EXTENDING AND REFINING AN ABSTRACT OPERATIONAL SEMANTICS OF THE WEB SERVICES ARCHITECTURE FOR THE BUSINESS PROCESS EXECUTION LANGUAGE

by

Roozbeh Farahbod
B.Sc., Sharif University of Technology, 2001

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
in the School
of
Computing Science

© Roozbeh Farahbod 2004
SIMON FRASER UNIVERSITY
July 2004

All rights reserved. This work may not be
reproduced in whole or in part, by photocopy
or other means, without the permission of the author.

APPROVAL

Name: Roozbeh Farahbod

Degree: Master of Science

Title of thesis: Extending and Refining an Abstract Operational Semantics of the Web Services Architecture for the Business Process Execution Language

Examining Committee: Dr. Anoop Sarkar
Chair

____

Dr. Uwe Glässer, Senior Supervisor

____

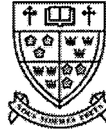
Dr. David Mitchell, Supervisor

____

Dr. Eugenia Ternovska, SFU Examiner

Date Approved: August 4. 2004

SIMON FRASER UNIVERSITY



Partial Copyright Licence

The author, whose copyright is declared on the title page of this work, has granted to Simon Fraser University the right to lend this thesis, project or extended essay to users of the Simon Fraser University Library, and to make partial or single copies only for such users or in response to a request from the library of any other university, or other educational institution, on its own behalf or for one of its users.

The author has further agreed that permission for multiple copying of this work for scholarly purposes may be granted by either the author or the Dean of Graduate Studies.

It is understood that copying or publication of this work for financial gain shall not be allowed without the author's written permission.

The original Partial Copyright Licence attesting to these terms, and signed by this author, may be found in the original bound copy of this work, retained in the Simon Fraser University Archive.

Bennett Library
Simon Fraser University
Burnaby, BC, Canada

Abstract

The Business Process Execution Language for Web Services (BPEL) is a forthcoming industrial standard for automated business processes, proposed by the OASIS[1] Web Services BPEL Technical Committee. BPEL is a service orchestration language which extends the underlying Web services interaction model and enables Web services to support long running business transactions.

We formally define an abstract operational semantics for BPEL based on the *abstract state machine* (ASM) paradigm. Specifically, we model the dynamic properties of the key language constructs through the construction of a *BPEL Abstract Machine* in terms of partially ordered runs of distributed real-time ASMs. The goal of our work is to provide a well defined semantic foundation for establishing the key language attributes by eliminating deficiencies hidden in the informal language definition.

This work combines two well-defined ASM refinement techniques to complement our previous efforts on the *core* model of the BPEL Abstract Machine. First, we elaborate the *core* model with regard to structural and behavioural aspects to make it more robust and flexible for further refinements. Specifically, we formalize the *process execution model* of BPEL and its decomposition into *execution lifecycles* of BPEL activities. We also introduce an *agent interaction model* to facilitate the interaction between different Distributed Abstract State Machine (DASM) agents of the BPEL Abstract Machine. We then extend the *core* model through two consecutive refinement steps to include *data handling* and one of the most controversial issues in BPEL, *fault and compensation handling.* The resulting abstract machine model provides a comprehensive formalization of the BPEL dynamic semantics and the underlying Web services architecture.

[1]Organization for the Advancement of Structured Information Standards

To my dear grandma Sareh,
for her utter love and her inspiring vision.

To Ozra and Nemat,
for their unconditional love and support.

"A fundamental new rule for business is that the Internet changes everything."

— BILL GATES

Acknowledgements

I am mostly grateful to my senior supervisor Dr. Uwe Glässer for his enthusiasm, friendship, and generous support and supervision throughout this work.

I would like to thank Mona, my beloved partner in life and a wonderful colleague, for her invaluable support and presence in every step of the way. I am also grateful specially to my parents and to my parents in law for their true love and continuous support and motivation.

I would also like to thank Dr. Eugenia Ternovska and Dr. David Mitchell for their valuable comments and suggestions for improvements of this thesis.

I would like to acknowledge the people in the School of Computing Science, the administrative and the technical support staff, and the Network Support Group at the Centre for Systems Science who together made this school a wonderful environment for research and graduate studies.

Contents

Approval ii

Abstract iii

Dedication iv

Quotation v

Acknowledgements vi

Contents vii

List of Tables xi

List of Figures xii

List of Specs xiii

Abbreviations and Acronyms xv

1 Introduction 1

1.1 Motivation 1
1.2 Objectives 3
1.3 Related Work 4
1.4 Significance of the Thesis 5
1.5 Thesis Organization 5
1.6 Notational Convention 6

2 Web Services and Business Processes **7**
2.1 Web Services Architecture . . . 7
2.1.1 Web Services Specifications . . . 10
2.2 Web Services Composition . . . 11
2.3 Overview of BPEL . . . 13
2.3.1 BPEL Activities . . . 14
2.3.2 Travel Agency: an Application Example . . . 15
2.3.3 Data Handling . . . 18
2.3.4 Correlation . . . 18
2.3.5 Long Running Business Transactions . . . 19

3 The Core of the BPEL Abstract Machine **20**
3.1 Distributed Abstract State Machines . . . 20
3.1.1 Concurrency and reactivity . . . 21
3.1.2 Real time behaviour . . . 21
3.1.3 Programs and rules . . . 22
3.1.4 Notational Conventions . . . 24
3.2 Overall Architecture . . . 24
3.3 The Formal Model . . . 27
3.3.1 Inbox Manager . . . 28
3.3.2 Outbox Manager . . . 28
3.3.3 Process . . . 30
3.3.4 Activity Agents . . . 31
3.4 Open Issues . . . 34

4 Elaborating the Core **36**
4.1 Refinement Notions . . . 37
4.1.1 The ASM Refinement Method . . . 37
4.1.2 Refinement Patterns and ASMs . . . 41
4.1.3 A Two Dimensional Refinement Approach . . . 43
4.2 Revising the Core . . . 45
4.2.1 Process Execution Model . . . 45
4.2.2 Agent Interaction Model . . . 48
4.2.3 Requirements Lists . . . 51

4.2.4 Input/Output Descriptors 54
4.2.5 Outbound Communication 55
4.3 Extensions to the Core 58

5 The Web Services Architecture of BPEL **60**
5.1 Data Handling Extension 60
5.1.1 Data Handling in BPEL 61
5.1.2 Requirements 62
5.1.3 Initial Definitions 64
5.1.4 Variables in Inbound/Outbound Communication 65
5.1.5 The Assign Activity 71
5.1.6 The Scope Construct 72
5.2 Fault and Compensation Extension 73
5.2.1 Fault and Compensation Handling in BPEL 73
5.2.2 Requirements 77
5.2.3 Process Execution Model: Fault Handling 77
5.2.4 Throwing Faults 83
5.2.5 Scope Agent: Refined 84
5.2.6 Fault Handling 88
5.2.7 Compensation Behaviour 90

6 Conclusion **100**
6.1 Validation 102
6.1.1 Termination Due to a Fault 102
6.1.2 Clarification on Activity Termination 103
6.1.3 Faults and the Compensate Activity 103
6.1.4 Invoking Compensation Handlers 104
6.1.5 Rethrowing a Fault 105

A Requirements Lists **106**
A.1 Receive-RL 106
A.2 Reply-RL 108
A.3 Data-RL 109
A.4 FC-RL 111

B The Revised Core **122**

B.1 Initial Definitions . . . 122

B.2 Programs . . . 133

C Data Handling Extension **147**

C.1 Initial Definitions . . . 147

C.2 Programs . . . 149

D Fault and Compensate Extension **152**

D.1 Initial Definitions . . . 152

D.2 Programs . . . 157

E Signaling **170**

E.1 Introduction . . . 170

F A Draft Proposal for Synchronized Request-Respond **172**

Index **182**

List of Tables

5.1 Requirement groups of fault and compensation handling in BPEL 78

List of Figures

2.1 Web services specifications 9
2.2 An example of a BPEL process: a travel agency 17

3.1 The composition of the BPEL service model and the network model 25
3.2 Sharpening informal requirements into executable specifications 26
3.3 High-level structure of the BPEL Abstract Machine 27
3.4 Control structure defined on DASM activity agents 32

4.1 The ASM Refinement Scheme 39
4.2 A two-dimensional refinement approach 43
4.3 Behavioural decomposition using incremental extensions 44
4.4 The process execution tree 46
4.5 Execution lifecycle of kernel agents: *core* 47

5.1 A *compensate* activity invokes the *compensation activity* of an enclosed scope 75
5.2 A compensation module cancels flight and room reservations 77
5.3 Activity execution lifecycle: extended by fault handling 80
5.4 Compensation handlers are invoked in their reverse order of completion . . . 92

List of Specs

3.1 The original inbox manager program . . . 29
3.2 The original process program of the *core* . . . 30
3.3 Behavioural specification of the *receive* activity . . . 32
3.4 The original program of flow agents . . . 34
4.1 The original SequenceProgram of the *core* . . . 51
4.2 The revised version of the SequenceProgram of the *core* . . . 52
4.3 Outbound communication behaviour in the original core model . . . 56
4.4 Revised Outbound Communication Behaviour . . . 57
5.1 Behaviour of the *receive* activity . . . 66
5.2 Inbound communication behaviour: Revised . . . 67
5.3 Extending inbound communication behaviour with data handling . . . 68
5.4 DeliverMessage in the *core* . . . 69
5.5 Extending outbound communication behaviour with data handling . . . 70
5.6 The behavioural definition of the *assign* activity . . . 71
5.7 Behavioural specification of the *scope* activity in data handling extension . . 72
5.8 Sequence program: extended by fault and compensation behaviour . . . 83
5.9 Extended specification of the *scope* activity in fault and compensation handling 88
5.10 Program of fault handler agents . . . 91
5.11 Program of compensate agents . . . 94
5.12 The ChooseNextCM rule is performed by compensate agents . . . 95
5.13 The behaviour of compensate agents in the *Running* mode . . . 96
5.14 The program of compensation handler agents . . . 99
F.1 Format of a *synchreceive* activity . . . 173
F.2 An example of using the *synchreceive* activity . . . 174

F.3 The *sequence* activity equivalent to the *synchreceive* example 175

Abbreviations and Acronyms

ASM	Abstract State Machine
BPEL	Business Process Execution Language for Web Services
BPEL4WS	Business Process Execution Language for Web Services
BPML	Business Process Modelling Language
DAML	DARPA Agent Markup Language
DARPA	Defense Advanced Research Projects Agency
DASM	Distributed Abstract State Machine
DRL	Data handling Requirements List
FCRL	Fault and Compensation handling Requirements List
HTTP	Hyper Text Transfer Protocol
HTTPS	Hyper Text Transfer Protocol Secure sockets (or Secure HTTP)
IDL	Interface Definition Language
IT	Information Technology
ITU-T	International Telecommunication Union - Telecommunication Standardization Sector
LRM	Language Reference Manual
SDL	Specification and Description Language
SMTP	Simple Mail Transfer Protocol
SOAP	Simple Object Access Protocol
OASIS	Organization for the Advancement of Structured Information Standards
TC	Technical Committee
UDDI	Universal Description Discovery Interface
URL	Uniform Resource Locater (world wide web address)

VHDL	VHSIC (Very High Speed Integrated Circuit) Hardware Description Language
W3C	World Wide Web Consortium
WSBPEL	Web Services Business Process Execution Language
WSCI	Web Service Choreography Interface
WSFL	Web Services Flow Language
WSDL	Web Services Description Language
XML	eXtensible Markup Language
XSD	XML Schema Definition

Chapter 1

Introduction

This thesis presents an abstract operational semantics for the Business Process Execution Language for Web Services (BPEL4WS), also called BPEL [4], based on the *abstract state machine* (ASM) paradigm [24]. We formally define a *BPEL Abstract Machine* in the form of a distributed real-time ASM (DASM) by modelling the dynamic properties of the key language constructs in terms of partially ordered machine runs.

BPEL is an XML based specification language for automated business processes. It provides expressive means for the process interface descriptions required for business protocols and executable process models. Version 1.1 of the BPEL language definition [4], henceforth called the LRM (language reference manual), is a forthcoming e-business standard proposed by OASIS[1]. As such, the language builds on other existing standards for the Internet and World Wide Web and, in particular, is defined on top of the service interaction model of W3C's Web Services Description Language, or WSDL [51]. A BPEL business process orchestrates the interaction between a collection of Web services exchanging messages over a communication network.

1.1 Motivation

IT organizations need the agility to respond to market changes, customer needs, and strategic requirements. In order to gain this agility, they need to streamline the information flow between different IT entities that perform the underlying business operations toward

[1]See the OASIS Web Services Business Process Execution Language Technical Committee (WSBPEL TC) web page at http://www.oasis-open.org.

obtaining an automated business process. This includes integrating both the organization's internal entities and those of its partners. Until recently, the price of integrating the IT entities of business partners with an organization's own entities has been very high. This was mainly due to the diversity of organization's proprietary interfaces and data structures [36].

Web services technology changed this situation by providing a platform-independent interface for application integration within an enterprise and between different enterprises. While Web services standards (like SOAP and WSDL[2]) facilitates simple integrations, business process specifications are required to specify various critical information of business processes, such as workflow, security requirements, and transaction management [36]. BPEL is proposed in this area to provide a language for formal specification of business processes and business interaction protocols [4].

To support an efficient integration of critical business processes, it is important to have *standard* business process specifications. To define such a standard for a business process language like BPEL, we need a precise specification of the language. While the LRM provides a comprehensive specification of this language, due to its natural language description, it is not void of ambiguities and inconsistencies. Our formal semantics is meant to complement the informal language description of the LRM by sharpening 'loosely defined' requirements into precise specifications. In this role, it serves as a robust mathematical framework for establishing the key attributes of BPEL in a well defined way; that is, by eliminating deficiencies – such as ambiguities, loose ends, and inconsistencies – that often remain hidden in the informal language definition (Issue #42, OASIS WSBPEL TC [35]):

> *There is a need for formalism. It will allow us to not only reason about the current specification and related issues, but also uncover issues that would otherwise go unnoticed. Empirical deduction is not sufficient.*

The abstract state machine (ASM) paradigm has been extensively used for formal specification of programming languages (e.g., Java [42], Prolog [6, 7]) and system modelling languages (e.g., SDL [39, 16, 22], VHDL [8, 9], SystemC [41]). The ASM formalism supports the integration of high-level modelling and analysis in the development cycle [11] which enables it to serve as a modelling basis in industrial standardization (e.g., ITU-T SDL-2000) [28]. Our work on BPEL builds on extensive experience from semantic modelling of various other industrial standards, including the IEEE language VHDL [9] and the ITU-T language

[2]See Section 2.1 for more details.

SDL [22]. The resulting SDL formal definition is part of the current ITU-T standard for SDL [28]. An important observation from this work is that the use of formal software models for practical purposes such as standardization demands for a gradual formalization of abstract requirements with a degree of detail and precision as needed [23]. To avoid a gap between the informal language definition and the formal semantics, the ability to model the language definition *as is*, without making any compromises, is often crucial. Practicability of the formalization approach further demands for flexibility and robustness as required for dealing with a moving target as standardization is a potentially open-ended activity.

1.2 Objectives

The goal of our work is to build from requirements elicitation and clarification a *ground model* ASM for BPEL. Intuitively, a ground model is an accurate yet abstract description which is [11, 5],

- *precise* at an appropriate level of abstraction,
- *flexible* for future changes and modifications,
- *understandable* by both domain experts and system designers,
- *complete* in the sense that every semantically relevant feature of the language is present,
- *operational* for validation through simulation and testing, and
- has a *precise semantic foundation.*

From such a ground model ASM, a hierarchy of intermediate models can be obtained through stepwise refinement which can eventually lead to an executable implementation of the language.

This work combines two well-defined ASM refinement techniques to complement our previous efforts on the core model of the BPEL Abstract Machine [18, 20, 19, 43]. First, we elaborate the core model with regard to structural and behavioural aspects to make it more robust and flexible for further refinements. Specifically, we formalize the *process execution model* of BPEL and its decomposition into *execution lifecycles* of BPEL activities. We also introduce an *agent interaction model* to facilitate the interaction between different

DASM agents of the BPEL Abstract Machine. We then extend the core model through two consecutive refinement steps to include *data handling* and one of the most controversial issues in BPEL, *fault and compensation handling.* Business processes normally involve long duration transactions which are based on asynchronous message communication that leads to a number of local updates at business partners. Handling faults and cancelling transactions in business processes often involve undoing partial work that is done in collaboration with different partners. BPEL provides its own method of handling faults and dealing with compensating activities, which is captured in the BPEL Abstract Machine by the *fault and compensation extension.*

Finally, the resulting abstract machine model provides a comprehensive formalization of the BPEL dynamic semantics and the underlying Web services architecture.

1.3 Related Work

There are various research activities applying formal methods to define, analyze, and verify the Web services composition languages. Closely related to our work is an approach of a group at Humboldt University in Berlin. This group is working on formalizations of BPEL for analysis, graphics and semantics using various modelling techniques. In [17], Dirk Fahland outlines an ASM model of the dynamic semantics of BPEL which is very similar to our view; however, their formalization just sketches this ASM model in terms of only two BPEL activities (*reply* and *sequence*). Alternatively, this group also proposed Petri-net models of Web services to analyze essential properties like usability [30, 32] and to address the composition problem of Web services [46].

Formal verification of Web services is addressed in [31] and [29]. The approach in [31] is based on Petri nets, while [29] uses a process algebra approach to derive a structural operational semantics of BPEL as a formal basis for building a tool to verify properties of the specification.

