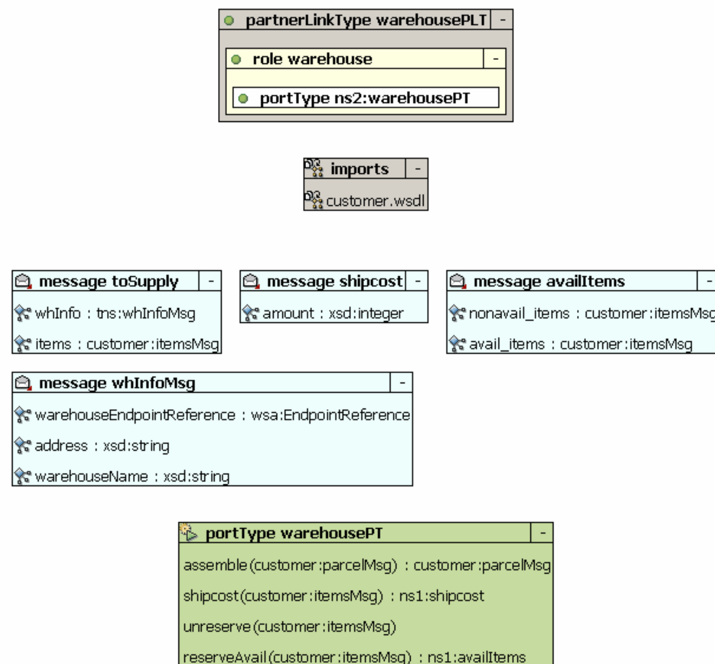


Figure 34: Definition of Warehouse Service

Overall process definition contain parts of defining partner links, import of namespaces from associated web-services, definition of process variables, correlation sets, and a definition of a workflow by itself.

PartnerLinks define which partner roles are associated with which portTypes from services. The non-trivial definition concerns the customer role, where "myrole" for the overall process is defined (see description above).

<partnerLinks>

<partnerLink myRole="service" name="customer"

partnerLinkType="customer:customerServiceLT" partnerRole="customer"/>

<partnerLink name="shop" partnerLinkType="shop:shopLT" partnerRole="shop"/>

<partnerLink name="supplier"

partnerLinkType="supplier:supplierLT" partnerRole="supplier"/>

<partnerLink name="warehouse"

partnerLinkType="warehouse:warehousePLT" partnerRole="warehouse"/>

</partnerLinks>

The overall diagram of the process is shown on the Figure 35 - Figure 38.

Variables part of the process definition contains description of all the variables, that process use to send to invocating operations of services and to store data after performing this actions. Among them we can find as common variables, related to process description, such as

<variable messageType="customer:itemsMsg" name="s_avail_items"/>

<variable messageType="customer:itemsMsg" name="s_nonavail_items"/>

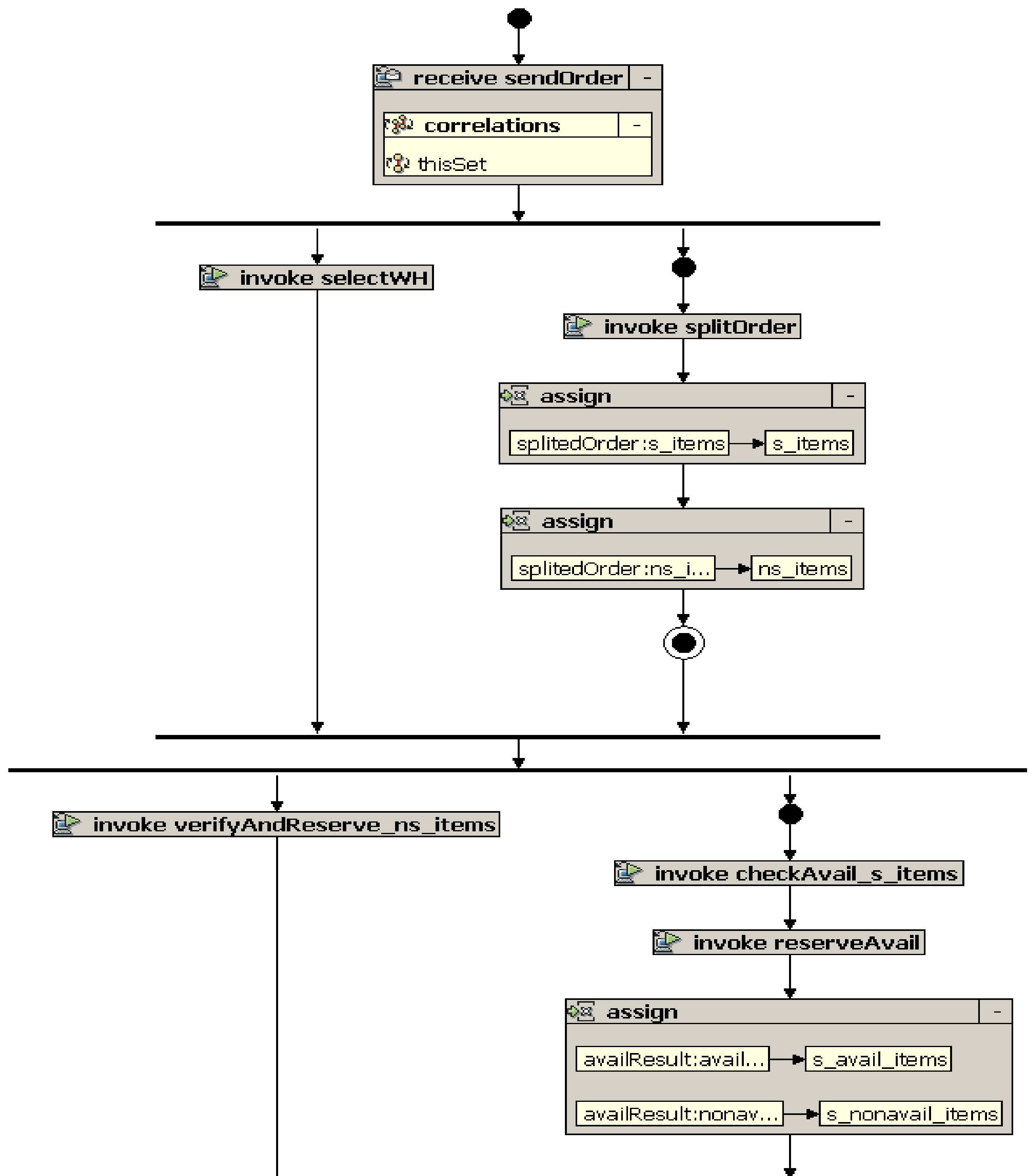


Figure 35: BPEL process model /1

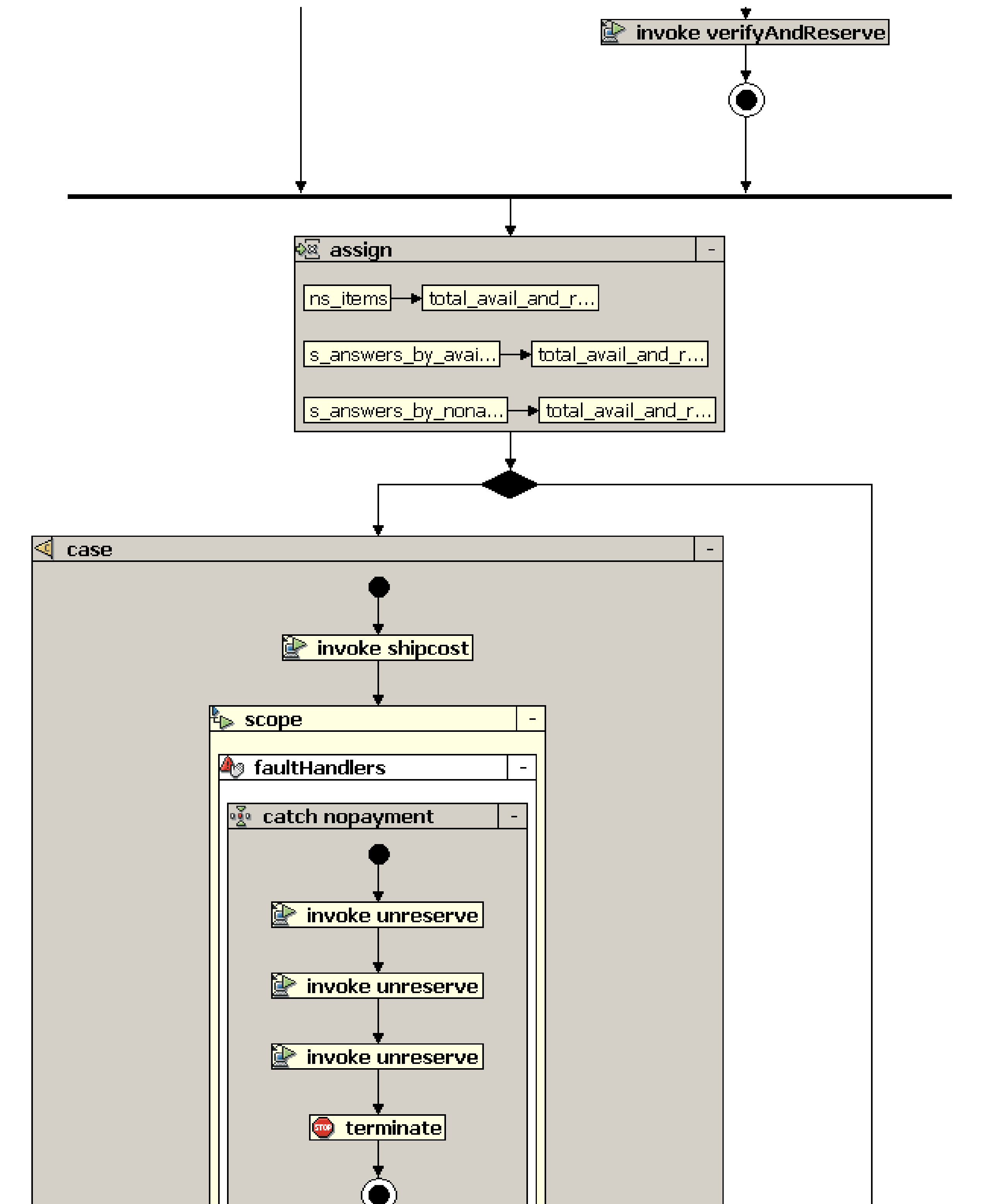


Figure 36: BPEL process model /2

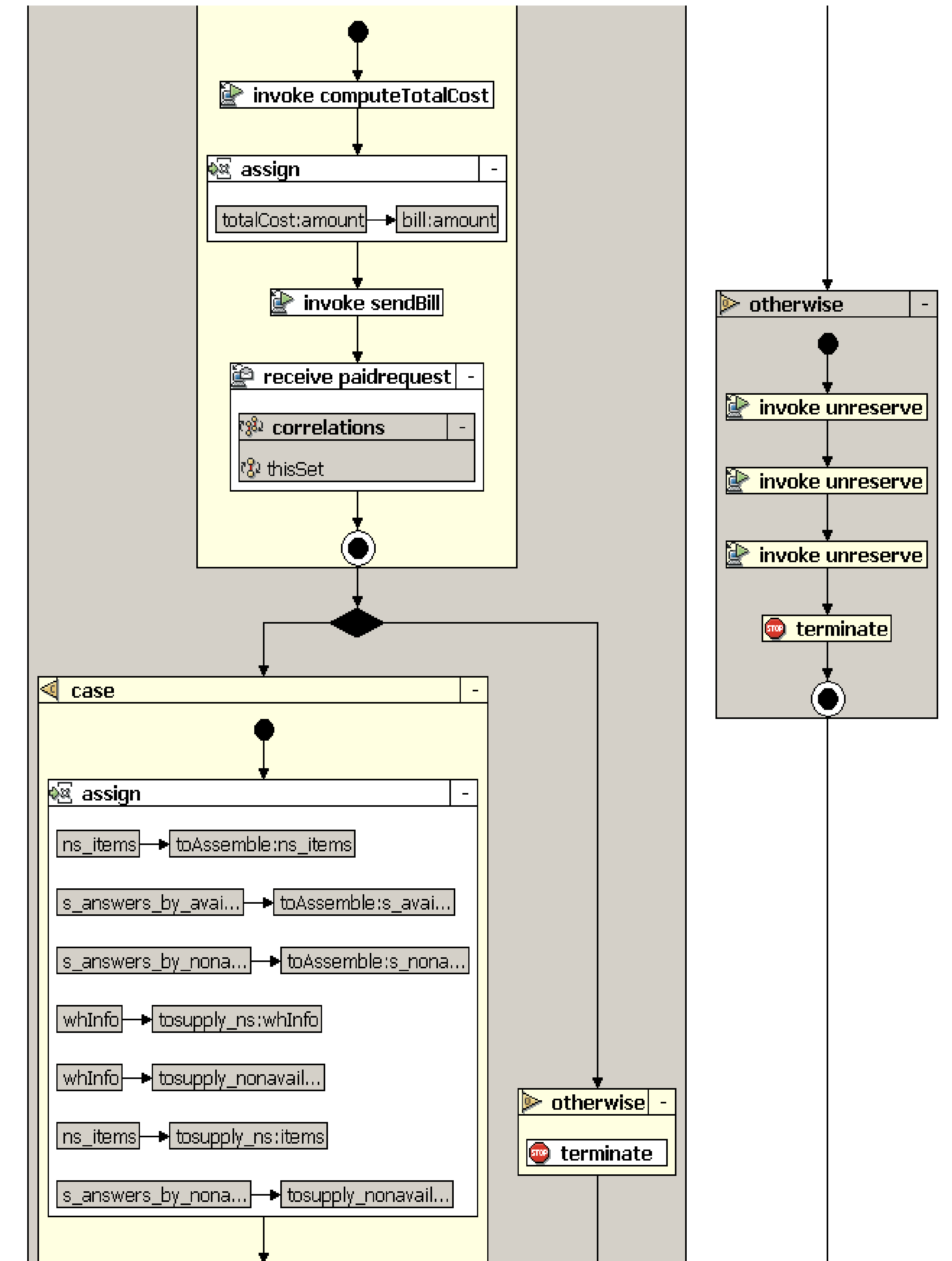


Figure 37: BPEL process model /3

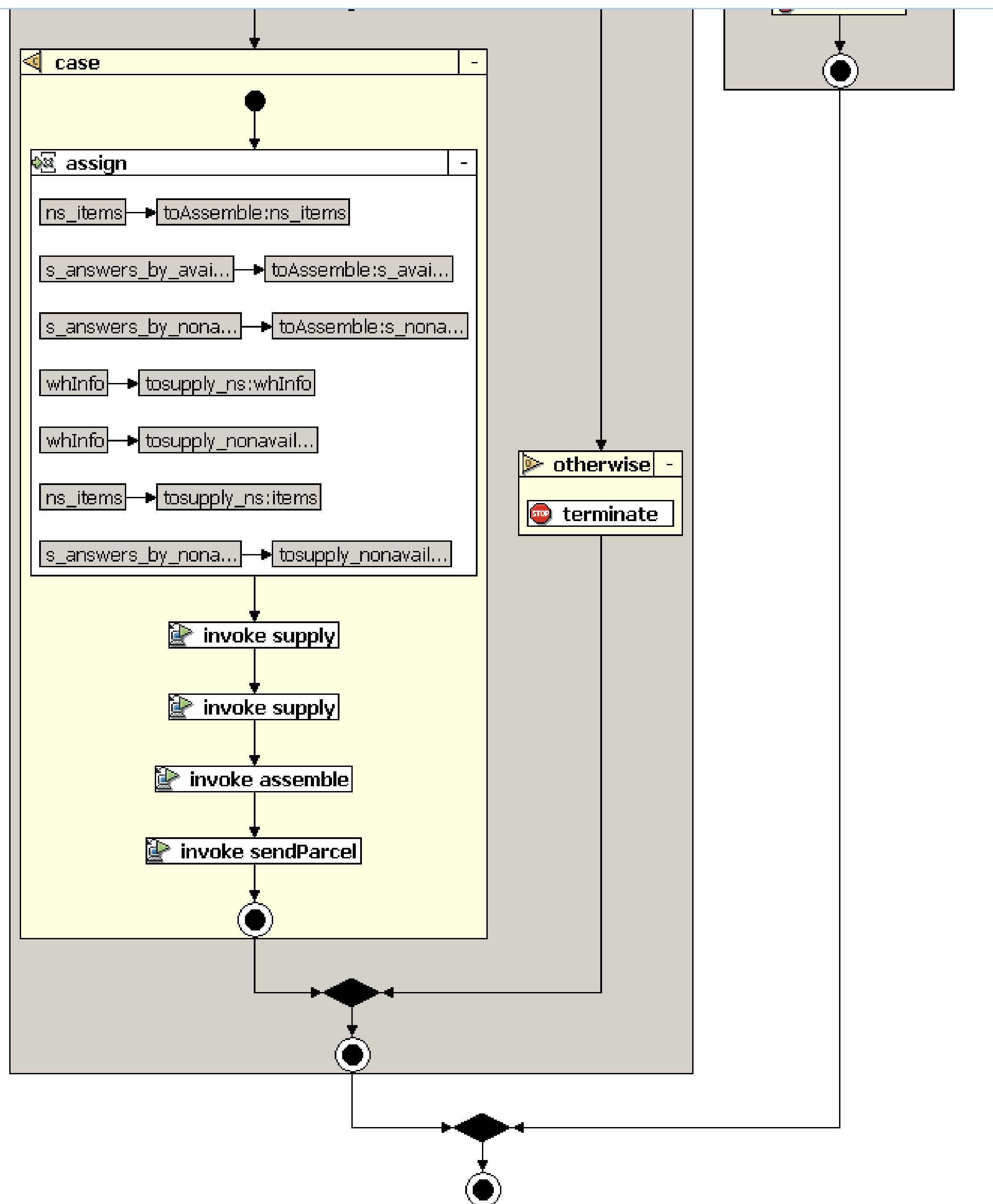


Figure 38: BPEL process model /4

and the temporal variables, that are needed only to organize the process, such as

<variable messageType="customer:status" name="tosupply_nonavail_status"/>

<variable messageType="customer:status" name="tosupply_ns_status"/>

The process contains two correlation sets. Correlation sets are needed to identify the instances of services when the process is started (instantiated). First of them contains the data, related to users order, this correlation is needed exactly on the start after "*receive*" of the *OrderMsg* from customer.

<correlationSets>

```
    <correlationSet name="thisSet" properties="customer:orderID"/>
    <correlationSet name="supplCS" properties="supplier:suppID"/>
  </correlationSets>
```

The second one is used to identify the supplier, that participates in transaction with warehouse of shop. Process description defines, that many suppliers may participate in one session of processing customers order. As were said above, this collaboration is defined using the WS-Addressing statements.

The overall process contains all the invocations of services operations, receiving data in synchronic mode, assigning values to different variables and checking their values on the "*switch*" blocks.

Activities "*invoke computeTotalCost*","*assign totalCost -> bill*","*invoke sendbill*" and "*receive paidrequest*" are grouped to one *scope*. On this scope defined faultHandler "nopayment", which invokes when no payment from customer is received. In this case, items, that were reserved at warehouse and at suppliers are to be un-reserved. This is defined in faultHandler action, by specifying sequence of actions for un-reserving items and terminating the whole process.

3.1.4 Diagnosis process

3.1.4.1 Case Studies

In this section we highlight some failure situations within the process. In the following section we will describe a sample diagnostic process for each of these situations.

We will study three situations that are started by an exception:

A. When computing the bill, the SHOP realizes that the ship cost sent by the WAREHOUSE is higher than the expected threshold (**HighShipCostException** of the SHOP).
B. When receiving the bill, the CUSTomer realizes that some ordered item is wrong (**WrongBillException** of the CUSTomer).
C. When assembling the package, the WAREHOUSE realizes that it received a wrong item from one of the SUPPLIERs (**WrongSupplyException** of the WAREHOUSE).

3.1.4.2 Diagnosis

From the point of view of diagnosis, exceptions are **symptoms** of something going wrong. There can be several possible causes for an exception; diagnosis must discard those that cannot have happened (due to further observations), possibly reducing the possibilities to the one that took place.

A. There can be two causes for a **HighShipCostException** in the shop: either the SHOP selected the wrong warehouse (thus choosing one that is far from the customer address), or the warehouse itself made a mistake in computing the ship cost. Diagnostic reasoning can find these two possible causes with backward reasoning, but without adding any observable data or test action it is not possible to discriminate between the two.
B. A **WrongBillException** is caused by someone reserving the wrong item, either the WAREHOUSE or one of the SUPPLIERs. By following backwards the path of the wrong item data, it is possible to discover who reserved that particular item and correctly diagnose the problem.
C. Let us look at the possible causes for a **WrongSupplyException**. Apparently there are three possibilities: (i) the SUPPLIER reserved the wrong item from the beginning; (ii) the SUPPLIER reserved the correct item but then made a mistake in updating its internal order DB, writing the wrong item code; (iii) the SUPPLIER did everything correct but sent the

wrong parcel to the WAREHOUSE. However, possibility (i) can be discarded by observing that it would have produced an error in the bill, while no **WrongBillException** was raised.
Thus only possibilities (ii) and (iii) remain as candidates. The SUPPLIER could discover the source of the error by comparing the reservation codes it sent to the SHOP with those it wrote down in its DB: if they are the same then (iii) holds; otherwise (ii) holds. It is worth noting that this further check must be described somewhere in the model, if we want it to be available during the diagnostic process.

3.1.5 Repair stage

3.1.5.1 Possible cases

A. If the WAREHOUSE computed a wrong ship cost, then the only possible repair is to correct the problem at the source and then compute it again. If on the other hand the SHOP selected the wrong warehouse, changing warehouse at this point could be too time-consuming. Thus the best solution is that (i) the SHOP corrects the problem for future conversations; (ii) for the current conversation, the SHOP keeps the wrong warehouse but lowers the ship cost for the customer (the shop itself will pay the difference).
B. In the case of wrong item reservation the only possibility is to repeat the reservation for the wrong item. A new bill (or negative answer, if the correct item wasn't available) is then computed and sent to customer.
C. If there was just a parcel mismatch (case (iii)) then it suffices to ask the SUPPLIER to send the correct parcel, possibly sending back the wrong one. Shipment costs to and from must be covered by the SUPPLIER who made the mistake. If on the other hand the SUPPLIER wrote the wrong data in its DB, it will have to correct the problem that caused this to happen, and besides sending the correct parcel it will also have to update its stock DB, that registered a wrong transaction.

3.1.5.2 Detailed analysis

In our approach faults may occur at three levels: Infrastructure and Middleware (due to failures in the underlying hardware, network, and system infrastructure); Web service level (due to failures in service invocation and orchestration); Web application level (malfunctions in the execution of Web applications due to data mismatches or coordination or choreography failures).
Repair actions are designed according to these three levels and originate different recovery strategies, according to the system components affected by the fault. For example, at the Web application level, the main goal is to provide: (i) services which respect the user requirements in terms of both functionalities and QoS, (ii) business continuity, and (iii) fault masking.
At the Web service level the main goal is to manage the service choreography correctly and to guarantee service continuity and QoS requirements by substituting corrupted services with compatible ones available in the network. Faults are considered as events and repair actions are triggered according to the Event Condition Action (ECA) paradigm. For example, a connection time out event at the middleware layer could be due by a fault or due to an overload of the provisioning server. Then in order to identify the fault the conditions which are evaluated are: (1) the network is available, (2) the provisioning server is available, (3) the server CPU usage rate is high. According to which of the above conditions are verified, different repair actions will be undertaken such as service substitution (in the first two cases) or resource re-allocation (in the last case).

In Table 3 we list possible draft categories of recovery actions. Once a fault has been diagnosed, our approach assumes that Web services deployed within our architecture are able to perform recovery actions, restore the correct state, and remove the causes that led to the failure. For each diagnosed fault, one or more recovery actions are executed. Two types of recovery actions are

identified: reactive and proactive recovery actions. Reactive actions are performed along with the execution of Web services and try/allow the recovery of running services. Proactive actions are mostly based on data mining techniques and can mainly be executed in an off-line mode. Proactive actions are complex and require the support of an environment able to execute services, to detect runtime faults, and to perform recovery actions with no damage to the running instances of the monitored Web services. A long term approach to self-healing is adopted, where recovery actions have the goal of improving Web Services and Web applications in order to avoid future failures. These actions can be oriented to provide a one-shot improvement action or to modify the service provisioning process for a permanent data and process improvement

In Table 3 the level, fault types and some examples thereof are reported. A Web Service execution fault is raised during invocation if a service is not responding, or due to a wrong security access authorization of the end user, or a parameter is missing in the input SOAP message. Notice that this kind of faults may occur even if the infrastructure properly works, so it can not be included into the infrastructure faults class. A mismatch in the structure of messages to invoke a service may be due, for instance, to an update in the published service interfaces which are not yet considered inside the invoking applications. In our framework, a Web service execution fault may occur when, upon substitution of a faulty Web service (e.g., an unreachable one) with a functionally equivalent one, no substitute is available. A service coordination fault is typically due to a violation of the order of invocation of service operations or messages (e.g., a book payment is received before the corresponding book reservation). Web Service coordination faults may occur when some of the Web Services in a composed service are unavailable. Another case occurs when a message is received that does not match the choreography protocol. Application level faults are related to the execution of a Web application based on Web Services. Most of the faults at this level can be captured by mechanisms at the infrastructure and middleware level (basically a timeout).

Examples of Application Coordination faults are a session fault, a phase time fault (e.g., the ordered item has been paid, but the confirmation of payment is received after an internal time out), a resource booking fault (e.g., not the whole resource pool necessary to complete the service has been reserved), or inter-process faults (e.g., service data have not been received at the correct time). Examples of Actor faults are the case the customer is not connected when a synchronous communication is needed, or an authorization fault. QoS violation faults are related to local and global constraints specified by the user or by the application designer.

For example, the user can require that the delivery time of the ordered item be lower than a given threshold or that the total price of the ordered item is lower than a given amount. Another category of faults is bound to the process design, that is, to the way the application workflow has been developed. For example, an Unavailable goods fault should be treated by exception handlers specifically included in the process workflow. In some cases, the handlers are already specified in the process able to manage the fault. In other cases, such handlers have not been designed, and hence the application could experience a deadlock or a total crash. Being the system self-healing, we expect the fault log to gather information useful to design the necessary handler. We assume that a fault event can be raised, and that no repair actions are explicitly executed, beyond a *notify* action. Generic application dependent faults are therefore not going to be considered. Internal Data faults include data quality faults related to data manipulated during the execution of a specific service. Examples are the wrong ID or name of an ordered good, or mismatched customer data. Since these faults specifically regard data internal to a service, they are evidenced as possibly bounded to specific filters and options to be then treated be recovery actions apart.
It is important to evaluate the quality of information flow along a specific service since failures can be caused by incorrect or missing information. The most important data quality dimensions are accuracy, completeness, consistency, and timeliness. These dimensions are objective dimensions and, therefore, are suitable for a quantitative evaluation and constitute a minimal set that provides sufficient information to evaluate the data quality level [R96].

In Table 3 we list possible draft categories of recovery actions. Once a fault has been diagnosed, our approach assumes that Web services deployed within our architecture are able to perform recovery actions, restore the correct state, and remove the causes that led to the failure. For each diagnosed fault, one or more recovery actions are executed. Two types of recovery actions are identified: reactive and proactive recovery actions. Reactive actions are performed along with the execution of Web services and try/allow the recovery of running services. Proactive actions are mostly based on data mining techniques and can mainly be executed in an off-line mode. Proactive actions are complex and require the support of an environment able to execute services, to detect runtime faults, and to perform recovery actions with no damage to the running instances of the monitored Web services. A long term approach to self-healing is adopted, where recovery actions have the goal of improving Web Services and Web applications in order to avoid future failures. These actions can be oriented to provide a one-shot improvement action or to modify the service provisioning process for a permanent data and process improvement.

Table 3: Levels of faults occurrence, type of fault, and examples

Recovery action type	Actions	Fault type	Type
Service-oriented recovery action	Retry	Infrastructural, WS execution	Reactive
	Substitute	WS execution, WS coordination	Reactive
	Completion of missing message parts	WS execution	Reactive
	Reallocate	QoS	Reactive/proactive
	Change process structure	All	Proactive
	Process-oriented methods	Application level	Proactive
Data quality recovery actions	Insert data quality block	Internal data	Reactive
	Process-oriented methods	Application level	Proactive

For instance, a variety of techniques for data improvement are proposed in the literature. The most straightforward solution suggests the adoption of one-shot data-oriented inspection and re-work techniques, such as data bashing or data cleaning [EN99]. These techniques focus on data values and can solve problems related to data accuracy and data consistency quality dimensions. A limitation of these techniques is that they do not prevent future errors. They are considered appropriate only when data are not modified frequently. On the other hand, a more frequent use of data bashing and data cleaning algorithms involves high costs that can be difficult to justify. To overcome these issues, several experts recommend for permanent improvement the use of process–oriented methods ([EN99], [R96], [SPP05], [SWZ00], [WAN98]). These methods allow the identification of the causes of data errors and their permanent elimination through a change in data access and update activities. These methods are appropriate when data are frequently created and modified. Organizations can also adopt mixed strategies in which they can decide to adopt a data-oriented technique or a process-oriented technique depending on data and process types.
Recovery actions can be also classified in *service-oriented* and *data quality recovery actions*. While the former deals with invocation, orchestration and choreography aspects of Web services, the latter pertains mainly to the management of data quality faults. For each of the fault types defined in the previous sections, several candidate recovery actions may be proposed, depending on the fault type and whether a reactive or proactive approach is taken. Such recovery actions are implemented by the modules that will be introduced in Section 5 (namely, a Reallocation module, a Substitution module, a Wrapper generator module, and a Quality module). In this way, whenever

the diagnosis step identifies a fault, the recovery action selector invokes the related repairing modules.

In our reference example, let us consider some examples of faults and repair actions. Let us assume a customer selects a food ware from the on-line site, and that his request includes the phases of food checking for availability (service activated by the customer), food selection from a warehouse (service activated by the shop service), book shipping (service executed by the shipper service), and payment (service by an external payment service). Let us suppose that the Shipper, Warehouse, and Supplier belong to a trust circle, that is, that no security faults can occur in the messages exchanged among these three services. Faults that may arise in the trust circle are a resource booking fault, due to mismatch of resource reservation to execute the application. An internal data fault may occur when the Shop sends order data to the Warehouse (e.g., a wrong ID). Another fault, of type Unavailable goods may occur during the execution of the Warehouse service, needing to store a log that asks to postpone the goods search process until a new event (Good in Stock) arises to signal that the Warehouse has been refilled. If we view the whole application as a workflow composed of three phases: Selection-and-Booking, Payment, and Delivery, a fault of type *phase time out* occurs if one phase exceeds the foreseen time schedule; a *session fault* occurs if a connection is lost among the phases, and the collected data are lost. Finally, consider that food reservation, payment, and shipping are regarded as services that have been orchestrated and attached to the customer context through e.g., the customer's mobile device or browser. If the shipping service arises a fault, e.g., a missed delivery due to a delayed delivery time, we regard this fault as a QoS violation in terms of delivery service time fault. As a consequence, the *repair action* can be a money refund service; this means a modification of the part of the order that was affected by the fault. At the Application level, this fault is notified by the client side controller invoked directly by the customer browser or device, which may undertake the following repair actions: 1) check the sequence of services which are affected by the fault; 2) reschedule or substitute the involved services (here, the payment service); 3) log the application level fault into the fault log; 4) notify the customer with the new schedule. For rescheduling, the system has to send a new payment form, and hence modify the payment service data. If, the payment requires for example to update the customer profile, the new profile is needed with the notification sent to the customer, with new preferences, options, or constraints for further orders. Such new profile may trigger updates to the customer security profile, in order to update the security logs contents.

Regarding data quality faults, a low accuracy value can characterize the shop catalogues, if there are some typos in it. This could imply the mismatch between the user request and the shop information and thus a book could be never retrieved. Another example of low accuracy can be described looking at the first phase where the users have to insert their own address for the book delivery. It could happen that the users gives wrong data and the parcel could never be shipped. In particular, a parcel delivery might fail if there is some internal inconsistency in the address written in the request, e.g., the zip code does not correspond to the right city. Inconsistency problems could be correct using data bashing techniques. A food search might fail also it is not completely described in the catalogues and thus requirements might be unfulfilled. This case regards poor data quality due to low value of the completeness dimension.

An important aspect that has to be considered in this example is the presence of many actors, each with its own database. Since actors are involved in the same business, databases could overlap and thus be affected by data misalignments. There could be database misalignments between bookshop and publisher and consequently the shop has out of date catalogues. In some cases this fault might imply the mismatch of users' requirements. In fact, it could happen that a producer does not communicate to the shop price variations. In this case, bookshop, along the user requirements and the available information contained in its own databases, might select that shop but the new prices do not satisfy the request. Users would receive a bill higher than the requested one. The error is due to the low values of timeliness associated with data owned by the shop. Misalignments between shop and other actors' databases can also cause completeness problem. The actors need

therefore to analyze their communication processes and adopt efficient synchronization mechanisms by choosing the most suitable time interval to perform the periodic realignment among databases.
In general, in the Repair Model we make the hypotheses that all needed information is available from the diagnosis model, namely: 1) which function was wrong; 2) which type of exception message was arisen. We also consider that a knowledge base of *fault-repair patterns* is available: containing patterns of two types: Domain Independent (Examples are: “delivery”, “missing db item”) and Domain Dependent (Examples of domains are: “University”,”Municipality”). Such patterns are then instantiated on the running cases, e.g., “delivery of perishable” for foodshop or “missing student information” for University domain.

The repair strategies should provide the system and the user to put in place have several alternatives. For example, considering the *Delivery* service, a rule, expressed as a simple triple format, can be:

<time_fault; delay_of_service (ok) → second_item_free>

whose meaning is that if a time fault occurred, bringing over a delay in the *Delivery* service but, in spite of the fault, the service was successfully completed (delay_of_service(ok)), then the customer will receive an item for free. For the WorkFlow this implies that a *cost* variable will be constrained for future executions of the WokFlow (*var cost=0*).

As another example, consider the variation of the previous fault, namely:

<time_fault; delay_of_service(not_ok) → (money_back) & (restart_WF)>

Here, the service could not be successfully completed (delay_of_service(not_ok)). The variables of the WorkFlow are not affected, but rather an activity has to added to the re-execution of the WorkFlow.

Sample faults at the *application level* are given below:

WRONG PRICE

e.g., the customer orders an item for 10$ and gets 20€ at payment time

The Diagnosis step will determine which component is wrong. Detection can be done using *data guards* at some points in the WorkFlow. Alternatively, it is necessary to put additional observations like:

1) Tracing back to determine the wrong component

2) Optimization of positioning data guards to *anticipate moment of symptom detection*

The Diagnosis *on* **WRONG PRICE** determines the possible wrong components and actions, such as:

Wrong computation by SHOP
Wrong ship cost by WAREHOUSE
Wrong data in catalogue by SHOP/WAREHOUSE (low data quality)
Wrong formulation of problem by CUSTOMER
Wrong communication (dialog) e.g. 20$ or 20€

Considering as a final example the fault: **INCORRECT ASSEMBLE OF PARCEL**, we have the following schema:

Diagnosis: before assemble of parcel action

Possible wrong components:

Wrong synchronization by SUPPLIER (e.g., goods are reserved but not available)
Wrong parts by SUPPLIER
Wrong reservation by SUPPLIER
Missing parts by SUPPLIER
Wrong parcel composition by WAREHOUSE

Repair action:

on line : supply new goods
off line : when delivery of both perishable and not perishable, make 2 parcels

3.1.6 Comparison

Cooperative review test case = CR;

Travel services test case = TS

Food shop test case = FS

The TS and the FS are quite similar with respect to a number of features, while the CR seems to focus on slightly different aspects. In this section we try to underline some points that could help in comparing the different test cases.

TS and FS

The FS and the TS model a scenario in which a customer accesses an on-line service in order to buy a complex product (food in the first case, flights and hotel accommodations in the second one). In both cases the customer expresses a set of requirements and the service tries to “build” a “product” that fulfils such requirements. The “product” is composed by different items, with different characteristics (perishable and unperishable items vs. flights and hotels), and for each item the system must check for availability and then reserve/buy it.

In both cases some sequences of activities must be treated as transactions, since they require to be “undone” as a whole if there is a failure. This aspect is very similar in both scenarios, although, currently, it is not explicitly described in the FS (but it can be easily added).

A difference is that in the TS there are no activities requiring human intervention, while in the FS at least the assembly of the parcel and its shipping must be performed by humans. A workflow that includes (possible) human activities seems to be more general, since: (a) it is closer to real cases (in many real business scenarios some operations are still performed by humans: think for example of a bank that handles everything electronically; it still has to physically send a credit card to its account holders and, for security reasons, the credit card must be activated on-line only upon reception of both the credit card and its PIN code sent in separate envelopes); (b) from a modelling point of view, human activities can be wrapped within a Web Service interface, such behaving in a homogeneous way with respect to the other Web Services involved.