Narayanan and McIlraith provide a model-theoretic semantics (based on situation calculus) and a distributed operational semantics (based on Petri nets) for the DAML-S language [2], a DAML-based Web service ontology language, which facilitates simulation, composition, testing, and verifying compositions of Web services [34].

Various research have been done to evaluate the capabilities and limitations of different languages proposed for Web services composition. Notably, van der Aalst et al. presented

a pattern-based analysis of BPEL [45], and BPML and WSCI [44] based on a collection of workflow and communication patterns which allows comparing the capabilities and limitation of these languages.

All these attempts either focus on approaches that are completely different from our approach, or provide models that are by far not as comprehensive as our model. Many of them are applying other formal methods pursuing different goals (e.g., pure verification) [31, 29, 45] or are not specifically focused on BPEL [31, 34]. To the best of our knowledge and based on existing publications, our work is the most comprehensive formal definition of BPEL semantics.

1.4 Significance of the Thesis

There is substantial industrial interest in the development, standardization, and implementation of BPEL. Hence, it is important to have a precise and reliable underlying semantic definition for the language. In this thesis, we present the most comprehensive formal semantics specification of BPEL based on a practical formal method that has a history of successful applications in industrial standardization [42, 6, 7, 39, 16, 22, 8, 9, 41, 28]. We address a number of inconsistencies, loose ends, and ambiguities in the informal definition of the language, as examples of how such a formal specification can support validation of the language definition in a way that effectively reveals weaknesses. Furthermore, this formal specification forms a basis for deriving an abstract executable semantics for BPEL that facilitates experimental validation through simulation and testing.

1.5 Thesis Organization

The thesis starts by introducing the Web services architecture, and provides an overview of BPEL in Chapter 2. Our original work on the core model of BPEL is briefly presented in Chapter 3 followed by a list of open issues. Chapter 4 substantially improves the core model to build a more robust and flexible foundation for further refinements. Chapter 5 extends the core model to build a comprehensive model of BPEL through two refinement steps by presenting the *data handling extension* and the *fault and compensation extension.* Chapter 6 concludes the thesis and provides a critical analysis of BPEL by addressing a number of weak points (loose ends, inconsistencies, etc.) in the LRM.

1.6 Notational Convention

As we frequently refer to parts of the LRM (specially in the requirements lists in chapters 4 and 5), we use the $ sign followed by a section number to refer to a section of the LRM [4]. For instance, '[$14.2]' refers to section 14.2 of the LRM.

When he [was] tired of official reports
and memoranda and minutes, he would
plug his foolscap-size Newspad into the
ship's information circuit and scan the
latest reports from Earth. One by one
he would conjure up the world's major
electronic papers...

Arthur C. Clarke, 1968

Chapter 2

Web Services and Business Processes

The World Wide Web, or *the Web* for short, has been serving us for more than a decade since 1993 when it started to become popular. As a human-to-machine interface of a computer-based network of information, it has provided a platform to share a variety of information in multimedia formats. Recently, efforts have been made to use the Web as a machine-to-machine interface through the notion of *Web services*.

This chapter starts with an overview of the Web services architecture, various specifications, and standard protocols that are designed and published by major IT vendors (Section 2.1). Section 2.2 introduces the notions of orchestration and choreography of Web services. The rest of this chapter then provides an overview of the Business Process Execution Language for Web Services (BPEL).

2.1 Web Services Architecture

What is a Web service? In a white paper published by IONA Technologies[1], a world leader in high performance integration solutions for IT environments, a Web service is defined as follows [40]:

[1] IONA Technologies PLC, http://www.iona.com

> *Simply put, a Web service is a software construct that exposes business functionality over the Internet. In the context of a Web service, "expose" means:*
>
> - *Identifying valuable business processes within the enterprise.*
> - *Defining loosely-coupled, service-oriented interfaces to those processes.*
> - *Describing those interfaces in a Web-based, industry-standard format.*

For more than a decade, the Web has been providing us with a way to share and distribute information, and it well served as a human-to-application (machine) interface. Today, with the development of electronic marketplaces and automated business-to-business transactions, the Web is also used as an application-to-application interface. In this new domain with large heterogeneous systems, *interoperability* becomes one of the most critical problems that software developers and business partners should deal with [13, 21]. Interoperability is generally defined as the ability of a system to use the parts or equipment of another system[2]. In the IT environment, interoperability is mostly about the ability of exchanging information with other systems. Web services, like many other distributed systems, are built upon the following two fundamental building blocks:

- *interoperability at the data exchange level*, which is provided by means of a simple, standard, and platform independent data exchange protocol, and
- *a unified functional representation of applications*, which can be achieved by an interface definition language (IDL).

Curbera et al. emphasize three key aspects in defining Web services: *interoperability*, *common representation*, and *heavy reliance on standards*, where the first two address the two fundamental building blocks mentioned above. Based on these aspects, they provide the following definition for a Web service [13]:

> *A Web service is a networked application that is able to interact using standard application-to-application Web protocols over well defined interfaces, and which is described using a standard functional description language.*

As a distributed computing platform to integrate a heterogeneous mix of platforms and programming models, it is important for Web services to converge to a small set of

[2]Merriam-Webster dictionary

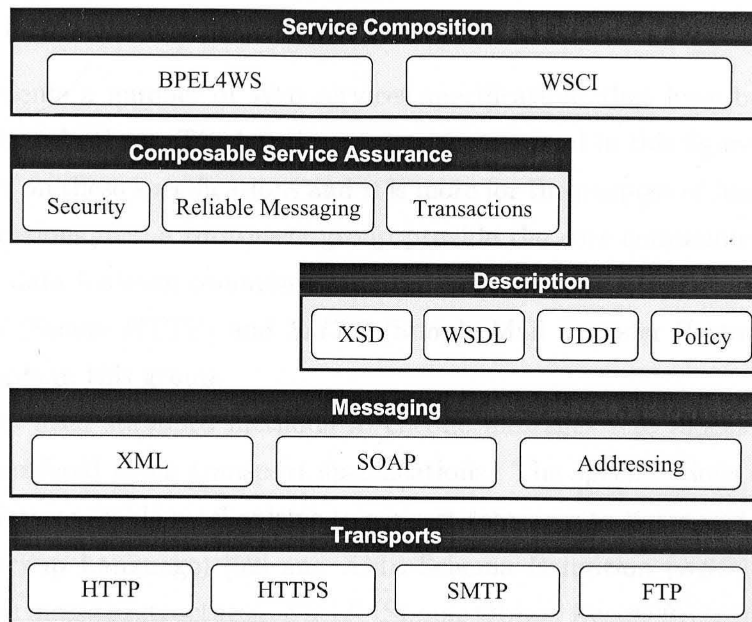


Figure 2.1: Web services specifications

well defined standards but not to become a common programming model. Considering the standards, the Web Services Architecture Working Group at the World Wide Web Consortium[3] gives a more specific definition of a Web service for the purpose of their working group and their proposed architecture [54]:

> *A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.*

In the following section we explore some of the pervasive standards and specifications related to the Web services architecture.

2.1.1 Web Services Specifications

Figure 2.1[4] presents a number of Web services specifications that have been published by Microsoft, IBM, and others. The layering structure presented in this figure does not impose an a priori order on these specifications and it is more for the purpose of functional grouping.

The specifications in the *transports* group provide the core communication mechanism to transfer raw data between communication endpoints. *HTTP* (Hyper Text Transfer Protocol), *HTTPS* (Secure HTTP) and *SMTP* (Simple Mail Transfer Protocol) are the most popular standards in this group.

Web services need standard methods to encode messages into blocks of bytes so that they can be transfered using transport specifications. The specifications in the *messaging* group provide interoperable mechanisms to convert messages to bytes and vice versa. *XML* (eXtensible Markup Language) [52] and XML Schema Definition (*XSD*) [48] are used to abstractly define message structures. *SOAP* (Simple Object Access Protocol) [50] provides a standard mechanism to encode XML messages into bytes that can be transfered by transport protocols.

Web Services Addressing (WS-Addressing) [3] provides an interoperable, transport independent mechanism to identify sender and receiver of messages. Today, most systems are using the same addressing mechanism that browsers and HTTP-servers are using over the HTTP transport. The sender specifies the destination of its message by placing a URL in the HTTP transport. The receiver finds the address of the sender by the return transport address. In this method the address information of the sender and receiver are not part of the message, which can cause communication problems (e.g., this information can be lost due to a timeout). WS-Addressing separates the address information from the underlying transport protocol by placing this information in the message without altering the message information. With this method, addressing information is not limited by the transport protocol. For example, when using HTTP without WS-Addressing, a response always goes back to the sender and a different destination cannot be specified. With WS-Addressing, however, one can specify that a response to a message should be sent to a communication endpoint different from the sender [3, 21].

---

[3]W3C, http://www.w3c.org

[4]The original idea of this figure is taken from [21].

The specifications mentioned so far support communication of Web services using messages. Nevertheless, before Web services can communicate using messages, they need to know what these messages are. A well-defined standard method is required for a Web service to document the structure of messages and describe the message-exchange patterns of the Web service (i.e., the interface to the Web service). The specifications in the Description group (Figure 2.1) enable a Web service to document and describe its service capabilities and its interface to the outside world.

XSD enables Web services to define XML data types which can be used in defining message structures. Web Services Description Language (*WSDL*) supports documenting and describing message structures (using XSD) and basic message interaction patterns of Web services. WSDL provides the following message interaction patterns [51]:

- input-only (receiving a message),
- input-output (receiving a message and sending a correlated message),
- output-input (sending a message and receiving a correlated message), and
- output-only (sending a message).

WSDL supports describing the interface of a service, but how do potential partners find this information? Currently there are two methods available. A potential user can either

- access a Web service and get all the required information about the service using *WS-MetadataExchange*, or
- use a *UDDI* (Universal Description Discovery Interface) service.

The UDDI specification defines a meta-data aggregation service that enables organizations to publish the services they provide and describe the interface to their services for potential users. Potential users then can query the UDDI service at design time or even at runtime to find services that fulfil their requirements.

2.2 Web Services Composition

While the Web becomes a widespread platform for automated application-to-application interactions and integration of business-partner applications, *Web services composition* becomes an important issue. System integration is much more than just a series of stateless

transactions. While transport standards, messaging standards and service description languages provide the underlying platform for automated application-to-application (service-to-service) interactions, composition protocols are required to enable integration of services within and across organization boundaries [4]. In today's fast growing electronic market, IT organizations need the agility to adapt to market changes and customer requirements. While the existing business process languages are not suitable for Web services, these organizations may define their own proprietary protocols for Web services composition which conceptually contradicts one of the key aspects of the Web services architecture, namely interoperability. When organizations build their own orchestration protocols and languages, integration of services from different organizations requires creation of new protocols or adaptation of organizations to their partners' proprietary protocols [38].

Leading companies in IT have been putting substantial effort into specifying standard protocols for Web services composition. Sun Microsystems, SAP and others proposed the Web Services Choreography Interface (WSCI), an XML-based interface description language that describes the message exchange of a Web service that participates in a collaborative interaction with other Web services [49]. IBM, Microsoft, BEA and others are proposing the Business Process Execution Language for Web Services (BPEL4WS), an XML-based business process language that provides a grammar to coordinate Web services interacting in a business process flow[5] [4].

There are two basically different types of Web services composition: *orchestration* and *Choreography*. Orchestration describes how Web services interact with each other through a message exchange flow. In orchestration, the overall process control is centred within one business partner. BPEL4WS is an example of an orchestration language. In contrast, choreography is more collaborative in nature. While there is no centric control over the entire process, each business partner in a choreography composition knows its part in the business interaction and message exchange flow. WSCI is an example of a choreography language [38, 4, 49].

As this work is focused on BPEL, the rest of this chapter provides an overview of this language and describes an example of a Web services composition using BPEL.

[5]Other Web services composition languages are also available which are not addressed here.

2.3 Overview of BPEL

The Business Process Execution Language for Web Services (BPEL) introduces a stateful interaction model that allows Web services to exchange sequences of messages between business partners. A BPEL process and its partners are defined as abstract WSDL services using abstract messages as defined by the WSDL model for message interaction. The major parts of a BPEL process definition consist of (1) *partners* of the business process (Web services that this process interacts with), (2) a set of *variables* that keep the state of the process, and (3) an *activity* defining the logic behind the interactions between the process and its partners. Activities that can be performed by a business process are categorized into *basic* activities, *structured* activities and *scope-related* activities. Basic activities perform simple operations like *receive*, *reply*, *invoke* and others. Structured activities impose an execution order on a collection of activities and can be nested. Scope-related activities enable defining logical units of work and delineating the reversible behaviour of each unit.

Business processes in BPEL can be described in two ways: *executable business processes* and *business protocols*. Executable processes model actual behaviour of a participant in a business interaction. Business protocols, however, do not describe the internal behaviour of business partners and only specify the visible message exchange behaviour between them. The process descriptions for business protocols are called *abstract processes*.

In April 2003, members of OASIS[6], including IBM and Microsoft among other leading companies in the e-business market, formed a Technical Committee in order to continue work on BPEL version 1.1 with the *"focus on specifying the common concepts for a business process execution language which form the necessary technical foundation for multiple usage patterns including both the process interface descriptions required for business protocols and executable process models."*[7]

In the following sections, we provide a brief overview of the BPEL activities and describe a simple application example. We then introduce the BPEL notions of fault handling and compensation behaviour which are of fundamental importance for the business process execution model.

[6]Organization for the Advancement of Structured Information Standards

[7]WSBPEL TC at OASIS, http://www.oasis-open.org

2.3.1 BPEL Activities

Basic activities perform simple Web services operations, including *receive*, *reply*, *invoke*, *assign*, *throw*, *terminate*, *wait*, and *empty*. Structured activities include *sequence*, *switch*, *flow*, *pick* and *while*. Scope-Related activities include *scope* and *compensate*. A short overview on each of these activities is presented in the following:

Receive The *receive* activity has an important role in the lifecycle of a business process. It provides both a fundamental Web services operation (which is receiving a message from a partner) and triggers the creation of new instances a of business process. If the *createInstance* attribute of a *receive* activity is set to 'yes', the *receive* activity is regarded as a *start activity*; i.e., whenever a message arrives for such a *receive* activity, a new instance of the business process must be created and the message must be assigned to the new instance.

Reply A *reply* activity must be defined in connection with a *receive* activity identifying a synchronous request-response interaction between two business processes. Thus, a *reply* activity sends a message to a partner in response to a request from this partner which was previously received by the associated *receive* activity.

Invoke A business process can access services provided by its partners by invoking an operation on such a service. An *invoke* activity can be used for invoking both synchronous and asynchronous operations of other Web services.

Wait The *wait* activity is used to introduce a delay in the business process execution. A *wait* activity identifies that a business process has to wait either *for* a period of time or *until* a certain deadline is reached.

Empty An *empty* activity does nothing. It is usually used in cases when a fault needs to be caught and suppressed without a reaction.

Assign An *assign* activity allows updating the state of a business process by copying data from one variable to another. It also allows performing simple computations assigning the value of an expression to a variable. The *assign* activity is part of the data handling mechanism provided by BPEL (see Section 2.3.3 for further details).

Throw A *throw* activity is used by a business process to generate an internal fault explicitly. A fault is identified by a globally unique name. An optional fault variable can also be defined. This variable contains further information on the fault and can be used by the fault handler to analyze the fault.

Sequence A *sequence* activity is a structured activity that enforces a sequential execution order on a collection of activities.

Switch A *switch* provides the ability to choose among a collection of activities. An ordered list of conditional branches (*case* elements) followed by an optional *otherwise* branch are defined in a *switch* activity. The first branch whose condition holds is chosen and its associated activity is executed. The *otherwise* branch is taken only if none of the conditional cases are true.

While A *while* activity iterates an activity while a certain condition holds.

Flow A *flow* activity enables concurrent execution of a set of activities. The concurrent execution is controlled by *synchronization dependencies* between the activities. Such dependencies are identified by execution *links* between activities.

Pick A *pick* activity awaits the occurrence of one event from a set of events and executes the associated activity to that event. If more than one event occurs then the pick activity will choose the first one that has occurred. As soon as an event is chosen, the pick activity no longer accepts any of the other events. Basically, there are two types of events on which *pick* activities can wait: *onMessage* events and *onAlarm* events. The semantics of an *onMessage* event is very similar to a receive activity. An *onMessage* event occurs as soon as its corresponding message is received. *OnAlarm* events are very similar to timers. They are triggered after a period of time or when a certain time deadline is reached.

2.3.2 Travel Agency: an Application Example

We provide here a simple example to illustrate the basic structures and some fundamental concepts of BPEL.

Suppose an online travel agency system (a Web service) that facilitates trip planning. For the sake of simplicity this travel agency has only three business partners: an airline

company, a hotel, and a credit card company. The process is simple: a client connects to the travel agency Web service and provides a suggested trip plan. The travel agency then books a round-trip flight based on the suggested dates and also reserves a hotel room for the period of the stay. It then sends a draft itinerary to the client. To purchase the tickets and finalize the the sale, the client then sends credit card information to the travel agency. The travel agency charges the credit card and returns a final itinerary back to the client[8]

From the client point of view there is only one Web service that provides the trip planning service. This Web service, however, is a composition of a number of Web services: the airline, the hotel, the credit card company, and the travel agency itself as the orchestrator. Figure 2.2 illustrates an abstract schema of the business process of this travel agency. The business process consists of seven basic activities, two of which being executed concurrently (as indicated by identical order numbers annotating these two activities). For each client the process execution is as follows:

1. The process starts with receiving a suggested schedule from the client using a *receive* activity.

2. The schedule is sent simultaneously to the airline Web service and the hotel Web service using two *invoke* activities. The simultaneous invocation is made by means of a *flow* activity that surrounds the *invoke* activities. The parallel execution path in the figure represents the *flow* activity.

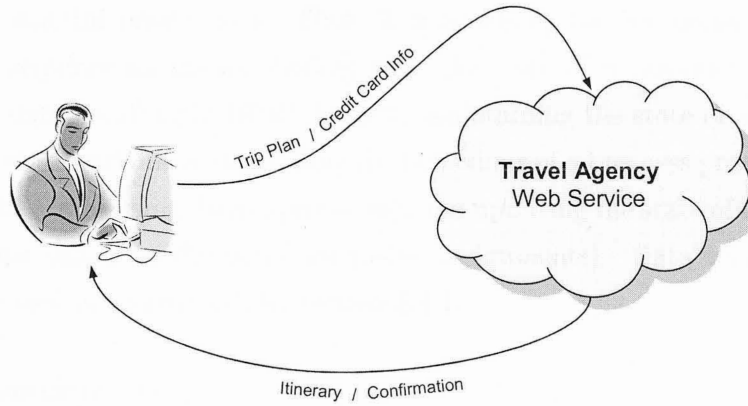
3. Based on the responses received from the airline and the hotel, a draft itinerary is returned to the client using a *reply* activity.

4. Upon receiving a confirmation from the client together with its credit card information, this information is sent to the credit card company. After the card is successfully charged, the final itinerary is sent to the client.

Later we will get back to this example to discuss other aspects of BPEL like fault handling and compensation behaviour.

---

[8] A real-life business process for a travel agency Web service is certainly more complicated and it involves a number of back and forth interactions with the client and its business partners.



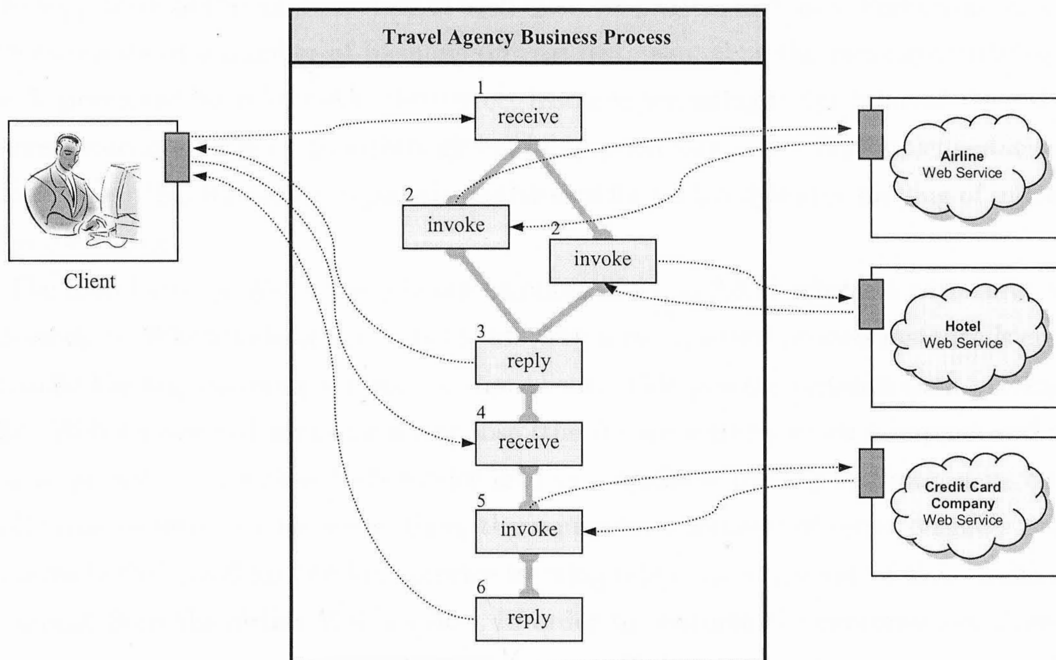


Figure 2.2: An example of a BPEL process: a travel agency

2.3.3 Data Handling

One of the main challenges in integrating Web services, and specifically business processes, is to deal with stateful interactions. Thus, it is necessary for any orchestration language to provide the required means for dealing with the state of a business process instance. The concept of data handling in BPEL includes maintaining the state of a business process instance (using state variables), controlling the behaviour of a business process by extracting the data from the state (using data expressions), and updating the state of a business process by assigning new values to the variables (using assignments). Data handling features of BPEL are discussed in more detail in Section 5.1.1.

2.3.4 Correlation

Business processes normally act according to a history of external interactions. The data handling features of BPEL facilitate dealing with stateful interactions by providing the ability to keep track of the internal state of each business process instance. Furthermore, a Web service consists of a number of business process instances, thus the messages arriving at a specific port must be delivered to the correct instance according to the internal state of such business process instance. To ensure global interoperability and avoid implementation dependencies, BPEL introduces a generic mechanism for such a dynamic binding of messages, called *correlation.*

The travel agency Web service in our example (Section 2.3.2) interacts with three other Web services. When a client connects to this Web service, a new process instance is created to handle the trip planning service for that client. This process instance then invokes the airline Web service and waits for a response (the *invoke* activity sends a request and waits for a response). The airline Web service creates a business process instance of its own to handle this request. At the same time, there may be a number of other business process instances in the travel agency Web service (serving other clients), some of them waiting for a response from the airline Web service. In order to continue the conversation, these two business process instances (one in the Travel agency Web service and the other in the airline Web service) need to *know* each other; i.e., when a response comes from a business process instance in the airline Web service to the travel agency, it must be delivered to the correct business process instance within the travel agency Web service. The mechanism supported by BPEL is to carry business tokens in all the messages belonging to a conversation so that

the destination of the messages can be recognized by the value of these business tokens. BPEL allows a business process to define a set of business tokens (*correlation tokens*). This set is called a *correlation set.* Once a correlation set is initiated for a conversation, all the messages in the conversation must carry the same correlation token values. In this way, an application-level conversation between business process instances is established. In our example, the name of the client along with the request time could be used as correlation tokens in message interactions. The travel agency Web service, upon receiving messages from its partners, will assign the messages to their corresponding process instances within the Web service based on the values of the these correlation tokens.

2.3.5 Long Running Business Transactions

Business processes normally involve long running transactions with non-negligible duration which are based on asynchronous message communication. Such transactions lead to a number of local updates at business partners. Consequently, when an error occurs, it may be required to reverse the effects of some or even all of the previous activities. This is known as *compensation.* The ability to compensate the effects of previous activities in case of an exception enables business processes to have so-called Long-Running (Business) Transactions (LRTs).

In BPEL, compensation and fault handling is performed using the *scope* activity. A *scope* activity defines a logical unit of work for which a *compensation handler* or a set of *fault handlers* can be defined. A compensation handler defines the compensating behaviour of a logical unit in case of an error. A fault handler defines the reaction of a logical unit to an error. The fault handling mechanism and compensation behaviour of BPEL are discussed in detail in Section 5.2.

Chapter 3

The Core of the BPEL Abstract Machine

This chapter presents our previous work [18, 20, 19, 43] on modelling the Business Process Execution Language for Web Services, in which we built the *core* model of the BPEL Abstract Machine. Focusing on the key aspects of the core concepts of the language, we formally define an abstract operational semantics based on the Distributed Abstract State Machine (DASM) paradigm in [18, 20, 19]. A comprehensive presentation of this work along with the complete formal model of the *core* is provided in [43]. The *core* model is extensively refined and extended in Chapters 4 and 5 to build a comprehensive semantic model that captures all different aspects of the language.

Section 3.1 provides a brief introduction to DASMs. The overall structure of the *core* model is then presented in Section 3.2. A brief overview of the complete formal model is presented in Section 3.3. Major open issues and possible further developments are then discussed in Section 3.4.

3.1 Distributed Abstract State Machines

This section briefly recalls the concept of distributed real-time ASM at an intuitive level of understanding and in a rather informal style. For a rigorous mathematical definition, we refer to the existing literature on the theory of ASMs [24, 25] and their applications [23, 11].

A DASM M is defined over a given vocabulary V by its program P_M and a non-empty

set I_M of initial states. V consists of some finite collection of *function symbols* and *predicate symbols*, each of a fixed arity. States of M are variants of many-sorted structures that express predicates in terms of their characteristic functions. Initial states yield valid interpretations of V.

M has a finite set AGENT of autonomously operating *agents*. The set of agents changes dynamically over runs of M as required to model varying computational resources. The behaviour of an agent a in a given state S of M is defined by its program $program_S(a)$. To introduce a new agent a in state S, a valid program has to be assigned to $program_S(a)$. To terminate a, $program_S(a)$ is reset to the distinguished value *undef* (not representing a valid program). In any state S reachable from an initial state of M, the set of agents is well defined as

$$\text{AGENT}_S \equiv \{x \in S : program_S(x) \neq undef\}.$$

The statically defined collection of all the programs that agents of M potentially can execute forms the distributed program P_M.

3.1.1 Concurrency and reactivity

Intuitively, agents of M model the concurrent control threads in an execution of P_M. They interact with each other by reading and writing shared locations of global machine states, where the underlying semantic model regulates such interactions so that potential conflicts are resolved according to the definition of *partially ordered runs* [24].

Interactions between M and its operational environment are restricted to actions and events as observable at well identified interfaces. The environment affects computations of M through externally controlled or *monitored* functions. Such functions change their values dynamically over runs of M, even though they cannot be updated by agents of M.

3.1.2 Real time behaviour

Real time behaviour imposes additional constraints on DASM runs ensuring that the agents react instantaneously [26]. Timing aspects are modelled based on an abstract notion of global system time. In a given state S of M, the time (as measured by some global clock) is given by a monitored unary function now_S taking values in a linearly ordered domain TIME. Time values are represented as positive real numbers. One can assume the values of *now* to increase monotonically over runs of M.

3.1.3 Programs and rules

A program is defined through a parallel composition of state transition rules. The canonical rule consists of a basic update instruction of the form $f(t_1, t_2, ..., t_n) := t_0$ where f is an n-ary function symbol and the t_i's $(0 \leq i \leq n)$ are terms. Complex rules are inductively formed by means of rule constructors. Two conventional rule constructors are the *block constructor* and the *conditional constructor*.

The *block constructor* is a collection of ASM rules $R_1 \ldots R_n$. To fire a block constructor, all the rules $R_1 \ldots R_n$ are fired simultaneously. This construct has the following form:

R_1
R_2
...
R_n

The *conditional constructor* has the form

---

if e **then**
  R_1
else
  R_2

---

where e is a Boolean term and R_1 and R_2 are ASM rules. If e is evaluated to *true* then R_1 is fired, otherwise R_2 is fired.

Non-determinism is often useful to abstract away from describing details of algorithms. Non-determinism is introduced in ASMs by the *choose* constructor in the following form:

---

choose $u \in U$ **with** φ
  $R(u)$

The meaning of this constructor is to fire rule R with an arbitrary u chosen from U satisfying φ. If there is no such u, nothing is done [24, 11].

When sequential execution is not required, parallelism (simultaneous execution of a rule) is a useful tool to abstract from sequentiality. The *forall* constructor in ASM provides simultaneous execution of a rule R for each u in U that satisfies a given condition φ. This

constructor has the following form:

---

forall $u \in U$ **with** φ
$R(u)$

In describing an algorithm, it is often required to dynamically allocate additional resources by introducing new elements. In ASMs, the *import* constructor, operating on a potentially infinite *reserve* set, provides allocation of new elements. The *import* constructor of the form

---

import u
$R(u)$

---

chooses an element u from the reserve set, removes it from the reverse set and fires rule R. The reverse set of a state cannot directly be updated by an ASM rule but is updated automatically through execution of an *import* constructor. The elements of the reserve set of a state are not allowed to be in the domain or range of any basic function of the state [11].

To extend a domain with a new element, we use the following notation:

---

extend U **with** u
$R(u)$

---

which imports a new element, binds the variable u to the newly imported element, adds u to the domain U, and fires rule $R(u)$.

To facilitate creation and termination of a given agent a of domain A, we introduce the two abbreviations *new* and *stop* in the following form:

---

new $a : A$
$R(a)$

stop a

The *new* operation creates a new agent a of domain A and sets *program*(a). It also adds agent a to the associated domain of agents. The *stop* operation removes agent a from the associated domain of agents and resets *program*(a) to *undef*.

To allow for partial updates of sets [27], the following operations are used to insert an element a into or remove a from a given set A.

---

add a **to** A

remove a **from** A

Finally, the reserved function symbol *self* has a special role: in a program (or rule) it refers to the agent executing the program (or rule).

3.1.4 Notational Conventions

The ASM specifications presented in this document use the following notational conventions for improved readability.

- Agent program names and ASM rule names start with a capital letter. The individual words also start with capital letters and there is no separator between individual words (e.g., ProgramName).
- The first time a program or a rule is defined, its name appears in boldface (e.g., **ProgramName**).
- Function names start with a lowercase letter. The individual words start with capital letters and the rest of the letters are written in lowercase (e.g., *functionName*).
- ASM keywords are written in lowercase using bold font (e.g., **else**).
- Domains are written in all capital letters and the individual words are separated by underscore '_' (e.g., DOMAIN_NAME).
- ASM specifications in the text are separated from the enclosing text by two horizontal lines: a thick line (———) indicating the start of the specification and a thin line (———) indicating its end.

3.2 Overall Architecture

This section introduces the overall architecture of the *core* model of the BPEL Abstract Machine in terms of a distributed real-time ASM. Logically, the BPEL Abstract Machine

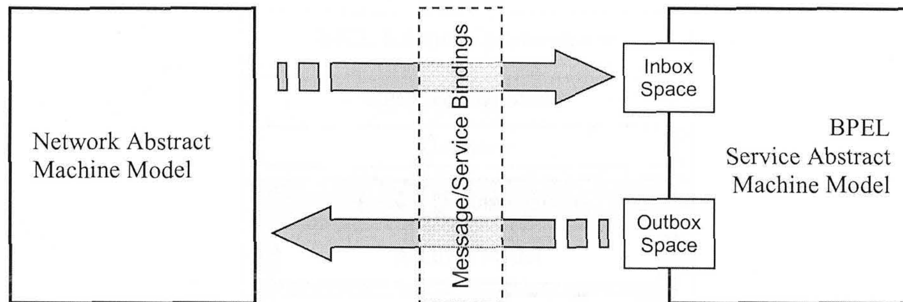


Figure 3.1: The composition of the BPEL service model and the network model

splits into a *service abstract machine* and a *network abstract machine*, so that the behaviour of the communication network is clearly delineated from that of the BPEL processes residing at the communication endpoints.

We focus on the service abstract machine model which captures the behaviour of services residing at communication endpoints while an executable ASM model of the network abstract machine is defined in [23]. Any interaction between these two models occurs at well-defined interfaces facilitating the composition of two models into the BPEL Abstract Machine (see Figure 3.1). Henceforth, we use the term BPEL Abstract Machine to refer to the service abstract machine.

The *core* model formalizes the key functional attributes of the BPEL Web services architecture based on the asynchronous computation model of distributed ASMs [24]. The primary focus is on dynamic process creation/termination, Web services communication primitives, message correlation, concurrent control structures, and core BPEL activities including *receive*, *reply*, *invoke*, *wait*, *empty*, *terminate*, *sequence*, *switch*, *while*, *pick* and *flow*. This model does not deal with data handling issues, faults or compensation behaviour, so does not include *assign*, *throw*, *scope*, and *compensate*.

The BPEL Abstract Machine architecture is organized into three layers of abstraction, called the *abstract* model, *intermediate* model and *executable* model, as illustrated in Figure 3.2. The abstract model formally sketches the behaviour of the key BPEL constructs. The intermediate model is the result of the first refinement step and provides a complete formal model of the key BPEL constructs. Finally, the executable model provides an abstract executable semantics of BPEL implemented in AsmL [33]. A graphical user interface (GUI) facilitates experimental validation through simulation and animation of abstract machine runs. Thus, the BPEL Abstract Machine forms a hierarchically defined ground model DASM

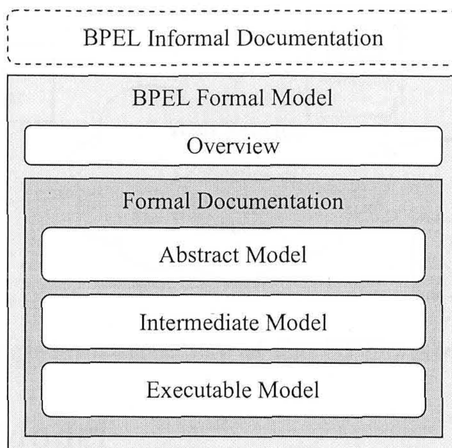


Figure 3.2: Sharpening informal requirements into executable specifications

[11, 5] obtained as the result of stepwise refinements of the abstract model.

Figure 3.3 shows an abstract view of the Web services interaction model underlying the BPEL Abstract Machine. A BPEL document abstractly defines a Web service consisting of a collection of business process instances. Each such instance interacts with the external world (i.e., the communication network) through two interface components, called *inbox manager* and *outbox manager*.

The inbox manager handles all the messages that arrive at the Web service. If a message matches a request from a process instance waiting for that message, the message is forwarded to the process instance. The inbox manager is also responsible for the new process instance creation which is further elaborated in Section 3.3. The outbox manager, on the other hand, forwards outbound messages from process instances to the network. The inbox manager, the outbox manager, and the process instances are modeled by three different types of DASM agents. While the inbox manager agent and the outbox manager agent deal with message exchange, each process agent models a single process instance. The major role of a process agent is to execute the main activity of a process; i.e., the activity that specifies the business logic behind process interactions.