The possible exceptions in TS and in FS are similar. For example, an alarm can be raised by the customer, in TS, to signal that the flight plan does not meet her requirements because there is an additional or a missing step in the itinerary; similarly, in the FS, the customer can raise an exception when, while checking the bill, she realizes that an item is missing, or it is wrong.

Also the examples of causes for exceptions seem to be quite similar in the TS and FS: mismatches between reservation ID numbers, database misalignments, and so on.

As a consequence of the similarity of the overall structure of the cooperating workflows and of the possible exceptions, diagnoses and recovery actions are very close in the two scenarios.

CR and FS

The CR focuses on QoS aspects, which are not explicitly considered in the FS (and in the TS). In this scenario the user provides a set of QoS requirements and the system tries to fulfil them. Moreover, QoS parameters are monitored during the execution of the workflow and detected mismatches raise exceptions.

Within the CR, exceptions related to QoS violations can be due to a violation of a QoS contract (possibly previously negotiated between provider and consumer) or due to a misinterpretation of some data (e.g. european vs american format for dates). The QoS contract can include generic parameters (such as availability, security, response time, …), which can be modelled in any scenario, and application-specific parameters, which are different in different domains, but that can probably be classified in quite general categories (see Chapter 3.2.3).

The CR does not include activities performed by humans and does not consider exceptions other than QoS violations. Moreover, it is not clear if and how it could be extended in order to manage transactions.

The FS, currently, does not include QoS parameter, but they can be added, since they definitely make sense in a real e-commerce scenario. It seems plausible, in fact, to imagine that an online shop aims at offering a service that fulfils a QoS agreement (probably based on a contract negotiated with the customer); moreover, the online shop can negotiate QoS levels with warehouses and with suppliers. The introduction of QoS monitoring would then enable the system to check QoS violations and to raise the corresponding exceptions.

An interesting aspect faced within the CR is the discovery phase, which make sense in an open environment. The FS was initially conceived as running in a closed scenario, where all the supplier were known a-priori, but it can be “rephrased” to take into account the discovery phase, if we imagine that the warehouses, before contacting a supplier, can “look for it” querying a public registry.

3.2 Test Case: cooperative review

3.2.1 Workflow

This example illustrates the functioning of a common collaborative activity addressing the problem of “review process”. This problem has several instantiations in different activity areas. In industrial activities such as engineering of complex and embedded systems, the “design review” activity is known to be as one of the most complex activities in aerospace industry (see the IST-DSE project). The “cooperative review” application scenario we study in the context of the DIAMOND project aims to support such activities. Most of the defined services, parameters, actors and processes are generic and may be applied to different domain-specific review processes. In order to facilitate the understanding and the collaboration within our project, we choose to instantiate the review process by the well-known scenario of the review process in scientific publishing activities. Namely, we consider the specific case of the scientific conferences looking for describing the automatic functioning of their different steps within a service-oriented approach.

This description takes into account the various steps concerning successively:

1. Search by the authors of conferences answering specific criteria (e.g., scientific topics, reputation of the conference, submission and publications deadlines, publishing house, etc…).
2. Search by the conference chair of reviewers answering specific selection criteria (e.g., research subject, expertise, institution, and so forth).
3. The submission process allowing the authors to deposit their papers.

The reviewing process, which involves the assignment of papers to the reviewers, the recovery of review reports, and the final decision concerning whether to accept or refuse submissions.

Table 4 : QoS parameters associated with conferences and reviewers.

Parameters	Classification	Conference	Reviewer
Topics	topic1, topic2, topic3	x	
Research subject	subject1,subject2,subject3		x
Paper's length	short, long, unlimited	x	
Submission deadline	soon, on-time, late	x	
Reputation	weak, average, strong	x	x
Acceptation degree	low, average, high	x	
Publisher's quality	low, average, high	x	
Acceptation notification date	soon, on-time, late	x	
Conference's date	soon, on-time, late	x	
Conference's place	site1, site2, site3, …	x	
Production level	low, average, high		x

Table 4 shows the different QoS parameters identified as being necessary to consider during the various steps of the "search process" required by the whole of the system's activities.

3.2.2 Architecture and Workflow

The actors

Several actors collaborate in order to accomplish the various management tasks of the cooperative reviewing process. The different types of actors are presented as follows:

The Conference Chair represents the principal actor in charge of organization of the conference. His activities proceed throughout the lifecycle of the system, from conference planning until the publication of proceedings. In practice, it is possible that this responsibility is shared between several people.

The Track Chair has in charge to manage a particular conference's session, whenever the conference is composed of several topics.

The Author represents each potential contributor to the different topics of the conference.

The Reviewer is an expert in one or more of the domains defined by the different topics of the conference, and he is skilled to produce an objective report on papers assigned to him.

General Architecture

As depicted in Figure 39 the description of the "Cooperative Reviewing System (CRS)" is made following a service-oriented approach. Namely, it is composed by Web services, orchestration elements, Web service registries (allowing discovering of Web services), WS clients, and QoS management elements.

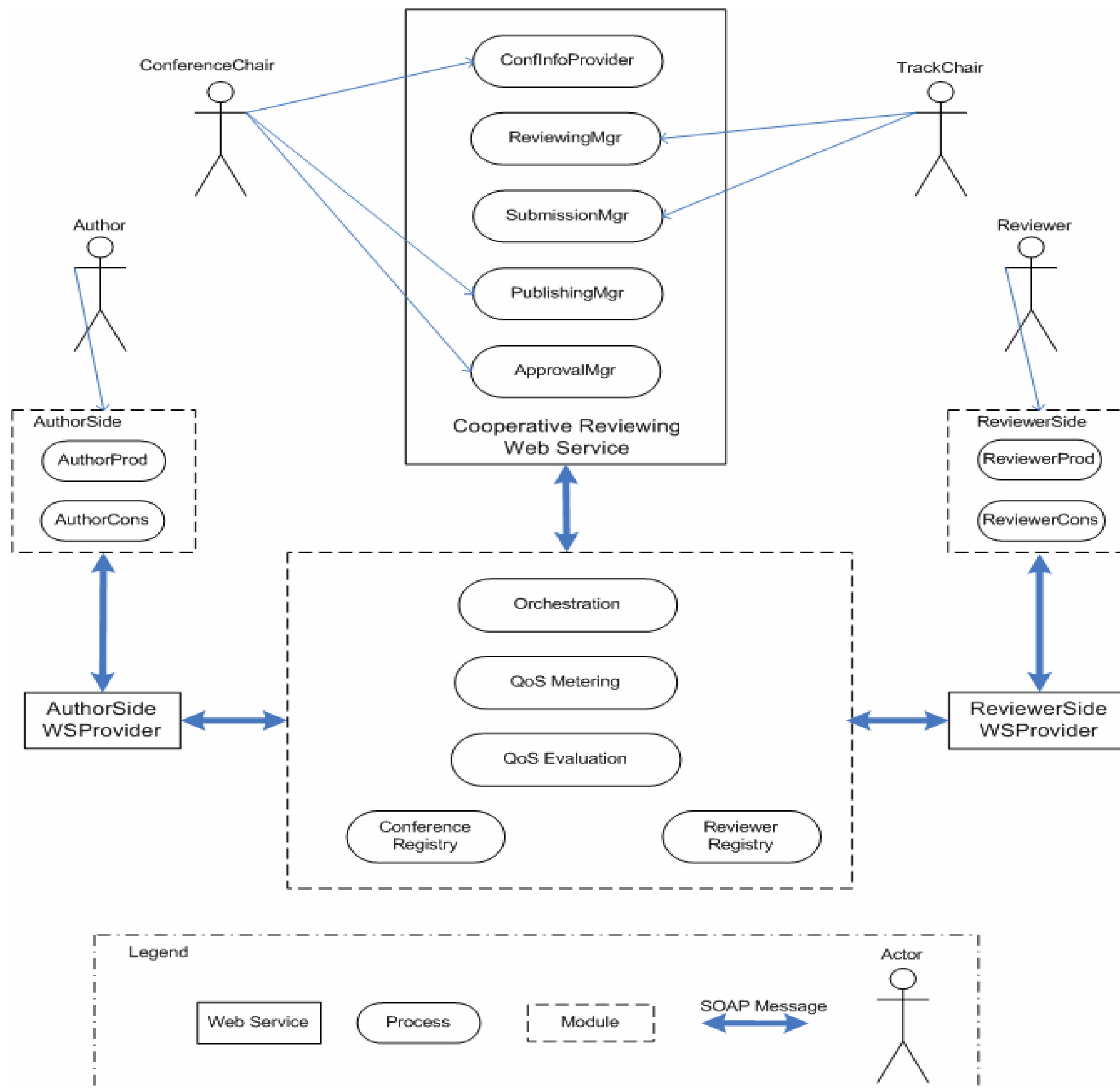


Figure 39: General architecture of the cooperative reviewing system.

The various components of the architecture and their functions are introduced into the following sections.

Activities Description

In this section we introduce the activities regarding the cooperative reviewing process. This description considers the components of each activity, their category (i.e. service, producer, consumer, registry, repository), the communication relationships between these components, as well as their multiplicity (i.e. 1:1, 1:*, *:1, *:*).

Activity 1. Conference Search

This activity describes the process where the authors seek “Calls for papers” being close to their research subject.

According to Figure 40, interested authors looking for conferences use an interface provided with producer/consumer processes in order to contact the *AuthorSideWSProvider* service. This service will start, in turn, an orchestration process which allows firstly, the *CooperativeReviewing* Web Service discovery; and secondly QoS management. In return, the author will obtain a list of conferences meeting his requirements and their related web services.

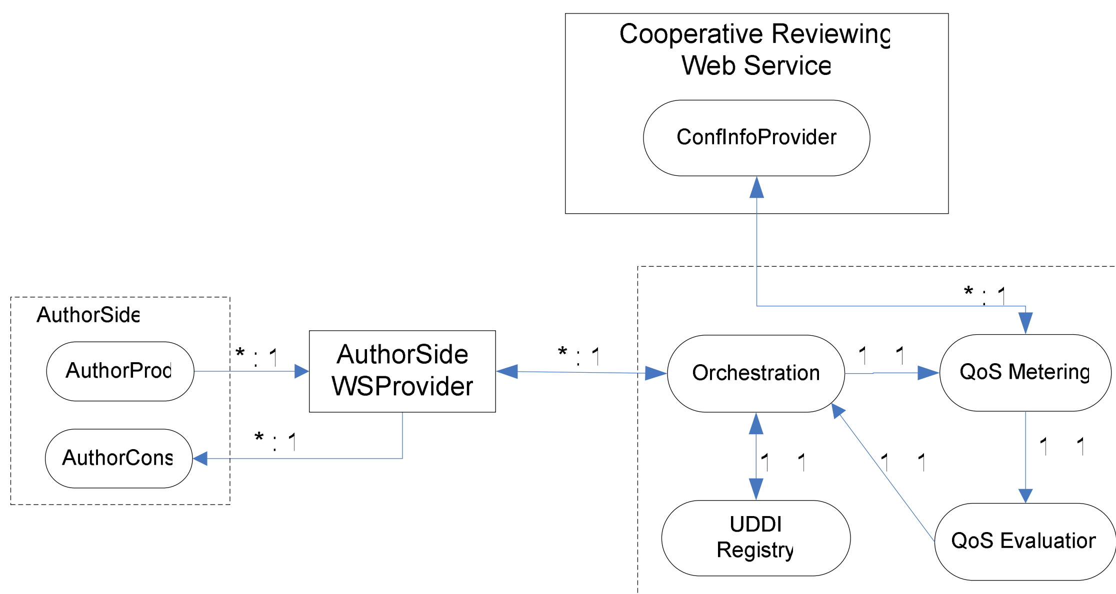


Figure 40: Looking for conferences.

The role of each component participating in this activity is detailed as follows:

- For Author side:

  AuthorProd:

  It represents an “information producing entity” that handles the user’s request and forwards it to his associated *AuthorSideWSProvider* service to be processed.

  AuthorSideWSProvider:

  This is a Web service that has in charge to starts the search of “*CooperativeReviewing* Web services” that map as much as possible QoS requirements specified by the author’s request.

  AuthorCons:

  It represents an “information consuming entity” that the *AuthorSideWSProvider* service has to contact, in order to forward the complete information related to all the conferences fulfilling his requirements. The list of these conferences results from the conversation involving the *AuthorSideWSProvider* and the *CooperativeReviewing* Web services.

- For Conference Chair side:

  ConferenceInfoProvider:

It is a process inside the *CooperativeReviewing* Web services deployed on the sites of the conferences; it is asked with respect to any kind of information relating to a conference.

UUDI registry:

This process manages registries allowing, in this case, discovery of *CooperativeReviewing* Web services.

Orchestration:

This process controls the global conversation among Web services involved on this activity.

QoS Metering:

This process allows measuring of QoS parameters among Web services conversations.

QoS Evaluation:

It must validate the obtained results with regard to QoS parameters, and sort the list of conferences before returning it to the implied author.

Figure 41 outlines the sequence diagram related to this activity. An author (across the *AuthorProd* process) sends a request message for conference search (*confSearch*) with QoS parameters satisfying his requirements. The *AuthorSideWSProvider* service receives this request and starts an orchestration process. The orchestration process queries the registry in order to discover *CooperativeReviewing* Web services. While each *CooperativeReviewing* service is solicited for conference information, after its answer (across the *ConfInfoProvider* process) QoS parameters are measured and evaluated. Finally, the conference list (*confList*) accomplishing with the QoS contract is delivered to the concerned author.

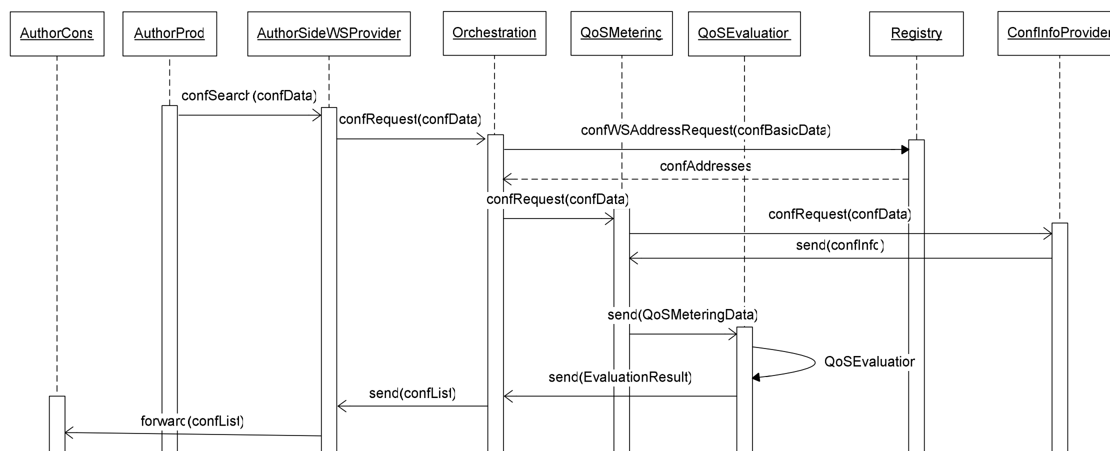


Figure 41: Sequence diagram for conference search activity.

Activity 2. Reviewers Search

This activity addresses the process of reviewers search and selection. Potential reviewers are selected by taking in account their qualification and expertise domain and contacted in order to manifest their interest to participate in the evaluation of submitted papers.

As presented in Figure 42, the *CooperativeReviewing* service, across its *ReviewingMgr* process, starts an orchestration process in order to find reviewers interested by the conference. This request contains the conference information and QoS requirements. The *ReviewerSideWSProvider* service

is contacted in order to invite its related reviewers. Each reviewer expressing its interest is stored into the right repository.

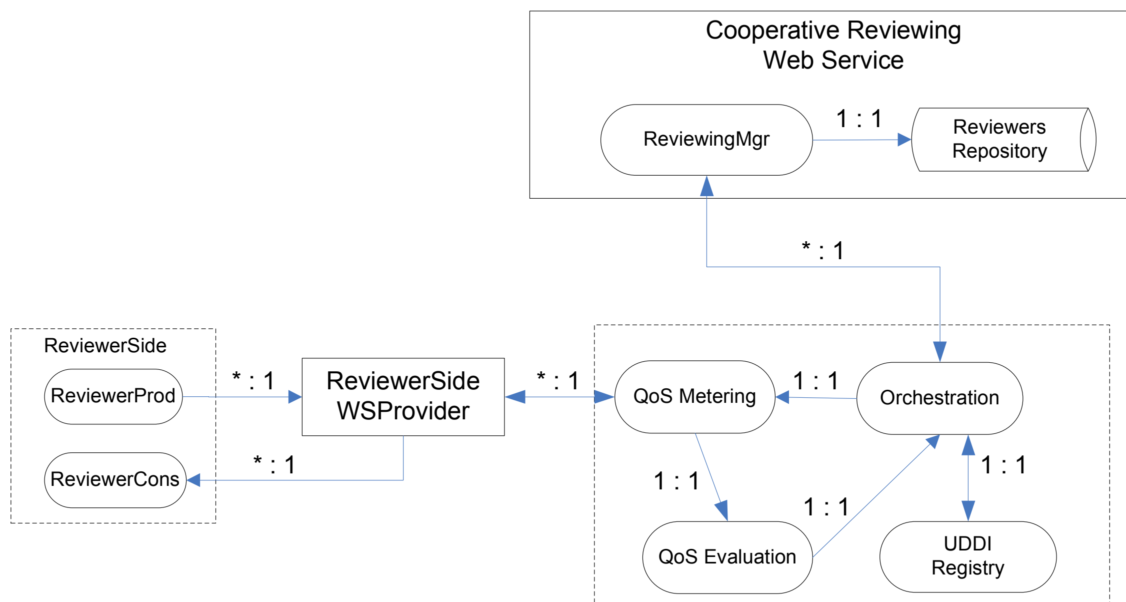


Figure 42: Looking for reviewers.

The role of each participant in this activity is described in what follows:

- For Track Chair side:

  ReviewingMgr:

  It has in charge starting the reviewers search activity according to the conference requirements.

  Reviewers Repository:

  It stores the data of reviewers accepting the invitation for participation.

- For Reviewer side:

  ReviewerSideWSProvider:

  It is contacted by the orchestration process so that it contacts and gets the response of its related reviewers.

  ReviewerProd:

  It represents the interface available to the reviewer to send its decision to the *ReviewerSideWSProvider* service.

  ReviewerCons:

  It constitutes the mechanism of notification available to the reviewer to interact with the *ReviewerSideWSProvider* service.

Orchestration:

This process makes it possible to control conversations among Web services involved on this activity. It contacts the *ReviewerSideWSProvider* services (across the QoS metering process, which catches the involved request) in order to collect the response of reviewers after invitation to participate in the review activity. It returns to the *ReviewingMgr* process the list and information related to reviewers accepting the invitation of the conference.

UDDI Registry:

This process manages registries allowing, in this case, discovery of *ReviewingSideWSProvider* Web services.

QoSMetering:

This process allows measuring of QoS parameters among Web services conversations.

QoSEvaluation:

It must validate the obtained results with regard to QoS parameters, and sort the list of reviewers before returning it to the implied ReviewingMgr process.

The sequence diagram related to this activity is depicted by Figure 43. The *ReviewingMgr* service sends a request (*searchReviewers*) with the QoS parameters characterizing the conference. This request launches the *Orchestration* process. The orchestration process queries the registry in order to discover *ReviewerSideWSProvider* services. Each *ReviewerSideWSProvider* service asks the reviewers that it manages about participation approval (*requestParticipation*) in the conference. The interested reviewers send a message of confirmation (*sendDecision(OK,reviewerid)*) to the *ReviewerSideWSProvider* service, which produces in turn a message (*sendConfirmation*) to the *QoSMetering* process. QoS parameters are measured and evaluated. Finally, the reviewer list (*reviewersList*) accomplishing with the QoS contract is delivered to the concerned *ReviewingMgr* process.

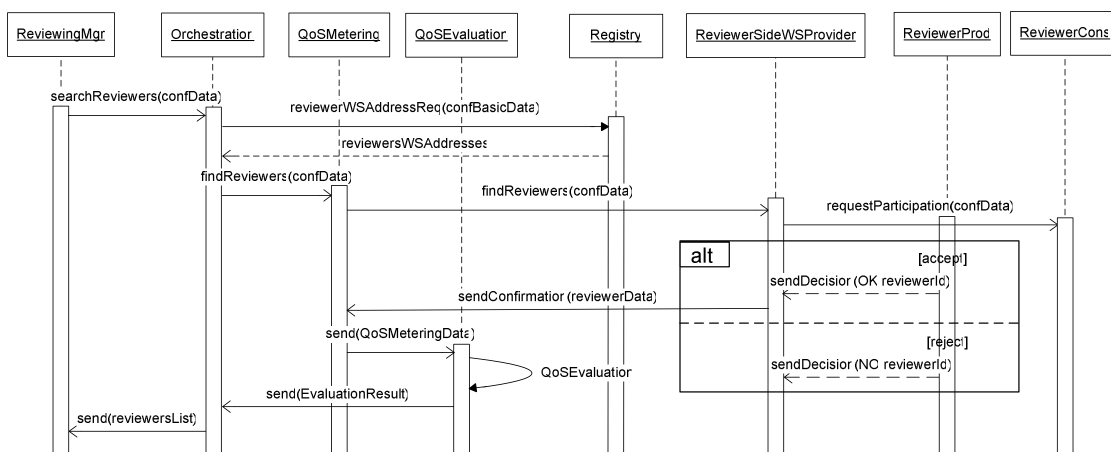


Figure 43: Sequence diagram for reviewers search activity.

Activity 3. Author Inscription

This activity considers the inscription of authors in order to submit papers to the conferences. Notice that figure describing this activity is intentionally omitted. Indeed, this one is much similar to that describing paper submission activity (activity 4).

The sequence diagram of Figure 44 is describing the interactions among the components intervening in this activity. An author asks for inscription in a conference (across the *AuthorProd* process). The *AuthorSideWSProvider* service transmits this request towards the *Orchestration* process. The *QoSMetering* process catches this request and interacts with the *SubmissionMgr* process in order to authorize the inscription. QoS parameters are measured and evaluated in order to return an “acknowledge message” to the concerned author (across the *AuthorCons* process).

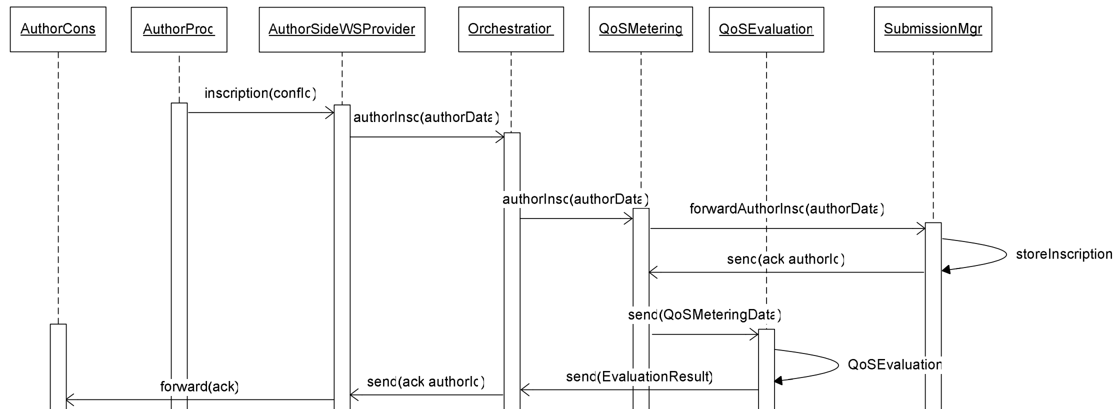


Figure 44: Sequence diagram for author inscription activity.

Activity 4. Paper Submission

This activity considers the paper submission activity by authors interested in conferences.

According to Figure 45, an author wishing to submit a paper must contact the *AuthorSideWSProvider* service. This service launches an orchestration process in order to establish a conversation with the *CooperativeReviewing* Web service. The *SubmissionMgr* service receives and authorizes this request and stores the author and paper data.

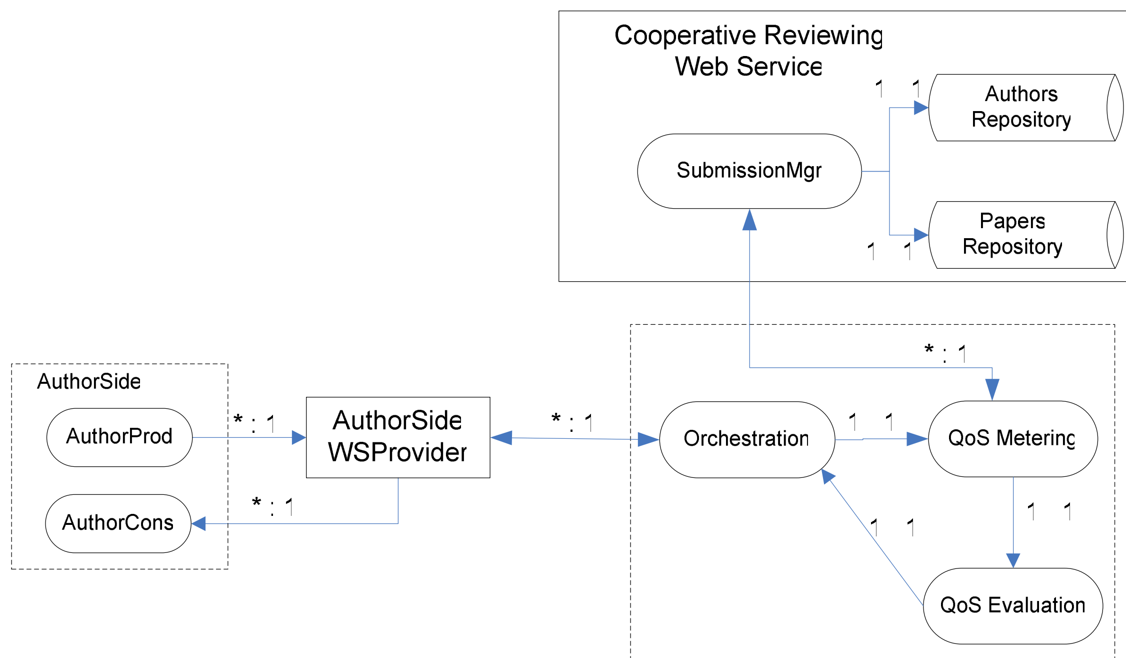


Figure 45: Paper submission

The roles of the participants related to this activity are described as follows:

- For Author side:

  AuthorProd:

  It represents the interface the author uses to contact the *AuthorSideWSProvider* service.

  AuthorCons:

  It constitutes the mechanism of notification available to the author to interact with the *AuthorSideWSProvider* service (i.e., submission acknowledge).

  AuthorSideWSProvider:

  It is the service allowing the author to start an orchestration process in order to submit papers.

- For Track Chair side:

  SubmissionMgr:

  This process receives submitted papers from authors and stores them.

  PapersRepository:

  It makes it possible to store the papers submitted by authors.

  AuthorsRepository:

  It makes it possible to store data of authors submitting papers.

Orchestration:

This process makes it possible to control conversations among Web services involved on this activity.

QoSMetering:

This process allows measuring of QoS parameters among Web services conversations.

QoSEvaluation:

It must validate the obtained results with regard to QoS parameters.

The sequence diagram involving this activity is showed in Figure 46. An Author submits a paper (across the *AuthorProd* process) for reviewing. The *AuthorSideWSProvider* service transmits the paper towards the *Orchestration* process. The *QoSMetering* process catches this submission and interacts with the *SubmissionMgr* process in order to deliver the paper. QoS parameters are measured and evaluated in order to return an "acknowledge message" to the concerned Author (across the *AuthorCons* process).

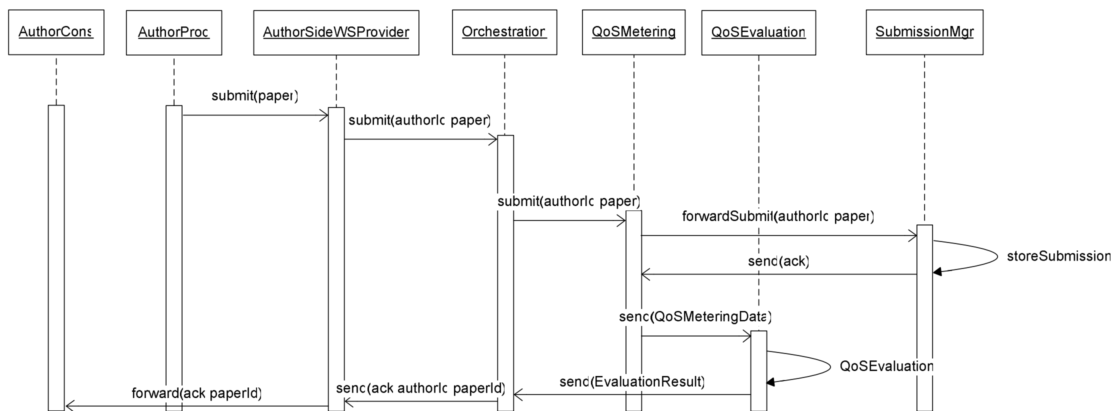


Figure 46: Sequence diagram for paper submission activity.

Activity 5. Paper Assignment

This activity considers the paper assignment by the Track Chairs to the reviewers.

According to Figure 47, each Track Chair (across the *ReviewingMgr* process) decides to assign a paper to some reviewers in order to get an objective report about its quality. To be so, the *CooperativeReviewing* Web service starts an orchestration process in order to deliver the paper to the qualified reviewers.

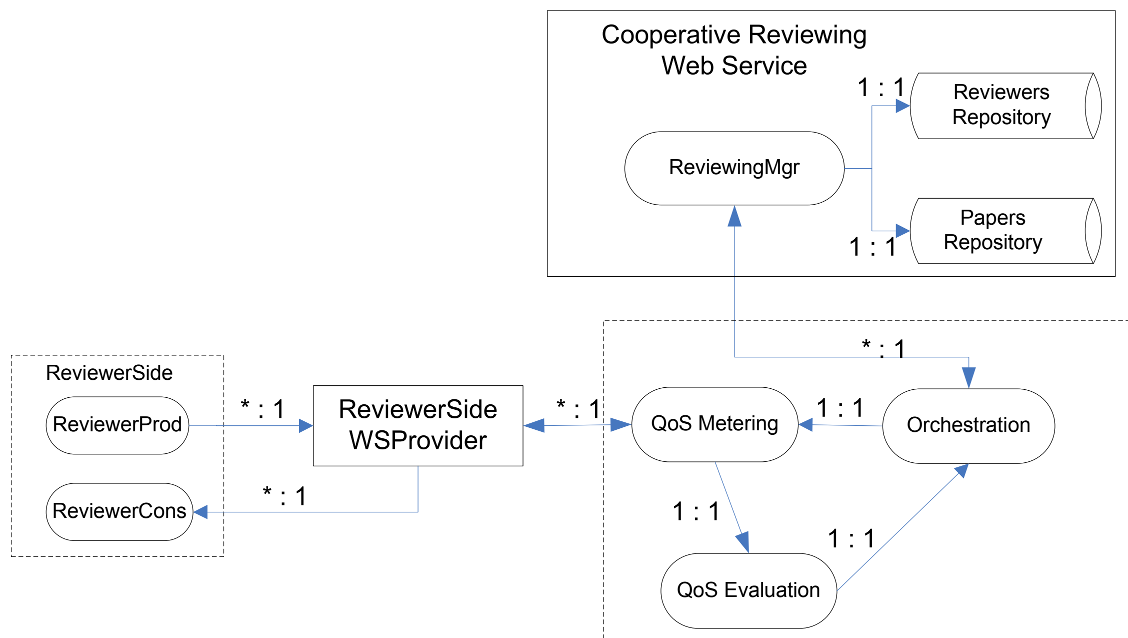


Figure 47: Paper assignment

The roles of the participants related to this activity are described as follows:

- For Track Chair side:

  ReviewingMgr:

  It carries out the assignment of the papers to the available reviewers.

  PapersRepository:

  It makes it possible to store the papers submitted by pre-inscribed authors.

  ReviewersRepository:

  It makes it possible to store the data of reviewers participating in the conference.

- For Reviewer side:

  ReviewerSideWSProvider:

  It establishes a conversation with the *CooperativeReviewing* Web service, across the orchestration and QoS-related processes, in order to send assigned paper to the concerned reviewers.

  ReviewerProd:

  It represents the interface available to the reviewer in order to communicate possible events related to this activity.

  ReviewerCons:

  It corresponds to notification mechanism allowing the reviewer to access its assigned papers.

Orchestration:

This process makes it possible to control conversations among Web services involved on this activity.

QoSMetering:

This process allows measuring of QoS parameters among Web services conversations.

QoSEvaluation:

It must validate the obtained results with regard to QoS parameters.

The sequence diagram involving this activity is depicted in Figure 48. A Track Chair (across the *ReviewingMgr* process) assigns a paper for reviewing and sends it to the selected reviewers. It is made in an orchestrated way (across the Orchestration process). The *QoSMetering* process catches this assignment and interacts with the *ReviewerSideWSProvider* service in order to deliver the paper to the assigned reviewer (across the *ReviewerCons* process). QoS parameters are measured and evaluated in order to return an "acknowledge message" to the *ReviewerMgr* process.

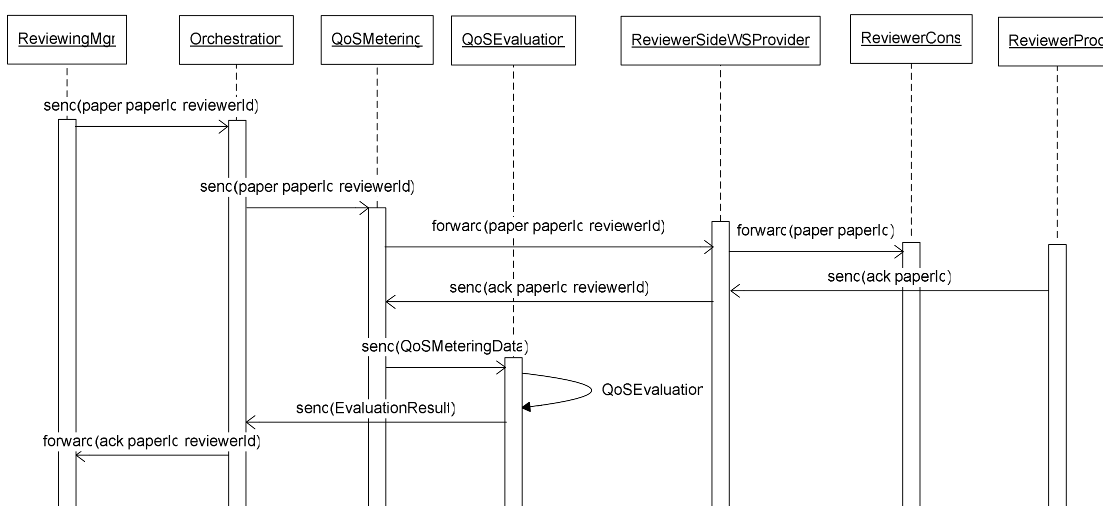


Figure 48: Sequence diagram for paper assignment activity.

Activity 6. Report Transmission

This activity considers the transmission of the papers review reports by the reviewers to the system.

According to Figure 49, in order to send the review reports, reviewers contact the *CooperativeReviewing* service through the *ReviewerSideWSProvider* service and the orchestration process involving this activity.

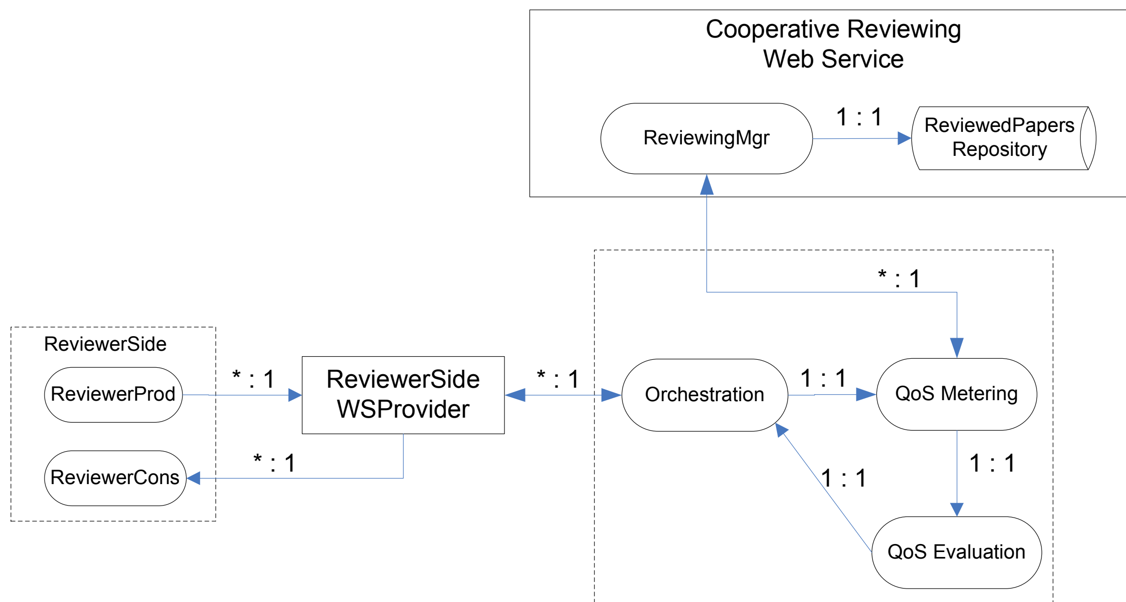


Figure 49: Getting reviewers' reports

The roles of the participants related to this activity are described as follows:

- For Reviewer side:

  ReviewerProd:

  It represents the interface of the reviewer to address review reports.