Section 3.3 provides an overview of the formal model of the *core* of the BPEL Abstract Machine.

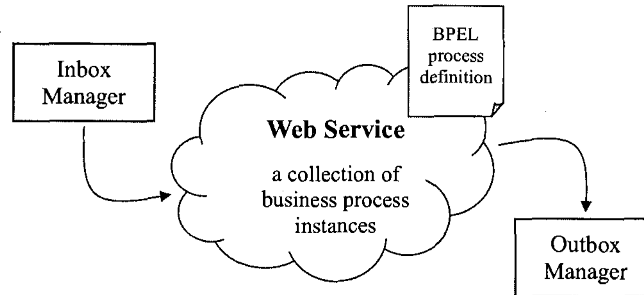


Figure 3.3: High-level structure of the BPEL Abstract Machine

3.3 The Formal Model

There are three major types of agents defined in the BPEL Abstract Machine, namely: inbox manager, outbox manager, and process instances. In addition to these agent types, another type of DASM agent called *activity agent* is introduced. Activity agents assist process agents in executing BPEL activities. Each process agent is responsible for executing a single process instance, and it uses dynamically created activity agents for executing complex (structured) activities.

$$\text{AGENT} \equiv \text{INBOX_MANAGER} \cup \text{OUTBOX_MANAGER} \cup \text{PROCESS} \cup \text{ACTIVITY_AGENT}$$

In the initial DASM state, there are only three DASM agents: the inbox manager, the outbox manager and a dummy process that facilitates creation of new process instances. In the following sections, we provide a brief overview on the behaviour of the inbox and outbox managers, process instances, and activity agents (see [43] for more details).

Modelling the behaviour of a BPEL process requires certain information that is specific for the given business process to be derived from the underlying BPEL document. We assume that the relevant information is generated automatically in a pre-processing step through static analysis of the underlying BPEL document using standard compiler techniques and formalized by a set of statically defined functions as part of the definition of the initial state of the DASM [43].

3.3.1 Inbox Manager

The inbox manager operates on the *inbox space*, a possibly empty set of inbound messages. In each computation step, it attempts to assign a message to a matching process instance. To wait for an incoming message to arrive, a process instance informs the inbox manager by adding an *input descriptor* to a set called, *waitingForMessage*. An input descriptor contains information on the waiting input operation and the waiting agent. The predicate $match(p, op, m)$ holds if message m can be assigned to operation op running in the process instance p according to the information specified by the input descriptor. The inbox manager uses this predicate to find an appropriate message that matches a waiting process instance. If the matching is successful, the message is assigned to the process instance using the AssignMessage rule which is comprehensively defined in [18, 43][1].

The inbox manager creates a new process instance whenever a matching message arrives for a *start activity*. Modelling process instance creation is simplified by introducing a nullary function *dummy* indicating a dummy process instance. The dummy process instance is not different from other process instances in its nature. However, there is always one and only one such process instance waiting on its start activity. By receiving the first matching message, the dummy process instance becomes a normal running process instance and a new dummy process instance will be created automatically by the inbox manager.

In [43], the intermediate model introduces an additional responsibility for the inbox manager which is captured by the PickActivityClearance. According to the LRM, whenever one of the expected messages is received by a pick activity, the business process must not accept any of the other messages (previously expected by the pick activity). Thus, once a message is assigned to a pick activity, the inbox manager is responsible for updating the waiting set such that no further message is assigned to that pick activity. The formal specification of the behaviour of the inbox manager and PickActivityClearance, as defined in [43], are recalled in Spec 3.1. The behaviour is, however, revised in Chapter 4 where the *core* model is elaborated.

3.3.2 Outbox Manager

The outbox manager operates on the outbox space, a possibly empty set of *output descriptors*. Each output descriptor represents an outgoing message to be generated and sent to

[1] A revised version of the AssignMessage is also presented in Appendix B

InboxManagerProgram $\equiv$
  if $inboxSpace(self) \neq \oslash$ **then**
    choose $p \in$ PROCESS, $m \in inboxSpace(self)$,
      $(agent, op) \in waitingForMessage(p)$ **with** $match(p, op, m)$
      AssignMessage$(p, agent, op, m)$
      PickActivityClearance
      // process instance creation
      if $p = dummyProcess$ **then**
        new $newDummy$: PROCESS
          $dummyProcess := newDummy$

PickActivityClearance (p : PROCESS, a : RUNNING_AGENT, op : IN_OPERATION) $\equiv$
  if $a \in$ PICK_MESSAGE_AGENT **then**
    forall $(a, op) \in waitingForMessage(p)$ **with** $op \neq op$
      remove (a, op) **from** $waitingForMessage(p)$

Spec 3.1: The original inbox manager program

the outside environment (the network). An output descriptor encapsulates the information on the message and its destination. In each step, the outbox manager chooses a single output descriptor and generates the corresponding message to be sent via the communication network. The following DASM program presents the behaviour of the outbox manager as defined in [43].

OutboxManagerProgram $\equiv$
  if $outboxSpace(self) \neq \emptyset$ **then**
    choose $od \in outboxSpace(self)$
      Send(od) // Effective send operation

However, the above mentioned behaviour of the outbox manager was defined considering the fact that the reply activity (or any other output operation) was regarded as a non-blocking activity. Although the LRM is not specific about the non-blocking behaviour of output operations, as our understanding of output operations in BPEL has improved, the behaviour of the outbox manager has been fully revised. This issue, along with a revised version of the outbox manager, is discussed in more detail in Chapter 4.

```
ProcessProgram ≡
  if ¬busy(self) then
    if ¬startedExecution(self) then
      startedExecution(self) := true
      busy(self) := true
    else
      stop self
  else
    ExecuteActivity(activity(self))
```

Spec 3.2: The original process program of the *core*

3.3.3 Process

Process agents model the behaviour of business process instances as defined by the underlying BPEL document. The major role of a process agent is to execute the main activity of a process. Once the execution of the activity is completed, the process agent terminates. The program of process agents is presented in Spec 3.2 [43]:

The *startedExecution* predicate specifies whether the execution of the main activity is started or not. When the process execution is started ($startedExecution(self) = true$), the process agent becomes busy (by setting the predicate *busy* to true) and remains busy during the execution. The ExecuteActivity rule takes care of the activity execution and is thoroughly defined in the intermediate model[2] [43]. The behaviour of each basic activity is modelled by a single ASM rule. To cope with the complexity of the execution of structured activities, the behaviour of structured activities are modelled by dynamically created activity agents which are further described in the next section.

Once the execution of a process is completed, the *busy* predicate is reset to false either by the process agent itself or by its child agent which is responsible for executing the main activity. This leads to the termination of the process agent in the next DASM step.

Here, we present the formal definition of the behaviour of the *receive* activity as an example of a BPEL basic activity. The behaviour of the *flow* activity, as an example of a structured activity, is then presented in the next section.

[2]See Appendix B for a revised version of this rule.

The Receive Activity

The *receive* activity is one of the most important activities in BPEL. It is used both as an input activity to receive a message from a partner and as an *start activity* to create new process instances. In the BPEL Abstract Machine, the creation of new processes is captured by the inbox manager as discussed in Section 3.3.1. The *receive* activity then only needs to inform the inbox manager of the expected message and wait for the message to arrive. The behaviour of this activity is captured by an ASM rule called ExecuteReceive.

Executing a *receive* activity in the BPEL Abstract Machine consists of two main tasks: (1) informing the inbox manger that a message is expected, and (2) waiting for the message to be received. Thus, ExecuteReceive works in two different modes, namely *initialization* mode and *waiting* mode, distinguished by a predicate *receiveMode*. In the initialization mode, the agent that is responsible for executing a receive activity (which can be either the process agent or one of its subordinate agents) informs the inbox manager that it is waiting for a message. This is done by adding an *input descriptor* to the *waitingForMessage* set of the root process. An input descriptor specifies the information on the expected message and the agent that is waiting for that message. In each step, the inbox manager inspects this set and checks whether any of the desired messages is received, and, if so, assigns it to the matching process instance. The agent then switches to the waiting mode ($receiveMode(self) := true$) and waits until the inbox manager assigns a message to it. Once a message is assigned to the agent, the inbox manager removes the corresponding input descriptor from the waiting set informing the agent that the assignment is performed.

The formalization of the behaviour of the *receive* activity is presented in Spec 3.3. The Synchronization rule addressed here fulfils the synchronization dependency requirements as specified in the LRM and is further elaborated in [43]. It is also worth pointing out that once the message is received and the execution of the receive activity is completed, the *(*busy) predicate is reset back to false.

3.3.4 Activity Agents

The execution of each structured activity inside a process instance is modelled by a single DASM agent of the type *activity agent* which is dynamically created by the process agent.

ExecuteReceive($activity$: RECEIVE) $\equiv$
  let $inputDescriptor = (self, activity)$ **in**
    if $\neg receiveMode(self)$ **then**
      $receiveMode(self) := true$
      // The running agent waits to receive a message
      add $inputDescriptor$ **to** $waitingSet$
    else
      if $inputDescriptor \notin waitingSet$ **then**
        $receiveMode(self) := false$
        $busy(self) := false$
        Synchronization($activity$)
where
  $waitingSet \equiv waitingForMessage(rootProcess(self))$

Spec 3.3: Behavioural specification of the *receive* activity

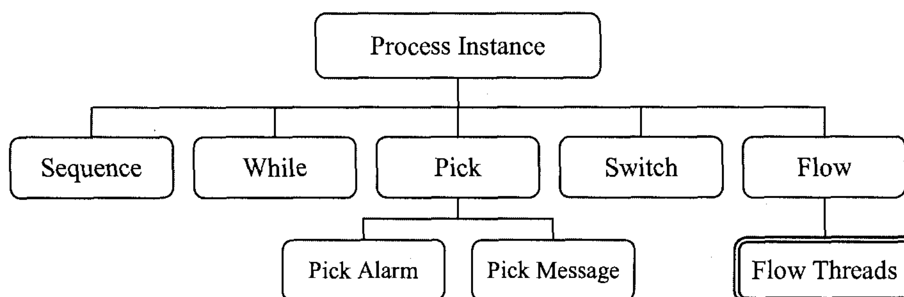


Figure 3.4: Control structure defined on DASM activity agents

Each activity agent can dynamically create other activity agents for executing nested structured activities. Moreover, to deal with concurrent control threads (like in *flow* and *pick*[3]) the responsible activity agent creates a number of auxiliary activity agents. For instance, to concurrently execute a set of activities, a flow agent assigns each enclosed activity to a separate *flow thread agent* [20]. Thus, at any time during the execution of a process instance, we may have a tree of DASM agents running under control of the corresponding process agent. Figure 3.4 shows the control structure of DASM activity agents, as presented in [43], where one can associate one branch from the root to a leaf with each single process instance. A revision of the execution tree will be presented in Chapter 4.

For maintaining the hierarchical relations between activity agents, we define a function

[3]One may argue that *pick* is not a concurrent control construct, but as discussed in [43], it can naturally be viewed as such.

parentAgent for linking a subordinate agent to its parent agent. For each activity agent, a derived dynamic function *rootProcess* is inductively defined for indicating the root of the execution tree. These functions are formally defined as follows:

$\text{RUNNING_AGENT} \equiv \text{PROCESS} \cup \text{ACTIVITY_AGENT}$
$parentAgent : \text{RUNNING_AGENT} \rightarrow \text{RUNNING_AGENT}$

$rootProcess : \text{RUNNING_AGENT} \rightarrow \text{PROCESS}$

$$rootProcess(a) \equiv \begin{cases} a, & \text{if } a \in \text{PROCESS}; \\ rootProcess(parentAgent(a)), & \text{otherwise.} \end{cases}$$

Flow Activity

The notion of concurrency in BPEL is provided by the *flow* activity and it is modelled by DASM agents in the core model of the BPEL Abstract Machine based on the principle of partially ordered runs [43]. A flow activity concurrently executes a set of activities and is completed when all the activities in the *flow* have completed their execution.

As for other structured activities, a flow activity is handled by an activity agent called a *flow agent*. A flow agent is responsible for executing a flow activity. To concurrently execute the activities declared inside the flow activity, the flow agent creates a set of flow thread agents and assigns each activity to one of these agents. The flow agent keeps track of its thread agents using a set called *flowAgentSet*. When created, the flow thread agents are added to this set. Once the execution of the activity assigned to one of the threads is completed, the thread removes itself from this set. Thus, whenever the *flowAgentSet* becomes empty, the execution of all concurrent activities is completed, and the flow agent releases its parent and terminates. Note that the operation of releasing the parent agent is performed by re-setting the *busy* predicate of the parent agent to false.

The behaviour of the flow agent is formally defined in Spec 3.4. For a complete list of all the function definitions, rules and agent programs of the original *core* of the BPEL Abstract Machine see [43].

```
FlowProgram ≡
  if ¬busy(self) then
    // Creates threads to concurrently execute activities grouped inside the flow.
    forall activity ∈ flowActivitySet(self)
      new fThread : FLOW_THREAD_AGENT
        Initialize(fThread, activity)
        add fThread to flowAgentSet(self)
      busy(self) := true
  else
    if flowAgentSet(self) = ∅ then
      // All threads are done, flow activity is completed.
      busy(parentAgent(self)) := false
      stop self
      Synchronization(baseActivity(self))
```

Spec 3.4: The original program of flow agents

3.4 Open Issues

The *core* of the BPEL Abstract Machine provides a high-level specification of the core concepts of BPEL, including concurrent control structures, communication primitives, message correlation, event handling mechanisms, and dynamic creation of services (process instances). To build a comprehensive model of BPEL around the *core* which includes all the fundamental aspects of the language, there are yet a number of open issues that need to be considered, such as

1. making the model more flexible for future refinements,
2. resolving ambiguities on outbound communication,
3. capturing data handling and state variables,
4. modelling fault handling, and
5. modelling the compensation behaviour of BPEL.

Chapter 4 elaborates the *core* of the BPEL Abstract Machine with regard to structural and behavioural aspects to resolve Issues #1 and #2. Resolving the first issue facilitates future refinements of the *core* toward a comprehensive high-level specification of the language. The second issue addresses the ambiguity of the LRM regarding outbound communication

of BPEL. Chapter 5 then extends and refines the *core* model to capture data handling, fault handling, and compensation behaviour of BPEL.

If there is to be any clarity at all,
it demands a certain assiduity.
Jason Dewinetz

Chapter 4

Elaborating the Core

Chapter 3 introduced the core model of the BPEL Abstract Machine and addressed some open issues in that model. This chapter substantially improves the core model with regard to structural and behavioural aspects, making it more robust and flexible for stepwise refinement. This chapter also resolves the open issues addressed in Chapter 3 by introducing: (1) a well-defined *Process Execution Model*; (2) a simple and efficient coordination platform for ASM agents; and (3) the notion of *requirements lists* which extract the key language requirements from the LRM to make these requirements accessible and to facilitate finding inconsistencies and ambiguities in the LRM. The outbound communication behaviour is also totally revised and a comprehensive specification is provided.

This chapter starts by an overview on commonly used refinement notions in software engineering and more specifically in Abstract State Machines (Sections 4.1.1 and 4.1.2). A two dimensional refinement approach to extend the core model of the BPEL Abstract Machine is then provided in Section 4.1.3. Various improvements on the *core* are presented in Section 4.2. Section 4.3 briefly presents the refinement of the core model using what is called *horizontal extensions*. The refinement is then discussed in detail in Chapter 5.

4.1 Refinement Notions

WordNet[1], a lexical database for English language at Princeton University, provides the following definitions for *refinement*:

> 1. a highly developed state of perfection; having a flawless or impeccable quality; 2. the result of improving something; 3. the process of removing impurities; 4. a subtle difference in meaning or opinion or attitude; 5. the quality of excellence in thought and manners and taste;

In software engineering, refinement can be defined more precisely as the process of improving an abstract model of a software system to a more concrete model, generally by reducing nondeterminism or uncertainty, which may eventually lead to a suitable implementation of the system.

In his well-known book, *The B-Book: Assigning Programs to Meanings*, Abrial informally defines refinement as a technique to transform an abstract mathematical model of a system to another mathematical model which is more concrete in the sense that it provides an 'implementation' of the abstract machine [1]. Woodcock and Davies, in their book on *Using Z: Specification, Refinement, and Proof* [53], provide a simple technical definition of refinement based on total relations: *"If R and S are total relations, then R refines S exactly when $R \subseteq S$."* Relation R can reduce the degree of freedom in S by omitting one or more tuples of the form (x, y_i) in S, where $x \in Dom(S)$ and each y_i is a distinct element of the set $Range(S)$.

In this section, we specifically focus on the ASM refinement method and address some of its frequently used forms.

4.1.1 The ASM Refinement Method

In this section, we recall some fundamental principles of ASM refinement techniques adopted from [12]. Most refinement notions in software engineering are based on a priori principles, like the Principle of Substitutivity which is described in [14] as:

> ***Principle of substitutivity:*** *it is acceptable to replace one program by another,* provided *it is impossible for a user of the program to observe that the substitution has taken place.*

[1] http://wordnet.princeton.edu

As an example one can refer to the concept of refinement in the B-method. In an informal approach to the refinement of generalized substitutions, we have: *"Roughly speaking, a substitution S (working within the context of a certain abstract machine M) is said to be* refined *by a substitution T, if T can be used in place of S without the 'user' of the machine noticing it."* [1, Section 11.1].

These refinement notions are restricted in various ways by their ground principles. Restriction to certain forms of programming is one example. As a consequence of restricting to sequential programming, refined programs are even structurally equivalent to their abstract versions; i.e., corresponding operations are occurring in the same order which almost prevents applying other forms of control structures such as parallelism. Restriction to certain pairs of input/output values or structures is another example, in which the possibility of refining abstract forms of input/output is ruled out.