  ReviewerCons:

  It corresponds to notification mechanism used by the reviewer

  ReviewerSideWSProvider:

  It represents the orchestration access point of the reviewer allowing to forward review reports to the system.

- For Track Chair side:

  ReviewingMgr:

  It is contacted by the orchestration process (across the QoS metering process, which catches the involved message) in order to forward review reports sent by reviewers.

  ReviewedPapersRepository:

  The repository used to store reviewers' reports.

Orchestration:

This process makes it possible to control conversations among Web services involved on this activity.

QoSMetering:

This process allows measuring of QoS parameters among Web services conversations.

QoSEvaluation:

It must validate the obtained results with regard to QoS parameters.

The sequence diagram involving this activity is showed in Figure 50. A reviewer sends a reviewing report for each paper (across the *ReviewerProd* process). The *ReviewerSideWSProvider* service transmits this report towards the *Orchestration* process. The *QoSMetering* process catches this submission and interacts with the *ReviewingMgr* process in order to store this report. QoS parameters are measured and evaluated in order to return an "acknowledge message" to the concerned Reviewer (across the *ReviewerCons* process).

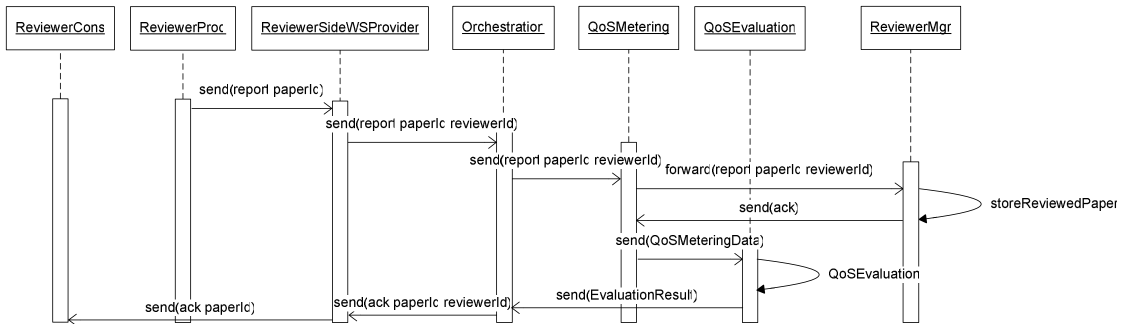


Figure 50: Sequence diagram for report transmission activity.

Activity 7. Author Notification.

This activity (Figure 51) addresses paper notification (acceptance of paper or its refusal). The decision is made by the *Conference chair* based on reviewing reports received from reviewers. This decision is transmitted to the author via the *AuthorSideWSProvider* service.

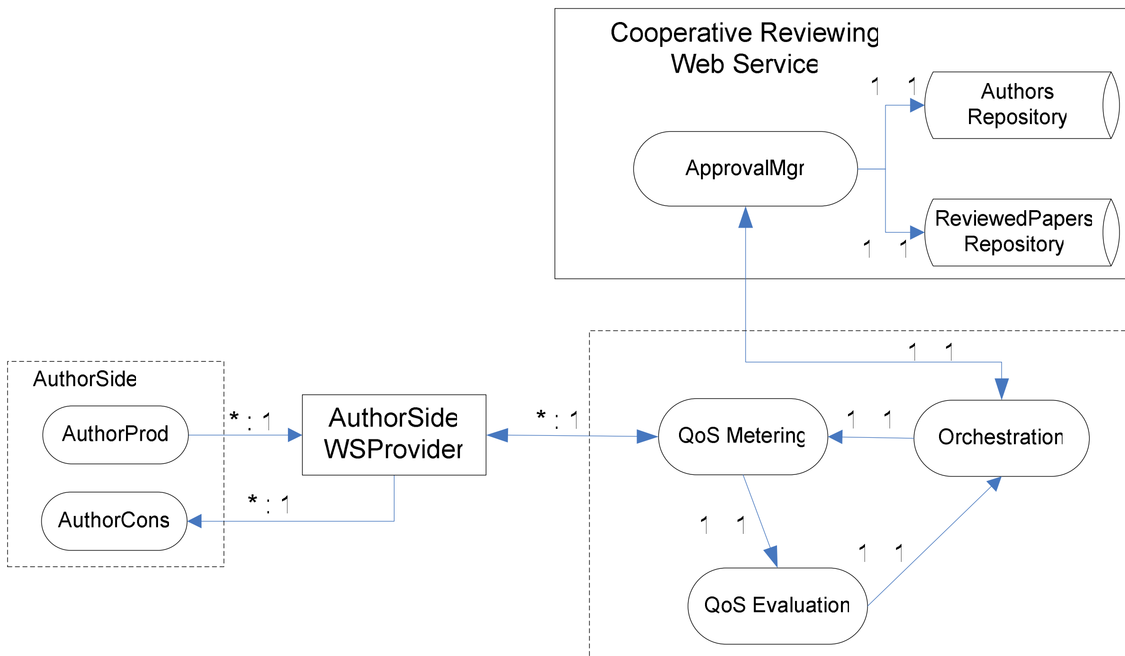


Figure 51: Getting approval decision

The components of this activity are described in what follows:

- For Author side:

  AuthorProd:

  It represents the interface available to the reviewer to communicate possible events related to this activity.

  AuthorCons:

  It corresponds to notification mechanism used by the author.

  AuthorSideWSProvider:

  Here this service constitutes the orchestrated access point of the authors making it possible to correctly transmit the notifications to them.

- For Conference Chair side:

  ApprovalMgr:

  It is a process of the *CooperativeReviewing* service that is in charge of transmitting the acceptance or refusal of papers.

  ReviewedPapersRepository:

  Repository where are stored the reports sent by the reviewers. The final decision of the *conference Chair* will be based on the combination of all the reviews related to a given paper.

  AuthorsRepository:

  Repository where are stored authors' data.

Orchestration:

This process makes it possible to control conversations among Web services involved on this activity.

QoSMetering:

This process allows measuring of QoS parameters among Web services conversations.

QoSEvaluation:

It must validate the obtained results with regard to QoS parameters.

The sequence diagram involving this activity is depicted in Figure 52. For each paper, the Conference Chair (across the *ApprovalMgr* process) sends decision of acceptance or refusal (and reviewing reports validating it) towards the *Orchestration* process, in order to notify the concerned author. The *QoSMetering* process catches this message and interacts with the *AuthorSideWSProvider* service in order to deliver this decision (to the author across the *AuthorCons* process). QoS parameters are measured and evaluated in order to return an "acknowledge message" to the *ApprovalMgr* process.

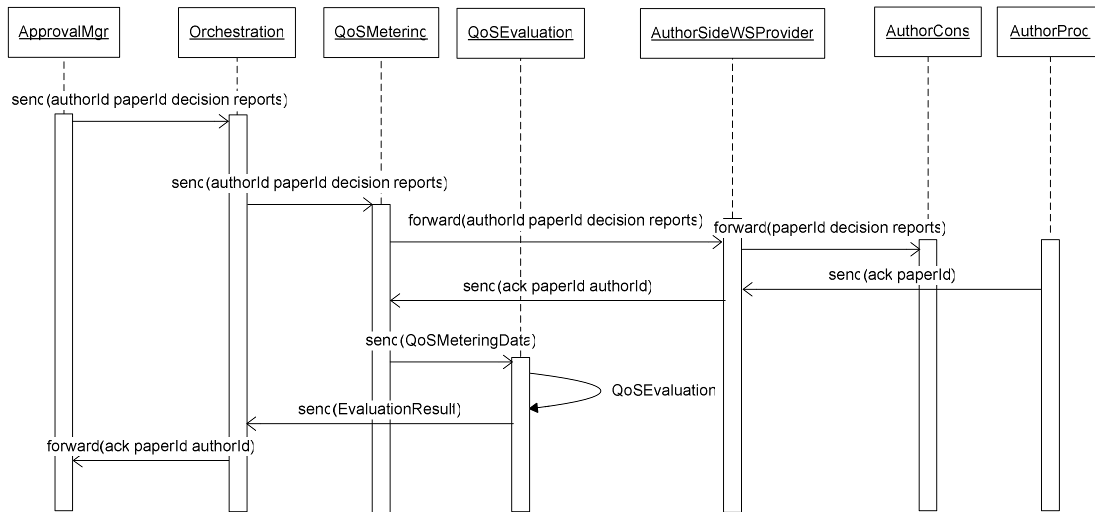


Figure 52: Sequence diagram for author notification activity.

Activity 8. Final paper submission

This activity (Figure 53) addresses final paper submission. In fact, this one is quite similar to paper submission activity, with the *PublishingMgr* process as element of distinction.

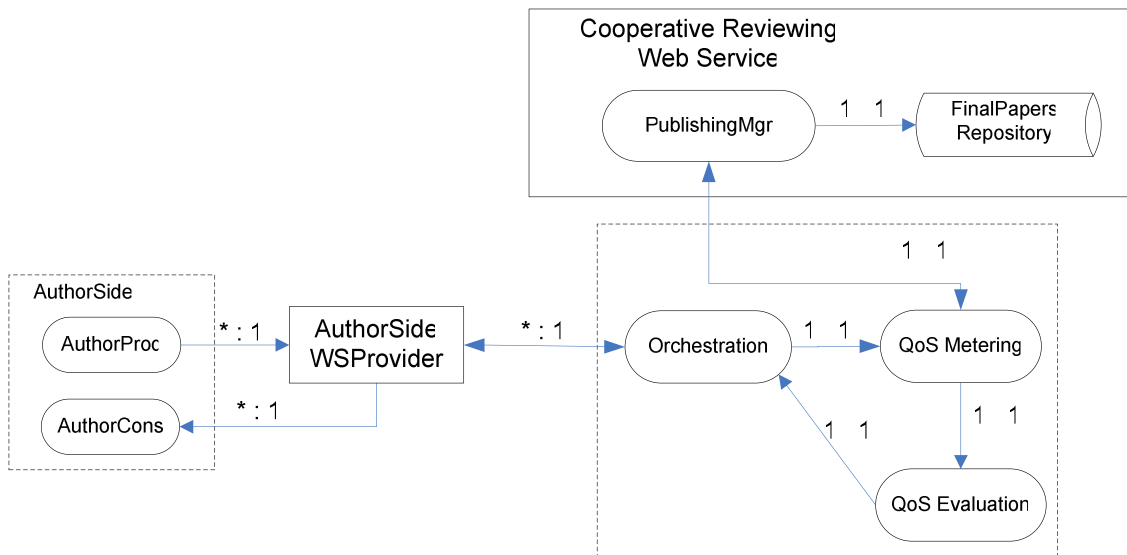


Figure 53: Getting final papers

- For Author side:

  AuthorProd:

  It represents the interface of the author to send final papers.

  AuthorCons:

  It corresponds to notification mechanism used by the author.

  AuthorSideWSProvider:

Here this service constitutes the orchestrated access point of the authors making it possible to correctly transmit the final papers.

- For Conference Chair side:

  PublishingMgr:

  It is a process of the *CooperativeReviewing* service that is in charge of receiving and storing the final papers.

  FinalPapersRepository:

  The repository used to store author's final papers.

Orchestration:

This process makes it possible to control conversations among Web services involved on this activity.

QoSMetering:

This process allows measuring of QoS parameters among Web services conversations.

QoSEvaluation:

It must validate the obtained results with regard to QoS parameters.

The sequence diagram involving this activity is depicted in Figure 54. An Author submits a final paper (across the *AuthorProd* process) for publishing in conference proceedings. The *AuthorSideWSProvider* service transmits this final paper towards the *Orchestration* process. The *QoSMetering* process catches this submission and interacts with the *PublishingMgr* process in order to deliver the final paper. QoS parameters are measured and evaluated in order to return an "acknowledge message" to the concerned author (across the *AuthorCons* process).

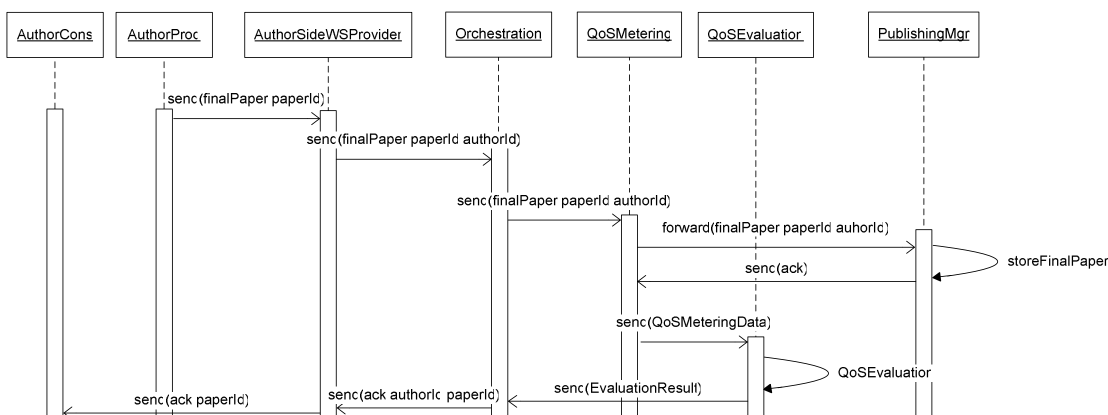


Figure 54: Sequence diagram for final paper submission activity.

Figure 55 presents the activity diagram of the conference management and cooperative review system. The first two parallel tasks are related to the search tasks (conference and reviewers). The conference and authors search tasks both lead to the inscription of the authors in a conference.

Then, the authors can submit their papers which will be attributed only after recruiting the reviewers (synchronization point). We have after, the different tasks of evaluation and author notification. If this notification implies the acceptance of paper, the authors must achieve the task of final paper submission.

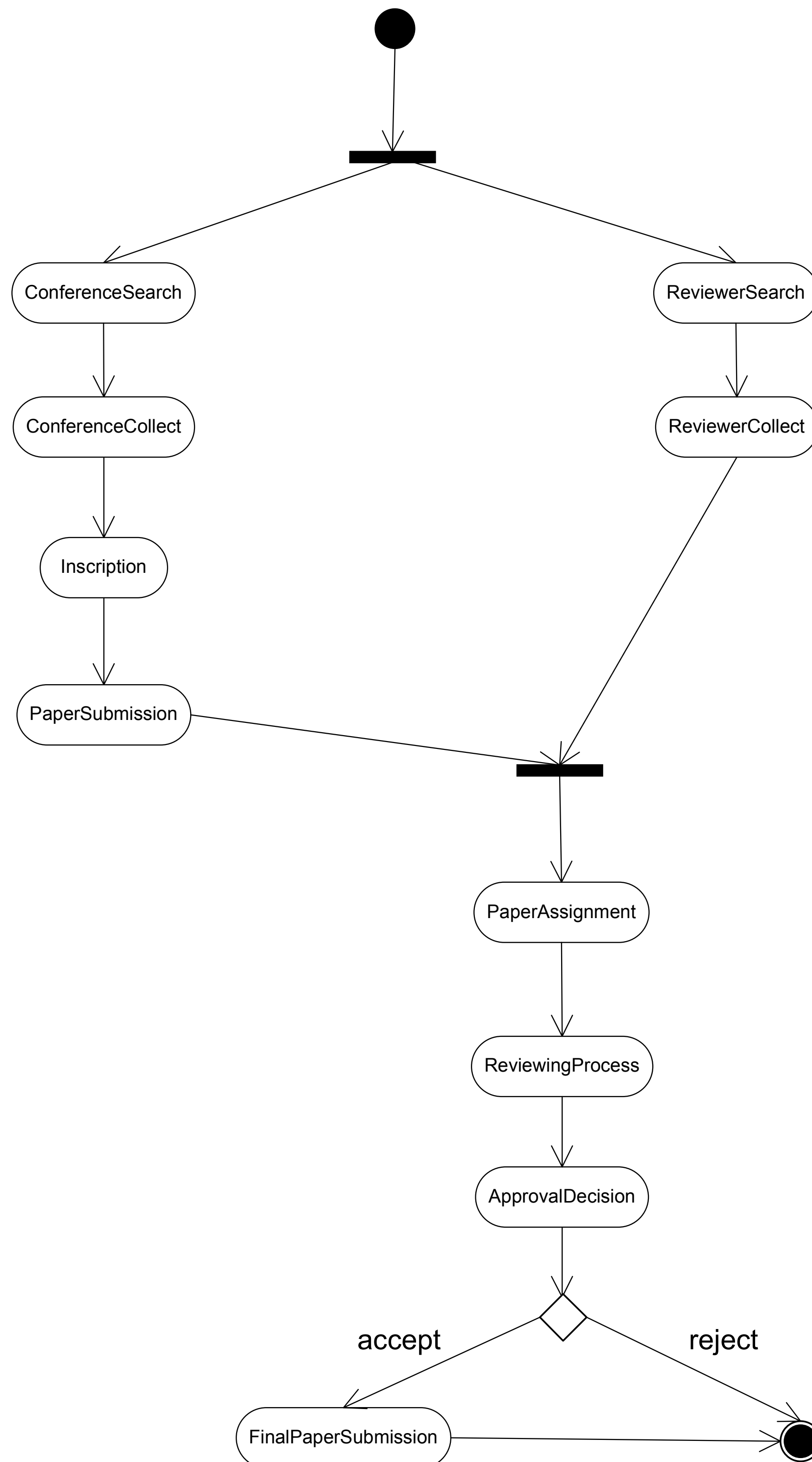


Figure 55: General activity diagram

3.2.3 Exceptions

Figure 56 presents a classification of exception categories. This classification considers three major exception classes: 1) QoS mismatches related to a non respect of the QoS contract between the QoS provider and the QoS requester, 2) semantical mismatches related to a signification misunderstanding of methods or parameters between the requester and the provider (i.e. date 1/2/2006 understood by the provider as 1st February and understood by the requester as 2nd January). The remaining non studied exceptions on mismatches are classified as 3) functional mismatches related to a faulty execution of a required service due for example to a wrong implementation but that still respect the QoS contract.

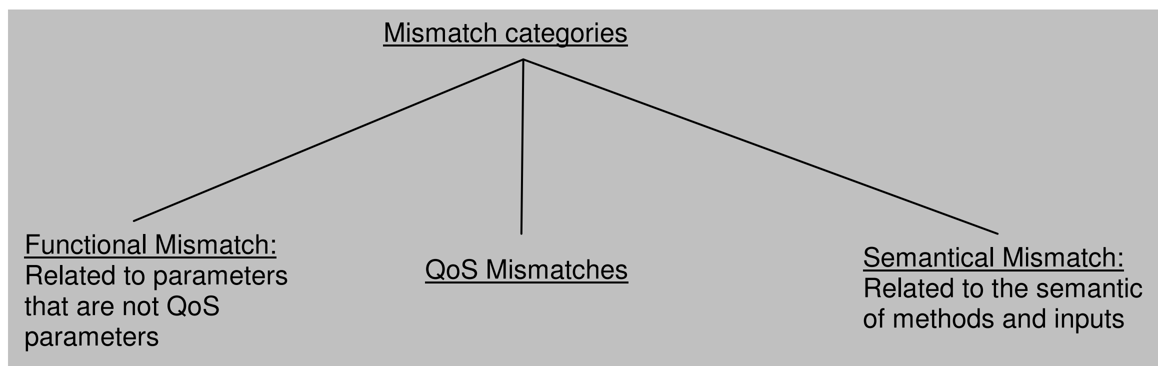


Figure 56: Classification of Considered Mismatches

The QoS contract (that can be obtained for example after a negotiation step) corresponds to the level of the QoS that a service provider accepts to deliver and that its requester accepts to receive. QoS parameters considered for this QAC are classified (see Figure 57) into the following categories: 1) generic QoS parameters such as availability, security, response time and throughput parameters, and 2) Application-specific QoS parameters that can be decomposed to 2.1) argument related mismatches corresponding to the correctness of (or the combination of) parameter values and domains that are exchanged between the requester and the provider, and 2.2) conversational-related parameters corresponding to the correct protocol supposed to mange the interaction between the provider and the requester. Conversational mismatches considers 2.2.1) time-related QoS violation of the applicative level and 2.2.2) mismatches related to the correct order of operation execution.

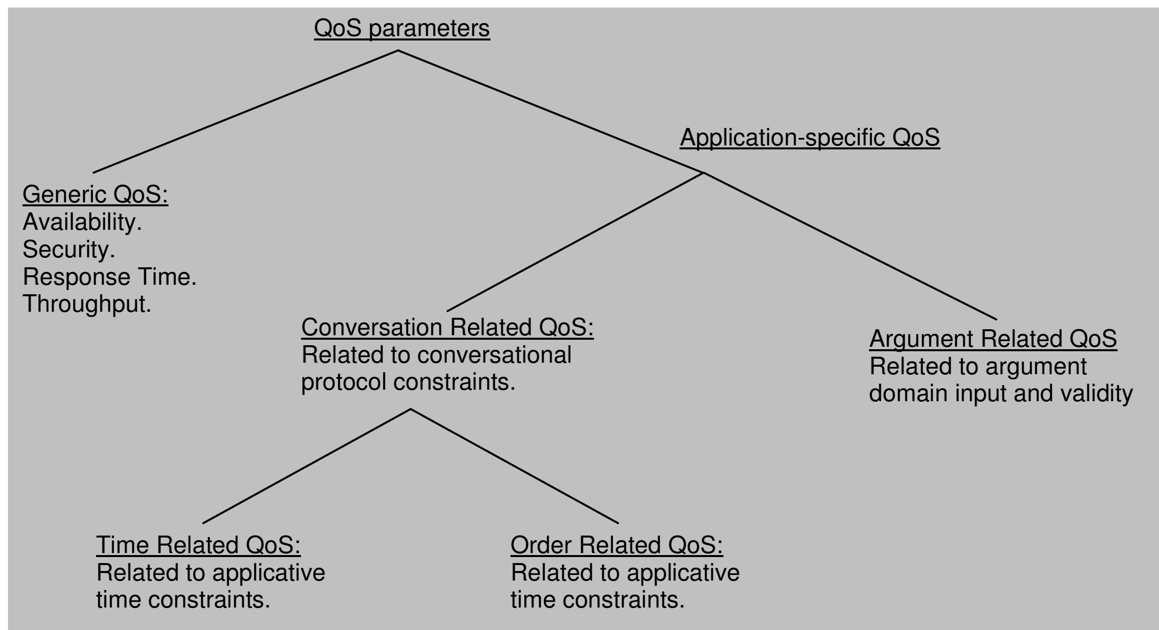


Figure 57 QoS parameter classification

3.2.3.1 Cooperative review QoS-related mismatches

In this section we present the mismatches related to the cooperative review application. We assume for the detection part, that this task is made by the Metering and Evaluation components of the orchestrator side. Thus, we suppose that all QoS metering, QoS evaluation and orchestration components can't be faulty and that the observable faults are those that can be detected by the QoS evaluation component. In the sequel, we present all mismatches but only the observable ones (written in bolt characters) will be considered for diagnosis and repair actions.

a. SEARCH STEP

a.1 Event search

a.1.1 Reception of an event whose deadline is not acceptable: *Argument-related mismatch* or *sematical mismatch*

a.1.1.1. Probable Origins :

a.1.1.1.1. date semantic misunderstanding, ConfInfoProvider failure, or communication delay between the ConfInfoProvider and the orchestrator.

a.1.1.2. Detection :

a.1.1.2.1. by the user.

a.1.1.2.2. by the AuthorSide service

a.1.1.2.3. by the evaluation component

a.1.2 Reception of an event whose topics are not valid : *Argument-related mismatch* or *sematical mismatch*

a.1.2.1. Probable Origins :

a.1.2.1.1. QoS metering, QoS evaluating, or ConfInfoProvider failure or topics semantic misunderstanding.

a.1.2.2. Detection :

a.1.2.2.1. by the user

a.1.2.2.2. by AuthorSide service

a.1.2.2.3. by the evaluation component

b. EXECUTION

b.1 Inscription & submission :

b.1.1 Non reception of the ack (within a given time defined by the Quality of service Agreement Contract (QAC)) the after author inscription (by the user from the SubmissionMgr): T*ime-related mismatch*

b.1.1.1. Probable Origins :

b.1.1.1.1. connection loss (between the user and the AuthorSide service, or between AuthorSide service and the orchestrator, or between the orchestrator and the SubmissionMgr)

b.1.1.1.2. AuthorSide crash

b.1.1.1.3. SubmissionMgr crash

b.1.1.2. Detection :

b.1.1.2.1. only by the user (in case of AuthorSide crash or connection loss between the user and the AuthorSide service)

b.1.1.2.2. by the AuthorSide service (if connection loss between the AuthorSide service and the orchestrator)

b.1.1.2.3. by the evaluation component (in case of SubmissionMgr crash or connection loss between the orchestrator and the SubmissionMgr)

b.1.2 Non reception of the ack (before the deadline defined by the QAC) after paper submission (by the user from the SubmissionMgr) : *time-related mismatch*

b.1.2.1. Probable Origins :

b.1.2.1.1. connection loss (between the user and the AuthorSide service, or between AuthorSide service and the orchestrator, or between the orchestrator and the SubmissionMgr)

b.1.2.1.2. AuthorSide crash

b.1.2.1.3. SubmissionMgr crash

b.1.2.2. Detection :

b.1.2.2.1. by the user (if AuthorSide crash or connection loss between the user and the AuthorSide service)

b.1.2.2.2. by the AuthorSide service (if connection loss between the AuthorSide service and the orchestrator)

b.1.2.2.3. by the evaluation component (if SubmissionMgr crash or connection loss between the orchestrator and the SubmissionMgr)

b.1.3 Login problem after an inscription (to a user) *Functional mismatch*

b.1.3.1. Probable Origins :

b.1.3.1.1. connexion loss (between the user and the AuthorSide service, between the AuthorSide service and the orchestrator, between the orchestrator and the SubmissionMgr)

b.1.3.1.2. failure or crash of SubmissionMgr

b.1.3.1.3. failure or crash of the AuthorSide service

b.1.3.2. Detection :

b.1.3.2.1. by the user (if connexion loss between the user and the AuthorSide service or AuthorSide service failure or crash)

b.1.3.2.2. by the AuthorSide service (if connexion loss between the AuthorSide service and the orchestrator)

b.1.3.2.3. by the evaluation component (in case of connexion loss between the orchestrator and the SubmissionMgr or in case of SubmissionMgr failure or crash)

b.1.4 Non reception (before the deadline defined by the QAC) of a submission by the SubmissionMgr after user's inscription: *Time-related mismatch*

b.1.4.1. Probable Origins:

b.1.4.1.1. the user never send it

b.1.4.1.2. connexion loss (between the user and the AuthorSide service, between the AuthorSide service and the orchestrator, between the orchestrator and the SubmissionMgr)
b.1.4.1.3. failure or crash of the AuthorSide service
b.1.4.2. Detection
b.1.4.2.1. by the user (if connexion loss between the user and the AuthorSide service or AuthorSide service failure or crash)
b.1.4.2.2. by the AuthorSide service (if connexion loss between the AuthorSide service and the orchestrator)
b.1.4.2.3. by the SubmissionMgr (in all cases)
b.1.4.2.4. by the evaluation component (in all cases)

b.2 Reviewer Assignment

b.2.1 Non Assignment of a paper: *functional mismatch*
b.2.1.1. Probable Origins :
b.2.1.1.1. connexion loss (between the ReviewerMgr and the orchestrator, between the orchestrator and reviewerSide service, or between the reviwerSide service and the reviewer)
b.2.1.1.2. failure or crash of the reviewerMgr
b.2.1.1.3. failure or crash of the reviwerSide service
b.2.1.2. Detection :
b.2.1.2.1. by the reviewerMgr (all cases except reviewerMgr crash or failure)
b.2.1.2.2. by the author (all cases)
b.2.1.2.3. by the evaluation component (if connexion loss between the orchestrator and the reviewerSide service or if reviwerSide service crash or failure)
b.2.2 Assignment to non qualified reviewer (topics and skills non concordant) *argument-related mismatch*
b.2.2.1. Probable Origins :
b.2.2.1.1. failure of the ReviewerMgr
b.2.2.2. Detection :
b.2.2.2.1. by the reviewer
b.2.2.2.2. by the evaluation component
b.2.3 Assignment of a paper to one of its authors (or of the same institution) *parameter-related mismatch*
b.2.3.1. Probable Origins :
b.2.3.1.1. failure of the ReviewerMgr
b.2.3.2. detection :
b.2.3.2.1. by the reviewer
b.2.3.2.2. by the evaluation component

b.3 Review Process:

b.3.1 Paper (supposed to be sent by the ReviewerMgr to the reviewer) reception missing the deadline: *Time-related mismatch*
b.3.1.1. Probable Origins :
b.3.1.1.1. propagation of b.2.1
b.3.1.1.2. connexion loss (between the ReviewerMgr and the orchestrator, between the orchestrator and the ReviewerSide service, or between the ReviewerSide service and the reviewer)
b.3.1.1.3. failure or crash of the ReviewerMgr
b.3.1.1.4. failure or crash of the ReviewerSide service
b.3.1.2. Detection:
b.3.1.2.1. by the reviewer (all cases)
b.3.1.2.2. by the ReviewerMgr (if connection loss between the ReviewerMgr and the orchestrator)

b.3.1.2.3. by the evaluation component (if reviewerSide service crash or failure or connection loss between the orchestrator and the reviewerSide service)

b.3.2 Report (supposed to be sent by the reviewer to the ReviewerMgr) reception missing the deadline : *Time-related mismatch*

b.3.2.1. Probable Origins :

b.3.2.1.1. report not sent by the reviewer

b.3.2.1.2. connection loss (between the reviewer and the reviewerSide service, between the reviewerSide service and the orchestrator, or between the orchestrator and the ReviewerMgr)

b.3.2.1.3. failure or crash of the reviewerSide service

b.3.2.2. detection :

b.3.2.2.1. by the reviewerMgr (all cases)

b.3.2.2.2. by the evaluation component (all cases)

b.3.3 Report not related to the paper (paperId and report are not consistent): argument-*related mismatch*

b.3.3.1. Probable Origins :

b.3.3.1.1. the reviewer (confusing two assigned paper ids)

b.3.3.2. detection :

b.3.3.2.1. by the ReviewerMgr (if it is implemented to do so)

b.3.3.2.2. by the author

b.4 Decision and notification :

b.4.1 decision (made by the SubmissionMgr and supposed to be sent to the Author) missing the deadline *time-related mismatch*

b.4.1.1. Probable Origins

b.4.1.1.1. propagation of fault 2.2.1

b.4.1.1.2. connection loss (between the SubmissionMgr and the orchestrator, between the orchestrator and the AuthorSide service, or between the AuthorSide service and the Author)

b.4.1.1.3. chairman did not take it

b.4.1.2. Detection :

b.4.1.2.1. by the author

b.4.1.2.2. by the evaluation component

b.5 Final Paper :

b.5.1 Final paper reception (supposed to be sent by the author to the ApprovalMgr) missing the deadline *Time-related mismatch*

b.5.1.1. Probable Origins :

b.5.1.1.1. the author did not send it

b.5.1.1.2. connexion loss (between the Author and the AuthorSide service, between the AuthorSide service and the orchestrator, or between the orchestrator and the ApprovalMgr

b.5.1.1.3. crash or failure of the AuthorSide service

b.5.1.2. detection :

b.5.1.2.1. by the ApprovalMgr

b.5.1.2.2. by the chairman

b.5.1.2.3. by the evaluation component

Figure 59 gives, following the presented mismatches decomposition, a classification of the previously described mismatches of the cooperative review system.

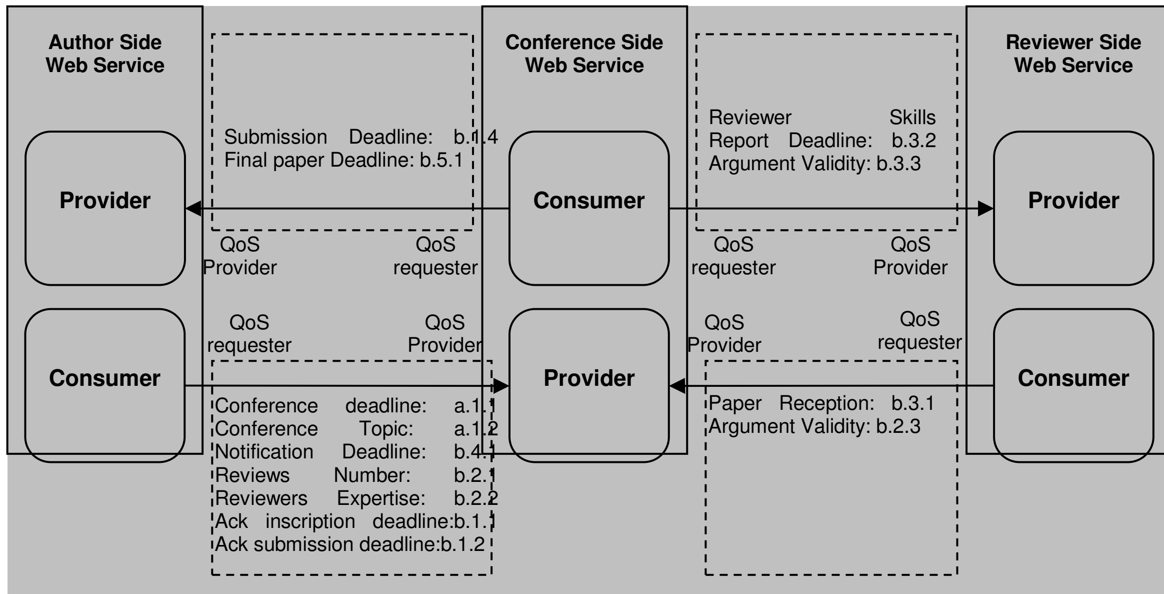


Figure 58 :QAC contract parameters

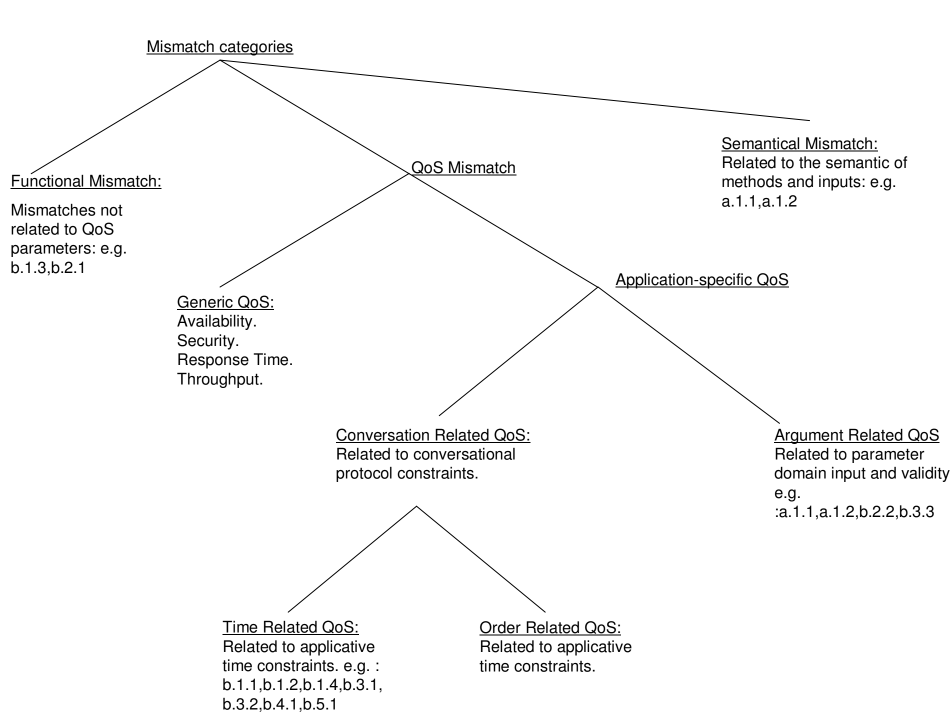


Figure 59 : Cooperative review Mismatches classification

3.2.4 Diagnosis process

The proposed architecture, illustrated in Figure 60, distinguishes three architectural levels: "Orchestration and QoS Monitoring Module", "Diagnosis and Recovery Module", and "Reconfiguration and Repair Module".