The ASM refinement method is not based on any a priori defined refinement principle; i.e., the notion of refinement can be defined without restricting to a given model of comparing program runs in different levels of a system. The freedom of abstraction in ASMs , defined as *"the availability of ASMs of arbitrary structures to reflect the underlying notion of state"* [12], provides the necessary means to fine tune the mapping of an abstract machine to a more concrete one, in such a way that the intended equivalence between runs of these two machines becomes observable [12, 42]. Instead of focusing on a generic notion of refinement which can be proved to work in every context (and for instance can only effect the program in a way that remains hidden from the user), the focus is to support a disciplined use of refinement which can correctly document and reveal intended design decisions by adding more details to an abstract description. It can be anything from improving a program by additional features, restricting a program through some boundary conditions to prevent undesired behaviour, or making an abstract program executable.

Utilizing the freedom of abstraction frees us from a predefined notion of state, program, run, equivalence or any particular method to establish correctness of a refinement. In fact, with ASMs, any feasible accurate method can be used to show that the refined model is loyal to the original design assumptions and its runs correctly translate the effects of the runs of the abstract one.

In particular, the ASM refinement method (by being appropriately instantiated) can capture various more restricted refinement notions in the literature. This means, it can provide a uniform framework to reflect various refinement notions available in the literature.

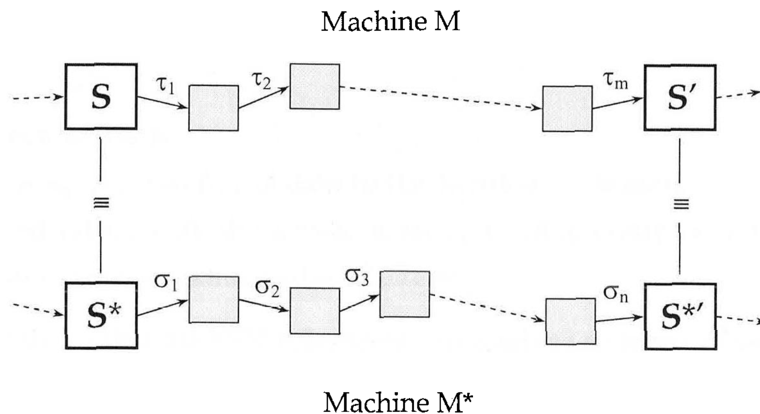


Figure 4.1: The ASM Refinement Scheme

Figure 4.1 illustrates the general scheme for an ASM refinement step. To refine an ASM M to an ASM M^* one has the freedom to define the following notions:

- **refined state**
- **states of interest and the correspondence between them**
  The states of interest are states of machine M and machine M^* that are related through the refinement process and are of particular interest. State S and its corresponding state S^* in Figure 4.1 are two states of interest of machines M and M^*.
- **computation segments**
  Computation segments of the form $\tau_1, \ldots, \tau_m$ between two sates of interest in M and corresponding refined segments of the form $\sigma_1, \ldots, \sigma_n$ between the corresponding states of interest in M^* can be defined, where each τ_i and σ_j represent a single M-step and a single M^*-step respectively. Figure 4.1 illustrates a computation segment of machine M and its corresponding computation segment in M^*. The resulting diagrams are called (m, n)-diagrams and the corresponding refinement is called (m, n)-refinement, where m steps of an abstract machine is refined to n steps of its refined machine.
- **locations of interest**
  A notion of locations of interest (in M-states) and corresponding locations (in M^*-states) can be defined, where locations represent abstract containers of data in states of M and M^*. The pairs of these locations (in M) and their corresponding locations

(in M^*) are then used to define the notion of equivalence of corresponding states of interest.

- **equivalence of data**
  A notion of equivalence ($\equiv$) of data in the locations of interest can be defined, which is then used (along with the notion of locations of interest) to define the notion of equivalence of corresponding states of interest.

The scheme shows that an ASM refinement can combine a change of signature (defining states of interest, corresponding states, locations of interest and their corresponding locations, and equivalence of data in those locations) with a change of control (computation segments and their corresponding segments), which are kept separated in many notations of refinement in the literature, like data refinement (e.g., in Z [53]) and operation refinement (e.g., in B [1]).

Once the notion of states and their equivalence are defined, M^* can be considered as a correct refinement of M if and only if every refined run simulates an abstract run with the equivalent corresponding states, according to the following definition [12]:

Definition 1. Fix any notions $\equiv$ of equivalence of states and of initial and final states. An ASM M^* is called a *correct refinement* of an ASM M if and only if for each M^*-run $S^*{}_0, S^*{}_1, \ldots$ there is an M-run $S_0, S_1, \ldots$ and sequences $i_0 < i_1 < \ldots$, $j_0 < j_1 < \ldots$ such that $i_0 = j_0 = 0$ and $S_{i_k} \equiv S^*{}_{j_k}$ for each k and either

- both runs terminate and their final states are the last pair of equivalent states; or
- both runs and both sequences $i_0 < i_1 < \ldots$, $j_0 < j_1 < \ldots$ are infinite.

Now a *complete refinement* is defined as:

Definition 2. M^* is called a *complete refinement* of M if and only if M is a correct refinement of M^*.

Note that when M^* is a correct refinement of M, there can be an M-run that has no corresponding M^*-run; i.e., M^* does not need to have an equivalence run for every run of M to be a correct refinement of M. On the other hand, when M^* is a correct refinement of M, for every M^*-run there is a corresponding M-run, so M is a complete refinement of M.

The pairs of initial and possibly final states are considered to be corresponding states; so, refinement correctness and completeness imply, for terminating runs, the equivalence of input/output behaviour of the abstract and the refined machine.

4.1.2 Refinement Patterns and ASMs

This section briefly presents *conservative refinement*, *data refinement*, and *procedural refinement* (also known as *operational refinement*), three widespread refinement patterns in the literature that are applied in practical system design and analysis.

Conservative Refinement

Conservative refinement, which is also called *conservative extension* or *horizontal refinement*, is a purely incremental refinement method which is suitable for introducing new behaviour in a modular approach. As an elegant example of applying this method, one can refer to the various extensions in the refinement of the Java machine in [42]. In order to define a conservative extension of an 'old' machine, the following steps should be performed:

1. *Define the 'new' condition*, in which the 'new' machine should be executed and the 'old' machine either has no defined behaviour or should not be executed. For instance, for adding fault handling to the BPEL Abstract Machine, this condition could be the execution mode of the machine being in Fault-Handling mode.

2. *Define the 'new' machine*, which defines the appropriate behaviour in case of the 'new' condition. In our example, it would be the *fault and compensation extension* which takes care of the fault handling behaviour.

3. *Guard the behaviour of the 'old' machine* using the negation of the 'new' condition, to prevent its execution when the new condition holds.

Data Refinement

Data refinement is mostly a $(1,1)$-refinement, where the effect of the refined operations on refined data types are the same as the effect of the abstract operations on abstract data types. Woodcock and Davies in [53] provide a nice definition for refinement of abstract data types. They define a data type $\mathcal{X}$ in a global state G to be a tuple of the form $(X, xi, xf, \{xo_i | i \in I\})$, where

- X is the space of values;

- xi: $G \rightarrow X$, is an initialization;

- xf: $X \to G$, is a finalization;
- $\{xo_i | i \in I\}$ is an indexed collection of operations, such that $xo_i : X \to X$
- xi and xf are total functions but each xo_i may be partial.

A program $P : G \to G$ is then defined as a composition of operations over data types that start with an initialization and ends with a finalization. This definition allows programs to be parameterized by data types. Any two abstract data types $\mathcal{A}$ and $\mathcal{B}$ that use the same index set of operations will support the same set of programs. According to this, the notion of refinement of abstract data types can be defined as follows: *if data types $\mathcal{A}$ and $\mathcal{C}$ share the same indexing set, then $\mathcal{A}$* is refined by $\mathcal{C}$ *if and only if for each program $P(\mathcal{A})$*, $P(\mathcal{C}) \subseteq P(\mathcal{A})$ [53][2].

In ASMs, a frequently used form of data refinement which uses the generalization of 'operation' to 'ASM rule' is through *instantiation*, where the ASM rules remain unchanged and only the abstract functions and predicates occurring in the rules are specified in more detail [12]. An example of data refinement in the *core* of the BPEL Abstract Machine is the refinement of an abstract function *message_is_received* in the description of the *receive* activity to the following definition [43]:

$$\begin{aligned}&\textit{message_is_received}\ (\textit{activity} : \text{ACTIVITY}) \equiv \\ &\qquad (\textit{self}, \textit{activity}) \notin \textit{waitingForMessage}(\textit{rootProcess}(\textit{self}))\end{aligned}$$

Procedural Refinement

In a given machine, replacing a rule (or a submachine[3]) by another rule (or another machine) is called *procedural refinement* or submachine refinement. This form of refinement in most cases is either a $(1, n)$-refinement (in compiler verification replacing one line of source code by a chunk of target code) or an (m, n)-refinement where m is usually less than n (replacing an abstract machine/rule by a more concrete one).

A distinctive example is the refinement of the Prolog ASM in [10] in which an abstract function *unify* is refined to a submachine which implements a unification procedure. Another

[2]Strictly speaking, the condition is $P(\dot{\mathcal{C}}) \subseteq P(\dot{\mathcal{A}})$ in which the $\dot{\mathcal{A}}$ is a totalised version of $\mathcal{A}$. See [53] for more details.

[3]See [15] for definition of an ASM submachine.

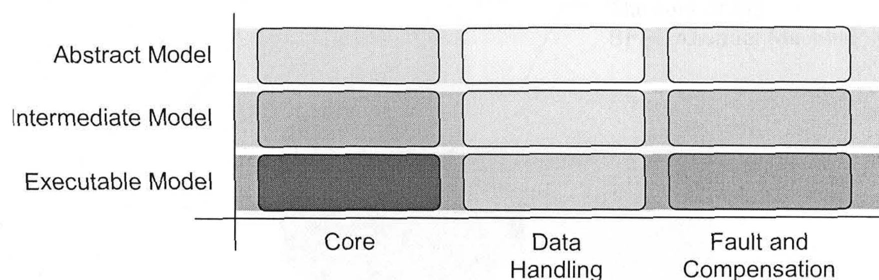


Figure 4.2: A two-dimensional refinement approach

example is the refinement of the abstract rule ExecuteActivity in the ProcessProgram of the abstract model of the *core* of the BPEL Abstract Machine into a complex rule in the next level (i.e., the intermediate model) which implements activity execution in BPEL.

Procedural refinement is more general than the principle of substitutivity mentioned above. A submachine which is refined in this way, can capture new features that are not included in the behaviour of the abstract machine (though they are related to the behaviour of the abstract machine).

We combine the refinement methods addressed here for constructing a two dimensional refinement approach to elaborate the *core* of the BPEL Abstract Machine and extend it to capture other aspects of BPEL. Section 4.1.3 introduces this refinement approach. The rest of this chapter applies this refinement method to elaborate the *core* and then Chapter 5 extends the *core* toward a comprehensive formal model of BPEL.

4.1.3 A Two Dimensional Refinement Approach

To deal with the complexity of the BPEL Abstract Machine and the required expansions to cover data handling and fault and compensation handling behaviour, we introduce a two dimensional refinement approach:

- **vertical refinement** which provides step by step elucidation using a combination of data refinement and procedural refinement in a three layered structure, and
- **horizontal refinement** which facilitates behavioural decomposition using conservative (incremental) extensions.

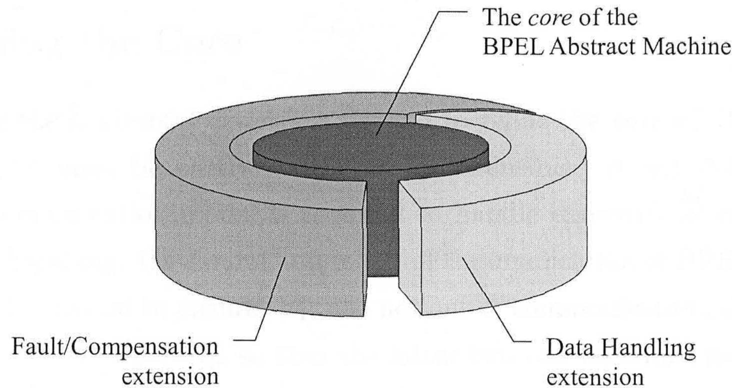


Figure 4.3: Behavioural decomposition using incremental extensions

Figure 4.2 illustrates the structure of the enhanced BPEL Abstract Machine. Based on this approach, the BPEL Abstract Machine comprises three basic building blocks reflecting its horizontal organization: *core*, *data handling extension*, and *fault and compensation extension*. The *core*, which is the revised version of the original core model, focuses on:

- dynamic process creation/termination,
- Web services communication primitives,
- message correlation,
- concurrent control structures,
- and core BPEL activities as addressed in Chapter 3.

The *core* does not deal with data handling issues, faults or compensation behaviour. The *data handling extension* adds BPEL variables and data handling behaviour to the *core* and replaces abstract message values with actual values received from the environment thus enables business processes to create and manipulate message values. The *fault and compensation extension* complements the model by providing fault handling and compensation behaviour. These extensions are fully discussed in Chapter 5.

Vertically, the architecture is organized into three layers of abstraction, called *abstract* model, *intermediate* model and *executable* model, as illustrated in Figure 3.2. The vertical refinement of the core model is described in more detail in Chapter 3.

4.2 Revising the Core

Before applying the horizontal extension method to refine the *core* of the BPEL Abstract Machine, the *core* must be partly revised to be extensible. A well-defined, flexible and extensible process execution model is required to handle extended forms of execution, in particular fault handling. Outbound and inbound communication of BPEL processes in the model needs to be revised to ensure that the notions of communication, data handling, and fault handling are well separated so that the latter two can be added as extensions to the first one. As more aspects of BPEL are captured by the model and the number of agents involved in the execution of a business process increases, a well-defined interaction framework is required to be defined to avoid complexity and ambiguity of interactions between agents.

4.2.1 Process Execution Model

In the original core model, activity agents go through at least two different phases during their execution. For modelling the required state transition behaviour, we define the predicate *busy* to distinguish between two general states of an activity agent:

- *busy(agent)* being true indicates that the *agent* is executing an activity;
- *busy(agent)* being false indicates that the *agent* is either in the initialization mode, or is finished executing an activity. The agent can again start executing an activity (in case of a *sequence* or a *while* agent) and thus may become busy again.

While this was a good choice to start with, after incorporating more activities into the model, we found that we need a better structure to deal with state transitions of activity agents. The *busy* predicate could cause ambiguity in some cases. The interpretation of this predicate being false was not completely consistent in the model; the corresponding agent could be either in the initialization phase or in the termination phase. For some activity agents, like *switch*, careful consideration was required to separate these two phases. It was also the responsibility of child activity agents to reset the value of the *busy* function of their parent activity agents back to false, indicating that they (the child activity agents) finished their execution. This would restrict the behaviour of parent agents and in some cases make the model ambiguous. Furthermore, this two-phase model was not flexible enough for future extensions of the model, for example to incorporate fault handling behaviour.

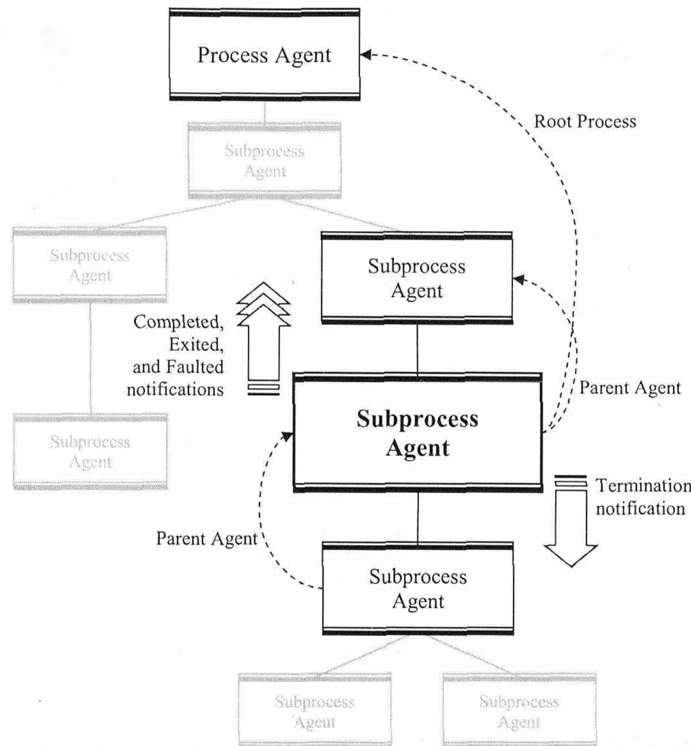


Figure 4.4: The process execution tree

To enhance the state transition model of the activity agents and to cope with the complexity of the process execution model, we introduce the notions of *process execution tree* and *execution lifecycles*.

At any time during the execution of a process instance we may have a tree of DASM agents running under control of the enclosing process agent (Figure 4.4). Each of these sub-agents monitors the execution of its child agents (if any) and notifies its parent agent in case of normal completion or fault. This structure provides a general framework for the execution of BPEL activities. The DASM agents that model BPEL process execution are called *kernel agents*. They include process agents and subprocess agents. In the *core*, however, we define subprocess agents to be identical to activity agents. Figure 4.4 sketches the process execution tree.

KERNEL_AGENT ≡ PROCESS ∪ SUBPROCESS_AGENT

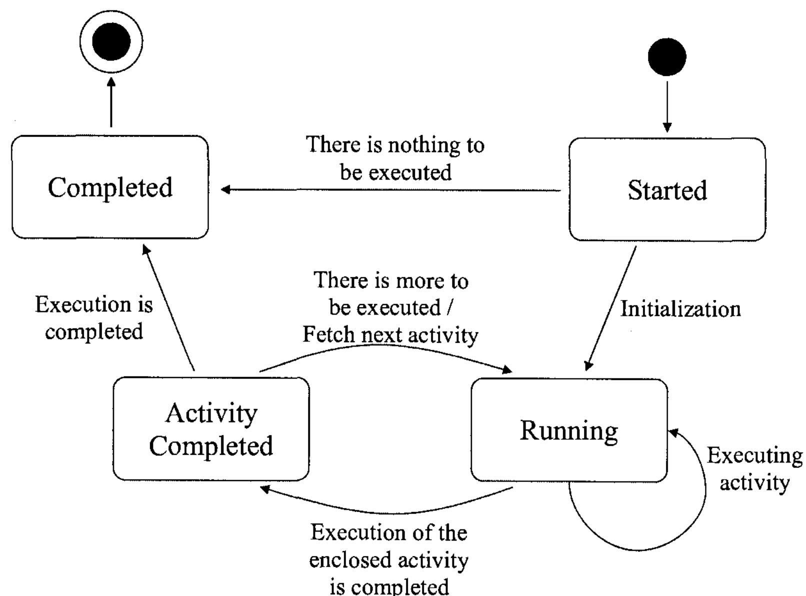


Figure 4.5: Execution lifecycle of kernel agents: *core*

For maintaining the hierarchical relations between kernel agents, we use the function *parentAgent* for linking a subordinate agent to its parent agent. For each kernel agent, a derived dynamic function *rootProcess* indicates the root of the execution tree, similar to the one that was defined over running agents in the original core model.

---

rootProcess : KERNEL_AGENT → PROCESS

We decompose the execution of a process instance into a collection of *execution lifecycles* for the individual BPEL activities. The state diagram in Figure 4.5 illustrates the normal execution lifecycle of kernel agents in the *core.* When created, a kernel agent is in the *Started* mode. After initialization, the kernel agent starts executing its assigned task by going into the *Running* mode. When the execution is completed, the agent goes to the *ActivityCompleted* mode, where it can decide (based on the nature of the assigned task) to either go back to the *Running* mode or finalize the execution and switch to *Completed.* Activity agents that may execute more than one activity (like *sequence*) or execute one activity more than once (like *while*) can go back and forth between the *ActivityCompleted* mode and the *Running* mode.

4.2.2 Agent Interaction Model

To avoid changing the state of an agent by its child agent(s), and to make the model flexible for future changes and extensions, we provide a simple yet elegant framework for agents to communicate with each other.

Communication between agents is provided by *signals*. Every kernel agent can send a signal to another agent using the following operation:

---

trigger s : SIGNAL_DOMAIN, agent
  $Rule_1$

A kernel agent can respond to a received signal using the following operation:

---

onsignal s : SIGNAL_DOMAIN
  $Rule_1$
otherwise
  $Rule_2$

Trigger and **onsignal** are the only interfaces of kernel agents' communication framework. To each process, we assign a set of signals which acts as a container for the signals coming to any kernel agent under control of the process. This set is addressed by the function *signalSet* defined on PROCESS. We define SIGNAL to be the set of all defined signal domains. In the *core* however, SIGNAL is defined as follows:

---

domain AGENT_COMPLETED
SIGNAL ≡ AGENT_COMPLETED

For every signal, *signalSource* and *signalTarget* are defined to indicate the source and the target agents of that signal.

---

signalSource : SIGNAL → KERNEL_AGENT
signalTarget : SIGNAL → KERNEL_AGENT

When an agent **triggers** a signal for another agent, a new element of the specified signal domain is created, its source and target agents are assigned, and the signal is added to the signal space of the target agent (which is the signal space of its root process). The following

syntactical transformation provides this behaviour:

---

trigger s : SIGNAL_DOMAIN, agent
  Rule
$\equiv$
extend SIGNAL_DOMAIN **with** s
  $signalSource(s) := self$
  $signalTarget(s) :=$ agent
  add s **to** $signalSet(rootProcess(self))$
  Rule

To respond to a signal, the target agent looks for an element of that signal in its corresponding signal space, removes that element from the signal space and performs the intended operations.

---

onsignal s : SIGNAL_DOMAIN
  $Rule_1$
otherwise
  $Rule_2$
$\equiv$
if $\exists s(s \in signalSet(rootProcess(self)) \wedge$
    $signalSource(s) = self \wedge s \in$ SIGNAL_DOMAIN$)$
  choose $s \in signalSet(rootProcess(self))$ **with**
      $s \in$ SIGNAL_DOMAIN $\wedge\ signalSource(s) = self$
    remove s **from** $signalSet(rootProcess(self))$
    $Rule_1$
else
  $Rule_2$

The complete definition of the related rules and agent programs is provided in Appendix B.

Sequence Program: An Example

Kernel agent programs of the original core model are revised with regard to the *process execution model* and the *agent interaction model* discussed earlier. We present here, as an example, the original version and the revised version of the sequence agent program

(respectively referred as the $\text{SequenceProgram}_{original}$ and the SequenceProgram).

A *sequence* activity is a structured activity that enforces a sequential execution order on a collection of activities. The $\text{SequenceProgram}_{original}$ has three phases: (1) fetching the next activity for execution, (2) executing the activity, and (3) going back to phase (1) if there is more activity for execution, otherwise finalizing the execution of the sequence. Spec 4.1 presents the original version of the sequence program. The three phases mentioned here are distinguished in the program by a combination of the *busy* predicate and the value of the *currentActivity*. When *busy*(*self*) does not hold, the agent gets the next activity for execution and switches *busy*(*self*) to *true*. When *busy*(*self*) holds, the agent either executes the current activity or finalize its execution. These two cases are distinguished by checking the value of *currentActivity*(*self*). If this value is undefined (*undef*), it indicates that there is no more activity for execution and the agent must finalize its execution (i.e., the execution of the sequence). Otherwise, the agent executes the current activity. The interpretation of the *busy* predicate and the separation of different execution phases in the program, in spite of being precise and concise, is not easy to follow.

The revised version of the sequence program is presented in Spec 4.2. Different execution modes of the agent (see Figure 4.5) are distinguished by the *execMode* function which is defined as follows:

EXECUTION_MODE ≡ {*emStarted*, *emRunning*, *emActivityCompleted*, *emCompleted*}
execMode : KERNEL_AGENT → EXECUTION_MODE
// initial value: *emStarted*

The sequence program starts in the *Started* mode, assigns the first activity to the *currentActivity* function and switches to the *Running* mode to execute the activity. If the executing activity is a basic activity, when the execution is completed, the ExecuteActivity rule will change the execution mode of the agent to the *Activity-Completed*. If the executing activity is a structured activity, when its execution is completed, its associated activity agent will send an *agent-completed* signal to the sequence agent. Upon receiving the *agent-completed* signal, the sequence agent switches to the *Activity-Completed* mode.

In the *Activity-Completed* mode, the sequence agent fetches the next activity to be executed. If the value of this activity is *undef* (i.e., there is no more activity to be executed), the sequence agent finalizes its execution using the FinalizeKernelAgent rule, which also switches

SequenceProgram$_{original}$ ≡
if $\neg busy(self)$ **then**
$currentActivity(self) := sequenceCounter(baseActivity(self))$
$busy(self) := true$
else
if $currentActivity(self) \neq undef$ **then**
ExecuteActivity($currentActivity(self)$)
else
stop *self*
$busy(parentAgent(self)) := false$
Synchronization($baseActivity(self)$)

Spec 4.1: The original SequenceProgram of the *core*

the execution mode to *Completed*. The FinalizeKernelAgent rule switches the execution mode of the agent to the *Completed* mode, sends an *agent-completed* signal to its parent agent, and uses the Synchronization rule to handle synchronization issues[4]. The sequence agent then, like all other kernel agents, terminates in the *Completed* mode. The formal definition of the FinalizeKernelAgent rule is presented below:

FinalizeKernelAgent ≡
$execMode(self) := emCompleted$
trigger s : AGENT_COMPLETED, $parentAgent(self)$
Synchronization($baseActivity(self)$)

4.2.3 Requirements Lists

The original core model is built upon version 1.1 of the LRM, where the requirements are scattered over nearly 140 pages, making it hard to analyze semantic aspects of the language and to extract key language properties. We built the original core model by frequently referring to the LRM to make sure that all the required aspects are considered faithfully. Despite our careful conformance to the LRM, there were some ambiguities in the LRM that we missed to address in the original core model. Blocking behaviour of the *reply* activity is an example which is further discussed in Section 4.2.5. s Before starting to build the revised

[4]See [43] for more detail on synchronization of activities.

```
SequenceProgram ≡
  case execMode(self) of
    emStarted →
      currentActivity(self) := sequenceCounter(self)
      execMode(self) := emRunning

    emRunning →
      if normalExecution(self) then
        onsignal s : AGENT_COMPLETED
          execMode(self) := emActivityCompleted
        otherwise
          ExecuteActivity(currentActivity(self))

    emActivityCompleted →
      currentActivity(self) := sequenceCounter(self)
      if currentActivity(self) = undef then
        FinalizeKernelAgent
      else
        execMode(self) := emRunning

    emCompleted → stop self
```

Spec 4.2: The revised version of the SequenceProgram of the *core*

version of the model, and in order to make it easier to further extend the model, we decided to extract lists of requirements related to the core aspects of BPEL. These lists are called *Requirements Lists*, covering major requirement elements related to different aspects of the language. These requirement elements are all extracted from the LRM and are all referenced precisely to the corresponding sections in the LRM. Requirements lists have been revised many times during the refinement process, to be as comprehensive and concise as possible. As an example, we present here the requirements list of the *reply* activity (Reply-LR):

1. "The `<reply>` construct allows the business process to send a message in reply to a message that was received through a `<receive>`. The combination of a `<receive>` and a `<reply>` forms a request-response operation on the WSDL *portType* for the process." [$6.2] "A *reply* activity is used to send a response to a request previously accepted through a *receive* activity. Such responses are only meaningful for synchronous interactions." [$11.4]

2. "The correlation between a request and the corresponding reply is based on the constraint that more than one outstanding synchronous request from a specific partner link for a particular *portType*, operation and correlation set(s) MUST NOT be outstanding simultaneously." [$11.4]

3. "For the purposes of this constraint [Reply-RL-3], an *onMessage* clause in a *pick* is equivalent to a *receive* (see 12.4. Pick)." [$11.4]

4. "Moreover, a reply activity must always be preceded by a receive activity for the same partner link, *portType* and (request/response) operation, such that no reply has been sent for that receive activity." [$11.4]

5. "Note that the `<reply>` activity corresponding to a given request has two potential forms. If the response to the request is normal, the *faultName* attribute is not used and the variable attribute, when present, will indicate a variable of the normal response message type. If, on the other hand, the response indicates a fault, the *faultName* attribute is used and the variable attribute, when present, will indicate a variable of the message type for the corresponding fault." [$11.4]

6. A reply activity MAY specify a variable that contains the message data to be sent [$11.4].

7. "Variables associated with message types can be specified as input or output variables for *invoke*, *receive*, and *reply* activities (see 11.3. Invoking Web Service Operations and 11.4. Providing Web Service Operations)." [$9.2]

8. "If the variable reference is omitted for an outgoing message, then any properties of the message are considered to have been initialized through opaque assignment..." [$15.1]

9. "If a reply activity is being carried out during the execution of a business process instance and no synchronous request is outstanding for the specified *partnerLink*, *portType*, operation and correlation set(s), then the standard fault *bpws:invalidReply* MUST be thrown by a compliant implementation." [$14.5]

10. "*correlationViolation* is thrown when the contents of the messages that are processed in an *invoke*, *receive*, or *reply* activity do not match specified correlation information." [$20.1]

11. "*invalidReply* is thrown when a reply is sent on a partner link, *portType* and operation for which the corresponding *receive* with the same correlation has not been carried out." (similar to #9) [$20.1]

12. "In case of activity termination, the activities *wait*, *reply* and *invoke* are added to *receive* as being instantly terminated rather than being allowed to finish." [$4.3]

The complete collection of requirements lists is presented in Appendix A.

4.2.4 Input/Output Descriptors

In the original core model, agents use tuples of two or more elements as descriptors to indicate that they are waiting for an incoming message or they have a message that needs to be sent out. Inbox and outbox managers look for these tuples and provide the required service to the corresponding agents.

There were two problems with using tuples as descriptors:

1. Every module (agent program or ASM rule) that deals with a descriptor should be aware of the exact structure of that tuple, even if only one component of that tuple is of interest. This reduces the flexibility of the model for future refinements; while if all

modules could only see the parts of the descriptor that they require, future refinements of some modules could simply extend the structure of the descriptor without affecting other modules.

2. This approach is not a good choice for future changes and improvement. Once the structure of the tuple representing the descriptor is fixed, this structure is hard-coded in all the modules dealing with that tuple. Future changes thus would require all the involved modules to be changed even if they are not directly affected by the change. For example, adding variable references to a descriptor requires all the modules using that descriptor to be changed even if they don't deal with variable references.

To prepare the model for future extensions, descriptors are defined as abstract data types with associated functions that represent their intended data structure. For example, for input descriptors (which inform the inbox manager that an 'agent' is waiting for a message on a specific 'operation') we provide the following definition:

---

domain INPUT_DESCRIPTOR
dscAgent : INPUT_DESCRIPTOR $\rightarrow$ KERNEL_AGENT
// Assigns to each descriptor, a running agent that created that descriptor.
dscOperation : INPUT_DESCRIPTOR $\rightarrow$ INOUT_OPERATION
// There is an Input/Output operation that is bound to every input descriptor.

With this structure, additional information can be flexibly attached to a descriptor without changing those modules that are not affected by the new property. For example, to add a time tag to descriptors indicating the completion time of the operation, the following function is defined:

---

dscCompletedTime : INPUT_DESCRIPTOR $\rightarrow$ TIME

4.2.5 Outbound Communication

The LRM states that "*The* `<receive>` *construct allows the business process to do a blocking wait for a matching message to arrive*" [4, Section 6.2]. While the LRM explicitly states that the *receive* activity is a blocking activity, for the reply activity it leaves this aspect unclear by declaring that "*the* `<reply>` *construct allows the business process to send a message in reply*

```
// — Reply Activity —
ExecuteReply_original(activity : REPLY) ≡
  AddOutputDescriptorToOutboxSpace(activity)
  busy(self) := false

AddOutputDescriptorToOutboxSpace_original(activity : ACTIVITY) ≡
  add outputDescriptor to outSpace
where outputDescriptor = (self, activity),
      outSpace = outboxSpace(outboxManager(rootProcess(self)))

// — Outbox Manager —
OutboxManagerProgram_original ≡
  if outboxSpace(self) ≠ ∅ then
    choose od ∈ outboxSpace(self)
      Send(od)// Effective send operation
```

Spec 4.3: Outbound communication behaviour in the original core model

to a message that was received through a `<receive>`" [4, Section 6.2] without mentioning any blocking or non-blocking behaviour for *reply.*

In the original core model the *reply* activity was considered to be a non-blocking activity. The behaviour of a *reply* activity was only modelled by adding the outgoing message to a set of messages which are supposed to be sent out, and the continuing the execution of the process. Spec 4.3 provides the ExecuteReply rule and the outbox manager program of the abstract layer of the original core model.

However, an in-depth analysis of the LRM revealed that the *reply* activity, like its counterpart, should be a blocking activity. In Section 4.3 of the LRM, "Feature Changes", it is stated that "*In case of activity termination, the activities* ***wait***, ***reply*** *and* ***invoke*** *are added to* ***receive*** *as being instantly terminated rather than being allowed to finish.*" This declaration indicates that the authors assume that the *reply* activity, like *wait* and *receive*, is in fact a blocking activity. Indeed, the lack of a formal and precise definition of these constructs of the language is the main reason behind the ambiguity in the semantics of *reply.*

According to this view, we model the behaviour of a reply activity in two phases: first, creating an appropriate output descriptor and adding it to the *waitingSetForOutput*; and second, waiting to receive a successful sent confirmation from the outbox manager. Spec 4.4 provides the revised version of the ExecuteReply and the oubox manager program.

To execute a reply activity, the request-response condition addressed by Requirement

```
// ———————— Reply Activity ————————
ExecuteReply(activity : REPLY) ≡
  if requestResponseConditionSatisfied(activity) then
    if ¬replyMode(self) then
      replyMode(self) := true
      GenerateOutputDescriptor(activity)
    if replyMode(self) then
      choose descriptor ∈ completedOutOperations(self) with
          dscAgent(descriptor) = self ∧ dscOperation(descriptor) = activity
        replyMode := false
        FinalizeActivity(activity)

GenerateOutputDescriptor(operation : OUTPUT_OPERATION) ≡
  extend OUTPUT_DESCRIPTOR with descriptor
    SetInOutDescriptor(operation, descriptor)
    add descriptor to waitingSetForOutput(rootProcess(self))

SetInOutDescriptor(operation : OUTPUT_OPERATION, dsc : INOUT_DESCRIPTOR) ≡
  dscAgent(dsc) := self
  dscOperation(dsc) := operation

// ———————— Outbox Manager ————————
OutboxManagerProgram ≡
  choose p ∈ PROCESS, descriptor ∈ waitingSetForOutput(p)
    DeliverMessage(p, descriptor)

DeliverMessage(p : PROCESS, descriptor : OUTPUT_DESCRIPTOR) ≡
  if variable(opr) = undef then
    add opaqueMessage(opr) to outboxSpace(self)
    if initiateCorrelation(opr) then
      InitiateCorrelation(p, descriptor, opaqueMessage(opr))
  dscCompletedTime(descriptor) := now
  add descriptor to completedInOperations(p)
  remove descriptor from waitingSetForOutput(p)
where
    opr ≡ dscOperation(descriptor)
```

Spec 4.4: Revised Outbound Communication Behaviour

#4 of DRL[5] must be satisfied. The two phases of the execution of a reply activity are distinguished by *replyMode*, a predicate defined on KERNEL_AGENTs. When *replyMode*(*self*) is false, an output descriptor is created from the corresponding *reply* activity and is added to the set of output descriptors waiting to be sent out (*waitingSetForOutput*). This behaviour is formulated in the GenerateOutputDescriptor. At the same time, *replyMode*(*self*) becomes true.