- The first architectural level is the "Orchestration and QoS Monitoring Module" (O&QoSM). Each pair of web service requester and web service provider is associated with a pair of QoS evaluation and QoS metering components. The monitoring components may be implemented on the provider-side or on the requester-side, or distributed on both sides. These components may store QoS information in journals for off-line diagnosis.
- The second architectural level deals with processing QoS mismatches for fault diagnosis and with decision related to Recovery. It is named the "Diagnosis and Recovery module" (D&RM). Different architectural and algorithmic choices may be done for implementing this module. In the first case, a unique diagnoser will centralize collecting QoS mismatches from all the QoS evaluators for different WS requesters and different WS providers. Other choices are possible: a diagnoser may be associated for a unique WS provider, or for a group of WS providers related to the same application, or having the same role in a given application, etc… Hierarchical distribution may also be considered.
- The third architectural level deals with performing the repair and reconfiguration actions with decision related to recovery. It is named the "Reconfiguration & Repair Module" (R&RM). This part needs an additional work to provide appropriate abstractions for reasoning about the different choices for implanting this module. LAAS-CNRS will work on this part.

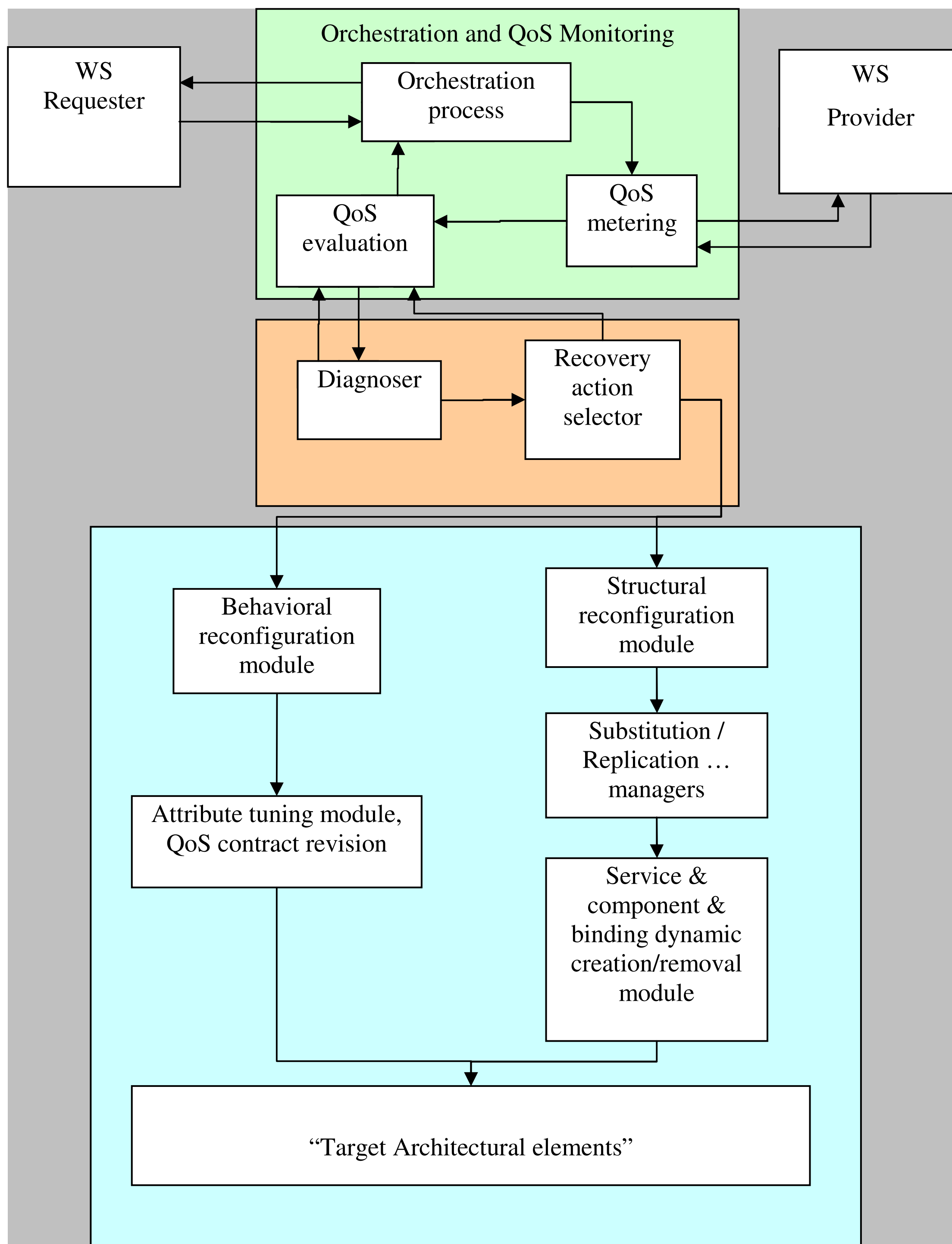


Figure 60 : General architecture for diagnosis and reparation

The diagnosis process relies on the detection of QoS mismatches. The QoS evaluation and metering processes analyze the conversations between web services, and fire observable events when mismatches are detected. From the diagnosis point of view, these processes behave like logical probes, filtering the network activity into a pertinent sequence of events. In some cases this sequence of events may need to be compared with other sequences recovered by other QoS related processes in distant sites in order to establish proper diagnosis.

The Diagnosis and Recovery architectural elements are included in the "Diagnosis and Recovery module". This module is compound of the "Diagnoser" component which is in charge of collecting and analyzing QoS mismatch events transmitted by the QoS evaluation component. It executes an algorithm that allows it to decide if there was a "fault" behind the observed QoS mismatch(es) and to identify the origin of this possible "fault". The diagnoser may react by telling the evaluator that there is no fault and that the response is acceptable and must be transmitted to the requester. If the diagnoser estimates that there is a fault, it contacts the recovery action selector (RAS).

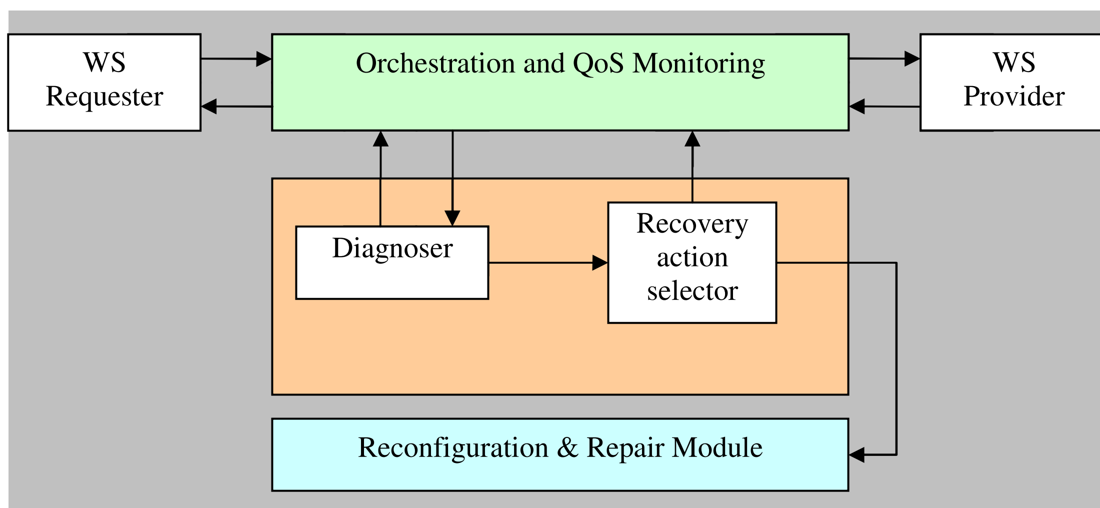


Figure 61 : Proposed architecture

The methods and algorithms used to perform diagnosis depend on the information available in each implementation site. Nevertheless, the approach described in [Ardissono, Console & al.] may be of interest whatever the implementation imposes, since it offers the possibility to establish QoS parameters dependencies between several conversations involving different web services. For example, the mismatch due to a delay for the paper reception by the reviewer may be due to a non-assignment of this paper to any reviewer. Hence the variable "date of paper transfer" is elaborated by the ReviewerMgr in function of the date of this paper's submission. An incorrect value for the former can result of an incorrect value of the latter, or of an incorrect computing.

In some cases, the repair action will involve QoS contract modifications, like for example, a deadline report. A QoS mismatch may also result of an inconsistent or out of date QoS contract, it is then important that the O&QoSM keeps track of past reconfigurations for the diagnosis process.

The diagnosis process will not aim at the same precision in internal or external perspectives. In the first case, it is necessary to establish exactly which fault occurred (discriminate), whereas in the second case, knowing in which web service it occurred suffices (isolate). When an inscribed user does not send a paper before submission deadline and the responsibility is clearly affected to the AuthorSide service, discriminating the user not sending, failure, and crash cases only in internal perspective.

According to the information provided by the diagnoser, the RAS may take two kinds of decisions:

- In the first case (for example the fault is not critical or is evaluated as transitory), the RAS tells the QoS evaluator to throw an exception that will be caught by the orchestrator. The orchestrator reacts by retrying the invocation.
- In the second case (the fault is critical),
    - o the RAS, first, contacts the R&R module for reconfiguration of architecture & behavior,
    - o then it sends to the orchestrator a sequence of "recovery actions" (new requests for example) to process the request that has failed. The orchestrator executes these actions and collects and sends the resulting responses to the end service requester (ESC).

It may be possible that repair succeeds and will be helpful for future requests but **past** requests cannot be compensated. In this case, it throws an exception towards the orchestrator. The orchestrator forwards it the ESC. These scenarios are described as sequence diagram in Figure 62.

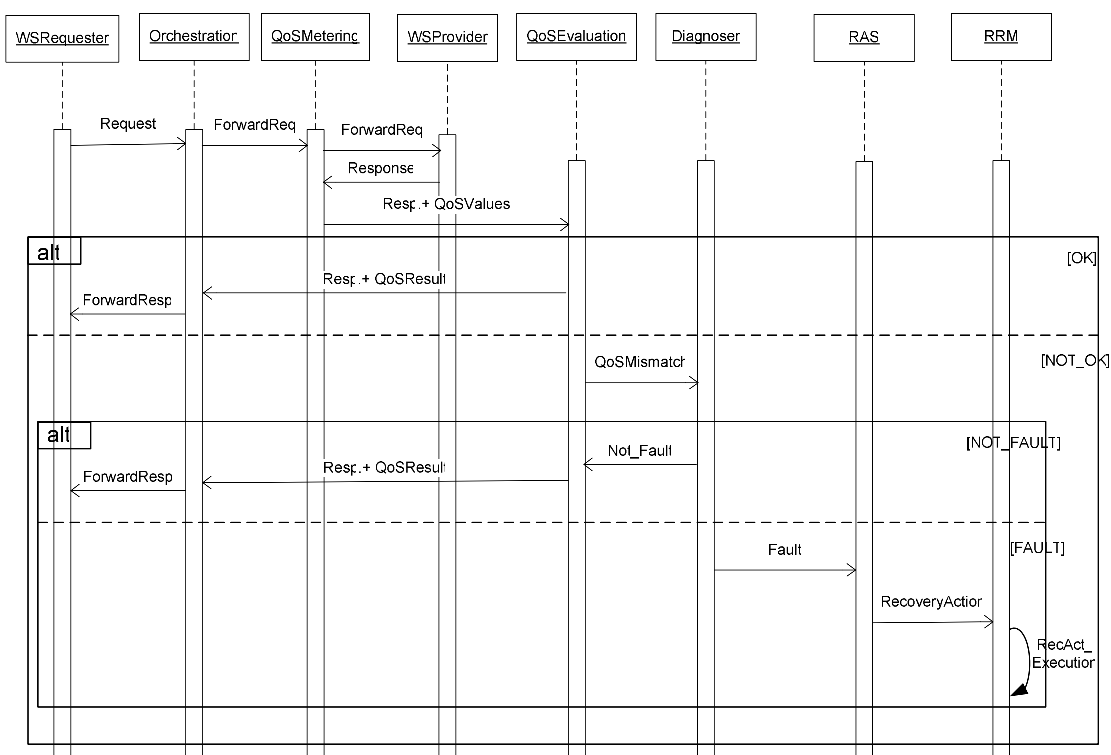


Figure 62 : Sequence Diagram for diagnosis and repair

3.2.5 Repair stage

There are two kinds of repair actions generated by the D&R module:

- Structural reconfiguration actions (SRAs) which consist in :
    - o substituting, replicating, wrapping, adding, or removing services and components, on the provider-side, and/or
    - o binding, unbinding or rebinding components and services to each others.
- Behavioral reconfiguration actions (BRAs) may be achieved by :
    - o tuning attributed of components, QoS parameters of services while this is possible,
    - o when tuning is not possible, a QoS contract may be revised (acceptable degradation will be defined). This leads to modification of the evaluator parameters.

The repair actions are performed on the "Target Architectural elements". The SRAs affect mainly the provider-side only for creation/removal actions. BRAs affect the provider-side as well as the orchestrator module, and the D&R module.

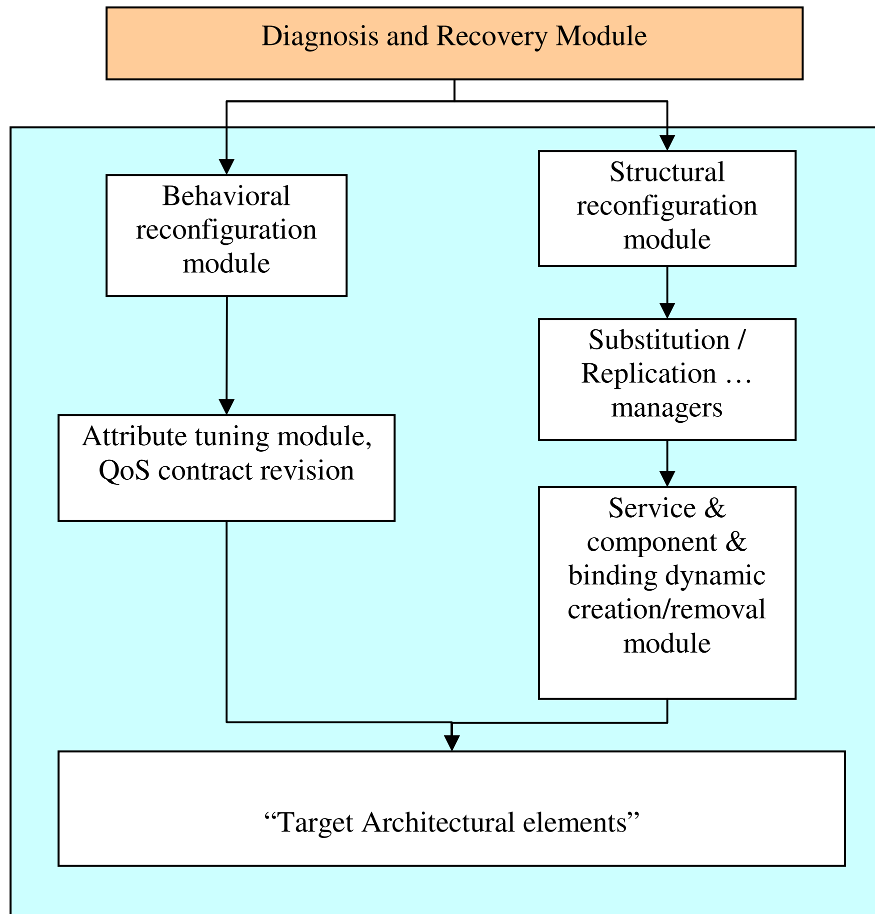


Figure 63 : Diagnosis and Recovery module

3.2.6 Evaluation and conclusions

Two different design options are possible.

In the first case, we can devise the whole system so that the orchestrator and the WS requesters are aware of the diagnosis and repair/recovery intermediates. They even may be involved in the diagnosis and repair/recovery processes. The QoS evaluator and metering components may be implemented at the applicative level as part of the orchestration process or the requester and provider sides.

- The advantages :
  - repair, recovery processes may be simply designed
- The disadvantages of this kind of design solution are:
  - loss of modularity. Changing/Updating of QoS managing components will also lead to modifications in the orchestration process

In the second case, we can devise the system so that the orchestrator and the WS requesters are totally unaware of the diagnosis and repair/recovery intermediates. The QoS evaluator and metering components may be implemented as interceptors at the SOAP level.

- o The advantages of this kind of design solution are :
  - genericity: the workflow of the orchestrator is not affected by handling QoS, Diagnosis, repair and recovery
- o The disadvantages :
  - repair, recovery processes may be more complex

Elaborating architectural abstractions is useful and may be driven by two requirements at the design and the diagnosis level:

- Optimise the diagnosis process and minimise the reconfiguration statement at the level of repair processes. This would allow reasoning on reconfiguration actions at a high level of abstraction.
- Optimise the design process and maximise genericity for the reuse of architectural elements.

General Discussion

The following notations are used in the sequel:

FS: Foodshop example,

CR: Cooperative Review example,

TS: Travel Services example.

The three examples seem to be of equivalent interest. They cover different kinds of application domains. They were developed addressing different but *complementary* functional characteristics, namely cooperation [CR], coordination [TS] and orchestration [FS].

Their functional descriptions consider different but *complementary* specification details (internal behaviours [FS, TS] vs. external interaction [CR]), specification levels (abstract [FS] vs. operational [CR], design-time [CR] vs. run-time or deployment [TS],) and specification points of view (business process described by complex workflow diagrams involving human and software actors [FS,TS] vs. cooperation process described by complex interaction sequences between software components [CR]).

The non-functional descriptions also consider different but *complementary* specification details for symptom characterisation (transactional properties [TS] vs. ad-hoc properties [FS] vs. QoS contracts [CR]), specification levels of diagnosis and repair (operational management architecture for reconfiguration based on dynamic adaptability [CR], mechanisms for global state consistency management based on transactions [TS], process-specific corrective actions [FS]).

Detailed Comparison

- The examples are specific instances of more general activity templates that cover two kinds of application domains:
  - o The “supplier/consumer chain” [FS,TS]. This domain of activity is a good application domain in general. It is considered by several projects and

publications. It becomes a more challenging activity domain with the cooperative characteristics between the actors. Recent research efforts go in this direction.

- The "cooperative design review" [CR]. This kind of activity belongs to the more general "Engineering process". The "review step" is an intermediate step between the "design step" and the "production step". It has been considered in different projects such as IST-DSE for aerospace industry. It is also addressed in different research works, namely for system engineering, and software engineering.

- Different *complementary* functional characteristics are addressed by the present descriptions (namely for *monitoring* and *Diagnosis / repair*):
  - coordination-oriented, transaction-guided, rollback-based repair : [TS]
  - cooperation-oriented, contract-guided, reconfiguration-based repair : [CR]
  - orchestration-oriented, scenario-guided, ad-hoc repair: [FS]
- Different *complementary* specification points of view and detail levels are addressed by descriptions:
  - "Business process" point of view focusing on internal behaviour (workflow) of system components [FS, TS] / "Cooperation process" point of view [CR] focusing on describing interaction between system components
  - For [CR]:
    - an operational description of architecture (modules, components and connections)
    - behaviour (sequence and activity diagrams. Workflow description for internal behaviours was not provided) distinguishing design-time elements and run-time elements
  - For [FS]:
    - Abstract description of architecture is provided.
    - presentation and reasoning address the behavioral (BPEL) sequences
  - For [TS]:
    - High level deployment architecture is provided.
    - The description gave the activity diagrams with extended UML notations, more expressive than BPEL.
  - Exceptions and faults:
    - For [CR]:
      - Specification includes a concrete classification for "faults" by identifying QoS categories, mismatches leading to exceptions, and different alternatives of handling/reacting
      - For each category, an example where detection may be done was defined. Cases where only human users can detect, were discarded in order to respect the self-healing property
    - For [TS, FS]:
      - Described exceptions belong to "business point of view": For [TS], they are of kind: "no available flight", or "no available hotel", etc… For [FS], they are of kind: "parcel items are wrong",… For several of the specified exceptions, the detection is made by the "end user".
  - Diagnosis and Repair:
    - For [FS,TS], description addressed business-level compensation: actions are not seemless, always of kind: "re-ask another transportation means" [TS], "repeat the reservation for the wrong item" [FS], "ask the SUPPLIER to send the correct parcel" [FS]
    - For [CR], focus was put on operational repair architecture for generic repair actions by dynamic reconfiguration.
  - Complexity of interaction schemas:

- [CR] specification scenarios show explicitly the complexity of interactions:
    - The activity involves different "business-level steps" for each actor:
        - For authors: service lookup, registration, first submission, final submission.
        - For Conf Service: reviewer lookup, assignment of reviews (bidding) and sending the papers to reviewers, notifying authors, etc…
        - For reviewer: accepting invitation, sending reports,
    - For a unique "business-level step", there is different asynchronous ("one way") and synchronous interactions ("request/response") in different directions between different pairs of the global WS actors
- [TS] and [FS] specification scenarios do not show the complexity of interactions:
    - The *specified* cooperation scenarios are composed of "single" independent interaction schemas of kind "request and wait", one operation by actor:
        - From the consumer point of view: I ask for booking [TS], or for food [FS], I wait the response, (a unique blocking type of interaction from the client point of view).
        - From the provider point of view: contact the "hotel" and the "transport company" for [TS] (or food supplier for [FS]) and commit or recover.

Synthesis

It should be possible to cover the different identified issues by each example, but this would need additional effort for involved partners.

- for [FS] and [TS] : adding cooperation and describe more complex interactions:
    - Consider asynchronous steps to implement distributed decisions between actors:
        - Different loops of requests:
            - User asks for booking/buying,
            - the system proposes different solutions,
            - the user confirms/cancels -definitely/or not, immediately/or later with/or without delay- a given choice or asks for modification, gets back new offers, etc…
    - Qos:
        - Identify QoS parameters,
        - define agreement contracts considering generic, conversation-related and argument-related QoS).
    - Reconfiguration:
        - Refine the structural description and identify scenarios involving repair actions based on dynamic architecture adaptability: substitution, wrapping, etc…),
    - Service discovery step description and analysis (looking and selecting suppliers, sub-contractors, booking services).
- for [CR]:
    - behavioral description:
        - Provide the workflow describing the internal behaviours of actors,

  - consider user-level decisions, interactions, exceptions and activity-specific corrective actions:
    - Consider content and semantic-related errors:
      - inconsistent content of a review report,
      - Confusion in interpretation of terms in selecting reviewers based on skill key-words: assigning a paper related to software architecture to urban architect, etc…)
    - transactional behavior:
      - managing complementary lists for acceptance and publishing,
      - re-assignment of reviews …
- business-level repair actions:
  - Replace the reviewer,
  - re-send the paper,
  - re-assign papers, etc…

3.3 Test Case: travel services

The example is about a a multi-step travel organizer. The scenario involves four kinds of web services: (i) a customer, (ii) a travel agent service, (iii) an airplane online ticket seller service and (iv) a hotel booking portal(see Figure 64).

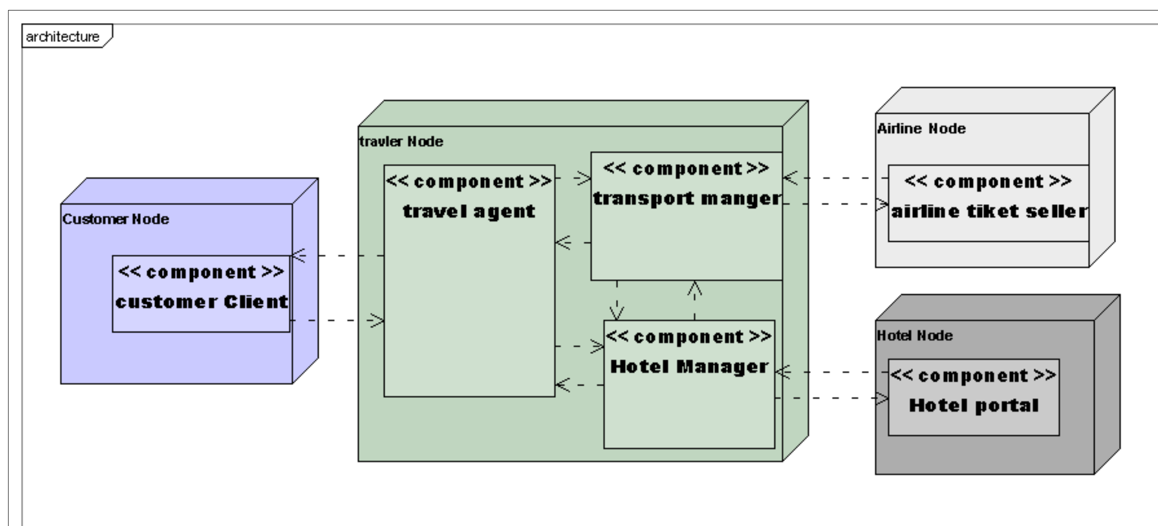


Figure 64 :The architecture of the travel agent Example

The service begins with a customer request. The request has the following semantic : The customer wants to visit several successive locations; with each location but the first, he associates an arrival deadline and an earliest departure time. There is no time constraints for the first departure point. Obviously, for a location, the arrival deadline must be before the departure time. The travel agent which receives this request has to organize the customer travel planning by buying the tickets and by booking the hotels if the arrival and the departure times -for a given location- occur on different days. An instantiation for this example could be, for a professor, the planning of his successive communications in different conferences all around Europe, in a quite tight period of time. An example of request in this context could be:

{(location=Paris,date=10/11/05,{}),
(location=Turin, date=10/11/05, deadline=12:30),
(location=Turin, date=11/11/2005, departure=16:00),
(location=Amsterdam,date=12/11/2005, deadline=13:00)\}

The rest of the report is organized as follows. In section ... we list, for each service, the set of atomicactivities. In section 3.3.1. we introduce the used notation. After that we design the individual behaviors of each service (the WSDL interface and the BPEL process description). In section 3.3.2 we list the possible faults and we try to trace some corresponding scenarios. The section 3.3.3, We propose a classification the repairing and reconfiguration tasks types according to their nature and their design needs, each of them is illustrated by a scenario from the travel agent example.

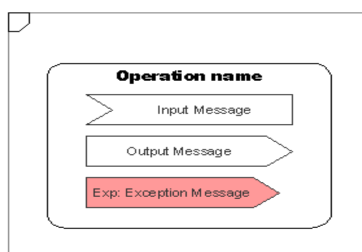
3.3.1 Workflow

3.3.1.1 The actors Interface descriptions

In the **WSDL** specification, a service is described by a set of operations. An operation is described by an "***input***" message and/or an "**output**" message and an optional "***fault***" message. According to the **WSDL** semantics, we present, For each service, the set of atomic operations. For simplicity purposes, we use the following **BNF** notation, instead of **XML**:

grammer	*WSDLoperation::= <operationName> "[" ("?" <message_name>[,"!" <message_name>])| ("!"<message_name>[,"?" <message_name>]) ["Exp : " message_name>] "]"* *?: input message* *!: output message* *Exp: Exception message *
example	*divide[?oprand, !result, Exp: byZeroException]*

The Figure 65 represents a graphical representation of a WSDL operation. use also a graphical



representation of a wsdl opération

Figure 65 : Graphical representation of a WSDL operation

3.3.1.1.1 The customer

The customer Agent service is the portal of the whole service. It offers a set of accessible operation to human user (see Figure 66).

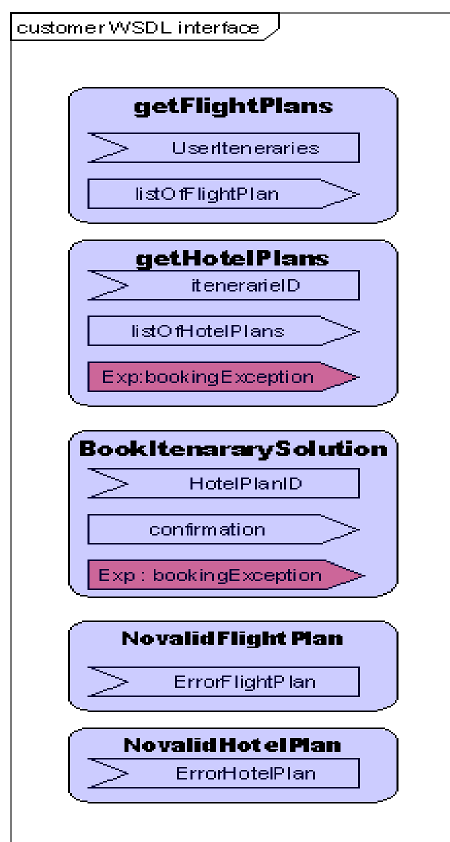


Figure 66 : Cutomer WSDL interface

GetItineraries[!UserItinerary, ?listofFlightPlan]

The customer formulates its request by specifying the list of locations and the dates windows. This message is sent to the Travel agent Web services. The travel agent web service sends back a set of possible flight plans satisfying the customer requirements. An empty list means that the user itinerary constraints cannot be satisfied.

ChooseItenerary[!IteneraryID, ?listofHotels, Exp: ?problemOfBooking]

After receiving the list of itineraries, the user can, according to the specified criteria, choose the most appropriate. After receiving the user selected itinerary, the travel agent web service composes and sends to the user a set of possible solutions for the hotels according to the geographical and time constraints for the transport solution. An exception can be raised by the travel agent if a problem occurs while booking the Flight composing the choosen Flight Plan.

NoValideItenary[!novalidItenaries]

Here we introduce a possible checkpoint by offering to the user the possibility to raise an alarm if the solutions offered by the previous operation do not respect her/his constraints.

chooseHotelSolution[!HotelsID, ?Cofirmation, Exp: ?problemOfBooking]

After receiving the list of hotel solutions, the user can - according to her/his criteria - choose the suited one. The travel agent either sends a confirmation message containing an ID and a password for the Flight tickets and the Hotel book or raises an exception if the hotel reservation failed.

NoValidHotels[!novalidItenaries]

As for the itinerary operations, we introduce a possible checkpoint by offering the user the possibility to raise an alarm if the solutions provided do not correspond to the transport plan.

3.3.1.2 The travel agent

Here we present the atomic operations which compose the travel agent services. We presents only the operations exposed by the service to the customer service partner (Figure 67).

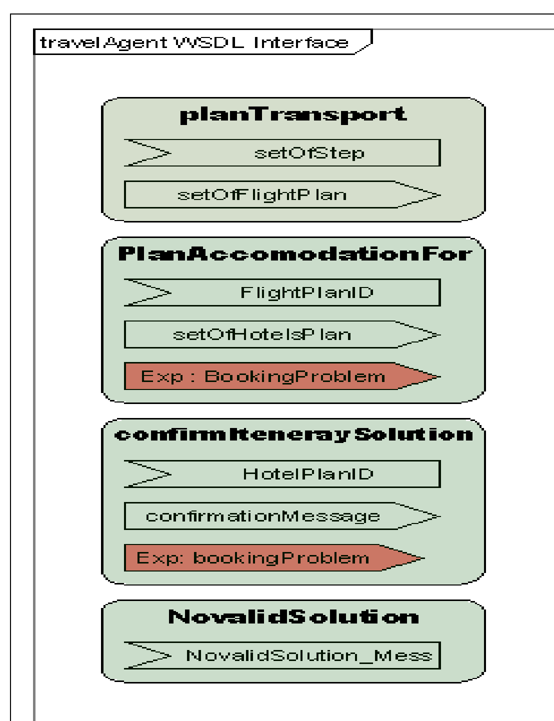


Figure 67 : Travel agent WSDL interface

3.3.1.2.1 From the customer point view

PlanTransport[?setOfStep,!setOfFlightPlan]

This operation is visible for the customer. It receives the specification of the itinerary (steps and time constraints) and returns a set of flight plan propositions (possibly empty).

NoValidSolution[?noValidesolution_Mess]

This operation can be invoked by the customer if the proposed Flight Plans or hotel plan do not correspond to the time and locations constraints.

PlanAccomodationForAFlightPlan[?FlightPlanID, !setOfHotelPlan, Exp: ?Bookingproblem]

it receives the user choice (itinerary) and - according to the flight plan – the service, using the hotel service, tries to compose a set of hotel solutions. Before composing a hotel solution the Travel agent tries to book the list of Flight. Tow case are possible, either its succeeds then its aggregates the result of the hotel portal services, the agent send a list of hotel solutions, or it fails then an exception is raised.

ConfirmItenerarySolution[?HotelPlanID, !confirmationMessage, Exp: ?Bookingproblem]

It receives the hotel solution chosen by the user. The travel agent tries to book the list of Hotels. Tow cases are possible: (i) either its succeeds then it sends to the user a confirmation message containing its personnel information about the Flights and the Hotel booking (ii) or it fails then an exception is raised.

3.3.1.3 The airplane ticket service and the hotel service

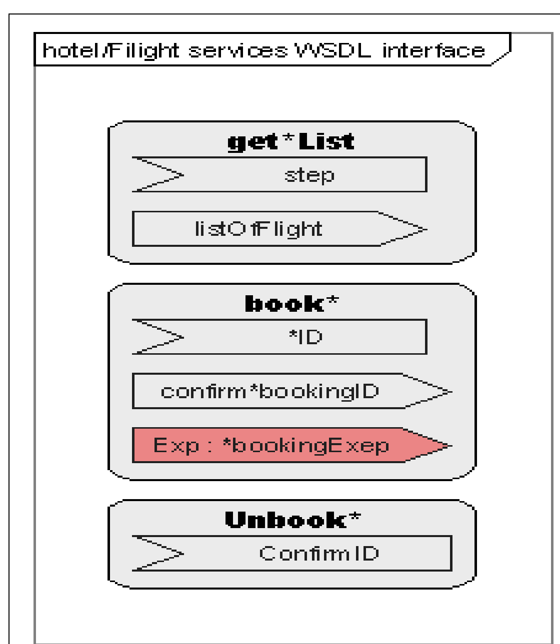


Figure 68 : Hotel/Flight WSDL interfaces

These services have similar set of operations. The "*" represents here "flights" and "hotels". See Figure 68.

get*List[?step, !listof*]

operation offring the possibility of consulting airline ticket and the hotel available offers.

book*[?*ID, !confirm*bookID, Exp: *bookingExcep]

It allows to the travel service for booking the customer choices. The operation can raise an exception if the product is not available any more or has a wrong ID etc. Booking problem exception is a class of exceptions which can occur within these operations (see **section Exceptions**).

Unbook*[?unbook*]

it offers the possibility to unbook a flight or hotel. This operation is useful for the case of customer cancellation or if no hotel solution was found.

3.3.1.4 The actors process descrptions

In this section we present the behaviors of each actor. We describe there processes using a specific codification of the **BPEL4WS** using **UML** activities diagram constractor. The Figure 69 summarises the way that we translate **BPEL4WS** artefacts using activities diagram constructors.

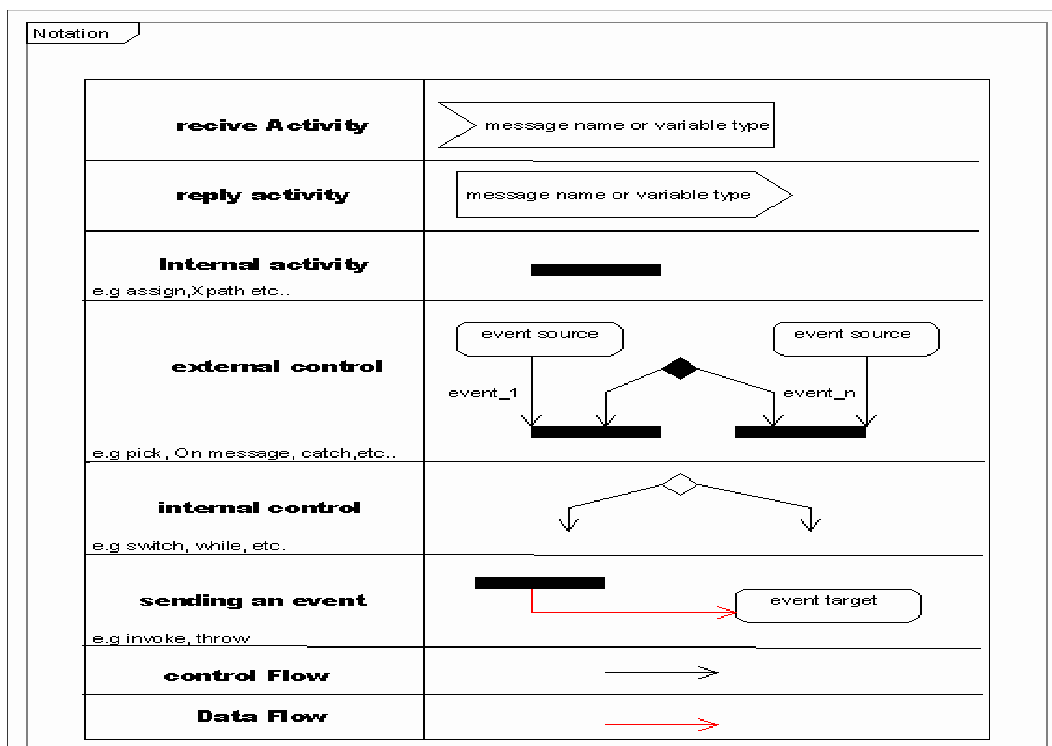


Figure 69 : Notation UML vs BPEL constructors

3.3.1.4.1 Customer process

The customer begins its interaction by sending the itinerary description to the travel agent. Then it receives a list of transport solutions (flights, companies, dates and times, costs, etc.). If the list is empty, the interaction ends; otherwise, the customer gives the control to the user. The user can choose a transport solution or detect a wrong plan according to her/his constraints. In the latter case the interaction ends. Here we can imagine that the "noValidFlightPlan" Message is a possible alarm.

When choosing a Flight plan, the user receives either a list of hotel solutions or a message indicating that one (or more) of the flights composing the plan can not be booked (no more available seats) or if a problem occurs within the IDs, etc (see Exceptions section). The interaction ends. Similarly, when receiving the hotel list, the user can choose a solution or raise an alarm when the hotel solutions do not correspond to the flight plan (additional or missing day in a hotel, a step without a hotel solution, etc.). If the list is empty or the customer raises an alarm the interaction ends otherwise the customer receives a message either confirming the booking or

indicating that a problem occurred within the booking action. The Figure 70 represents the customer behaviour activities.

3.3.1.4.2 The travel agent process description

The service is instantiated when it receives an itinerary request from the customer. The service decomposes the description into a set of flights and asks the airline ticket service for the flights list. For each location and a date constraints (get flight from Paris to Turin for date1 when the time of arrival is less than t1). When it receives the list, the travel agent tries to compose a set of solutions according to the internal criteria (cost for example, minimum of stopover). The list of itinerary solution is returned to the customer for choice. The customer can have one of the three behaviours:

- **(i)** ends the interaction (when the list is empty), the travel agent ends too.
- **(ii)** raises an alarm, the travel agent can here introduce a diagnostic process and ends the interaction. We can also imagine the possibility of reconfiguration once the fault is localized, etc.
- **(iii)** sends an itinerary ID. The travel agent tries to book the list of flights composing the itinerary solution. Two cases are possible :
    - At least one flight cannot be booked, a message is sent to the customer service to inform it that a problem has occurred while booking. The cause of the booking failure can be a wrong data or the unavailability of places. In the first case a diagnosis process can rectify the fault while in the second case a reconfiguration solution can be used.
    - When the booking action succeeds, the travel agent asks the hotel service for each night the itinerary contains. According to the flight plan solution a set of possible hotels is sent to the customer for choice. The customer response can be analogous to the flight plan choice process (empty list, choose one, or send no valid solutions).

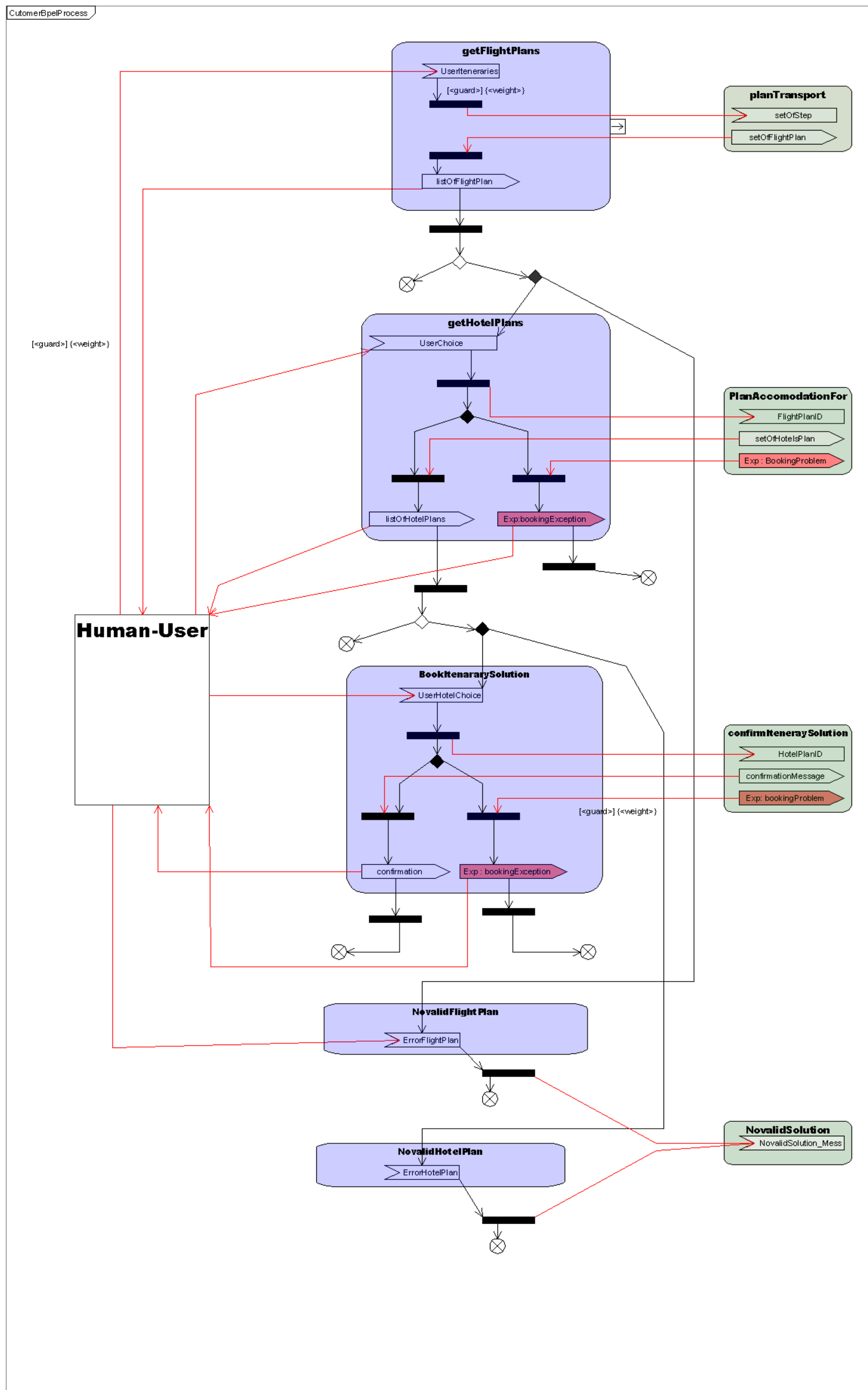


Figure 70 : The Customers BPEL process

If the customer chooses a solution, the travel agent tries to make the booking. If the booking succeeds, a confirmation message is then sent to the customer otherwise the travel agent "unbooks" the flights list and sends to the customer a message informing that an error occurred while booking the hotel. The Figure 71 represents the travel agent process behaviour.

3.3.1.4.3 The hotel and airline process description

The hotel and airline services are quite similar. They are composed of three independent operations. The first allows for consulting the databases to answer to a request (list of flight or list of hotels). The second offers an operation of booking a flight or a hotel place using the product ID. This operation can raise an exception if a problem occurs. The third one is an "unbook" operation. In this version we present the services behaviour as a simple web service but we can imagine a complex interaction process e.g (i) get the offers (ii) wait for a booking operation and (iii) if it is not booked in a period of time the product is released and we can suppose that if a booked one is not unbooked within an other period (a day for example) then the service persist it.

3.3.2 Exceptions

First of all, we emphasise some points: (i) the example involves only Web services: there is no human intervention. (ii) The final output is not a physical action (independent from the service execution), but a confirmation message, an on line diagnosis is thus possible. (iii) The customer in our service is solicited more than once so it is involved in the interaction and it can be a source of faults.

During the description of the services and their behaviours, we pointed out a set of specific messages which can be considered as possible alarms. Four types of message are considered here:

- **No valid Flight plan** : message raised by customer to signal that the flight plan does not meet its requirements.
- **No valid Hotel solution** : It is raised by the customer to signal that the Hotel solution does not correspond to the chosen flight plan.
- **Flight booking exception** : This type of message corresponds to a set of fault types, it is first raised by the airline services and sent to the travel agent. This message occurs while the travel agent tries to book a flight which does not exist or it is not flagged by that instance. The same type of exception is raised by the travel agent to the customer.
- **Hotel booking exception** : It is raised by the Hotel service and sent to the travel agent. This message occurs while the travel agent tries to book a Hotel which does not exist and or it is not Flagged by that instance. As the previous one this exception leads to the propagation of an exception to the customer.

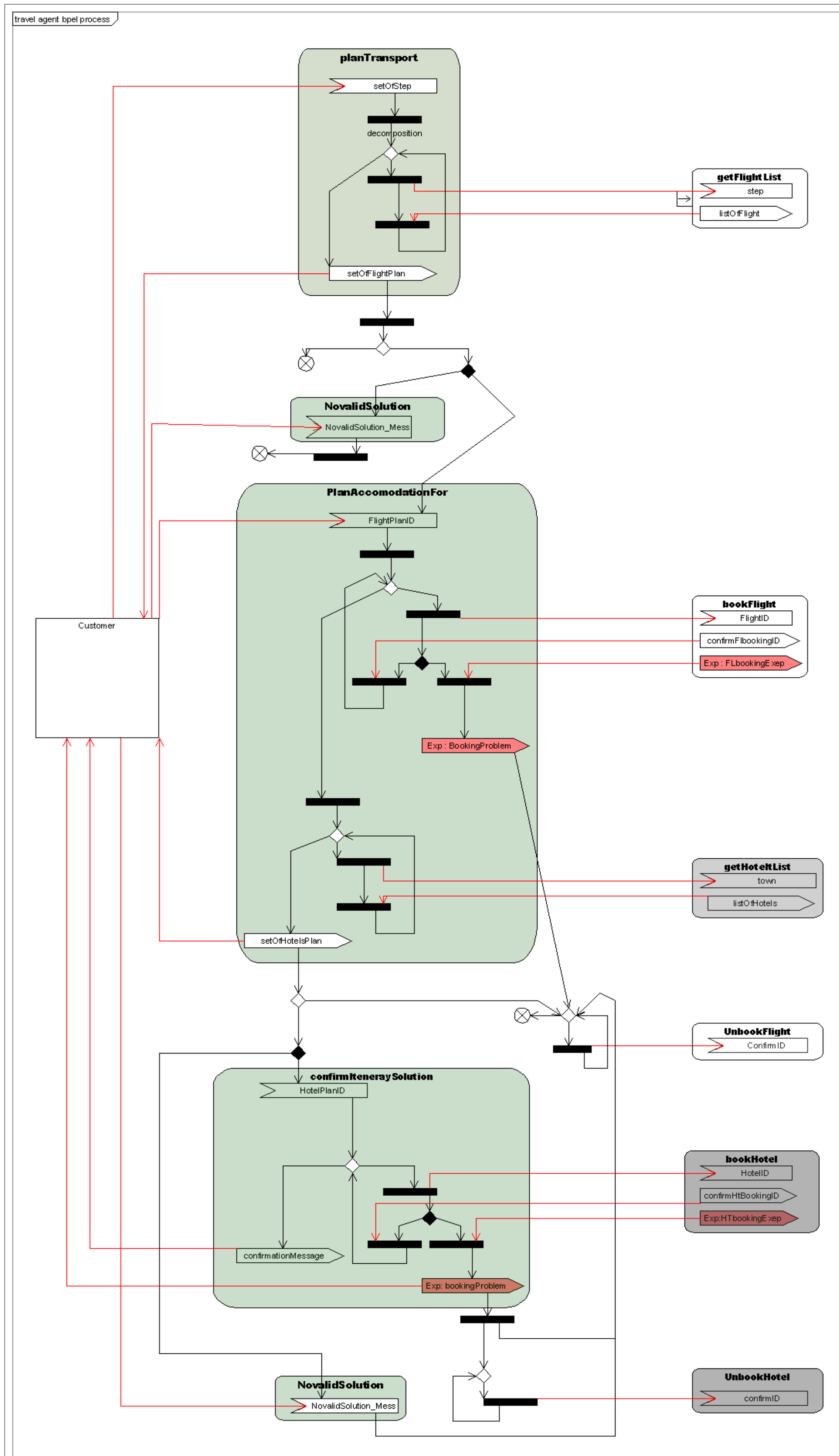


Figure 71 : The travel agent process behavior

3.3.2.1 No valid Hotel and Flight plans

The interaction is about itineraries which are complex structures that obey to a certain logic. The different steps must be a sequence of different towns. The time windows must be totally ordered and non overlapping etc. While the travel agent constructs a flight plan using basic information from the airline service, some faults may occur. So, let we consider that the customer, after receiving the list of flight plans, it detects that the plans are note valid. At that point of the interaction, only three parts are involved: the customer itself, the travel agent and the airline ticket seller service. Here are possible classes of data faults which can be raised at that point.

Missing or additional steps in the itinerary

The cause of such faults can be multiple and each of the three involved services can start a local analysis of log files in order to have an idea about its implication. Here are the possible causes of faults by involved actor:

- **The customer** can be the faulty if it expresses wrongly its itineraries message.
- **The travel agent** can be faulty in two possible cases:
    - While decomposing the itinerary : The travel agent try to extract from the Customer request a set of steps [departure town , arrival town, departure date window, arrival date Window] for each step it tries to extract independently all the possible flights for each step. It composes, for each step, a request and invokes the Airline service. At the end of this process, the travel agent has, for each step, a set of possible solutions. A set of faults may then occur: (i) it can express a wrong date window. (ii) it can miss one (or several) step(s). (iii) it can mix data of two steps.
    - While composing the solution : from the list of solutions, the travel agent tries to sequence a Flight plan.During the composition, some faults can happen: (i) it can choose two solutions with very closed or overlapped dates. (ii) it can compose a plan with two solutions of one step. The fault which can happen here could be the result either of a bad algorithm or of a consequence of a propagation of the previous faults. In the second case, the fault must be identified as a result of previous faults i.e. the context of the fault has to be correctly established.
- **The Airline** service can be faulty if it has a data base problem: it returns for given steps request a wrong list of solution by changing the town or flight outside the requested date windows.

Missing or additional nights in the hotel solution

The hotel solution construction is based on the user Flight plan chosen by the user and the hotel offered by the hotel Portal to construct solution. The diagnosis of this fault lead in addition to the diagnosis of the travel agent actions (constructing the list of nights, merging the results to construct a proposition etc.) and the Hotel portal service response, the diagnosis of the flights plan construction which mean that all the interaction and all the services are involved during the diagnosis process. The possible causes of a wrong Hotel solution are:

- **Bad Flight plan** the fault may occurs during the construction of the flight plan and the it doesn't be detected before. In that case, all the services are involved and the lists of possible faults listed befor are considered.
- **The travel agent** the set of Hotel requests are constructed by the travel agent according the flight plan. As the flight plan the travel agent can make errors while constructing the list of hotel requests. its can make error while aggregating the results of the hotel portal service.
- **The hotel portal data base fault** the hotel service could execute wrongly the request of the travel agent. we can imagine that it returns a wrong hotel in a wrong place or booking one additional night or a wrong date window.

3.3.2.2 Hotel And Flight booking Exception

These exceptions are raised respectively by the Airline and the hotel services. Booking Exception can correspond to a class of faults. Here are the set of such exceptions and the possible causes.

Data base error

This instance of exception are not diagnosable since it is caused by a crash of the data base.

No such Flight or Hotel ID

The data sent from the travel agent to book a flight do not correspond to an entry of the services' data bases. The causes of this fault can take source either in the travel agent which deteriorates the Identifier or in the Airline and hotel services which changed the data base entry despite of the flag puted for this entry during the consultation process.

3.3.2.3 Behavioral faults

This Kind of faults is detected by a non valid exchange message sequence. First of all, we must distinguish two types of behaviour faults (i) behavioural faults which result from a bad protocol engineering (ii) behavioural faults which can be resulted from a fault and thus can be interpreted as alarms. Here, we don't take into account engineering problems. We suppose that the overall choreography schema is correctly designed (e.g no deadlock problems), so we consider only behavioural faults resulting from data faults.

The behavioural faults considered here are the result of a set of monitoring properties according to business logics. The properties express some rules of the services process evolution according to

the handled data values.

For a given sequence δ of possible events $(\delta = e_1,...,e_{n})$ and a set of predicates $\psi(\upsilon)$ (where ψ is a set of predicates and υ is a set of variables : events, messages or local data), the services must behave as ρ.

We use this notation for this type of rules the notation is influenced by the CTL logic wen we expresse a classe of a set of sounf sequences and states according to a set of executed actions:

$$[\delta\ ]: \psi(\upsilon) \rightarrow \ \rho$$

Within our travel agent service we can imagine this rule for example :

let δ_1=*!itinerariesMessage,!ListOfstep,?listOFlightStep,!ListOfFlightplan*

$$\delta_1\text{: } EmptySet(ListOfFlightplan) \rightarrow 0 \vee \delta_1\text{: } EmptySet(ListOfFlightplan) \rightarrow \tau^*$$

which means that after sending the list of possible flight plans to the customer, if this list is empty then the only possible behavior of the travels agent is the termination.

We distinguish two subtypes of behavior faults; (i) local behavioral faults and global behavioral faults. The former is the result of a non valid sequences of messages according to a set of local business rules while in the second type each local sequence is valid (projection) but the composition (interaction schema) is not valid according to a set of cooperation rules.

3.3.3 Diagnosis and Repair stage

Face to the listed faults and while most of them are detected during the execution of the interaction, the diagnosis process can be done on-line and thus we can imagine for each type of fault a set of reconfiguration or repairing scenarios. According to the fault type we can distinguish three types of repairing actions.

3.3.3.1 Compensation actions to reach the services purpose

This type of repairing action concerns the situation then the service must find other service or a set of services to repair a fault. In our example this corresponds to the situation when no valid flight message is caused by a missing step in the flight plan. To repair this fault the travel agent can re-ask the airline service for a flight to the missing step. If no flight is possible we can imagine that the travel agent, if it is possible, proposes a train transport solution for this step.

3.3.3.2 Compensation actions for a proper termination

This kind of repairing is about the situation when the service purpose can not be reached and the interaction terminates negatively. For some consistency consideration (transactions for examples, data coherence, etc.) the service must coordinates some actions in order to terminate properly by undoing some performed actions. In our service we can imagine that the final confirmation message may contain some faults (propagation of one the cited faults (missing step for example) and that there is no airline or train solution (the pre-cited type of configuration). In this case, the service must undo all the booking tickets and hotels.

3.3.3.3 Repairing by re-executing a part of the interaction.

Some faults (for example data transmission faults) can be repaired by coordinating a re-execution of a specific part of the interaction. After detecting the source of the fault, using the coordination schema and equivalence relation over services states, a possible repairing can be a re-instantiation of the corresponding behaviours parts in each involved service. The instantiation and the execution can be made by the supervisors of each service. At the end, they pass the control to their services instances.

Note here that the three types of repairing need different types of services capabilities. The first type supposes that the service implements planing capabilities while the second type imposes that the service knows the main operation and the way to undo them. The last type of reconfiguration is the most simple as it is possible to implement an algorithms in the supervisors in order to synthesise the extended behaviours and thus without any change in the services implementation.

3.3.4 Comparison

The travel agent example is about planning a travel. The plan is constructed by first decomposing the user request and then composing the solution from different data sources. It is true that the food shop example can be viewed also as a plan construction since processing a command is processing all the requested products. But the travel agent example presents a set of specific additional features that can be interesting. We present here its main advantages.

Complexity of the diagnosis scenarios

In the travel agent example two plans are constructed: the transport plan and the accommodation plan. In the whole process until the transport plan construction, the travel agent presents the same complexity as the diagnosis scenarios in the food shop example (one has just to replace step by product). However within the travel agent example, the transport plan – which is a source of faults as mentioned in the document – is used to construct the accommodation plan. So we have here a causal relation between two sub-processes of the whole process. This allows more complicated

diagnosis scenarios as faults in the transport plan can be propagated to the accommodation plan construction. For example, if a fault (symptom) is detected in the accommodation plan then the diagnosis scenario can be either a local fault in the hotel plan construction or the result from a propagation of fault in the transport plan solution. In the latter case the reasoning is deeper than the reasoning needed for a transport plan fault (and by the same way the faults detected in the food shop example). Notice also that this causality between activities, viewed necessarily here as transactions, is present even in absence of faults: undo actions at the level of transport plan have to be undertaken in case of an impossibility to fulfil a step in the subsequent accommodation plan.

Another important feature dealing with the propagation is the learning aspect. In fact, as transport faults can affect the accommodation plan solution, it could be very interesting to characterise the fault by observing the manifestation of the faults causalities at the level of the process, e.g. a missing step in the flight plan leads obviously to at least one missing step in a hotel plan.

Flexibility: centralised, decentralised and distributed versions

The present version is more suitable for centralised or decentralised diagnosis approach. In fact, the travel agent Web service centralises the whole process and from this point is similar to the food shop. In the future we project to design a more distributed version, by dividing the travel agent into two complex Web services: transport manager and accommodation manager. The transport manager will try to find all the transport plans and the accommodation manager will do so for hotels in parallel. They will then interact which each other and the customer in order to build a travel solution. This version will distribute the travel agent and will make possible distributed diagnosis and repair. It gives rise also to more interesting fault scenarios involving interaction between transport and accommodation manager (some have already been identified).

Diversified repair actions

The example is about a spatial-temporal planning and so requires more reasoning about semantic knowledge. This makes the repair activities more attractive and more interesting. Within the food shop example the repair activities (by substitution for example) involve Web services that are very similar to the original Web services (check for another Warehouse Web service for example). However in the travel agent the repair activities – to deal with a missing step for example – will be taking a train or renting a car, or it can be a multi-step solution such as train plus car, etc. The reasoning about the distances and the time constraints will influence the repair activities.

Quality of service and degraded modes

The final product of the process can be evaluated with multi-criteria (number of steps fulfilled in the global plan, category of flights or hotels, cost, time of transport, etc.), not just black or white, and we can assume that the service can give only a partial solution (hotel missing at a step, hotel category lower than requested, dates of a step not exactly coinciding with the requested ones, etc.) for the travel plan and that the user can accept it. This can be viewed as QoS metrics. All the usual QoS, dealing for example with the response time, can be added as metrics measurements too.

User interaction for on-line vs. off-line diagnosis

The human user can use the customer agent operations (novalidFlightplan and novalidHotelplan) to raise exceptions and this at two occasions: after the transport plan and after the accommodation plan. The fact that the human user, which is involved in different critical steps of the whole process (three times), can detect faults can be viewed as an interesting source of symptoms in fault detection. In the food shop example the human user, after having sent his/her command, can react only when he/she receives physically the products (or the bill) that means after the process ends. In our example the final product of the process is a message containing the plan details information, so until the end of the process faults may happen and on-line diagnosis with user information is still possible. Of course, as in food shop example, the user can also (or only) detect faults after the

end of the process, for example when he tries to get his ticket and the flight is not reserved or is cancelled, etc., which can be used for offline diagnosis.

4 General Requirements for the proposed solution

4.1 Requirements for Web Service composition and execution for self-healing environments

The provided WS Diamond Description language should enable to express:

- General Service aspects, regarding service behaviour, compensability, negotiability, quality of service, and so on.
- Semantic annotation of services, operations and parameters referring to Web Service ontology.

4.1.1 General Service aspects

In order to state the requirements for service description, we consider that a Web Service is modelled as a software component which implements a set of operations and possibly returns structured data as a result of an operation invocation. A composite service is specified as a high-level business process (e.g., in BPEL language) in which the composed Web Service is specified at an abstract level. We refer to *abstract Web services* as a task t_i, while Web Services selected to be executed are called *concrete Web Services*. To support adaptive concretization, *semantic annotations* to the BPEL process may be defined to specify either intrinsic characteristics of the process, or requirements by the user of the composite service. As language requirements, it should be possible to describe both abstract services (see Figure 72) and their materialization as concrete services or as flexible (that is, adaptable to the run-time requirements) WS-Diamond services. WSDL extensions are needed in order to address abstract service as well as their instantiations.

In Figure 73, we consider that services have a functional and non functional description and that they are associated with providers, users (through a delivery channel). A service can be in the status of “required”; supposing a concrete service has been found on the WS-Diamond network with best matching characteristics, such concrete service has a public view (published) and a private view (internal and manageable).

Accordingly, the WSDL extensions are required to describe the public view on the service process including constraints on parameters, constraints on possible ordering and mandatory order of operations, and faults. To describe the private view of process, language constructs must be available to express the quality and local and global constraints, as well as private faults and compensations actions (derived from BPEL and workflow literature).

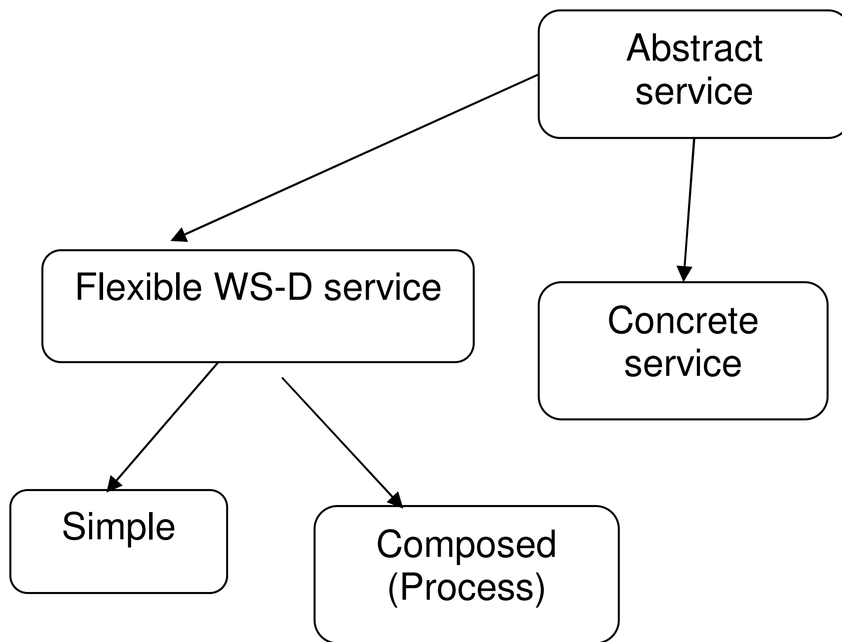


Figure 72 :Abstract and instantiated services

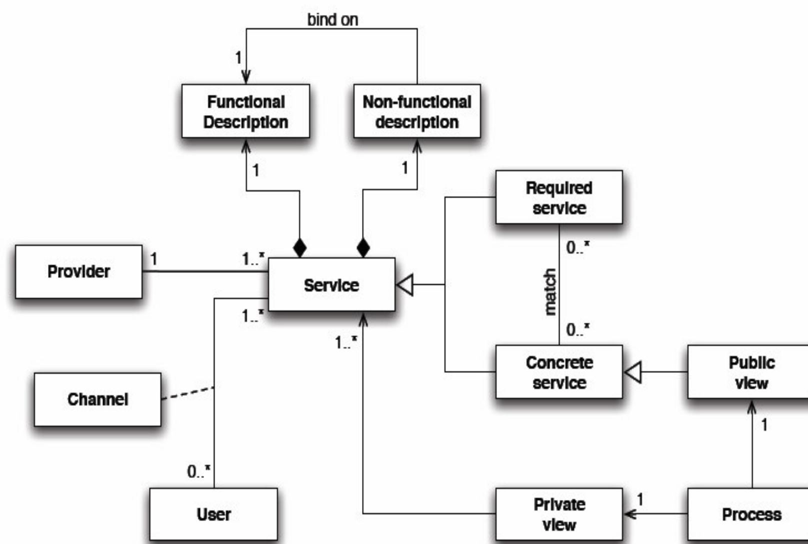


Figure 73 : Schema of services environment

In order to define the Service Management Interface (see Section 5, General Architecture of one WS-Diamond node), the Description Language should provide the ability to perform actions to manage the service, such as to activate the service, enquiry about its status, modify its parameters, and so on. Referring to the example in Figure 74, showing a Travel-Service with its operations, a

Management Interface provides operations such as State-request/control, Negotiation (to be able to contract values of parameters e.g., for QoS), or to mange events through a publish&subscribe mechanism. The QoS characteristics can be described according to a WS-Policy notation, while the Service behaviour is specified as a workflow of steps.

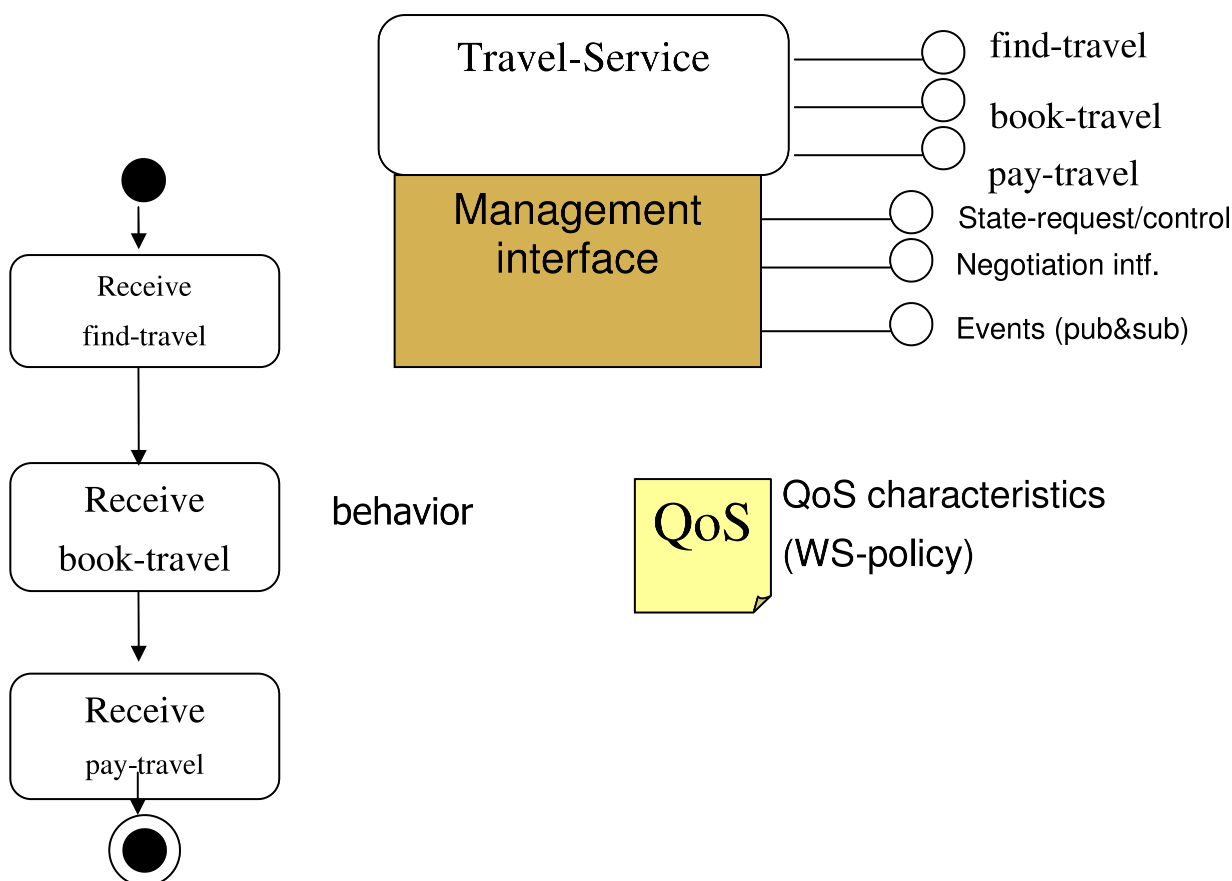


Figure 74 : A Travel-Service example, its Management Interface, process flow, and annotation for Quality

In particular, a *Negotiation* phase should be allowed for the services and their composition. To such aim, the language must allow the description of negotiation parameters and protocols. Reference to negotiation handler modules should also be possible. Participation to the auctions is delegated to negotiation handlers in order to separate business (functional) logic from negotiation logic. Moreover agents may be reused by describing them as web services (see Figure 75). Negotiation capabilities extensions are needed. For Negotiator implementation, a WS-Coordination for negotiation Process should be specifiable.

A Negotiator has two roles: Broker of messages among participants and Controller of protocol compliance.

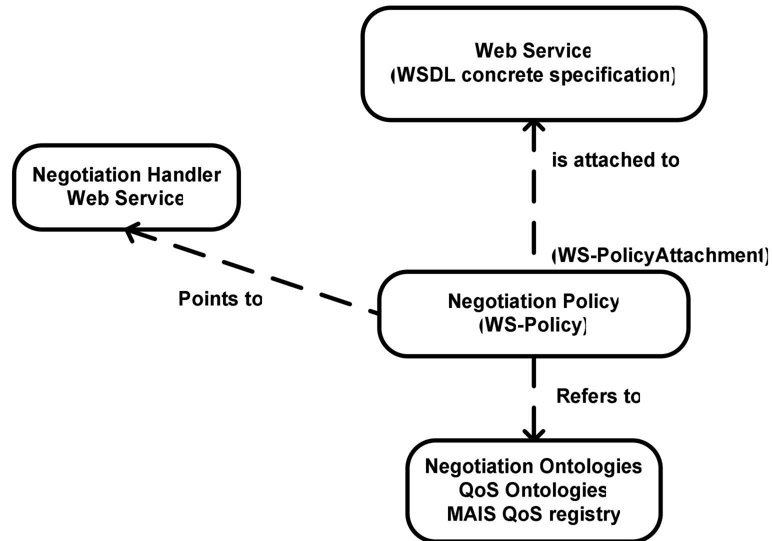


Figure 75 : Negotiation Handlers

For Composed services, an internal view has to be defined composed services (orchestrated in a process).

The *Service request language* constructs should allow expressing User preferences (context dependent) as shown in the following fragment:

```
<User name = "XXX">
        <avTimeWeight value = "1">
        </avTimeWeight>
        <priceWeight value ="0">
        </priceWeight>
        <reputationWeight value = "0">
        </reputationWeight>
        <availabilityWeight value  = "0">
        </availabilityWeight>
        <dataQualityWeight value = "0">
        </dataQualityWeight>
        <restarTime value = "0.5">
        </restarTime>
        <avTimeConstraint value = "2000">
        </avTimeConstraint>
        <priceConstraint value ="15">
        </priceConstraint>
        <reputationConstraint value = "0.0005">
        </reputationConstraint>
        <availabilityConstraint value  = "0.00000001">        </availabilityConstraint>
        <dataQualityConstraint value = "0.01">
```

```
        </dataQualityConstraint>
        <degDurationConstraint value = "20000000">
        </degDurationConstraint>
</User>
```

Other management issues regard the Context State, Contracting, Monitoring, and Certification. In particular, for contract specification (in WSLA language, or WS-Agreement), the following capabilities should be supported:

- Describe service elements touched by the contract (portTypes, operations,…)
- Report guarantees, recovery actions, penalties,…
- Identify and monitoring third parties.

Related open problems are the aspect of contract parameter and service QoS specification, QoS ontologies, Service composition, the relation between internal and external QoS, contract enactment/enforcement frameworks, to provide a description of the module interfaces (manager service, event notification format,…). Instead rules for recovery actions and mechanisms for their enforcement should not be provided, since they are left to domain specific applications

Contract violation is itself a fault; moreover, recovery actions should be included in contract specification, while composition specification should be considered the nominal behaviour of the system to be monitored (diagnosis). In classical approach, the contract manager enforces the recovery actions execution. In the WS-Diamond approach, self-healing web services do not need a centralized service manager.

4.1.2 Semantic annotations

The following s*emantic annotations to the BPEL specification* need be defined:

- *probability of execution of conditional branches*: for every switch *s*, the probability of execution $\{p^s_1, p^s_2, \ldots, p^s_{NBs}\}$ of conditional branches is specified ($sum_{i=1}^{NBs}\, p^s_i=1$, NB^s indicates the number of disjoint branch conditions of *s*)
- *loop constraints*: the expected maximum number of iteration NI^l is defined for every loop *l*; the probability distribution $\{p^l_0, p^l_1, \ldots, p^l_{NIl}\}$ of the loop number of iterations ($sum_{i=0}^{NIl}\, p^l_i=1$) is specified ($p^l_0$ indicates the probability that the loop is not executed, p_1^l indicates the probability that the loop is executed once, and so on)
- *global and local constraints on quality dimensions*: global constraints specify requirements at process level, while local constraints define quality of Web services to be invoked for a given task in the process. We assume that quality constraints may be defined on a set of *N* pre-defined quality dimensions q_n. Furthermore, local constraints can limit the set of Service Provider which can support the execution of an abstract service.
- *Web service dependency constraints*: impose that a given set of operations in the process is executed by the same Web service. This type of constraint allows considering both stateless and stateful Web services in composite services
- *user preferences*: a set of normalized weights $\{w_1, w_2, \ldots, w_N\}$, $sum_{n=1}^{N}\, w_n=1$, indicating the end user preferences with respect to the set of quality dimensions

The probability of execution of conditional branches and the distribution of loops number of iterations can be evaluated from past executions by inspecting system logs or can be specified by the composite service designer. We assume that for every loop *l* an upper bound NI^l for the loop number of iterations is determined. Otherwise, if an upper bound does not exist, the

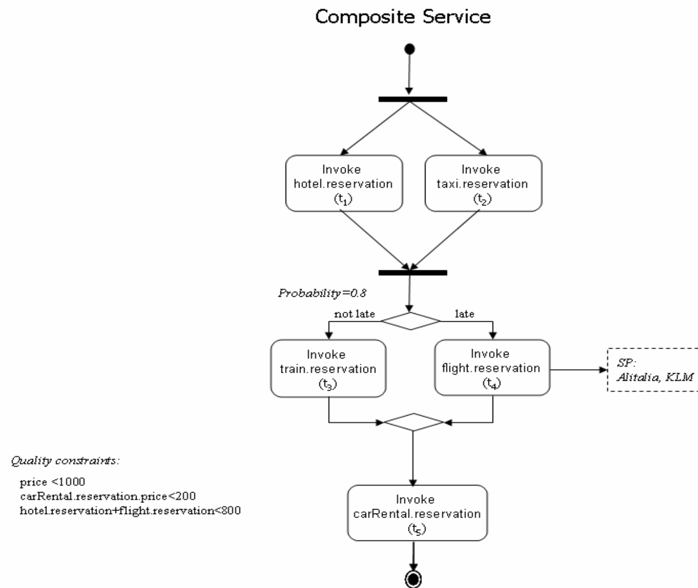
process cannot be optimized since infinite resources might be needed for its execution and global constraints cannot be guaranteed.