When *replyMode*(*self*) is true, the agent is basically waiting for the outbox manager to send out the message and move the output descriptor to the set of completed output descriptors (*completedOutOperations*). Thus, once added to the *completedOutOperations* set, the agent removes the descriptor, resets *replyMode*(*self*) back to false, (for future reply activities) and finalizes its execution using the FinalizeActivity rule.

The outbox manager on the other hand, looks for output descriptors from running processes and employs the DeliverMessage rule to send out the message to the network. In the original core model, the behaviour of delivering a message is only defined when there is no variable defined for the outgoing activity. In this case, the DeliverMessage rule puts an appropriate *opaque message* into the *outboxSpace*(*self*) set. An opaque message is a message with non-deterministic property values. The opaque message is provided by an abstract function *opaqueMessage* defined on output operations (i.e., *reply* and *invoke*), which should satisfy Requirements 13 to 16 of DRL[6]. DeliverMessage then adds a completion time tag to the output descriptor, removes it from the set of waiting output descriptors, and adds it to the *completedOutOperations* set.

A complete definition of all the corresponding functions, rules and programs are provided in Appendix B.

4.3 Extensions to the Core

It is mentioned in Section 3.4 that data handling, fault handling, and compensation behaviour of BPEL is not covered by the core model of the BPEL Abstract Machine. This chapter substantially revised the core model making it more robust and flexible for future refinements that ultimately form a comprehensive model of the Web Services Architecture of BPEL.

[5]See Appendix A.3.

[6]See Appendix A.3.

A two-dimensional refinement approach is presented in Section 4.1.3 which facilitates further refinements of the *core* toward capturing new aspects of BPEL using the notion of incremental extensions, and enables step by step elucidation of the extensions through a combination of data refinement and procedural refinement approaches.

In addition, appropriate requirements lists are extracted from the LRM[7] facilitating precise modelling of the language. For a clear separation of concerns, the aspects of data handling, fault handling, compensation behaviour, and the core of the language are carefully separated from each other. This is mostly visible in the concise definition of the *data handling extension*.

An in-depth discussion of the extensions of the *core* requires a whole chapter of its own. Thus, Chapter 5 introduces these extensions and explores them in detail.

[7]See Appendix A for a complete list of these requirements.

How obvious — how necessary — was that mathematical ratio of its sides, the quadratic sequence 1 : 4 : 9! And how naive to have imagined that the series ended at this point, in only three dimensions!

Arthur C. Clarke

Chapter 5

The Web Services Architecture of BPEL

Chapter 4 elaborates on the *core* of the BPEL Abstract Machine making it more robust and flexible for further refinements. The *core* of the BPEL Abstract Machine abstracts from data handling, fault handling, and compensation behaviour and mainly focuses on modelling BPEL abstract processes (business protocols). This chapter completes the mathematical definition of our BPEL Abstract Machine by presenting the *data handling extension* and the *fault and compensation extension* as two horizontal refinements of the *core* of the BPEL Abstract Machine. The resulting model provides a comprehensive high-level specification of the Web services architecture of BPEL.

5.1 Data Handling Extension

The *data handling extension* is a horizontal refinement of the *core* of the BPEL Abstract Machine which supplements the *core* with data handling behaviour and the notions of *variable* and *scope.* This section starts with an overview of the notion of data handling in BPEL, provides a requirements list on data handling extracted from the LRM, and then refines the *core* of the BPEL Abstract Machine by presenting the *data handling extension.*

5.1.1 Data Handling in BPEL

Web services orchestration languages, like BPEL, provide stateful interactions over a stateless communication platform like the one defined by WSDL (Chapter 2). To maintain the state of a business process, BPEL uses state variables, simply called *variables*. To process and manipulate the data collected in variables, BPEL provides *data expressions*. Furthermore, a notion of *assignment* is required to update the state of a business process, which is available in BPEL through its `<assign>` activity. Thus, data handling in BPEL is delivered through concepts of variables, expressions, and assignments.

Variables

In BPEL, variables are mainly used as message containers. Messages from other partners are stored in variables for further processing and manipulation. In this case, variables are defined as WSDL message types [51]. Variables can also hold other forms of data which is not used for communication with partners, using an XML Schema simple type or an XML Schema element [4].

Each variable is defined by a name and a type. The type of a variable is defined using one of the three available tags: *messageType* (for WSDL message type), *type* (for XML Schema simple type), or *element* (for XML Schema elements). Variables can be defined in the `<variables>` area of a BPEL document. These variables are called *global variables* and are valid in the entire process program unless redefined locally (see Scopes).

In BPEL variables are mainly used as message containers by input and output operations (activities) like *receive*, *reply*, and *invoke*. A variable reference in a *receive* activity, indicates that a copy of the incoming message must be stored in the referenced variable. A variable reference in a *reply* activity, refers to the variable that contains the outgoing message. Respectively, *invoke* and *pick* can also reference to variables. The use of variables in input/output activities is optional in an abstract process.

Scopes

Scopes in BPEL provide the behaviour context for activities. A Scope is a special form of a structured activity which can have its own variable definitions, correlation sets and fault and compensation handling behaviour (see Section 5.2). Each scope has a primary activity which defines the normal behaviour of the scope. This activity can be any basic or

structured activity. BPEL allows scopes to be nested arbitrarily.

Variables defined in a scope are called *local variables*. Local variables are only valid in the scope in which they are defined. If a local variable is defined with the same name and same type as a variable in an enclosing scope, the local variable will hide the variable of the enclosing scope within the local scope and all its enclosed scopes.

Expressions

Expressions in BPEL are either boolean-valued, deadline-valued, duration-valued, or a general expression based on an expression language which is referenced at the beginning of a BPEL document. In the current version of BPEL (version 1.1) general expressions must conform to the XPath (XML Path Language) 1.0 Expr production [47] where the evaluation of the expression results in an XPath value type (string, number, or Boolean) [4].

Assignment

The *assign* activity is introduced in BPEL to enable business processes to assign values to variables. It can be used to copy the value of a variable or some part of it to another variable, or to evaluate and assign the value of an expression to a variable. The *assign* activity is a set of *copy* elements. Each *copy* element has one pair of *from* and *to* elements. A *from* element refers to a variable (or just one part of a structured variable), an expression or a literal value. A *to* element refers to a variable[1] (or just one part of a structured variable). The LRM does not indicate any execution order on *copy* elements of an assign activity.

5.1.2 Requirements

Like other issues in the LRM, data handling behaviour of BPEL is addressed at least in four different sections. Thus, to start modelling the data handling behaviour of BPEL, a complete list of related requirements were extracted from the LRM. Some of these requirements to which we referred later in this section are presented below, while the complete list is presented in Appendix A.3. We further refer to this data handling requirements list as *DRL*. The numbering of these requirement elements presented here are kept consistent to the complete list in the appendix.

[1]Both *from* and *to* elements can also address a BPEL *partnerLink* which is related to the dynamic partner bounding behaviour of BPEL and is out of the scope of this work.

Partial List of Data Handling Requirements in BPEL

1. "The type of each variable may be a WSDL message type, an XML Schema simple type or an XML Schema element." [$9.2]

2. "The name of a variable should be unique within its own scope." [$9.2]

3. "If a local variable has the same name and same *messageType/type/element* as a variable defined in an enclosing scope, the local variable will be used in local assignments and/or *getVariableProperty* functions." [$9.2]

4. "It is not permitted to have variables with [the] same name but different *messageType/type/element* within an enclosing scope hierarchy. The behavior of such variables is not defined." [$9.2]

5. "Variables associated with message types can be specified as input or output variables for *invoke*, *receive*, and *reply* activities." [$9.2]

6. "When an invoke operation returns a fault message, this causes a fault in the current scope. The fault variable in the corresponding fault handler is initialized with the fault message received." [$9.2]

7. "Each variable is visible only in the scope in which it is defined and in all scopes nested within the scope it belongs to." [$9.2]

8. "A global variable is in an uninitialized state at the beginning of a process. A local variable is in an uninitialized state at the start of the scope it belongs to." [$9.2]

   ...

13. "...it is permissible, in abstract processes, to omit the variable reference attributes from the `<invoke/>`, `<receive/>`, and `<reply/>` activities. The meaning of such an omission must be stated clearly." [$15.1]

14. "If no variable is specified for an incoming message, then the abstract process may not refer subsequently to the message or its properties (if any)." [$15.1]

15. "If the variable reference is omitted for an outgoing message, then any properties of the message are considered to have been initialized through opaque assignment..." [$15.1]

5.1.3 Initial Definitions

In the BPEL Abstract Machine, for a clear separation of concerns, we abstract from data types (types of variables) and data values by introducing a well-defined interface between the behavioural model and the data model, so that future changes in the data model will not result in changes of the behavioural model, and the behavioural model will be re-usable for other similar architectures. Thus, three domains are defined in this extension representing three types of variables[2]. A static function *varType* holds the relation between a variable and its type.

---

domain VARIABLE
domain MESSAGE_TYPE
domain XML_TYPE
domain XML_ELEMENT

$varType$: VARIABLE $\rightarrow$ MESSAGE_TYPE $\cup$ XML_TYPE $\cup$ XML_ELEMENT

To satisfy Requirement 3 of DRL, we assume that all the variable names are unique in the entire process program (which helps us not to deal with scopes of variables). Assuming that all local variable names are unique within their scope[3], we provide the following procedure to be performed in the pre-processing phase. All local variable names are prefixed with their corresponding scope names (which are unique according to the LRM [4, Section 13]) to make them unique in the entire process. Local variable definitions can then be moved to the global variable definition.

Unique variable names and a single global set of variable definitions help us to have a one-to-one matching of variable names and their representations in our model, and eliminates the need to define a separate domain for variable names.

We define a dynamic function *varValue* that holds the value of a variable in a specific process. Data values are abstracted by introducing domain VALUE that contains all the possible data values. Similarly, data expressions are generally represented by a domain EXPRESSION, where an external function *expValue* represents the result of the evaluation of an expression in the global state of a specific process. A derived function *value* is then

---

[2]See Requirement #1 in Appendix A.3.

[3]See Requirement 2 of DRL, Appendix A.3.

defined based on these two functions (*varValue* and *expValue*), to provide the value of both variables and expressions.

---

domain VALUE
domain EXPRESSION

$varValue$: VARIABLE × PROCESS → VALUE
$expValue$: EXPRESSION × PROCESS → VALUE

$value$: (EXPRESSION ∪ VARIABLE) × PROCESS → VALUE

$$value(x,p) \equiv \begin{cases} varValue(x,p), & \text{if } x \in \text{VARIABLE;} \\ expValue(x,p), & \text{if } x \in \text{EXPRESSION.} \end{cases}$$

The complete list of initial definitions is presented in Appendix C.

5.1.4 Variables in Inbound/Outbound Communication

In this section, we present how the *data handling extension* enables the BPEL Abstract Machine to use variables in input/output operations. The ultimate purpose of having variables in an orchestration language like BPEL is to used them in input/output operations. Messages coming from business partners must be stored to allow further processing of each message. To interactively communicate with business partners, BPEL processes need to perform computations on inbound messages (requests) and store the results so that they can further be used to generate outbound messages (response). Thus, an output operation must be able to retrieve the outbound message from a stored location in the process instance.

The *core* of the BPEL Abstract Machine abstracts from BPEL variables. It is valid to have BPEL process definitions that do not deal with variables. The LRM calls such processes, an *abstract process* or a *business protocol* [4, Section 1]. Data handling extension supplements the BPEL Abstract Machine with variables and data handling. Careful attention has been made to clearly separate the outbound communication behaviour of BPEL from its data handling behaviour in the BPEL Abstract Machine, so that there are only three ASM rules that need to be refined in this extension.

ExecuteReceive(*activity* : RECEIVE) $\equiv$
  if $\neg receiveMode(self) \wedge \neg outstandingReceiveConflict(activity)$ **then**
    $receiveMode(self) := true$ // The running agent is waiting to receive a message
    GenerateInputDescriptor(*activity*)

  if $receiveMode(self)$ **then**
    choose $descriptor \in completedInOperations(self)$
        with $dscAgent(descriptor) = self \wedge dscOperation(descriptor) = activity$
      $receiveMode(self) := false$
      FinalizeActivity(*activity*)

GenerateInputDescriptor(*operation* : INPUT_OPERATION) $\equiv$
  extend INPUT_DESCRIPTOR **with** *descriptor*
    SetInOutDescriptor(*operation*, *descriptor*)
    add *descriptor* **to** $waitingSetForInput(rootProcess(self))$

SetInOutDescriptor(*operation* : OUTPUT_OPERATION, *dsc* : INOUT_DESCRIPTOR) $\equiv$
  $dscAgent(dsc) := self$
  $dscOperation(dsc) := operation$

Spec 5.1: Behaviour of the *receive* activity

Input Operations

Input operations, like *receive*, perform a blocking wait for an inbound message to arrive. Their behaviour is modelled in the *core* using two consecutive phases, which are distinguished by *receiveMode*, a predicate defined on KERNEL_AGENTs. In the first phase, they create an *input descriptor* and add it to the *waitingSetForInput* of their corresponding process. In the second phase, they are basically waiting for the inbox manager to receive an appropriate message and move the input descriptor to the set of completed input descriptors (*completedInputOperations*). The corresponding ASM rules are presented in Spec 5.1. Other input operations (activities) in BPEL practically follow the same approach. The complete specification of these activities is provided in Appendix B.

Spec 5.2 presents the inbound communication behaviour of the *core*, in particular the behaviour of the inbox manager which serves input operations that are waiting for inbound messages. At each run, if there is any arrived message in the $inboxSpace(self)$, the inbox manager picks a matching set of a process, a message and an input descriptor, and employs

// ———— ———— – Inbox Manager Program —— ——
InboxManagerProgram $\equiv$
  if $inboxSpace(self) \neq \emptyset$ **then**
    choose $p \in$ PROCESS, $m \in inboxSpace(self)$,
      $descriptor \in waitingSetForInput(p)$ **with**
      $waitingOnIO(dscAgent(descriptor), p) \wedge match(p, operation, m)$
    AssignMessage$(p, descriptor, m)$
    if $p = dummyProcess$ **then**
      new $newDummy$: PROCESS
        $dummyProcess := newDummy$
where
  $operation \equiv dscOperation(descriptor)$

// ———— ———— Assign Message —— ———— ——
AssignMessage$(p$: PROCESS, $descriptor$: INPUT_DESCRIPTOR, m : MESSAGE$) \equiv$
  if $initiateCorrelation(op)$ **then**
    InitiateCorrelation$(p, descriptor, m)$
  $dscCompletedTime(descriptor) := now$
  add $descriptor$ **to** $completedInOperations(p)$
  remove m **from** $inboxSpace(self)$
  remove $descriptor$ **from** $waitingSetForInput(p)$
where
  $op \equiv dscOperation(descriptor)$,
  $agent \equiv dscAgent(descriptor)$

Spec 5.2: Inbound communication behaviour: Revised

the AssignMessage rule to assign the message to the waiting operation. A detailed specification of the AssignMessage is provided in Spec 5.2. Assign Message basically checks for correlation requirements, removes the input descriptor from the *waitingSetForInput*, assigns a receive time (operation completion time) to the descriptor and adds it to the set of *completedInOperations*. An in-depth description of this rule is provided in [43, Section 5.1]. As the *core* does not deal with BPEL variables, the arrived message is not stored anywhere.

To extend the inbound specification of the *core* with data handling behaviour, there is only one rule that needs to be extended: AssignMessage.AssignMessage$_{data}$ is defined to assign the message to the variable which is referenced in the input operation. Spec 5.3 presents this definition. The static function *variable* refers to the variable of an input/output operation:

// ——————— AssignMessage Extended ———————
AssignMessage($p, descriptor, m$) $\equiv$
  AssignMessage$_{core}$($p, descriptor, m$)
  AssignMessage$_{data}$($p, descriptor, m$)

AssignMessage$_{data}$(p : PROCESS, $descriptor$: INPUT_DESCRIPTOR, m : MESSAGE) $\equiv$
  if $variable(dscOperation(descriptor)) \neq undef$ **then**
    $value(variable(dscOperation(descriptor)), p) := m$

Spec 5.3: Extending inbound communication behaviour with data handling

$variable$: INOUT_OPERATION $\rightarrow$ VARIABLE

AssignMessage is then extended to execute AssignMessage$_{data}$ in parallel to AssignMessage rule of the *core* (which is now called AssignMessage$_{core}$). Message assignment is performed only when the variable reference of the input operation is not undefined, which preserve the behaviour of the *core* when there is no variable reference for the input operation.

Output Operations

Outbound communication is discussed in Section 4.2.5. To extend outbound communication behaviour with data handling behaviour, there are only two rules that need to be extended: *SetInOutDescriptor* and DeliverMessage.