Local constraints can be specified for every task by the composed service designer. The end-user can specify local constraints but only on the public view of the composite service.

Web service dependency constraints are specified by the composite service designer. User references may be either specified explicitly by the user requesting the service, or can be implicit in the user profile and therefore the same for all service requests, or not specified at all. In the last case, all quality dimensions are considered at the same level of preference giving each dimension a weight 1/N.

Figure 76 shows a virtual travel agency example. To introduce switch and loop probability distribution they have to be labelled by introducing the BPEL <*name*> standard attribute.In this example, a local constraint is introduced on the carRental invocation price and local constraint limits the set of Service Provider for the flightReservation task. Finally, a global constraints guarantees that the price of the overall process is lower than 1000 and a global constraint entails that the price of the reservations of the hotel and the flight is lower than 200.

In the example in Figure 77, constraints for multiple channels are reported. Different global constraints, local constraints, and user preferences can be associated with different channels in the user profile. Furthermore, the example introduces Web service dependencies constraints which entail that the set of task t_1, t_2, and t_3 will be executed by the same concrete service as task t_4 and the task invoked in the while loop.



travelAgency.bpel

```
<process>
  <sequence>
     <flow>
         <invoke hotel.reservation>
         </invoke>
         <invoke taxi.reservation>
         </invoke>
     </flow>
     <switch name="checkPoint">
         <case condition=isLate>
              <invoke flight.reservation>
              </invoke>
         <otherwise>
              <invoke train.reservation>
              </invoke>
         </otherwise>
         </case>
     </switch>
     <invoke carRental.reservation>
     </invoke>
  </sequence>
<process>
```

annotation.xml

```
<annotation>
    <switch name=checkPoint>
         <case condition=isLate probability=0.2>
         </condition>
         <otherwise probability=0.8>
         </otherwise>
    </switch>
</annotation>
```

constraints.xml

```
<UserProfile name=user1>
    <exeTimeWeight value=0.5>
    </exeTimeWeight>
    <reputationWeight value=0.5>
    </reputationWeight>
    <priceConstraint value=1000>
    </priceConstraint>
    <carRentalReservationPriceConstraint
         value=200>
    </trainReservationPriceConstraint>
    <flightReservationProvider
         value=Alitalia,KLM>
    </flightReservationProvider>
    <globalConstraint>
        <WSInvocations>
        <invocation>hotelReservation</invocation>
        <invocation>flightReservation</invocation>
        </WSInvocations>
        <qualityDimension>price</qualityDimension>
        <operator>="<"</operator>
        <value> 800 </value>
    <globalConstraint>
</UserProfile>
```

Figure 76 : A Virtual Travel Agency Example

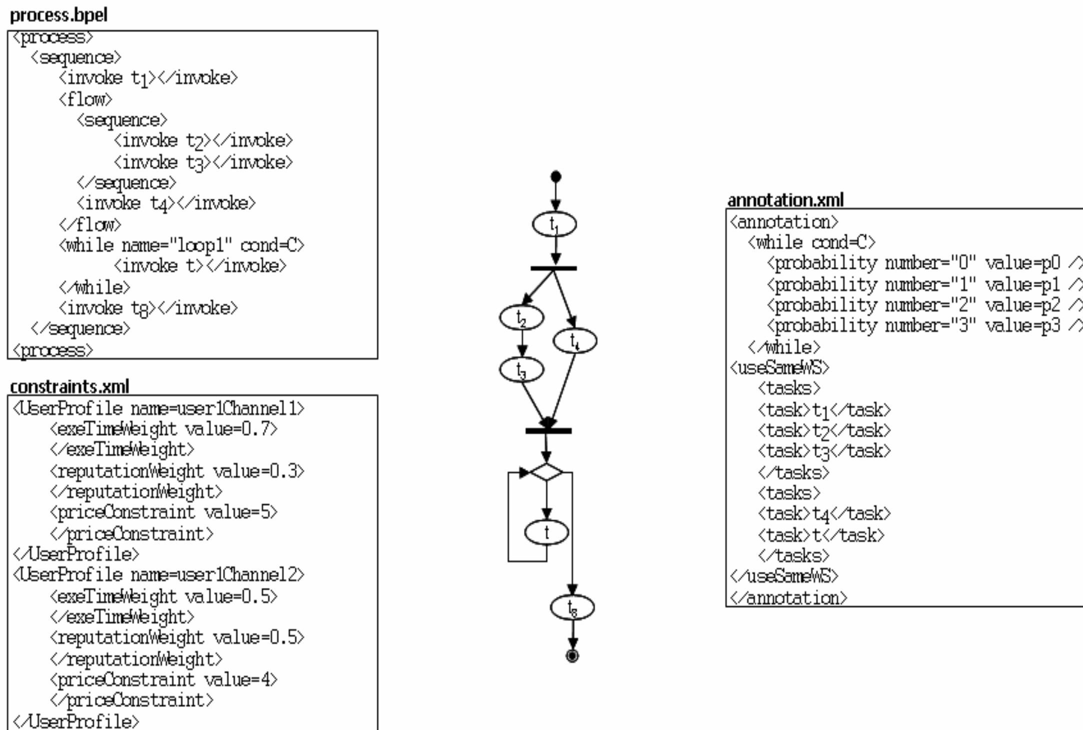


Figure 77 : A Multi-Channel Constrained Example

4.2 Requirements for model-based Diagnosis and repair of Cooperative Web Services

This set of requirements is related to the work package 4 whose goal is to be able to develop a surveillance platform decicated to web services in order to guarantee that the global behaviour satisfies the requirements, i.e that the services are available and reliable and that a given Quality of Service (QoS) is fulfilled. It includes:

- detecting abnormal situations that compromise the quality of service (QoS),

- diagnosing the primary causes of those deficiencies, which can be identified as a faulty component, a bad communication between two services, an unsuitable configuration of the network ...

- doing recovery in the best possible way, by reacting instantaneously and adapting the planned interactions, or by deciding to replacing a faulty component, modifying communication parameters or reconfiguring the network.

In order to do so, model-based techniques will be used. The challenge is to apply recent results and techniques developed for monitoring, diagnosing and reconfiguring complex physical systems to web services networks.

4.2.1 Current trends in model-based diagnosis and repair.

Research on model-based diagnosis developed since the mid 70's and led to several new methodologies, solutions and applications, mainly applied to static systems. Those systems, with a

unique non-changing state, supposed instantaneous observations, fault effects visible in the diagnosis window and no evolution of the system in this window. Associated methods resulted in timeless descriptions of the systems.

Diagnosis progressively stopped being considered as a subsidiary activity, performed off-line. The current trend of research, starting from the 90', concerns complex systems and aims at integrating monitoring, on-line diagnosis and repair in dynamic systems. The idea is to detect, as soon as possible, every discrepancy between the real behavior of the system and the expected one, and to react as soon as possible by selecting the repair action to perform, in order to get the expected system behavior. It allows to alternate diagnosis and repair phases. Taking into account repair actions often leads to a planning problem [FG92], [SUW93], [NEB93]. A more recent trend is to focus on reconfigurable systems whose structure can evolve, for instance after a reconfiguration repair decision.

In order to model such complex and dynamical systems, it has been proposed to represent them as discrete-event systems that can be formalized by transition systems. Many formalisms have been proposed such as automata, Petri nets and Markov model, process algebra ... More recently, decentralized and distributed diagnosis approaches [PEM02], [LAZ03], [PR02] needed for large systems, have been developed. In this case, diagnosis is performed both in a decentralized way (at subcomponent level) and on the overall system (e.g. making decentralized diagnosers cooperate). Still more recently, an extension from distributed systems to multi-agent ones has been explored [ROB02] and a model-based approach was applied to on-line monitoring and diagnosis of multi-agent systems [MIC04].

As said before, our challenge in this project is to apply these techniques recently developed for monitoring, diagnosing and reconfiguring complex physical systems to web services networks.

4.2.2 Model-based diagnosis and repair faced to Web Services

For a long time, diagnosis was only applied to physical systems for which system components and artifact parts were homologous. Progressively, the focus moved from traditional application fields to new fields such as economical systems, software, communication networks. Particularly significant with respect to this project is the application to software diagnosis in which the same basic technologies have been successfully applied to debug and component-based software.

To exploit the existing techniques to web services networks is a challenging task due to their main characteristics:

1. They are composite networks, and the global quality depends on the individual quality of each service but also of the quality of their interactions. We have thus to take into account individual behavior models for each web service, but the communication models.

2. They are reconfigurable systems, and thus components themselves, their connections, and even communication protocols may change during the process.

3. They are complex distributed systems. Decentralized/distributed diagnosis approaches appear to be well-suited to this kind of networks which are not so far away from reconfigurable telecommunication networks, but a flexible architecture has to be designed so that diagnosis/repair tasks can be locally and globally handled.

4.2.3 Requirements

The logical organization of the diagnosis and repair task is as follows :

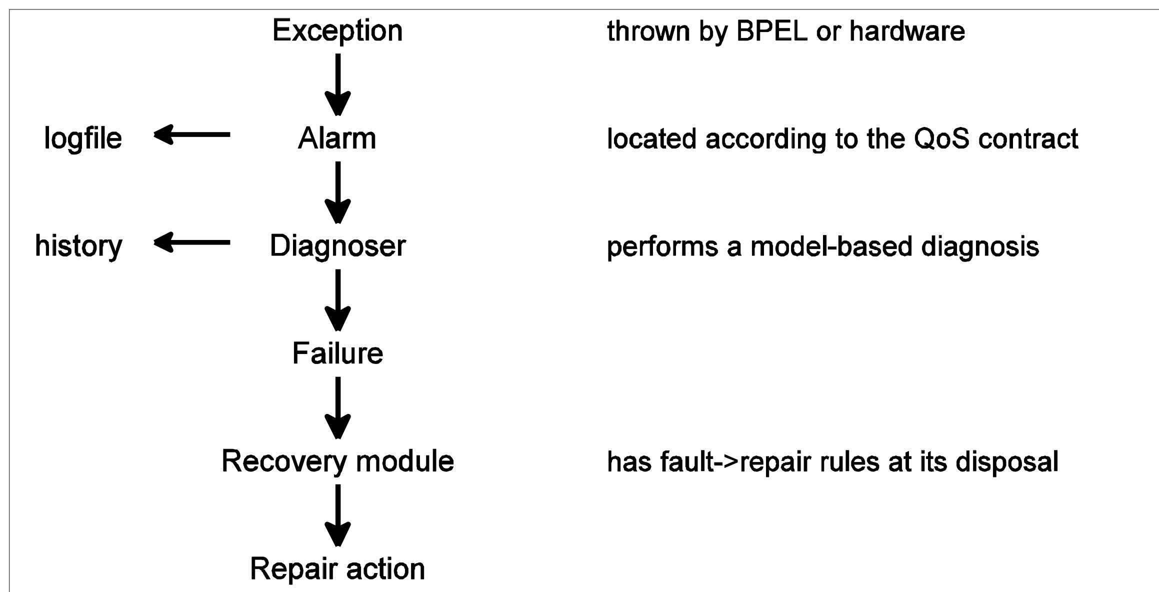


Figure 78 : The logical organization of the diagnosis and repair task

The first step in WP4 will be to characterize the diagnosis and repair problem and for this, it is necessary to identify the symptoms which will trigger the diagnosis task (QoS metering and symptom elaboration), to establish a catalog of all the faults and QoS infractions, and to identify the set of possible repair and reconfiguration actions.

QoS-metering

The main objective of the diagnostic task is to maintain the best level of QoS in the web service network. It means that the required QoS has to be well specified in a "quality contract" (SLA: Service Level Agreement), in order to detect any discrepancy between the current quality and what was expected. This quality contract usually deals with notions indirectly linked with the service itself and not necessarily perceptible by the end user: consider, for instance, quality of service at application, transport or network levels. Then, evaluating the overall QoS (SLM: Service Level Management) will consist in evaluating operators such as packet loss, bandwidth (in streaming or burst modes), latency (transport delay) or jitter (latency variations), but also operators defined for a single web service.

QoS parameters can be classified into the following categories: the generic QoS parameters such as availability, security, response time, etc.; the application-specific QoS parameters as the input-output mismatches between the requester and the provider or a conversational mismatch as a time delay violation.

As a matter of fact, each service is designed for a precise function on which one can rely in order to describe locally the quality constraints that have to be respected so as to offer an optimal service. A question which has to be debated is the language which will be used to describe such a quality contract and its link with semantic annotations.

Symptoms/alarms elaboration

As mentioned above, each web service must be provided with a quality contract expressed in the shape of constraints. Violating one of those constraints will result in triggering an alarm. The alarms will be redirected to the diagnoser and/or to a database in order to be treated as effectively as possible. So we will have to elaborate high-level alarms (or symptoms) which can be processed by the diagnoser.

These alarms can be classified according to their corresponding occurrence level. We can distinguish infrastructure and middleware alarms due to failures in the underlying infrastructure (hardware, network); web service alarms due to failures in service invocation and orchestration (they are mainly application-specific alarms) and web application alarms due to data mismatches, actor faults, coordination or choreography failures.

A large part of these alarms are directly related to the QoS-metering task as they can be expressed as quality contract violations.

Symptoms related to the Food Shop case could be, for instance, the fact that:

(a) the CUSTomer realizes that some ordered item is wrong,

(b) when assembling the package, the WAREHOUSE realizes that it received a

wrong item from one of the SUPPLIERs,

(c) when computing the bill, the SHOP realizes that the ship cost sent by the

WAREHOUSE is higher than the expected threshold.

A question concerning the alarms is the transmission language and method to use. In fact, BPEL already allows to "transport" exceptions, and a solution could be to enrich the BPEL exception language in order to make it able of treating part of the alarms. Another would be to define an ad-hoc transmission language adapted to this diagnostic mechanism.

Faults types

We will have to consider two kinds of failures: the *individual web service related* failures, and the *choreography related* failures.

The diagnoser task is to identify failures from a set of symptoms and to find the primary causes responsible of these symptoms. To facilitate this identification, we can classify these failures as physical faults (connection loss, network cut, ...), programming faults (bad communication protocols, ...), human or i/o faults (bad conversions, syntactical mismatch, data cut down, ...). This failures classification is directly related to the alarms classification as the latter ones can be seen as observable manifestations of the former ones.

Considering once again the Food Shop example, it is possible to determine the failures that fathered the symptoms in the previous section:

(a) the WAREHOUSE or one of its SUPPLIERs reserved the wrong item,

(b) the SUPPLIER reserved the correct item but made a mistake updating its

internal order database, or the SUPPLIER did everything correct but sent

the wrong parcel to the WAREHOUSE,

(c) the SHOP selected the wrong WAREHOUSE, or the WAREHOUSE itself made a

mistake in computing the ship cost.

Model and algorithms

The model-based approaches relies on comparing expectations and observations. They rely on a model which represents in a less or more abstracted way the system behaviour. It can be restricted to the normal expected behaviour or includes the faulty behaviour. The model design will be one of the important task in the following. We first have to decide the kind of model we plan to use (level of abstraction, formalism), to build it and to think about a way of giving means to assist its acquisition. We will have to distinguish the component model and the conversational model describing the information exchange between the components. These models are directly related to the description languages used to describe the web services themselves and their communication

protocols. For instance, this diagnosis model is clearly related to the behavioural description at the component level for which dedicated languages exist (as BPEL). In the same way, the conversation model is certainly related to the orchestration and choreography languages. However, diagnosis specific information has probably to be added to allow a precise diagnostic task.

Algorithms will be then specified according to the characterisation of diagnosis, the chosen architecture, and the model development. The general architecture has first to be decided (see Chapter 5 for a discussion on this point).

Repair and reconfiguration actions

The definition of repair and reconfiguration actions requires a representation language which allows the definition of requirements for composite Web services. These requirements specify if an instantiation of the overall process of interacting Web services is regarded as correct and complete. Such process requirements have to express functional as well as quality of service requirements including time constraints. Note, that these process requirements may depend on the participating actors (e.g. customers or shops may formulate different requirements).

Beside process requirements a description of possible changes of the overall process must exist. These changes may comprise exchange of services, re-invoking of services, messages to services (e.g. a request for compensation), structural changes of workflows etc.

The repair and reconfiguration process has to operate on the current state of the composite Web service process as well as on the planed future process steps. Consequently, appropriate representations methods for processes and their states are required.

Changes of processes may be associated with costs. Hence, there must be the possibility of assigning costs to changes and the ability to rank competing reconfigurations and repairs of processes. In addition to costs we may require the specification of additional attributes which are associated to Web services and their composition (e.g. the degree of trust in a Web service).

The diagnosis process usually outputs a set of possible diagnoses which explain faulty behaviour. In order to compute the “best next” repair and reconfiguration actions, the likelihoods of these diagnoses have to be computed.

Regarding the architecture, we currently do not impose any requirements. Consequently, both centralized and de-centralized approaches for repair and reconfiguration may be explored.

Finally, all the above mentioned descriptions must be easy to formulate since they have to replace explicit exception handling. In addition, repair and reconfiguration has to be performed online which implies satisfying running time behaviour.

4.2.3.1 Requirements for Enrichment of (Semantic) Web Service Description Languages

With respect to the different types of faults as listed in table 1, different levels of fault detection make different extensions to existing web service and semantic web service description standards necessary.

- Internal data faults: We have to distinguish between semantic mismatches and other data faults. To detect semantic mismatches, the web service description needs to be annotated with semantic datatypes. It must be possible to compare the semantic datatype of the data exchanged at runtime with the semantic datatype as in the service description. For fault detection at this level, it is not necessary to have a description of the workflow or interaction model. Existing semantic web service languages like WSDL-S or OWL-S are suitable for this task and need not be enhanced further. Other data faults include the violation of implicit constraints. The description of parameters is often underspecified. For example, if we specify that a certain

numerical value is meant to be a book price, the fact that the number must be positive is often omitted. In order to detect data faults efficiently, it is therefore necessary to make these constraints explicit. We propose to use (semi-) automated learning techniques to address this problem.

- Application coordination faults: To detect application coordination faults, it is necessary to have a description of the workflow and the interaction model. While existing languages suitable for describing workflows such as BPEL4WS or OWL-S are sufficient, it is desirable, with respect to diagnosis, to extend the workflow description with hierarchical information. OWL-S provides with the SimpleProcess construct a way how to encapsulate more complex workflows into a black box. Such a construct similar to the definition of subroutines should be introduced in the process language used in DIAMOND to allow for hierarchical decomposition.
- Actor faults: With respect to the requirements of the description language an actor faults are equivalent to application coordination faults.
- Quality of Service violation fault: To correctly detect quality of service violation faults it is required that the description language contains definitions of quality of service for both a single service as well as a complex workflow. Some WS-* standards as well as WSOL address the issue of quality of service. Semantic Web Service Languages also provide slots for quality of service information, this is the subject of extensive research (e.g. Cardoso 2002). The description languages used in DIAMOND must support quality of service annotations.

4.2.3.2 Underspecified Descriptions

As opposed to the (usually automated) generation of WSDL descriptions, any additional formal description of a web service and a workflow process means extra work for the software engineer or programmer without a direct feedback. This has two implications for DIAMOND: First, good tool support for the creation of any additional metadata is needed. This issue is partially addressed by the development of semi-automated tools for the acquisition of markup. The second implication is that – also when created with these semi-automated tools – this additional metadata will sometimes be incomplete. It is therefore a requirement that the diagnosis algorithms must be able to deal with possibly underspecified descriptions. As described above, to diagnose different classes of faults, different metadata is needed. Depending on the metadata actually present it is required that the diagnosis algorithm at least provides a fall-back solution (on a different level), if the metadata is underspecified.

[/VU – 2 Feb.]

4.3 Requirements for design for diagnosability and repairability

Diagnosability and repairability analysis is part of the design stage and is based on models of the system. Diagnosability analysis provides information about the classes of faulty behavior of the system that can be diagnosed, which is a mandatory step in self-healing system design. Repairability analysis aims at classifying the fault situations from which the system can recover. Both properties can be evaluated at design time and may be involved in the software validation criteria.

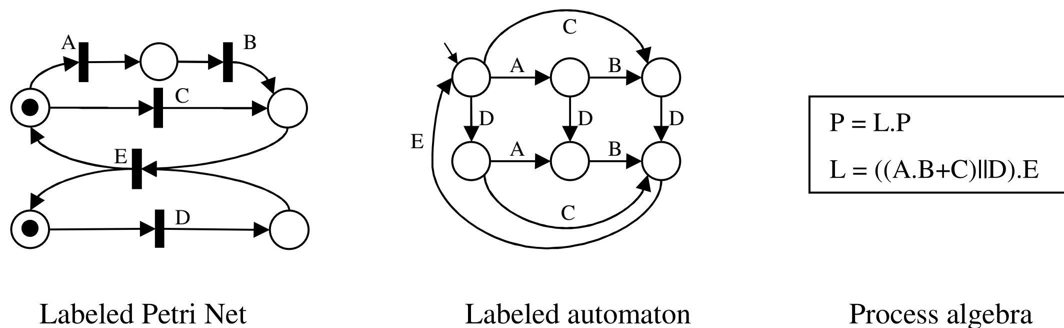
4.3.1 Models for diagnosability and repairability

Diagnosability properties for Web services are related to diagnosability properties on discrete-event systems. Their analysis is based on a structural and behavioral model of the system which is

usually represented as a set of event-driven and interacting components and which includes an observable model (what is observable or not). In the following, we present a set of classical formalisms for discrete event systems that are used in the literature before introducing several notions of diagnosability and reparability.

4.3.1.1 Formalism for discrete-event systems

The most classical formalisms for discrete event systems are Petri nets and automata. Both of them involve the notions of states and transitions. By associating events to transitions, labeled transition systems or labeled Petri nets are built. Process algebra is also useful to model discrete event systems and has been used for diagnosability checking.



Labeled Petri Net Labeled automaton Process algebra

Figure 79 : Petri net, automaton, and process algebra models of a simple system

When modeling distributed and/or very complex systems, considering the system as a set of communicating components is necessary; in this case, each component is modeled separately. In the case of Petri nets, communication is modeled by common places or transitions between several components. The global model of the whole system is obtained by merging the different nets over the shared places and transitions. In the case of automata, communication is modeled by common (communication) events. The global model is obtained by applying synchronous product to all the system's components. In process algebra, communications are modeled by synchronized processes. The component oriented approach is clearly preferred when modeling web services. A design framework should allow specifying the workflow for only one web service, independently from other web services.

We distinguish two types of models.

- *Transition-based models*: the behavior of the system is represented by a set of event sequences (automata, process algebra). Among these events, some are observable, some are faulty. This type of model is commonly used to model *permanent faults* (once a faulty event has occurred, the system cannot recover from the fault) but is also used to model *intermittent faults* (in that case, two events are required at least, one when the fault starts and the other when the fault ends).
- *State-based models:* Another approach consists in considering faults and observations as part of the state (variable states). Transitions are usually labeled with actions or commands that change the state of the system.

From such models, and with respect to the faults and the observability of the system, it is possible to express diagnosability in a formal way, and implement methods to analyze it.

4.3.1.2 Notions of diagnosability

The notion of diagnosability gathers a set of properties that define the diagnosis quality that could be expected from a monitor of the system. Depending on the knowledge and the class of the monitored system, several diagnosability levels can be investigated.

- *Detectability:* if the system verifies detectability properties, then the observable behavior is sufficient enough to detect a faulty behavior of the system. In other words, the monitor is able to detect if the system is faulty or not with a finite number of observations.
- *Isolability:* this property means that not only the monitor is able to detect whether a system is faulty or not but also it can locate the component where the fault has occurred in a finite time. If the system contains only one component, isolability is equivalent to detectability.
- *Identifiability:* a fault of a given type is identifiable if the monitor is able to decide after a finite set of observations that a fault of this type has necessarily happened or not. The system is said to be identifiable if every anticipated fault is identifiable.

In the case of a single fault type per component, identifiability is equivalent to isolability. In the case of one single fault type for the whole system, identifiability, isolability and detectability are equivalent. Therefore, the property that is generally considered in the literature is diagnosability, and is brought back to identifiability, isolability or detectability given the set of fault types that is anticipated.

The different faults that may occur in the system are generally gathered in a set of fault types. The goal of diagnosis is to assess the type of the fault that occurred. This allows the designer to consider some faults that do not need to be discriminated (i.e. considered as different in diagnosis). Two faults are said to be *discriminable* when their belonging to different fault types keeps the system diagnosable.

Besides the different diagnosability levels, two different definitions are used: strong and weak diagnosability. Strong diagnosability is a theoretical definition describing the concept, but experience has led designers to consider a more tolerant definition, weak diagnosability, which is more relevant from a practical point of view.

- *Strong diagnosability:* a system is strongly diagnosable when any fault occurrence is necessarily followed by observables allowing the monitor to detect/isolate/identify the fault.
- *Weak diagnosability:* a system is weakly diagnosable when, after any fault occurrence, it is possible to make the system generate observables allowing the monitor to detect/isolate/identify the fault.

4.3.1.3 Notions of reparability

To our knowledge, very few works have been published about repairability in discrete event systems. This subsection presents interesting approaches as a basis for the study of reconfiguration and repair capabilities in Web Services.

Different approaches [FG92],[CRR91],[GRC04] have been explored to model configurations and reconfigurations. The reparability notion is generally related to safety: a repair action may not be safe to perform in any state of the system.

Repairability relies on the possibility to perform some repair actions after the occurrence of a fault. A system may not be reparable after a fault occurrence for several reasons:

- No repair action can be provided to get the system back to work
- The system is in a state in which no repair actions can be performed safely, and cannot evolve to a state allowing repair actions.

Basically, it is possible to distinguish two kinds of repair actions.

- Functional repairs consist in changing input parameter values for some components
- Structural repairs consist in modifying the interconnections between the system components.

In the context of web services, repair actions may consist in:

- Substituting the faulty component with one or several equivalent one(s);
- Offering reduced services (degraded mode) (and adding new observables?);
- Debug the logical part of the system or repair the underlying physical service;

Some repair actions may be followed by changes in some of the system's parameter values, e.g. the type of service, or the quality of service.

Diagnosability levels and repair types are strongly correlated. The decidability of repair actions relies on the diagnosability level, and conversely: a low diagnosability level may limit the possibility to implement some types of repairs. Relationships between repair type and diagnosability are illustrated in Table 5.

Table 5 : Correspondences between diagnosability levels and repair actions

Repair type	**Diagnosability level**
Degraded mode	Detectability
Component substitution	Isolability
Revision	Identifiability

A web service entering degraded mode should always be detected, both by the provider and the consumers, as dependent services may suffer from the service degradation. Moreover, when a component only ensures fault detectability, the best repair that could be performed is trying to keep offering the service even with some degradation.

In order to replace a faulty component with one or more equivalent ones, it is necessary to determine which component to replace; this repair action thus requires isolability. On the other hand, when some faults in a component are not discriminable but the component is isolatable, the best repair action that can be decided is to replace the faulty component.

Finally, in order to perform an efficient revision, it is necessary to have the best information about the fault. This is what corresponds to identifiability for the diagnosis module.

Internal and external perspectives:

In an external perspective (the monitor does not own the web service(s)), revision is not allowed for confidentiality reasons. The diagnosis ideal goal consists in isolating the fault in the external web service. This allows substituting the faulty web service by another one, or relying on its degraded service if it offers one.

In an internal perspective (the monitor owns the web service(s)), the designer aims at complying with the security policy, and the ideal diagnosis goal is to achieve the finest identifiability.

4.3.2 Architecture of the supervision system, impacts on diagnosability and repairability

Web Services are distributed systems and several supervision architectures for diagnosis/repair may be deployed.

- *Centralized supervision:* a unique entity is in charge of computing a global diagnosis from the observations and deciding repair actions on the system. In a centralized architecture, the flow of observations is global and the set of faults to diagnose is the set of faults of the system. In this context, the analysis of diagnosability and reparability is global.
- *Supervision with a coordinated monitoring system:* several entities known as local diagnosers are in charge of diagnosing one or several components. These local diagnosers communicate with a supervising entity that is in charge of performing any diagnosis that local diagnosers may not achieve. There may be several diagnosis levels. The advantage of this architecture is to provide a 'divide and conquer' paradigm (decentralized computations) by maximizing the local diagnosis computations and minimizing the coordination. In order to achieve an optimal architecture, the diagnosability analysis must be performed on subsystems in order to guarantee that some faults can be fully diagnosed locally (a fault is said to be locally diagnosable or decentrally diagnosable) or with a minimal coordination. Given this analysis, it is possible to help in the decision of the placement of the local diagnosers. Local diagnosability is usually a more restrictive property that requires a better observability of the system (a fault may be globally diagnosable but not locally diagnosable). Reparability analysis follows the same schema as diagnosability.
- *Distributed supervision:* as in the previous architecture, local diagnosers/supervisors diagnose the system's components, but in the distributed architecture, there is no supervising entity. Local diagnosers may communicate between them to reinforce the diagnosis accuracy. From a diagnosability point of view, the analysis is almost the same as in the coordinated monitoring architecture. The difference is just a matter of implementation of the communications between the diagnosers (exchange of messages between diagnosers instead of exchange of messages with a coordinator). As far as the reparability is concerned, the supervision is performed locally in coordination with the other supervisors. In this architecture, an analysis of what can be repair locally or not must be performed.

From the composition point of view, locally diagnosable web services are safer to invoke, as they should facilitate the isolation of faults on external web services and allow safe dynamic web service substitutions.

4.3.3 On-line versus off-line supervision activities for Web Services

The context in which diagnosis and repairs are performed may result in different constraints for diagnosability and repairability. The most important aspect of the context is whether they are run on-line or off-line.

On line: the supervision activity is performed on-line; in this case, the supervision system observes an information flow and must decide repair actions on the fly.

- *Diagnosability for on-line diagnosis* depends on observables acquired and processed on-line.
- *Repairability for on-line repair* relies on the types of faulty situations that can be recovered on-line, i.e. on the subset of actions that can be applied at run time. Repair is then performed without interruption of the service provision.

Off line: the supervision activity is performed off-line; in this case, the supervision system reads observations in a log or journal and decides repair actions with very loose time constraints.

- *Diagnosability for off-line diagnosis* on observables that can be recorded from the system, i.e. from historical/logged data.
- *Repairability for off-line repair* relies on the types of faulty situations that can be recovered off-line, i.e. on the subset of actions that can be applied off-line. Repair involves a service provision interruption.

Mix approaches may also be considered: it is possible to achieve different levels of diagnosability for on-line and off-line diagnosis by storing a part of the observations in a log file, and considered the other subset at run time. Diagnosability and repairability are mutually dependent as shown in Table 6.

Table 6 : Consistent situations for diagnosability and repairability

	Consistent situations			
Detectability	On line	On line	On line	Off line
Isolability / Discriminability	On line	On line	Off line	Off line
Repairability	On line	Off line	Off line	Off line

4.3.4 Design requirements

This section describes our early approach of the design stage for self-healing web services, according to the various notions we presented in the previous sections. It explains how taking diagnosability and repairability into account may involve specific steps in the design procedure.

4.3.4.1 Design procedure

The main input for the designer when analyzing diagnosability and repairability is the safety policy the system needs to comply with. The designer, when considering these properties, is brought to answer several questions:

- What is the criticity of each service?
- Which faults must be considered?
- What is the impact of each fault on the services provisioning?
- Which repair actions must be offered after each fault?
- How fast must each fault be diagnosed?

Answering these questions should lead the designer to choose the supervision architecture, and an on-line/off-line/mix approach. The design procedure may follow the two following scenarios.

- *Safety policy → repairability requirements → diagnosability requirements:* in this scenario, the designer considers the web service safety policy (response time, availability, exactitude …) and deduces the repair actions that must be available in order to comply with the policy. This availability involves some minimum diagnosability levels for the different components.

- *Diagnosability levels → repairability constraints → safety constraints:* in this scenario, the designer is faced to diagnosability limits, which make some repair actions impossible to perform. Some alternatives in this case may be adapting the safety policy to match the reachable specification, or use additional means to ensure safety (duplicate servers, change software platform…).

4.3.4.2 Design framework

Diagnosability and repairability require the use of specific tools, which can be provided by a design framework. This section lists some requirements about the framework for self-healing systems design.

Diagnosability analysis relies on an operation called *projection* on observables, which computes, from the system's model, the model of its observable behavior. A diagnosability oriented design framework should allow the designer to specify projected models. This can simply be done by allowing the designer to define some relations between models, as one model being the observable projection of the other one.

A repair aware framework should allow the designer to describe model transformations, or to relate different models, as one model being the result of a repair action performed on the other model. It should also allow the designer to specify constraints on these transformations, such as state based preconditions and post conditions. This information could be given as input of a model checking entity which would evaluate repairability.

4.4 Requirements for semi-automatic acquisition of semantic markup for Web services

The Semantic Web Services vision of automatic discovery, composition and invocation of web services requires that each service is annotated with semantic metadata. Emerging standards such as OWL-S, WSMO or WSDL-S address this topic. However, in order to facilitate for automatic diagnosis, these annotations are not enough to detect *all* fault types. First, we have to distinguish between:

- Faults that can be detected without semantic annotations
- Faults that can be detected with semantic annotations
- Faults that cannot be detected with annotations using existing standards and require extended annotations

Furthermore, we also have to distinguish between:

- Annotations that describe the functional properties of a single service
- Annotations that describe the non-functional properties of a single service
- Annotations that describe a composition of services

To learn annotations, we have to distinguish between three sources of information:

- Static information such as existing descriptions in WSDL
- Information that is gathered while a single service is being executed
- Information that is gathered while a composition of services is being executed

Furthermore, we have to distinguish between supervised learning that requires training data that is annotated by humans or unsupervised learning, also known as clustering, that does not require a human in the loop and arranges the data in meaningful groups based on a proximity measure. For the purpose of annotation it is clear that clustering performs less accurate than supervised methods.

Previous work has addressed how to learn abstract functional properties (a) from static information (i.) in both supervised and unsupervised setups, and messages that are exchanged while one service is being executed (ii.).

4.4.1 Gathering functional properties from static information

While gathering functional properties from static information can be seen as a special form of schema matching, web service annotation as a special problem has also already been discussed in literature.

The ASSAM tool addresses the problem of annotating web services with semantic information from ontology by assigning classes or properties in an ontology to operations and parts in a web service. ASSAM treats web service annotation as text classification. Text samples are drawn from identifiers and comments in the WSDL file and a machine learning algorithm is trained to classify these texts. Furthermore, ASSAM uses structural information from the web service to improve classification accuracy.

We propose to develop ASSAM further with respect to the requirements in DIAMOND.

4.4.2 Gathering functional properties from executing a single service

While ASSAM gathers functional properties from the (static) description of a service only, it ignores a valuable source of information, namely the concrete data that is passed between services.

The OATS algorithm makes use of this information. It uses string matching to aggregate the output of several web services that have been invoked with the same input. While it is assumed that mappings for the input parameters of the services exist (this includes conversion between different formats or even semantic transformations such as unit conversions), the output values are matched in order to identify equal output parameters.

For the detection of data faults (see above) it is also a requirement that we make implicit constraints on the data explicit. Knowing the semantic datatype of a parameter is necessary, but not sufficient. Learning constraints on the data is a requirement for DIAMOND. We propose to enhance OATS or to devise a new algorithm with similar intuitions with respect to the requirements imposed by DIAMOND.

4.4.3 Gathering data from executing a composition of services

Both ASSAM and OATS address the problem of generating metadata to facilitate for matching on the level of (semantic) datatypes, all aspects regarding workflow are ignored. While such semantic markup is necessary and also sufficient to detect certain classes of faults (see above), metadata on both workflow but also quality of service aspects is very important in DIAMOND. Therefore, we propose to explore new ways how to acquire metadata while a composition of services is being executed. To facilitate for algorithms to harvest data that is being exchanged, it is required that the messages sent are being logged to be processed offline. An online processing of sent messages is deemed both unnecessary and unrealistic for two reasons: First, it is unclear how the messages

could be processed in such a way that metadata could be derived that is of immediate use for the execution of the service. Only in this case, however, is an online-processing useful. If the metadata that is being acquired should be used in a later invocation of the service (composition), there is no need to process messages directly when they are sent. Second, the sent messages will – depending on the desired level of metadata – have to be annotated. We propose three levels that require different amounts of annotations:

- Learn Quality of Service constraints: We propose to learn quality of service annotations from message logs. To accomplish this, the logs will have to be annotated with a flag that signs whether the invocation was successful and within acceptable limits with respect to the desired service level or whether the quality of service fell short of expectations. The quality of service constraints can then be directly learned from the annotated logs.
- Learn abnormality: We propose to learn a classifier that can detect whether an invocation of a web service is abnormal or not. For this task, we have to distinguish between two cases: First, if the logs of invocations that serve as training data contain only positive examples (i.e. where the invocation was successful), an additional requirement for the learning algorithm is that it must be able to learn a classifier from positive examples only. (Muggleton 1996, Bostrom 1998) Second, if the logs contain both positive and negative example (i.e. where the invocation failed), the logs have to be annotated at least with a flag. It is desirable for the classifier that its internal model can be exported in the form of rules that can be embedded in the web service description.
- Learn cause directly: It might also be possible to learn classifier that can predict the cause of a fault directly. For this level of fault detection it is required that the message logs are not only annotated with a binary ”correct/fault”-flag but also with the root cause. Learning on this level is considerably harder than just learning abnormalities or quality of service constraints. We regard this as future work.

5 Preliminary architecture

The architectural aspects are illustrated in the following with respect to the following dimensions:

- cooperation among distributed nodes
- a proposal for an architecture to link diagnosis and repair actions
- centralized and distributed diagnosis architectures

Cooperation among distributed nodes

As shown in the following Figure 80, we envision a global self-healing system in which WS-Diamond nodes can cooperate with non WS-Diamond node. In addition, each WS-Diamond node in the distributed architecture might provide only a subset of the modules developed in the project.

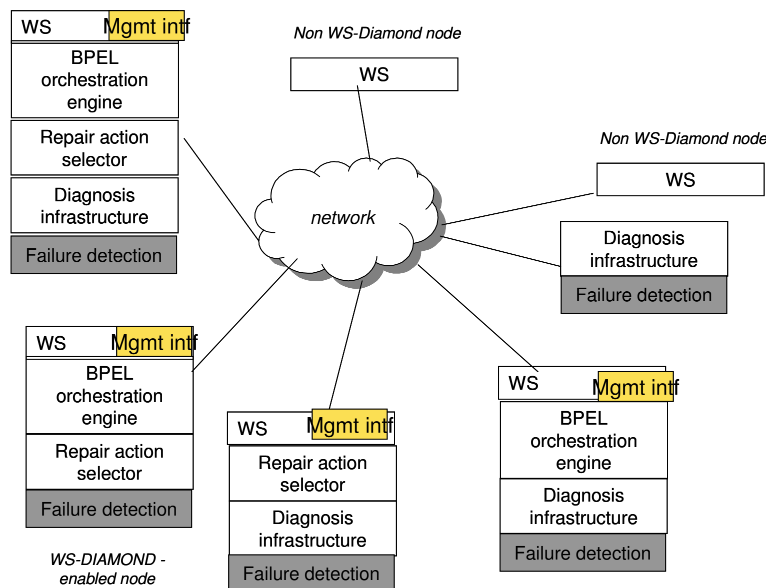


Figure 80 : WS-Diamond environment

We assume that a WS-Diamond node may provide self-healing web services or only an infrastructural support to the self-healing system.

The main modules of a complete WS-Diamond node are:

- a management interface for web services
- a process orchestration engine for enacting composed services
- a repair action selector
- a diagnosis infrastructure
- a fault detection infrastructure (the development of which will be out of the scope of the project)

The following Figure 81 illustrates how the different modules cooperate inside a node.

The diagnoser will be notifies of fault events through messages or events generated by the hardware/software infrastructure (including the self-healing system itself). The diagnoser identifies which fault occurred and needs to be recovered. All fault events are stored in a fault log. The Recovery action selector performs a choice among a set of possible recovery actions associated to each type of fault as indicated in a fault registry. The selection triggers a recovery action request to a recovery module associated to the required action.

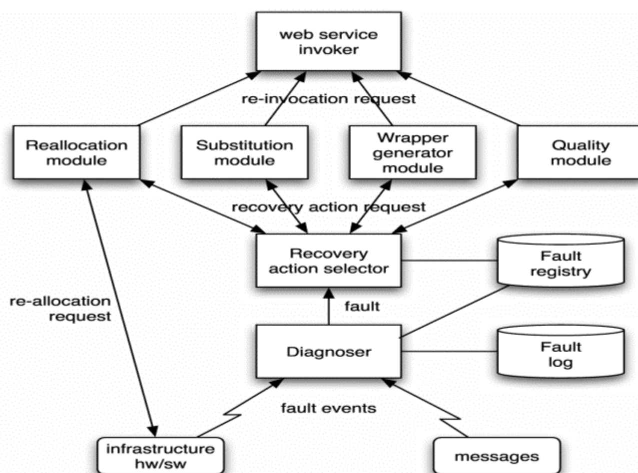


Figure 81 : Cooperation of modules.

A list of such modules (not all-inclusive) is provided in the Figure 81: substitution to replace services during the orchestration of a composed service, wrapper generation to change parameters to solve incompatibility problems during invocation, quality module to perform data quality checks and improvements (e.g. correct typos or incorrect codings), reallocation module to change allocation of resources to services.

5.1 Diagnosis architectures

Defining a diagnosis architecture raises a set of important questions:

- What is the nature of a diagnoser?
- What is its relation with the Web service (coupled or decoupled design)?
- What is its relation with partner diagnosers?
- Is it a service offered by the execution environment?
- Is it implicitly or explicitly handled during the service design?

The architecture of the whole environment will depend on the answers to these questions.

In our architecture we separate the Web service definition and the diagnoser definition. In order to let the Web services useful for both diagnosis and non diagnosis scenarios we consider that the published diagnosis functionalities are defined in separate Web services that we call Diagnosis Web Services (DWS). Our architecture considers three types of Web services: Web services without diagnosis/recovery service; Web services with diagnosis and/or recovery service; and Web services offering only diagnosis/recovery service. See Figure 82 for the framework model.

In the following we make a proposition of an architecture. The architecture is detailed in different levels of granularity. First of all we argue why a diagnoser must be itself a Web service and we

define it in this way as DWS. We present next the interaction between the DWSs and the Web services in the case of a centralised, decentralised/supervised or distributed diagnosis and recovery process. Then we discuss the advantages and drawbacks of each approach. The last section gives a zoom on each Web service execution environment to emphasize the interaction between the extended modules and the platform components.

5.1.1 The diagnoser is a Web service

Considering diagnoser as a Web service has several design and implementation advantages:

- Interoperability for possibly distributed diagnosis processes. Diagnoser cooperation can be viewed and designed as choreography between a set of associated DWSs.
- Possibility of on-line cooperative diagnosis.
- Re-usability of Web services Standards for self-healing features description and implementation. For example we can use WSDL standard to describe the diagnosis operation offered by a DWS and BPEL to describe its behaviour or WSDL-S to annotate semantically its goals.

The DWS may differ by the offered services. We define three kinds of services.

- Information service: This consists in a set of operations invoked by partner diagnoser in order to get information about the service state and logged information for a given instance. This operation answers a request about the log file content. For example a diagnoser can ask another diagnoser for branching condition values, etc.
- Diagnosis service: Represents a set of operations dealing with diagnosis activities. The DWS may perform diagnosis locally or in cooperation with other DWSs (see section below for diagnosis architecture).
- Recovery service: DWSs can exchange (send and/or receive) information about the recovery decision or possible recovery strategies and implement them.

According to the chosen diagnosis process architecture and the chosen and recovery process architecture: centralised, decentralised/supervised or purely distributed, the DWS will need to provide a combination of the three kinds of services.

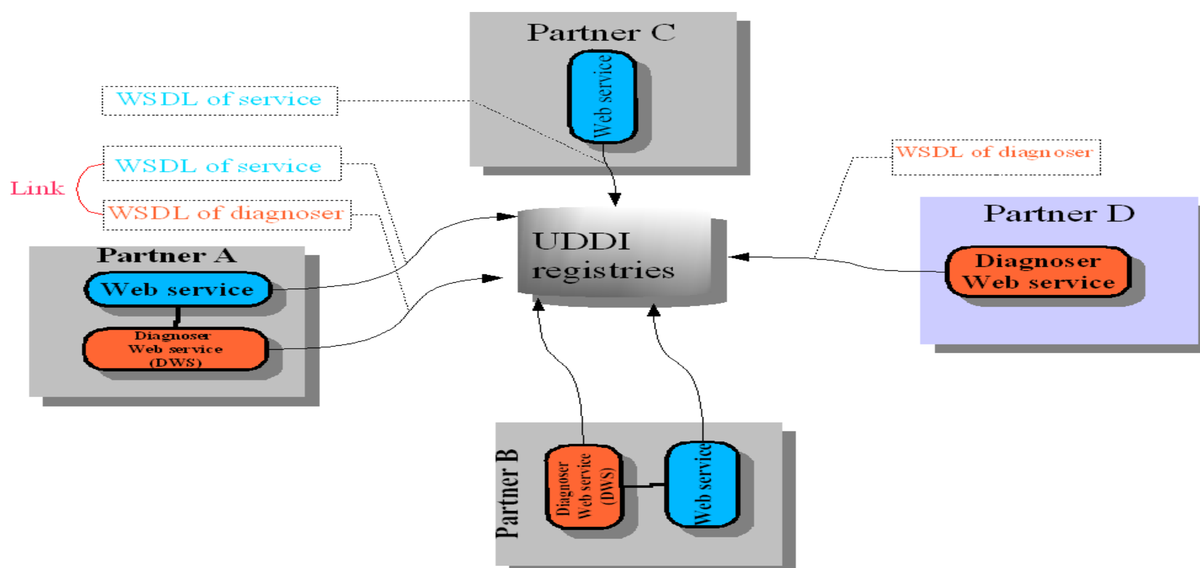


Figure 82 : Self-healing Web services framework model

5.1.2 The diagnosis architecture

In evaluating diagnostic architectures, we will consider both decentralised/supervised architectures and purely distributed architectures. In both cases, each Web Service that enters as partner in a Web Service composition may be associated with a DWS. The composition of and the interaction between the DWSs offered by each partner will depend on the chosen architecture. In the following we detail the needed capabilities (information, diagnosis, and recoveries) and the interaction schemas of the DWSs according to the diagnosis architecture. Here are the notations used:

- Green arrows represent information interactions.
- Red arrows represent diagnosis interactions.
- Blue arrows represent recovery interactions.
- Green, red, and blue databases represent respectively that the DWS offers information, diagnosis, and recovery capabilities.

5.2 Centralised diagnosis and recovery architecture

In centralised diagnosis/recovery approach we consider a distinguished partner which offers diagnosis and recovery capabilities that we call DWS Coordinator (DWSC). The composite Web service may be either a decentralised or a centralised workflow. In the first case we can imagine that the DWSC is the DWS of the central Web service. In the second case it represents an independent partner involved only in the diagnosis/recovery cooperation (the supervisor). The DWS of each partner in the centralised approach offers only information capabilities. They are

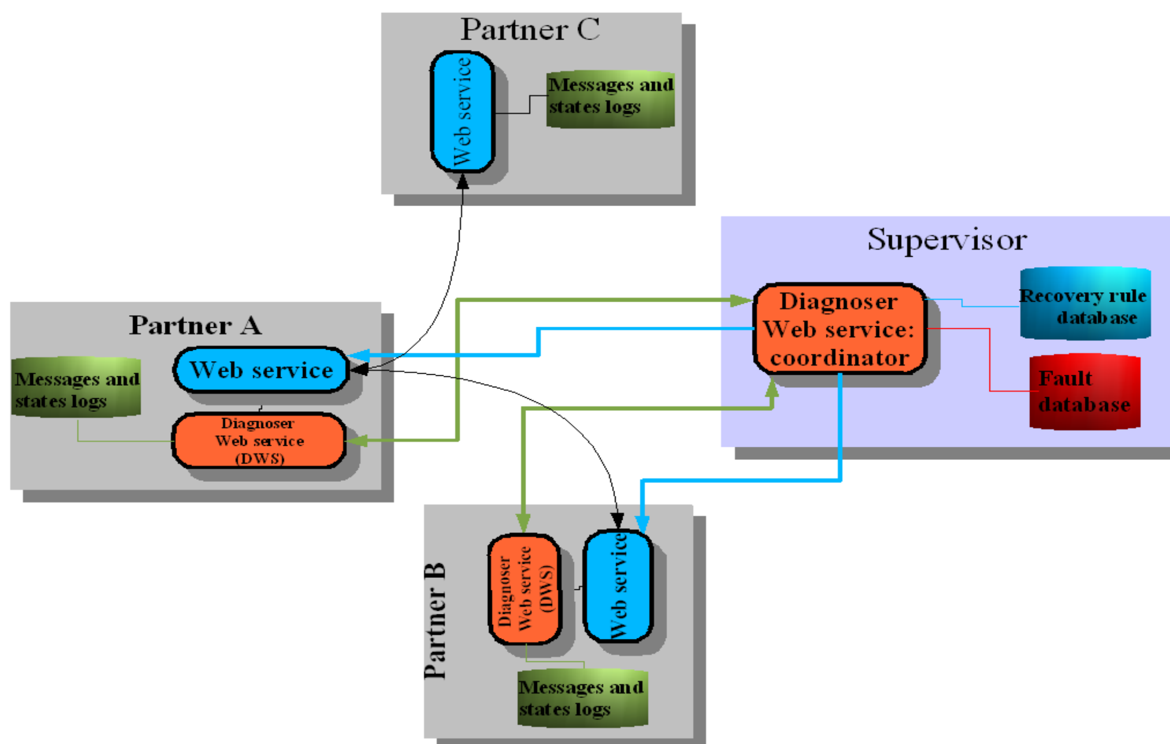


Figure 83 : Centralised diagnosis and centralised recovery

used by the DWSC to get information about the associated WS states and messages value in order to perform diagnosis and repairing activities. The repairing actions are decided by the DWSC,

which sends to each partner its role in the repairing task. Note that no diagnosis message is exchanged between the DWSs and the DWSC (see Figure 83).

5.3 Decentralised/supervised diagnosis and centralised recovery architecture

In the decentralised diagnosis approach the DWS of each partner may perform local diagnosis activities and give information about its WS states. The DWS may be invoked locally by the WS itself or by an internal detection mechanism. It can be used by the DWSC (the coordinator is still present) in order to perform local diagnosis. The DWSC uses the local diagnosis results from each DWS to decide about the diagnosis (e.g. by resolving local diagnosis conflicts and merging local diagnoses). The supervisor can initiate itself a diagnosis for certain types of faults based on its own fault detection mechanism (for example a fault which cannot be detected locally but needs a global view). The repairing activities are centralised and performed by the DWSC once the fault localised.

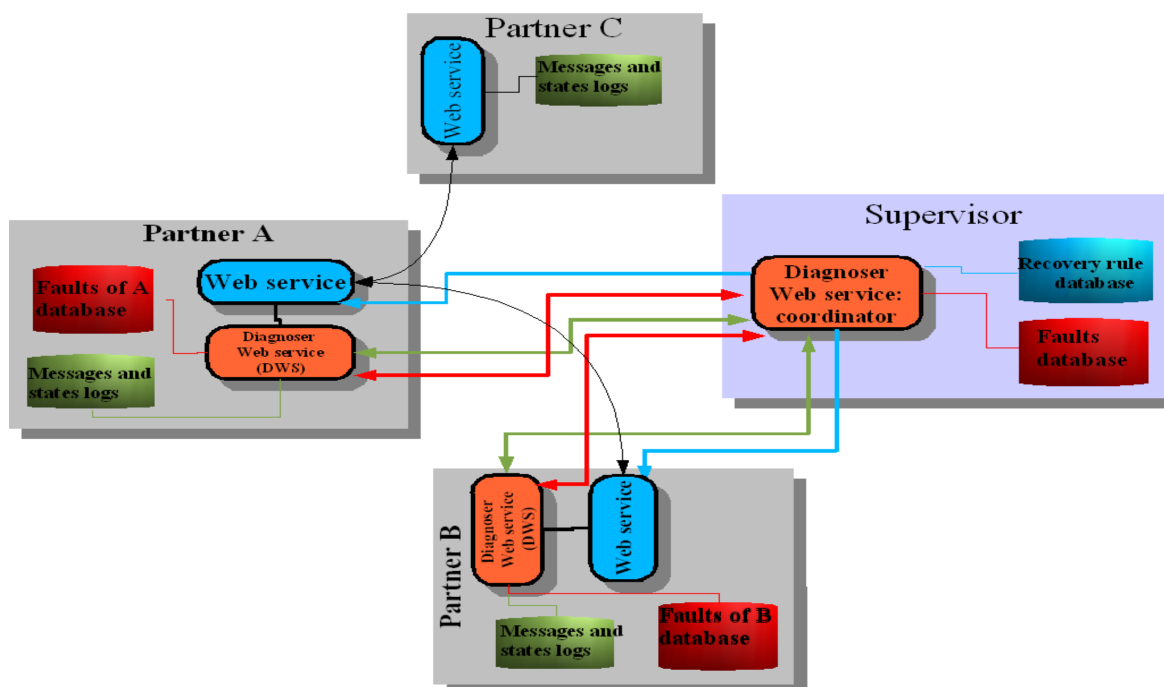


Figure 84 : Decentralised diagnosis and centralised recovery

It sends to each partner its role in the repairing tasks. Note that we can imagine in a decentralised approach that there is no information messages (green arrows) exchange between the DWSC and the DWSs (e.g. for security reasons). The Figure 84 represents this decentralised diagnosis architecture. Note that the recovery task could be also decentralised (see next section), leading to a decentralised diagnosis and decentralised recovery architecture.

5.4 Distributed diagnosis and decentralised recovery architecture

A distributed diagnosis approach supposes that each DWS holds local diagnosis capabilities and diagnosis coordination capabilities. Each DWS can perform diagnosis activities in order to ask another DWS a request or to answer the request of another DWS as well as for internal fault

detection. The DWS interacts with each other to decide about a common explanation for the fault. The resulted fault causes are communicated to the DWSC in order to organise the repairing tasks, in a centralised way as in Figure 84, or in a decentralised/supervised way as shown here. In this latter case, the DWSC defines the repairing tasks by integrating the repairing information gathered from each DWS and then it sends to each partner its role in the repairing tasks. The Figure 85 represents the interaction between the DWSs and the DWSC in the distributed diagnosis (and decentralised/supervised recovery) approach.

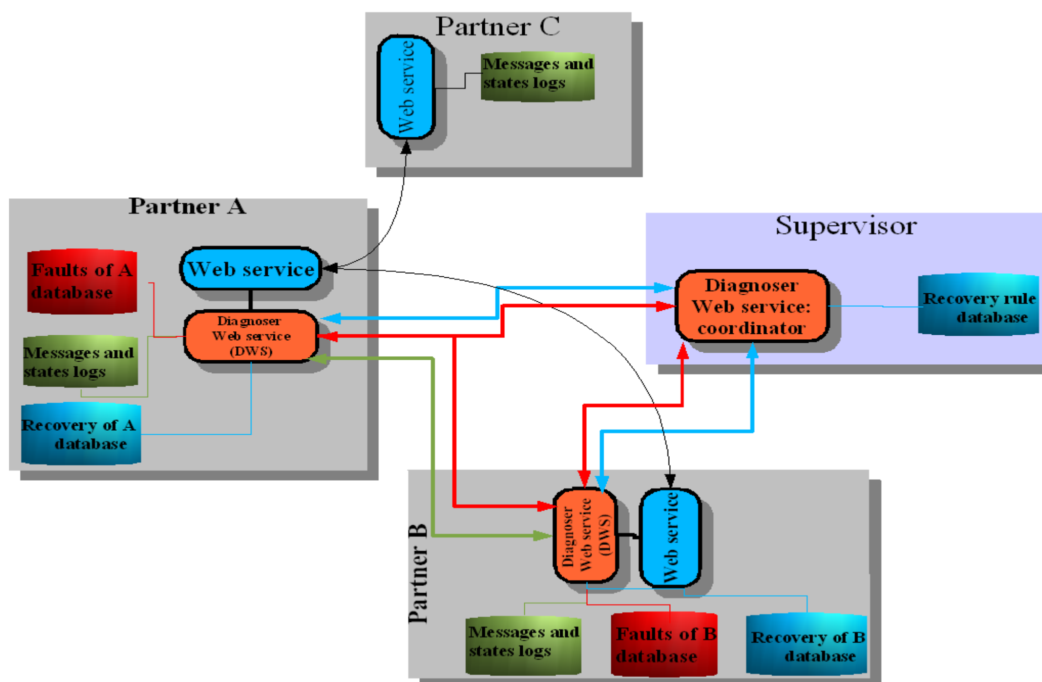


Figure 85 : Distributed diagnosis and decentralised recovery

In the same way, nothing prevents a priori to consider also complete distribution of the recovery process, i.e. a distributed diagnosis and distributed recovery architecture, so without any supervisor, but this seems to be out of the scope of feasibility in the present technological state of the art.

5.4.1.1 Discussion about the diagnosis architecture with regard to feasibility

Compromise between decentralised/supervised and distributed diagnosis architecture will have to be carefully studied w.r.t. feasibility, in particular concerning amount and nature of diagnostic information to be exchanged between DWSs..

It is worth pointing out that a purely distributed architecture does not necessarily lead to a looser approach or to local diagnosers having more independence.

In this regard, a supervised architecture has rather some advantages:

- Some diagnoses, and even more some repair actions, can require the coordination of the activities of different local diagnosers. This implies that a purely distributed approach would be tightly coupled, since in order to reach an agreement local diagnosers need to exchange a great amount of information following a complex interaction protocol; on the other side a supervised architecture could guarantee a quite loose interaction (among local diagnosers), thanks to the mediation of the diagnostic coordination service.
- Without any coordinator, local diagnosers need a quite large amount of extra knowledge, in order to be able to handle (complex) interactions with the other local diagnosers; this would

impose an overload at definition time to all those organisations willing to expose Web services with diagnostic capabilities, and would probably need them to forsake privacy requirements. In other words, in order to reach an agreement on a diagnosis, in a purely distributed architecture, local diagnosers would need to exchange more information than they would exchange with the coordinator in a supervised architecture. This is due to the fact that the coordinator can obtain a complete (though very abstract, without the internal private details) picture of what is going on, which is missing in the purely distributed approach.

- In a purely distributed approach it is hard for local diagnosers to keep track of the correlation between several different (but logically related) interactions taking place among them. In a supervised architecture the logical relations between different invocations of a local diagnoser are handled by the coordinator, and each invocation to a local diagnoser can be independent of the others from the point of view of the local diagnoser state.

5.4.1.2 The platform architecture

Here we propose our vision of the architecture of a self-healing Web services environment. Some assumptions are considered:

We consider that the diagnosis and repair tools will be defined for an existent execution environment. Implementing diagnosis and repair activities is considered as advanced features of Web services.

The architecture must offer a set of tools to handle diagnosis and repair design aspects. For example, a by-default diagnoser and repair service.

We do not make assumptions about the diagnosis algorithm or the characteristics – centralised, decentralised/supervised, distributed – of the approach. The architecture is general.

For that, we consider first a generic Web services process execution environment (see Figure 86) and then we represent the different components of the diagnosis extension and the possible interactions within the generic environment components.

5.4.1.3 A generic WSP execution environment

The generic architecture represents a set of possible components. This allows an abstract description of the main execution steps of a WSP. Based on these steps we can sketch, from control and data point of view, the diagnosis and repairing interventions.

- Compiler: verifies the User BPEL4WS specification.
- Deployer: transforms the User specification in an internal data format. This transformation depends on the implementation of the BPEL4WS engine. It generates what we call Deployable Web services process (DWSP) (see Figure 86).
- Communication manager: responsible of the relation between the Web service instances and their partners (manages both input and output).
- Instance manager: handles the creation and the rooting mechanisms (correlation mechanisms of BPEL4WS).
- State manager: execution engine. It implements the control semantics of BPEL4WS constructors.
- Web services invoker: responsible for client management. It executes essentially the invoke activities for the state manager.
- Data manager: maintains a database and stores different events and data of the WSP life cycle. It represents the capacity of an environment to register different data at a different level of granularity. For example, handling the log files and error archives or saving the process states.

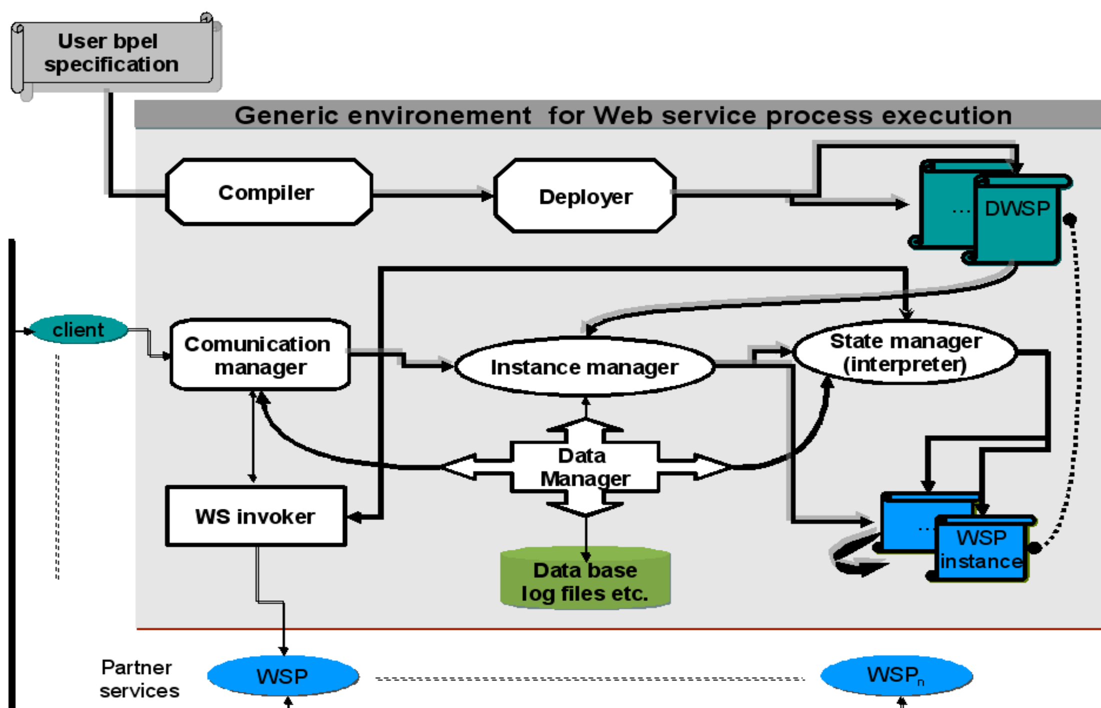


Figure 86 : Generic Web services execution environment

5.4.1.4 The WS-Diamond architectural extension for self-healing Web services

WS-Diamond aims at realising an additional set of tools that can be plugged into existent Web services execution environment and can provide self-healing features during the design, the execution and the management of the Web service life cycle. The main features in the WS-Diamond extension are fault detection, diagnosis capabilities, repairing capabilities. We do not assume that within one partner all the features are used. We call WS-Diamond node the partner infrastructure that provides WS-Diamond features as show on Figure 80.

We illustrate in next Figure the cooperation of the different modules inside a node and within the execution environment main components (the color code green for information, red for diagnosis and blue for repair is maintained).

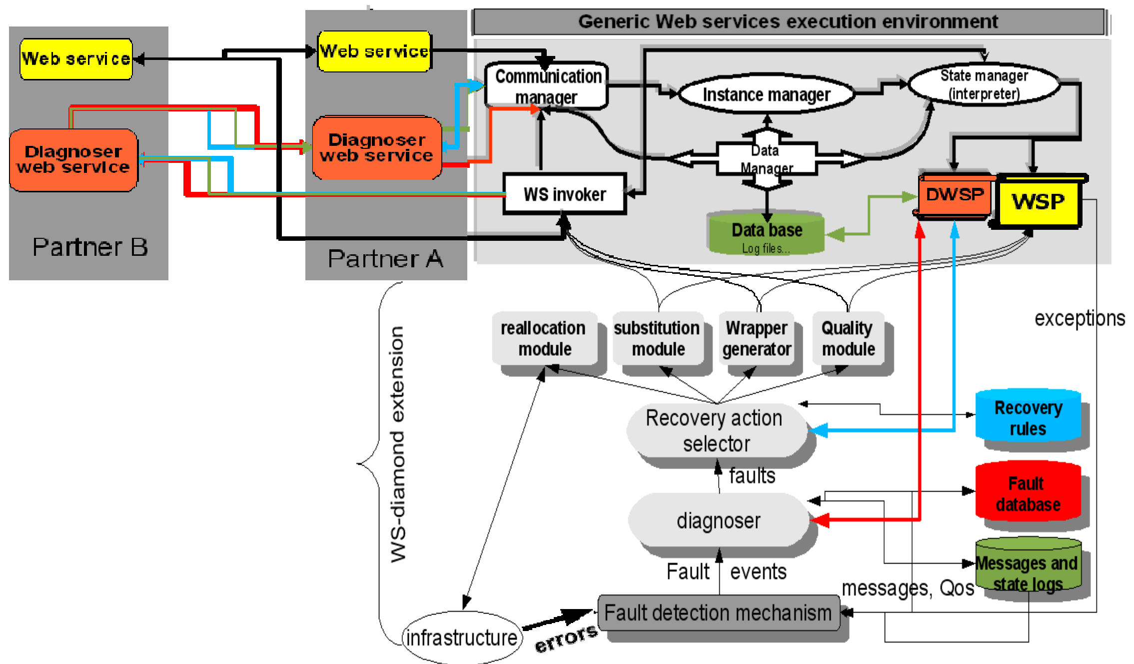


Figure 87 : Cooperation of the modules inside a node and with the execution environment

The diagnoser is notified of fault events through messages or events generated by the hardware/software infrastructure (including the self-healing system itself). Using access to messages and states logs and to a fault database (all fault events are stored in a fault log), the diagnoser identifies which fault occurred and needs to be recovered. The recovery action selector performs a choice among a set of possible recovery actions associated to each type of fault as indicated in a recovery rules registry. The selection triggers a recovery action request to a recovery module associated to the required action. A list of such modules (not all-inclusive) is provided in the figure: substitution module to replace services during the orchestration of a composed service, wrapper generator to change parameters to solve incompatibility problems during invocation, quality module to perform data quality checks and improvements (e.g. correct typos or incorrect coding), reallocation module to change allocation of resources to services. Note that the substitution, wrapper and Qos modules can either perform direct recovery actions by executing the recovery actions (e.g the substitution module invokes the new web service by using the Web service invoker module) or indirectly by changing the Web Services Process instance state (e.g the substitution can replace in the WSP instance an invoke activities by another one automatically generated).

We describe in the following a sequence diagram illustrating the interaction between the WS-Diamond extension modules and the generic architecture ones. Note that the sequence diagram is a simple example and do not cover all the possible interactions presented in the previous paragraph.

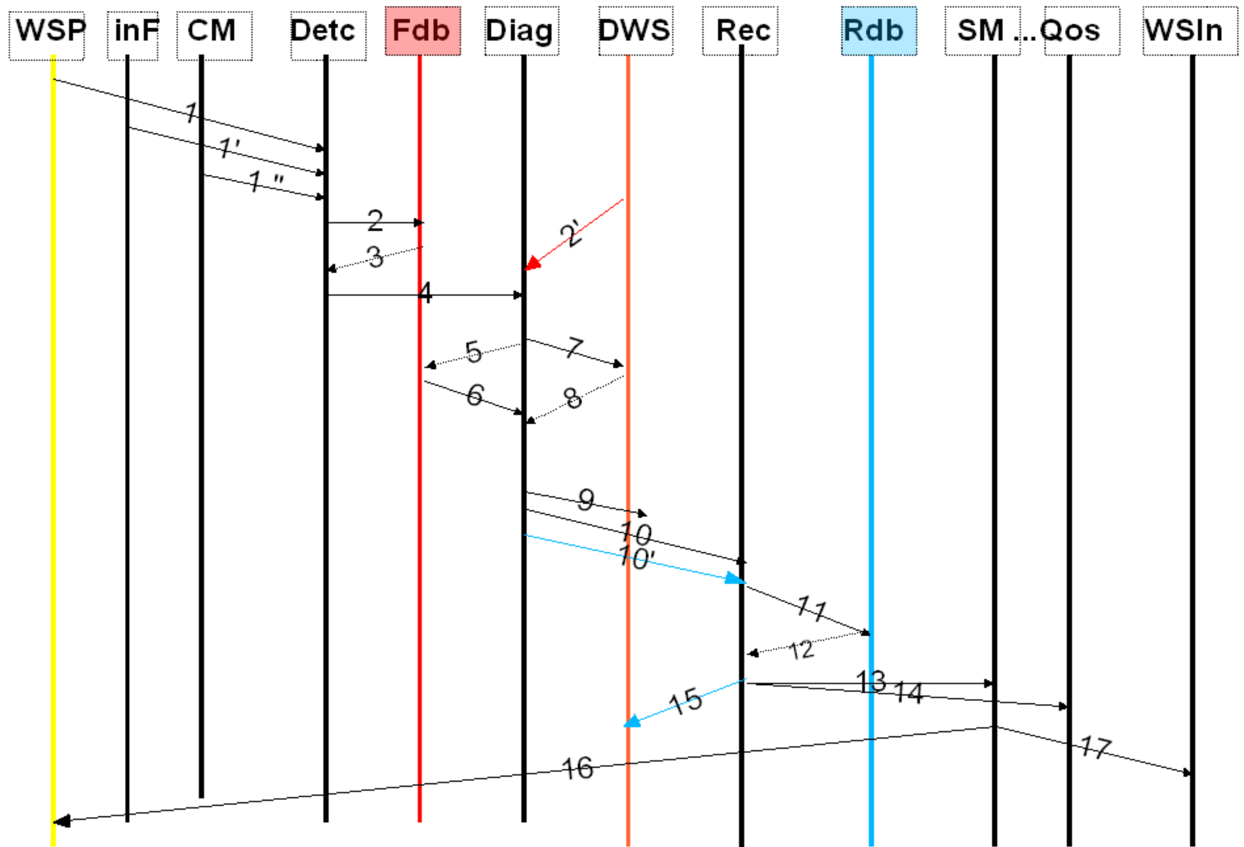


Figure 88 : Sequence diagram of modules interaction

- 1 , 1' or 1" represent respectively an exception raised by a WSP, an error occurred in the infrastructure (Inf) and a message coming from the Communication Manager (CM). All these events are given to the Detection module (Detc).
- The detection module can request the fault Database (Fdb) (message 2) in order to get more information about the fault event.
- Stored Fault events information (3)
- After identifying the fault event the Detection module (Dect) throw it to the diagnoser module (Diag), 4.
- The diagnosis module can receive a diagnosis request coming from an other DWS of one of the partners either for distributed or decentralized case (message 2'). The 2' message can be an alternative case to all the previous ones.
- The diagnoser either gets access to the fault Database (e.g. case base diagnosis) (message 5), or requests information from other DWS of partners using the DWSP (associated to the target service instance) (message 7). The 6 and 8 messages represent the response of the two previous ones (5 and 7). This process can be repeated as necessary.
- The 9 and 10 messages represent the diagnoser output (the diagnosis) for the received fault event. The 9 is a diagnosis message sent either to the supervisor DWS or to a DWS of one

of the partners. The 10 message is a possible explanation of the fault source sent to the recovery module (Rec).

- The recovery module will use the diagnosis output (decision) in order to choose the most suited repair actions. For that it can ask the recovery Database (Rdb) for recovery rules (11 and 12).
- The recovery module can be requested by supervisor and partner for doing some repair actions (message 10').
- The recovery module activates one or more of the recovery modules (messages 13 and 14).
- The recovery module can send its repair decision to a supervisor or one of the partners DWS. It uses the DWS (message 15).
- The selected recovery module can either change the Web service instance state or use the invoker module (message 16 and 17).

6 Summary and outlook

This report summarized the work performed and the results achieved in the first part of the project and corresponding to Milestone M1. In particular we reported on:

- Requirements for self-healing Web Services. We defined that requirements that are guiding and will guide the work in the project workpackages
- Selection of test-beds. We defined the application scenarios that will be used as test-beds during the design and testing of the surveillance platform
- Common working environment and standards. We reported on the choices we made as regards these two aspects, moving from the state of the art and motivating the specific decision and selections we made.

We also presented a preliminary diagnostic architecture that complement the definition of the requirements and that is a concrete starting point for the work in the project.

Last but not leas we provided a glossary that summarizes and unifies the terminology adopted by the Web Services and Diagnosis community, providing the basis for all the future project reporting and documentation.

7 Glossary of terms

Correct service: identifies a service implementing the designed system functions.

Failure: an event occuring when a discrepancy between the delivered service and the correct is observable.

Error: part of the system state that arises as a consequence of a fault and may cause a subsequent failure.

Fault: a malfunctioning in the service execution that can produce an erroneous state and, as a consequence, a failure.

Fault-Error-Failure Chain: summarizes the relationships among a fault, an error, and a failure. As shown in Figure 1, when a fault occurs it causes an error which, in turn, is manifested as a failure. A fault can be either active or dormant. In the former case the fault produces an error; in the latter case does not.

Exception: notification of failure to a diagnoser.

Error/Failure detection: activity related to the discovery of the error which cause a failure. It represents the input of the diagnoser and state how the failure occurs.

Fault identification: one of the results of diagnosis. It defines why the failure occurs.

Recovery: process subsequent to the diagnosys aiming at providing a dependable system. It could be either reactive or proactive and may include repair actions. Reactive recovery (also called on-line recovery) aims at solve a fault after the related exception is caught. Pro-active recovery aims at avoiding the fault.

Self-healing system: system able to automatically recover possible failures.

Failure modes: a characterization of the way a process fails. Refers to a rather complete description, including the pre-conditions under which failure occurs, how the thing was being used, proximate and ultimate/final causes (if known), and any subsidiary or resulting failures that result.

Observation: Set of system (or component) states at a given time.

Compensation: application-specific activities that attempt to reverse the effects of a previous activity that was carried out as a part of a larger unit of work that is being abandoned.

Repair action: Any activity, such as tests, measurements, replacements, adjustments and repairs, intended to restore or retain a functional unit in a specified state in which the unit can perform its required functions.

Alarm: give warning of a problem or of a condition, often audibly and/or visually.

Elementary activity: A specific task, that provides a specialized capability, service or product based on a requirements.

Complex activity: A grouping of tasks, that provides a specialized capability, service or product based on a requirements.

Task: any piece of work that is undertaken or attempted

Business process:A set of one or more linked procedures or activities, which collectively realize a business objective or policy goal, normally within the context of an organizational structure defining functional roles and relationships [WfM98].

A web service composition: is a process definition (workflow), which is composed of several activities. These activities are related to the parts of a business process and depend on each other. Each composition has a predefined start state, termination state and a process flow.

Composition (Composition Schema, Workflow):The automation of a business process, in whole or part, during which documents, information or tasks are passed from one participant to another for action, according to a set of procedural rules [WfM98].

The activities of composition will be actively controlled by web service composition management service [ET]. WSCMS is a (re)active system for control of process flow among involved web services or web service compositions according to a workflow specification (composition). It supports with it its components both design (build time components) and their control and execution (run time components) of web service compositions.

Web service composition management system (workflow management system):A system that defines, creates and manages the execution of compositions through the use of software, running on one or more composition engines that is able to interpret the composition definition, interact with composition participants (web services) and, where required, invoke appropriate IT tools and applications [WfM98].

Appendix A. FoodShop example BPEL code

```
<?xml version="1.0" encoding="UTF-8"?>
<process name="FastFoodShopingProject"
    targetNamespace="urn:FastFoodShopingProject" xml:ID="1"
    xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
    xmlns:bpws="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
    xmlns:customer="http://wsdiamond.com/wsdl/foodshopexample/customer"
    xmlns:shop="http://wsdiamond.com/wsdl/foodshopexample/shop"
    xmlns:supplier="http://wsdiamond.com/wsdl/foodshopexample/supplier"
    xmlns:tns="urn:FastFoodShopingProject"
    xmlns:warehouse="http://wsdiamond.com/wsdl/foodshopexample/warehouse"
    xmlns:wsa="http://schemas.xmlsoap.org/ws/2003/03/addressing"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <import importType="http://schemas.xmlsoap.org/wsdl/"
        location="warehouse.wsdl"
namespace="http://wsdiamond.com/wsdl/foodshopexample/warehouse"/>
    <import importType="http://schemas.xmlsoap.org/wsdl/"
        location="supplier.wsdl"
namespace="http://wsdiamond.com/wsdl/foodshopexample/supplier"/>
    <import importType="http://schemas.xmlsoap.org/wsdl/"
        location="shop.wsdl"
namespace="http://wsdiamond.com/wsdl/foodshopexample/shop"/>
    <import importType="http://schemas.xmlsoap.org/wsdl/"
        location="customer.wsdl"
namespace="http://wsdiamond.com/wsdl/foodshopexample/customer"/>
    <partnerLinks>
        <partnerLink myRole="service" name="customer"
            partnerLinkType="customer:customerServiceLT"
partnerRole="customer"/>
        <partnerLink name="shop" partnerLinkType="shop:shopLT"
partnerRole="shop"/>
        <partnerLink name="supplier"
            partnerLinkType="supplier:supplierLT" partnerRole="supplier"/>
        <partnerLink name="warehouse"
            partnerLinkType="warehouse:warehousePLT" partnerRole="warehouse"/>
    </partnerLinks>
    <variables>
        <variable messageType="customer:orderMsg" name="Order"/>
        <variable messageType="customer:itemsMsg" name="s_items"/>
        <variable messageType="customer:itemsMsg" name="ns_items"/>
        <variable messageType="shop:splitedOrder" name="splitedOrder"/>
        <variable messageType="supplier:answers" name="s_answers_by_avail"/>
        <variable messageType="supplier:answers" name="ns_answers"/>
        <variable messageType="customer:itemsMsg" name="s_avail_items"/>
        <variable messageType="customer:itemsMsg" name="s_nonavail_items"/>
        <variable messageType="warehouse:availItems" name="availResult"/>
        <variable messageType="supplier:answers" name="s_answers_by_nonavail"/>
        <variable messageType="customer:itemsMsg"
name="total_avail_and_reserved"/>
        <variable messageType="warehouse:shipcost" name="shipcostWH"/>
        <variable messageType="shop:totalCost" name="totalCost"/>
        <variable messageType="customer:totalCost" name="bill"/>
        <variable messageType="customer:status" name="paidStatus"/>
        <variable messageType="customer:parcelMsg" name="parcel"/>
        <variable messageType="customer:parcelMsg" name="toAssemble"/>
        <variable messageType="warehouse:whInfoMsg" name="whInfo"/>
        <variable messageType="warehouse:toSupply" name="tosupply_ns"/>
        <variable messageType="warehouse:toSupply" name="tosupply_nonavail"/>
        <variable messageType="customer:status"
name="tosupply_nonavail_status"/>
        <variable messageType="customer:status" name="tosupply_ns_status"/>
        <variable messageType="customer:status" name="sent_status"/>
```

```
    </variables>
    <correlationSets>
        <correlationSet name="supplCS" properties="supplier:suppID"/>
        <correlationSet name="thisSet" properties="customer:orderID"/>
    </correlationSets>
    <sequence xml:ID="2">
        <receive createInstance="yes" name="sendOrder"
            operation="requestorder" partnerLink="customer"
            portType="customer:ServicePT" variable="Order" xml:ID="3">
            <correlations>
                <correlation set="thisSet"/>
            </correlations>
        </receive>
        <flow xml:ID="4">
            <invoke inputVariable="Order" operation="selectWH"
                outputVariable="whInfo" partnerLink="shop"
                portType="shop:shopPT" xml:ID="5"/>
            <sequence xml:ID="6">
                <invoke inputVariable="Order" operation="splitOrder"
                    outputVariable="splitedOrder" partnerLink="shop"
                    portType="shop:shopPT" xml:ID="7"/>
                <assign xml:ID="8">
                    <copy>
                        <from part="s_items" variable="splitedOrder"/>
                        <to variable="s_items"/>
                    </copy>
                </assign>
                <assign xml:ID="9">
                    <copy>
                        <from part="ns_items" variable="splitedOrder"/>
                        <to variable="ns_items"/>
                    </copy>
                </assign>
            </sequence>
        </flow>
        <flow xml:ID="10">
            <invoke inputVariable="ns_items"
                name="verifyAndReserve_ns_items"
                operation="verifyAndReserve" outputVariable="ns_answers"
                partnerLink="supplier" portType="supplier:supplierPT"
xml:ID="11"/>
            <sequence xml:ID="12">
                <invoke inputVariable="s_items"
                    name="checkAvail_s_items" operation="checkAvail"
                    outputVariable="s_answers_by_avail"
                    partnerLink="shop" portType="shop:shopPT" xml:ID="13"/>
                <invoke inputVariable="s_items" operation="reserveAvail"
                    outputVariable="availResult" partnerLink="warehouse"
                    portType="warehouse:warehousePT" xml:ID="14"/>
                <assign xml:ID="15">
                    <copy>
                        <from part="avail_items" variable="availResult"/>
                        <to variable="s_avail_items"/>
                    </copy>
                    <copy>
                        <from part="nonavail_items" variable="availResult"/>
                        <to variable="s_nonavail_items"/>
                    </copy>
                </assign>
                <invoke inputVariable="s_nonavail_items"
                    operation="verifyAndReserve"
                    outputVariable="s_answers_by_nonavail"
                    partnerLink="supplier"
                    portType="supplier:supplierPT" xml:ID="16"/>
            </sequence>
```

```
        </flow>
        <assign xml:ID="17">
            <appendChild>
                <from variable="ns_items"/>
                <to variable="total_avail_and_reserved"/>
            </appendChild>
            <appendChild>
                <from part="resitems" variable="s_answers_by_avail"/>
                <to variable="total_avail_and_reserved"/>
            </appendChild>
            <appendChild>
                <from part="resitems" variable="s_answers_by_nonavail"/>
                <to variable="total_avail_and_reserved"/>
            </appendChild>
        </assign>
        <switch xml:ID="18">
            <case
condition="bpws:getVariableData('total_avail_and_reserved')=bpws:getVariableData
('Order','items')">
                <sequence xml:ID="19">
                    <invoke inputVariable="total_avail_and_reserved"
                        operation="shipcost" outputVariable="shipcostWH"
                        partnerLink="warehouse"
                        portType="warehouse:warehousePT" xml:ID="20"/>
                    <scope xml:ID="21">
                        <faultHandlers>
                            <catch faultName="nopayment">
                                <sequence xml:ID="22">
                                    <invoke
                                    inputVariable="s_avail_items"
                                    operation="unreserve"
                                    partnerLink="warehouse"
                                    portType="warehouse:warehousePT"
xml:ID="23"/>
                                    <invoke inputVariable="ns_items"
                                    operation="unreserve"
                                    partnerLink="supplier"
                                    portType="supplier:supplierPT" xml:ID="24"/>
                                    <invoke
                                    inputVariable="s_nonavail_items"
                                    operation="unreserve"
                                    partnerLink="supplier"
                                    portType="supplier:supplierPT" xml:ID="25"/>
                                    <terminate xml:ID="26"/>
                                </sequence>
                            </catch>
                        </faultHandlers>
                        <sequence xml:ID="27">
                            <invoke inputVariable="shipcostWH"
                                operation="computeTotalCost"
                                outputVariable="totalCost"
                                partnerLink="shop"
                                portType="shop:shopPT" xml:ID="28"/>
                            <assign xml:ID="29">
                                <copy>
                                    <from part="amount" variable="totalCost"/>
                                    <to part="amount" variable="bill"/>
                                </copy>
                            </assign>
                            <invoke inputVariable="bill"
                                operation="sendBill"
                                partnerLink="customer"
                                portType="customer:customerPT" xml:ID="30"/>
                            <receive operation="paidrequest"
                                partnerLink="customer"
```

```
                                portType="customer:ServicePT"
                                variable="paidStatus" xml:ID="31">
                                <correlations>
                                    <correlation set="thisSet"/>
                                </correlations>
                            </receive>
                        </sequence>
                    </scope>
                    <switch xml:ID="32">
                        <case
condition="bpws:getVariableData('paidStatus','status')=true">
                            <sequence xml:ID="33">
                                <assign xml:ID="34">
                                    <appendChild>
                                    <from variable="ns_items"/>
                                    <to part="ns_items" variable="toAssemble"/>
                                    </appendChild>
                                    <appendChild>
                                    <from part="resitems"
variable="s_answers_by_avail"/>
                                    <to part="s_avail_items"
variable="toAssemble"/>
                                    </appendChild>
                                    <appendChild>
                                    <from part="resitems"
variable="s_answers_by_nonavail"/>
                                    <to part="s_nonavail_items"
variable="toAssemble"/>
                                    </appendChild>
                                    <copy>
                                    <from variable="whInfo"/>
                                    <to part="whInfo" variable="tosupply_ns"/>
                                    </copy>
                                    <copy>
                                    <from variable="whInfo"/>
                                    <to part="whInfo"
variable="tosupply_nonavail"/>
                                    </copy>
                                    <copy>
                                    <from variable="ns_items"/>
                                    <to part="items" variable="tosupply_ns"/>
                                    </copy>
                                    <copy>
                                    <from part="resitems"
variable="s_answers_by_nonavail"/>
                                    <to part="items"
variable="tosupply_nonavail"/>
                                    </copy>
                                </assign>
                                <invoke
                                    inputVariable="tosupply_nonavail"
                                    operation="supply"
                                    outputVariable="tosupply_nonavail_status"
                                    partnerLink="supplier"
                                    portType="supplier:supplierPT" xml:ID="35"/>
                                <invoke inputVariable="tosupply_ns"
                                    operation="supply"
                                    outputVariable="tosupply_ns_status"
                                    partnerLink="supplier"
                                    portType="supplier:supplierPT" xml:ID="36"/>
                                <invoke inputVariable="toAssemble"
                                    operation="assemble"
                                    outputVariable="parcel"
                                    partnerLink="warehouse"
```

```
                                    portType="warehouse:warehousePT"
xml:ID="37"/>
                                <invoke inputVariable="parcel"
                                    operation="sendParcel"
                                    outputVariable="sent_status"
                                    partnerLink="customer"
                                    portType="customer:customerPT" xml:ID="38"/>
                            </sequence>
                        </case>
                        <otherwise>
                            <terminate xml:ID="39"/>
                        </otherwise>
                    </switch>
                </sequence>
            </case>
            <otherwise>
                <sequence xml:ID="40">
                    <invoke inputVariable="s_avail_items"
                        operation="unreserve" partnerLink="warehouse"
                        portType="warehouse:warehousePT" xml:ID="41"/>
                    <invoke inputVariable="ns_items"
                        operation="unreserve" partnerLink="supplier"
                        portType="supplier:supplierPT" xml:ID="42"/>
                    <invoke inputVariable="s_nonavail_items"
                        operation="unreserve" partnerLink="supplier"
                        portType="supplier:supplierPT" xml:ID="43"/>
                    <terminate xml:ID="44"/>
                </sequence>
            </otherwise>
        </switch>
    </sequence>
</process>
```

References

[AAFJ+02] Arkin, A., Askary, S., Fordin, S., Jekeli, W., Kawaguchi, K., Orchard, D., Pogliani, S., Riemer, K., Struble, S., Takacsi-Nagy, P., Trickovic, I. and Zimek, S. (2002). Web Service Choreography Interface (WSCI) 1.0. *W3C Note*, August 2002. Retrieved January, 2005, from http://www.w3.org/TR/wsci/

[AAZM04] Arpinar, B., Aleman-Meza, B., Zhang, R.and Maduko, A. (2004). *Ontology-Driven Web Services Composition Platform*. Proceedings of the IEEE International Conference on E-Commerce Technology, IEEE.

[ABW] Active BPEL official web-site: http://www.activebpel.org/index.html

[ACDG+03] Andrews, T., Curbera, F., Dholakia, H., Goland, Y., Klein, J., Leymann, F., Liu, K., Roller, D., Smith, D., Thatte, S., Trickovic, I. and Weerawarana,, S. (2003) *Business Process Execution Language for Web Services, Version 1.1, Specification*. http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/

[ACKM04] Alonso, G., Casati, F., Kuno, H. and Machiraju, V. (2004). *Web Services – Concepts, Architectures and Applications*. Springer-Verlag, Berlin Heidelberg.

[AES] Active Endpoints official web-site: http://www.active-endpoints.com/products/activewebflow/awf_faqs.html#top

[AGS] ActiveGrid official web-site: http://www.activegrid.com/what/products.php

[AMS02] Aissi, S., Malu, P. and Srinivasan, K. (2002). *E-business process modeling: the next big step.* IEEE Computer, Volume 35, Issue 5, pp 55 – 62.

[AK05] "Web Service Semantics – WSDL-S", W3C Technical Note, Version 1.0., Akkiraju et al., April 2005

[AOS] Agila Projectofficial web-site: http://incubator.apache.org/projects/agila/

[APA] Apache Axis. http://ws.apache.org/axis/

[APA2] Apache Axis2. http://ws.apache.org/axis2/

[ABC03] Atkinson, B., T. Bellwood und M. Cahuzac et. al.: *UDDI Version 3.0.1*. Technical report, OASIS, October 2003.

[BAM05] Alistair Barros, Marlon Dumas, Phillipa Oakes: *Standards for Web Service Choreography and Orchestration: Status and Perspectives*. In: BPM Workshops 2005, Springer-Verlag 2005

[BAP03] Bausch, W., C. Pautasso und G. Alonso: *Programmign for Dependability in a Service-based Grid*, In: 3rd International Symposium on Cluster Computing and the Grid, pp. 164-171, IEEE 2003

[BBG03] Baïna, K., Benali, K. and Godart, C. (2003). *Dynamic interconnection of heterogeneous workflow processes through services.* In 11th International Conference on Cooperative Information Systems (CoopIS'03), In Confederated International Conferences (DOA/CoopIS/ODBASE'03), (LNCS) 2888, Catania, Sicily, Italy, November 3-7, 2003. Springer-Verlag.

[BC01] Bieber, G. and Carpenter, J. (2001). Introduction to Service-Oriented Programming (Rev 2.1). Retrieved January, 2005, from http://www.openwings.org/download/specs/ServiceOrientedIntroduction.pdf

[BCT04] Benatallah, B., Casati, F. and Toumani, F. (2004). *Web service conversation modeling: a cornerstone for e-business automation.* IEEE Internet Computing, Volume 8, Issue 1, pp 46 – 54.

[BD05] Bianchini, D. and De Antonellis, V. (2005). *Ontology-based Semantic Interoperability Tools for Service Dynamic Discovery.* In Proc. of the First Int. Conference on Interoperability of Enterprise Software Applications (INTEROP-ESA'05), Geneva, Switzerland, February, 23rd-25th 2005.

[BDPP05] Bianchini, D., De Antonellis, V., Pernici, B. and Plebani, P. (2005). *Ontology-based methodology for e-Service discovery.* Information Systems, Elsevier Science, in press, published on line.

[BEFG+04] Ballinger, K., Ehnebuske, D., Ferris, C., Godgin, M., Liu, C. K., Nottingham, M. and Yendluri, P. (2004). *Basic Profile Version 1.1*. Web Services Interoperability Organization (August 2004), *http://ws-i.org/Profiles/BasicProfile-1.1.html*.

[BOS] bexee Project official web-site: http://bexee.sourceforge.net

[BPEL4People] Kloppmann, M. et al. (2005). *WS-BPEL Extension for People – BPEL4People,* A Joint White Paper by IBM and SAP.

[BPEL-SPE] Kloppmann, M. et al. (2005). *WS-BPEL Extension for Sub-processes – BPEL-SPE,* A Joint White Paper by IBM and SAP.

[BPMI02] BPMI.org (2002). *BPML/BPEL4WS - A Convergence Path toward a Standard BPM Stack.* BPMI.org Position Paper. Retrieved January, 2005, from http://www.bpmi.org/

[BPMI05] BPMI.org. (2005). *Business Process Management Initiative* (n.d.). Retrieved January, 2005, from http://www.bpmi.org/

[BSD03] Benatallah, B., Sheng, Q.Z. and Dumas, M. (2003). *The Self-Serv environment for Web services composition.* IEEE Internet Computing, Volume 7, Issue 1, pp 40 – 48.

[C03] Chinnici, R. (2003). *Java API for XML-Based RPC (JAX-RPC) Specification 1.1.* SUN Microsystems (October 2003), http://java.sun.com/webservices/jaxrpc/

[C04] Chappell, D. (2004). *Understanding BPM Servers*. Chappell Associates. Retrieved December, 2004, from http://www.microsoft.com/biztalk/techinfo/default.mspx

[CAM00] Cass, A., B. Lernerand, E. McCall, L. Osterwei, S. Sutton und A. Wise: *A Process Definition Language and Interpreter.* In: In Proceedings of the International Conference on Software Engineering (ICSE), Ireland, June 2000

[CAM01] Fabio Casati, Ming-Chien Shan: *Dynamic and adaptive composition of e-services.* Inf. Syst. 26(3): pp. 143-163, 2001

[CBS04] Cardoso, J., Bostrom, R.P. and Sheth, A. (2004). *Workflow Management Systems and ERP Systems: Difference, Commonalities, and Applications.* Information Technology and Management 5, Kluwer Academic Publishers, pp 319 – 338.

[CCOS] CapeClear Project official web-site: http://www.capeclear.com

[CCMW01] Christensen, E., Curbera, F., Meredith, G. and Weerawarana.,S. (2001). *Web Services Description Language (WSDL) 1.1*. W3C, Note 15, 2001. http://www.w3.org/TR/wsdl.

[CD98] Castano, S. and De Antonellis, V. (1998). A Framework for expressing Semantic Relationships between Multiple Information Systems for Cooperation. Information Systems, 19(4): 33-54.

[CDM04] Confalonieri, R., Domingue, J. and Motta, E. (2004). *Orchestration of Semantic Web Services in IRS-III.* In proceedings of the First AKT Workshop on Semantic Web Services (AKT-SWS04) KMi, The Open University, Milton Keynes, UK, December 8, 2004.

[CH85] C. A. R. Hoare: *Communicating Sequential Processes*, Prentice Hall International Series in Computer Science, 1985

[CIY00] Nihan Kesim Cicekli, Yakup Yildirim: Formalizing Workflows Using the Event Calculus. DEXA 2000, pp. 222-231, Springer-Verlag 2000

[CMPP+04] Cappiello, C., Missier, P., Pernici, B., Plebani, P. and Batini, C. (2004). *QoS in Multichannel IS: the MAIS Approach.* Proceedings of the International Workshop on Web Quality (WQ'04) in conjunction with the ICWE 2004, Munich, Germany.

[COL] Collaxa: http://www.collaxa.com

[COS] Collaxa Project official web-site: http://www.nttman.net/archives/000969.html

[CRR91] J. Crow, J. Rushby. Model-based reconfiguration: *Toward an integration with diagnosis*, Proc. 9th national Conference on Artificial Intelligence AAAI'91, Anaheim, CA, USA, pp.836-841, 1991.

[CSS] Creative Science Systems BizZyme BPEL Java Server Official web-site: http://www.creativescience.com/software/bpel.html

[DF97] Damiani, E., and Fugini, M.G. (1997) *Fuzzy Identification of Distributed Components.* In Proceedings of the 5th Fuzzy Days International Conference, LNCS 1226, pp 550-552.

[DMPP03] De Antonellis, V., Melchiori, M., Pernici, B. and Plebani, P. (2003). *A Methodology for e-Service Substitutability in a Virtual District Environment.* In Proceedings of the Conference on Information Systems Engineering (CAiSE 2003), Velden, Austria.

[DP06] Daniel, F. and Pernici, B. (2006) special issue of International Journal of E-business Research (IJEBR) on Web Services-Based E-Business Systems, Jan 2006

[DS05] Dustdar, S. and Schreiner, W. (2005). *A Survey on Web Services Composition*. Int. J. Web and Grid Services, Vol. 1, No. 1, 2005.

[ERG96] Ekkart Rudolph, Peter Graubmann, Jens Grabowski: Tutorial on Message Sequence Charts. In: Computer Networks and ISDN Systems, 1996, 28(12),1629-1641

[EN01] Eisenberg, B. and Nickull, D. (2001). *ebXML Technical Architecture Specification v1.04*. Retrieved January 2005, from http://www.ebxml.org/specs/index.htm

[EN99] L. English. Improving *Data Warehouse and Business Information Quality*. John Wiley & Sons, 1999.

[F04] Ferrara, A. (2004). *Web services: a process algebra approach.* In: ICSOC. pp. 242–251, ACM, 2004.

[FB02] Fensel, D. and Bussler, C. (2002). The Web Service Modeling Framework WSMF. Electronic Commerce: Research and Applications, 1(2002), pp 113-137.

[FG92] G. Friedrich, G. Gottlob, W. Nejdl. *Formalizing the repair process*, Proc. of the 10th European Conference on Artificial Intelligence ECAI-92, Vienna, Austria pp. 709-713, 1992.

[GK03] Goodwin, P. and Kassem, N. (2003). *SOAP with Attachments API for Java (SAAJ) Specification v1.2.* SUN Microsystems (October 2003), http://java.sun.com/webservices/saaj/index.jsp

[GKMR+05] Graham, S., Karmarkar, A., Mischkinsky, J., Robinson, I., Sedukhin I., (editors): *Web Services Resource 1.2 (WS-Resource)*, OASIS Public Review Draft, 2005, available at http://docs.oasis-open.org/wsrf/wsrf-ws_resource-1.2-spec-pr-01.pdf

[GPS99] Grefen, P., Pernici, B. and Sanchez, G. (1999). *Database Support for Workflow Management.* The WIDE Project. Kluwer.

[GRC04] A. Grastien, M.-O. Cordier, C. Largouët. *Extending decentralized discrete-event modeling to diagnose reconfigurable systems*, working notes 15th International Workshop on Principles of Diagnosis DX'04, Carcassonne, France, pp. 75-80, June 2004.

[H05] Haller, A. (2005). D7.3v1.0 *Mission Statement – WSMX. WSMX.* Working Draft, January 2005. Retrieved December, 2004, from http://www.wsmo.org/2005/d7/d7.3/v1.0/20050109/

[HDMC+04] Hakimpour, F., Domingue, J., Motta, E., Cabral, L. and Lei, Y. (2004). *Integration of OWL-S into IRS-III.* In proceedings of the First AKT Workshop on Semantic Web Services (AKT-SWS04); KMi, The Open University, Milton Keynes, UK.

[HP02] Hewlett-Packard Company (2002). *Web Services Conversation Language (WSCL) 1.0.* W3C Note, March 2002. Retrieved December, 2004, from http://www.w3.org/TR/wscl10/

[HSS05] Hinz, S., Schmidt, K., and Stahl, S. (2005). *Transforming BPEL to Petri Nets.* In: W.M.P. van der Aalst et al. (Eds.): BPM 2005, LNCS 3649, pp. 220–235, Springer Verlag 2005.

[IBI] IBM Wbsphere Business Integration Server http://www.alphaworks.ibm.com/tech/bpws4j

[JHKK04] Jung, J., Hur, W., Kang, S. and Kim, H. (2004). *Business process choreography for B2B collaboration.* IEEE Internet Computing, Volume 8 , Issue 1 , Jan-Feb 2004, pp 37 – 45.

[JBK85] J. Bergstra and J. Klop:Algebra of communicating processes with abstraction, Theoret. Comput. Sci., 37 (1985), pp. 77–121

[KBRF+04] Kavantzas, N., Burdett, D., Ritzinger, G., Fletcher, T. and Lafon, Y. (2004). *Web Services Choreography Description Language Version 1.0.* W3C Working Draft, October 2004. Retrieved December, 2004, from http://www.w3.org/TR/ws-cdl-10/

[KN03] Khalaf, R. and Nagy, W.A. (2003). *Business Process with BPEL4WS: Understanding BPEL4WS, Part 7, Adding correlation and fault handling to a process.* Research report, IBM developerWorks, April 2003. Retrieved January, 2005, from

http://www-128.ibm.com/developerworks/webservices/library/ws-bpelcol7/

[L03] Langdon, C.S. (2003). *The state of Web services.* IEEE Computer, Volume 36, Issue 7, pp 93 – 94.

[L04] Leavitt, N. (2004). *Are Web services finally ready to deliver?* IEEE Computer, Volume 37, Issue 11, Nov. 2004, pp 14 – 18.

[LAZ03]G. Lamperti, M. Zanella. *Diagnosis of Active Systems*, Kluwer, Academic Publishers, 2003.

[LKDK+03] Ludwig, H., Keller, A., Dan, A., King, R. P. and Franck, R. (2003). *Web Service Level Agreement (WSLA) Language Specification, Version 1.0.* IBM Corporation (January 2003), *http://www.research.ibm.com/wsla*.

[LRT03] Leymann, F., Roller, D. and Thatte, S.(2003). *Goals of the BPEL4WS Specification.* http://xml.coverpages.org/BPEL4WS-DesignGoals.pdf

[M03a] Martin, D. (2003). *The OWL Services Coalition. OWL-S: Semantic Markup for Web Services.* White Paper. Retrieved December, 2004, from http://www.daml.org/services/owl-s/1.0/owl-s.html

[M03b] Mitra, N., Ed (2003). *SOAP Version 1.2 Part 0: Primer.* W3C Recommendation 24th June. http://www.w3.org/TR/2003/REC-soap12-part0-20030624/

[MAIS] MAIS (n.d.). MAIS project Home Page; retrieved January, 2005, from

http://black.elet.polimi.it/mais/

[MIC04]R. Micalizio, P. Torasso, G. Torta. On-line monitoring and diagnosis of multi-agent systems: a model based approach, *Proc. 16th European Conference on Artificial Intelligence ECAI'04*, Valencia, Spain, pp. 848-852, August 2004.

[MIL89] R. Milner: *Communication & Concurrency*. Prentice Hall, 1989

[MM04] Milanovic, N. and Malek, M. (2004). *Current Solutions for Web Service Composition*. IEEE Internet Computing, Volume 8, Issue 6, Nov.-Dec. 2004, pp 51 – 59.

[MMMP04] Maurino, A., Modafferi, S., Mussi, E. and Pernici, B. (2004). *A framework for provisioning of complex e-services*. IEEE International Conference on Services Computing (SCC 2004), Shanghai.

[MMY05] Maamar, Z., Mostefaoui, S.K. and Yahyaoui, H. (2005). *Toward an Agent-Based and Context-Oriented Approach for Web Services Composition*. IEEE Transactions on Knowledge and Data Engineering, Volume 17, Issue 5, May 2005, pp 686 – 697.

[MS] Microsoft Corporation (n.d.). Microsoft BizTalk Server; retrieved January, 2005, from http://www.microsoft.com/biztalk/

[NEB93]W. Nejdl, J. Bachmayer. Diagnosis and repair iteration planning versus n-step look ahead planning, *working notes of the 4th International Workshop on Principles of Diagnosis DX'93*, Aberystwyth, Wales, UK, 1993.

[OBS] Oracle BPEL process manager Project Official web-site: http://www.oracle.com/appserver/bpel_home.html

[P02] Paulson, L.D. (2002). *Choreographing web services.* IEEE Computer Volume 35, Issue 11, Nov. 2002, pp 25 – 25.

[P03a] Peltz, C. (2003). *Web services orchestration - a review of emerging technologies, tools, and standards*. Hewlett-Packard Company, 2003.

[P03b] Peltz, C. (2003). *Web services orchestration and choreography.* IEEE Computer, Volume 36, Issue 10, Oct. 2003, pp 46 – 52.

[PEM02]Y. Pencolé, M.-O. Cordier, L. Rozé. Incremental decentralized diagnosis approach for the supervision of a telecommunication network, *working notes of the 12th International Workshop on Principles of Diagnosis DX'01*, Sansicario, Italy, pp. 151-158, 2001a. Also *Proc.IEEE Conference on Decision and Control*, Las Vegas, Nevada, USA, pp. 435-440, 2002.

[PBM] Parasoft BPEL Maestro official site:

http://www.parasoft.com/jsp/products/home.jsp?product=BPEL

[PL05] Polleres, A. and Lara, R. (2005). *D4.1v0.1 A Conceptual Comparison between WSMO and OWL-S.* WSMO Working Draft, January 2005. Retrieved January, 2005, from http://www.wsmo.org/2004/d4/d4.1/v0.1/20050106/

[POS] Pexee Official web-site: http://pxe.fivesight.com/wiki/display/PXE/Home

[PP04] Pernici, B. and Plebani, P. (2004) *A Reasoned Introduction to the Web Services World* . UPGRADE. Vol. V, No. 6, December 2004. p 55.

[PR02] G. Provan. A Model-Based Diagnosis Framework for Distributed Embedded Systems, *Proc. 8th International Conference on Principles of Knowledge Representation and Reasoning KR'02*, Toulouse, France, pp. 341-352, April 2002.

[R96] Redman, T.C., ed. (1996). Data *Quality for the Information Age.* Artech House: Boston, MA, USA.

[RLK04] Roman, D., Lausen, H. and Keller, U. (2004). *D2v1.0. Web Service Modeling Ontology (WSMO).* WSMO Working Draft; September 2004. Retrieved January, 2005, from http://www.wsmo.org/2004/d2/v1.0/20040920/

[ROB02]N. Roos, A. ten Teije, A. Bos, C. Witteveen. An analysis of multi-agent diagnosis, *Proc. AAMAS'02*, Bologna, Italy, pp. 986-987, July 2002.

[SWZ00] G. Shankaranarayan, R. Y. Wang, and M. Ziad. *Modeling the Manufacture of an Information Product with IP-MAP.* In Proceedings of the 6th International Conference on Information Quality, 2000.

[SPP05] M. Scannapieco, E. Pierce, and B. Pernici. IP-UML*: Towards a methodology for quality improvement based on the IP-MAP framework. AMIS (Advances in Management Information Systems)* Monograph on Information Quality, 2005.

[ST01] S. Thatte: *XLANG - Web Services for Business Process Design*. Technical report, Microsoft Corporation, 2001

[SUW93]Y. Sun, D. Weld. A framework for model-based repair, *Proc. of the 11th National Conference on Artificial Intelligence AAAI-93*, Washington, DC, USA, pp.182-187, 1993.

[SWM04] Smith, M.K., Welty, C. and McGuinness, D.L. (2004). *OWL Web Ontology Language Guide.* W3C Recommendation, February 2004. Retrieved January 2005, from http://www.w3.org/TR/2004/REC-owl-guide-20040210/

[TOS] Twister official web-site: http://www.smartcomps.org/twister/

[TPP02] Tosic, V., Patel, K. and Pagurek, B. (2002). *WSOL - Web Service Offerings Language.* In: Workshop "Web Services, e-Business, and the Semantic Web (WES)". In conjunction with CAISE '02.

[TPPE+03] Tosic, V., Pagurek, B, Patel, K., Esfandiari, B and Ma, W. (2003). *Management Applications of the Web Service Offerings Language (WSOL).* CAiSE 2003, pp 468-484.

[UDDI] UDDI : http://www.uddi.org/

[UDDI-XML] UDDI Group, UDDI Version 2.0 XML Schema, http://www.uddi.org/schema/uddi_v2.xsd

[V04] Vinoski, S. (2004). *WS-Nonexistent Standards.* IEEE Internet Computing, Volume 8, Issue 6, Nov.-Dec. 2004, pp 94 – 96.

[VTKB03] Van der Aalst, W. M. P., ter Hofstede, A. H. M., Kiepuszewski, B. and Barros, A. P. (2003). *Workflow Patterns*. Distributed and Parallel Databases, 14(3), pp 5-51, July 2003.

[W04a] Vambenepe, W. (Ed.) (2004). *Web Services Distributed Management: Management Using Web Services (MUWS 1.0) Part 1*. Committee Draft, OASIS, December 2004. Available at: http://www.oasisopen.org/apps/org/workgroup/wsdm/download.php/10558/cd-wsdm-muws-part1-1.0.pdf

[W04b] Vambenepe, W. (Ed.). (2004). *Web Services Distributed Management: Management UsWeb Services (MUWS 1.0) Part 2.* Committee Draft, OASIS, December 2004. Available at: http://www.oasisopen.org/apps/org/workgroup/wsdm/download.php/10557/cd-wsdm-muws-part2-1.0.pdf

[WAN98] R. Wang. A Product Perspective on Total Data Quality Management. Communications of the ACM, 41(2), 1998.

[WF02] Weerawarana, S. and Francisco, C. (2002). *Business Process with BPEL4WS: Understanding BPEL4WS, Part 1, Concepts in business processes.* Research report, IBM developerWorks, Aug. 2002. Retrieved January, 2005, from

http://www-106.ibm.com/developerworks/webservices/library/ws-bpelcol1/

[WFMC] WfMC - Workflow Management Coalition (n.d.). Retrieved January, 2005, from http://www.wfmc.org

[WfXML] Workflow Management Coalition. (2001). *Workflow Management Coalition Workflow Standard - Interoperability Wf-XML Binding.*

[WS-BPEL 2.0] Arkin, A. et al. (2005). *Web Services Business Process Execution Language Version 2.0.* Committee Draft, 21st December, OASIS.

[WWC92] Wiederhold, G., Wegner, P. and Ceri, S. (1992). *Toward megaprogramming*. Communications of the ACM, Volume 35 , Issue 11, 1992, pp 89 – 99.

[W3C] W3C - World Wide Web Consortium (n.d.). Retrieved January, 2005, from http://www.w3.org

[XPDL] Workflow Management Coalition. (2002). *Workflow Process Definition Interface - XML Process Definition Language (XPDL).*