To send out a message, it is important that outbox manager uses the exact value of the output variable at the time of the execution of the output operation. Parallel structures in BPEL (using the *flow* activity) makes it possible for the value of the variable referenced by the output operation to be changed before the outbox manager actually sends the corresponding message out. Thus, the value of the variable (the outbound message) should also be attached to the output descriptor.

The idea of using descriptors, which is discussed in Section 4.2.4, enables us to simply attach a new property to descriptors that holds the value of the variable that is referenced by the input/output operation. This is handled by the *dscVariableValue* function defined below:

```
DeliverMessage_core(p : PROCESS, descriptor : OUTPUT_DESCRIPTOR) ≡
  if variable(opr) = undef then
    add opaqueMessage(opr) to outboxSpace(self)
    if initiateCorrelation(opr) then
      InitiateCorrelation(p, descriptor, opaqueMessage(opr))
  dscCompletedTime(descriptor) := now
  add descriptor to completedInOperations(p)
  remove descriptor from waitingSetForOutput(p)
where
    opr ≡ dscOperation(descriptor)
```

Spec 5.4: DeliverMessage in the *core*

---

dscVariableValue : INOUT_DESCRIPTOR → VALUE

In the *core* of the BPEL Abstract Machine, SetInOutDescriptor is responsible to initialize input and output descriptors before they are actually added to the waiting sets. In the *data handling extension*, this rule should be refined to assign the actual value of the output operation to its corresponding descriptor. The extended SetInOutDescriptor is presented in Spec 5.5.

Since variables are not considered in the *core* of the BPEL Abstract Machine, there is no actual message to be sent out. Thus, after checking that there is no variable reference in the output operation[4], DeliverMessage sends out an opaque message (abstract message). The definition of $\text{DeliverMessage}_{core}$ is presented in Spec 5.4.

When there is a variable reference defined for the output operation, $\text{DeliverMessage}_{data}$ should send out the message that is stored in the referenced variable. This rule first checks that correlation requirements (by checking the *correlationSatisfied* predicate) and then adds the message value of the variable into the *outboxSpace*. DeliverMessage is re-defined to execute $\text{DeliverMessage}_{core}$ and $\text{DeliverMessage}_{data}$ together.

[4]This is a guard condition that enables future extensions of the rule (see Section 4.1.2).

// ——— —— SetInOutDescriptor Extended —— ————

SetInOutDescriptor$(descriptor, operation, agent) \equiv$
  SetInOutDescriptor$_{core}(descriptor, operation, agent)$
  SetInOutDescriptor$_{data}(descriptor, operation, agent)$

SetInOutDescriptor$_{data}(descriptor : \text{INOUT_DESCRIPTOR},$
      $operation : \text{INOUT_OPERATION}, agent : \text{KERNEL_AGENT}) \equiv$
  if $operation \in \text{IN_OPERATION} \wedge variable(operation) \neq undef$ **then**
    $dscVariableValue(descriptor) := value(variable(operation), rootProcess(self))$

// ———————— DelvierMessage Extended ———— ————

DeliverMessage$(p, descriptor) \equiv$
  DeliverMessage$_{core}(p, descriptor)$
  DeliverMessage$_{data}(p, descriptor)$

DeliverMessage$_{data}(p : \text{PROCESS}, descriptor : \text{OUTPUT_DESCRIPTOR}) \equiv$
  if $variable(operation) \neq undef$ **then**   // variable should contain a message
    if $correlationSatisfied(descriptor)$ **then**
      add $messageValue(dscVariableValue(descriptor))$ **to** $outboxSpace(self)$
where
  $operation = dscOperation(descriptor)$

Spec 5.5: Extending outbound communication behaviour with data handling

ExecuteAssign(*activity* : ASSIGN)
  forall *c* **in** *copyElements*(*activity*)
    ExecuteCopy(*fromSpec*(*c*), *toSpec*(*c*))
  FinalizeActivity(*activity*)

Spec 5.6: The behavioural definition of the *assign* activity

5.1.5 The Assign Activity

The *assign* activity in BPEL is a set of *copy* elements. Each *copy* element is a pair of a *from-spec* and a *to-spec* element corresponding to the *from* and *to* elements of a *copy* element in BPEL (see Section 5.1.1). The semantics of a *copy* element is to copy the value of *from-spec* to the *to-spec*. This activity is discussed in more details in Section 5.1.1.

We define the following domains for *copy* elements, *from-spec*, and *to-spec* elements:

domain COPY_ELEMENT
domain FROM_ELEMENT
domain TO_ELEMENT

The set of all the *copy* elements of an *assign* activity is represented by unary function *copyElements*. For each *copy* element, *fromSpec*, and *toSpec* refer to its *from-spec* and *to-spec*.

copyElements : ASSIGN → COPY_ELEMENT-set

fromSpec : COPY_ELEMENT → FROM_ELEMENT
toSpec : COPY_ELEMENT → TO_ELEMENT

The behaviour of the *assign* activity, specified by the ExecuteAssign rule, is presented in Spec 5.6. Since we abstract from the data model, we define an abstract rule called ExecuteCopy to copy the value of a *from-spec* to a *to-spec*. FinalizeActivity, which is defined in the *core* of the BPEL Abstract Machine, will set the execution mode to *Activity-Completed* and deals with synchronization issues [43].

```
ScopeProgram ≡
  case execMode(self) of
    emStarted →
      execMode(self) := emRunning
      InitializeLocalVariables(baseActivity(self))

    emRunning →
      if normalExecution(self) then
        onsignal s : AGENT_COMPLETED
          execMode(self) := emActivityCompleted
        otherwise
          ExecuteActivity(innerActivity(baseActivity(self)))

    emActivityCompleted → FinalizeKernelAgent

    emCompleted → stop self
```

Spec 5.7: Behavioural specification of the *scope* activity in data handling extension

5.1.6 The Scope Construct

We introduced *scope* in Section 5.1.1. A *scope* in BPEL is just a wrapper around an activity to provide a local context for the execution of the activity. This section presents a formal specification for the behaviour of scope excluding its fault and compensation handling behaviour. Since scopes are one of the fundamental constructs of fault and compensation handling in BPEL, we will get back to scopes in Section 5.2.

Without focusing on fault and compensation handling behaviour, the behaviour of scope will be reduced to a simple wrapper around its main activity. Similar to structured activities, we define a new type of agents called *scope agents*, to handle the execution of *scope* activities. Spec 5.7 presents the ScopeProgram.

Considering that we converted local variables to global variables (see Section 5.1.3), to satisfy Requirement 8 of FCRL[5], scope agents have to set the value of local variables to an uninitialized value in the *Started* mode. This is performed by the InitializeLocalVariables rule. This rule is formally defined in Appendix C.

In the *Running* mode, the behaviour of *scope* is only defined when the *normalExecution* predicate is true, which is the case in the core model and the *data handling extension*. When

[5]See Appendix A.3.

a scope agent is in the *Running* mode, receiving an *agent-completed* signal means that the child agent of this scope (there can only be one child agent) has completed its execution. The scope agent should then go to the *Activity-Completed* mode. Otherwise, the agent keeps executing its main activity.

Like many other activity agents, scope agents finalize their activity using the FinalizeActivity rule, which takes them to the *Completed* mode, where the agent stops its execution.

The complete ASM specification of the *data handling extension* of the BPEL Abstract Machine is provided in Appendix C.

5.2 Fault and Compensation Extension

The *fault and compensation extension* supplements the *core* of the BPEL Abstract Machine with compensation and fault handling behaviour. This is a fairly complex issue in the definition of BPEL. An in-depth analysis in fact shows that the semantics of fault and compensation handling, even when ignoring all the syntactical issues, is related to more than 40 individual requirements spread out all over the LRM. These requirements (some of them comprise up to 10 sub-items) address a variety of separate issues related to the core semantics, general constraints, and various special cases. This section provides an overview of the fault handling and compensation behaviour in BPEL (Section 5.2.1) and presents a list of the requirements on fault handling and compensation behaviour extracted from the LRM (Section 5.2.2). The process execution model underlying the BPEL Abstract Machine is extended in Section 5.2.3 to include fault handling and compensation behaviour of BPEL processes. We then provide a comprehensive definition of the *fault and compensation extension* of the BPEL Abstract Machine.

5.2.1 Fault and Compensation Handling in BPEL

Business processes typically perform durative transactions through asynchronous communication between partners. Such transactions normally cause local updates at the interacting partners. Consequently, when an error occurs, it may be required to reverse the effects of some or even all of the previous activities. This concept is known as *compensation*. The ability to compensate exceptions in an application-specific manner enables business processes to have so-called Long-Running (Business) Transactions (LRTs).

In BPEL, compensation and fault handling is performed using the *scope* activity. *Scope* provides a logical unit for which a *compensation handler* and a set of *fault handlers* can be defined. A compensation handler defines the compensating behaviour of a logical unit in case of an error. A fault handler defines the reaction of a logical unit to an error.

Compensation Handlers

A compensation handler is defined within a scope and forms a wrapper around an activity that is considered to be the *compensation activity* of that scope. Compensation handlers enable business process designers to define compensating behaviour for a scope in case of an error. The compensation activity can be any BPEL activity, including another *scope*. A scope can only be compensated after its successful completion. When a scope finishes successfully, the compensation handler of that scope is said to be *installed* for possible future invocations.

Compensation handlers can be invoked by means of the *compensate* activity. A *compensate* activity requires the name of the scope to be compensated and can only be called from a fault handler or a compensation handler *"of the scope that immediately encloses the scope for which compensation is to be performed."*[4, Section 13.3.2] Figure 5.1 illustrates this concept using an example of two scopes, A and B, where scope B is enclosed by scope A.

The semantics of the *compensate* activity is somewhat complex. At the time of producing this document, there were still ongoing debates among the OASIS WSBPEL-TC committee members on the semantics of this activity. Requirements B-1 to B-7 of the fault and compensation handling requirements list provided in Appendix A.4 address some of the issues regarding the behavioural semantics of the *compensate* activity.

Basically, the behaviour of a *compensate* activity with a reference to scope S is defined as executing the compensation handler (compensation activity) of scope S. Nevertheless, there are a number of cases to be considered, such as:

- If scope S does not have a compensation handler, a default compensation behaviour should invoke the compensation handlers for the immediately enclosed scopes in the reverse order of the completion of those scopes[6].

[6] See Requirement B6 of the fault and compensation requirements in Appendix A.4.

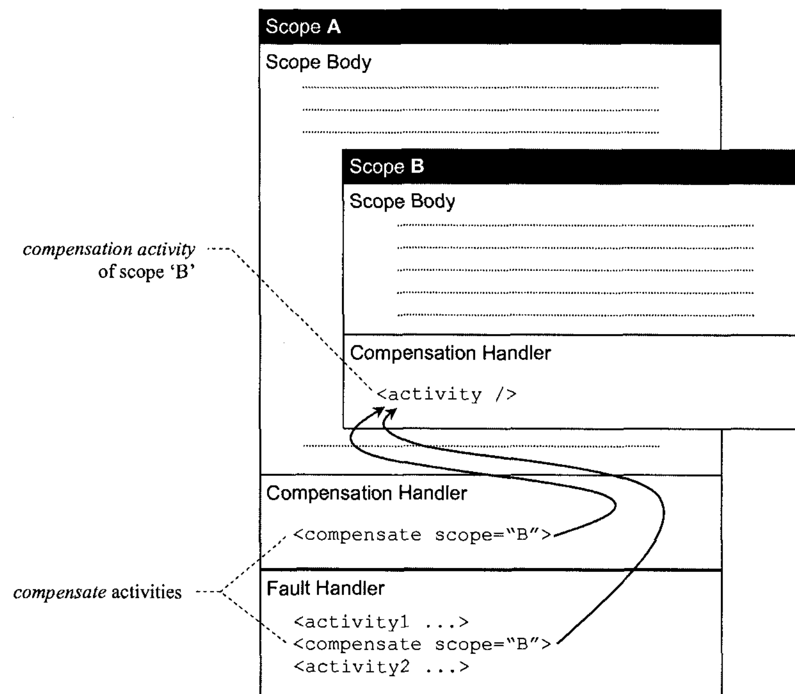


Figure 5.1: A *compensate* activity invokes the *compensation activity* of an enclosed scope

- If scope S was nested in a loop, the instances of the compensation handlers in the successive iterations are invoked in reverse order.

- If scope S was not successfully completed, invoking its compensation handler is equivalence to the *empty* activity.

For a better overview and understanding, we have extracted from the LRM individual issues that are related to requirements of compensation behaviour. A complete list of these issues is provided in Appendix A.4.

Fault Handlers

Fault handlers provide a way to define how scopes should react to an error. This reaction is meant to *undo* (i.e., reverse) the work of a successfully completed scope. A fault handler consists of a number of optional *catch* clauses for handling specific types of faults and one optional *catchall* clause to deal with all other faults. Each *catch* or *catchall* clause wraps around one BPEL activity that defines the response of that clause to related faults.

Requirement C8 of the fault and compensation requirements (see Appendix A.4) specifies how *catch* clauses are selected in a fault handler.

The completion of a scope in which a fault is thrown is never considered successful, even when the fault handler successfully handles the fault. Thus, a compensation handler for such a scope will not be installed and the scope cannot be compensated in the future. Such a scope is considered to be *exited*, rather than completed. If the scope has no suitable fault handler that can handle the fault, or if the fault handler in turn encounters a fault that cannot be handled, the fault is thrown to the next enclosing scope and the scope is considered to be *faulted*[7].

The LRM indicates that a fault handler starts its execution by implicitly terminating all those activities that are currently active and directly enclosed within the scope of the fault handler [4]. Thus, occurrence of a fault in a scope immediately (prior to any reaction) leads to termination of the execution of the scope[8]. When there is no suitable fault handler available for a fault, the fault will invoke a default fault handler. A default fault handler of a scope will run all available compensation handlers for immediately enclosed scopes in the reverse order of completion of the scopes, and then rethrows the fault to the next (higher) enclosing scope[9].

To explicitly signal an internal fault, BPEL introduces the *throw* activity. A *throw* activity gets a fault name and an optional fault variable (a variable that contains extra information about the fault) and throws a fault at the time of its execution[10].

Travel Agency: an Example of Compensation Behaviour

The description of the travel agency business process presented in Section 2.3.2 can be extended with a compensation module (see Figure 5.2). A fault may occur when a process instance in the travel agency Web service is waiting to receive a confirmation from its client along with the required credit card information. This fault could be a cancel response from the client, an incorrect piece of information or a communication problem. Due to such a fault, the normal execution of the process instance is cancelled and a compensation module

[7]Requirement B11 of the fault and compensation requirements list provided in Appendix A.4 explores this aspect in more detail.

[8]See Requirement B16 in Appendix A.4.

[9]See Requirement B8 in Appendix A.4.

[10]See Requirement B10 in Appendix A.4.

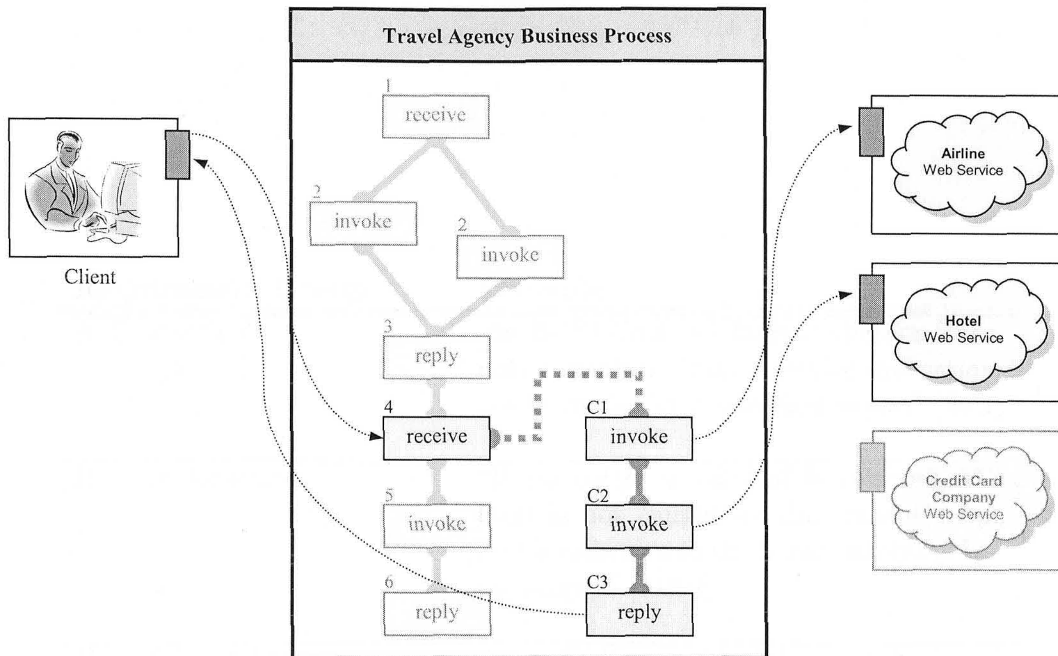


Figure 5.2: A compensation module cancels flight and room reservations

is invoked. In this example, the compensation module cancels flight and room reservations and sends a notification to the client.

5.2.2 Requirements

The semantics of fault and compensation handling, even when ignoring all the syntactical issues, is specified by more than 40 individual requirements scattered out over the LRM in 6 different chapters. A complete list of these requirements, classified in 6 different categories (*syntactical*, *core semantics*, *details and constraints*, *special cases*, *interpretation*, and *extensibility*), is presented in Appendix A.4. Table 5.1 presents some examples of these requirements. Henceforth, we use the term *FCRL* to refer to this list. This section presents a number of these requirements that are further discussed in this chapter.

5.2.3 Process Execution Model: Fault Handling

Section 4.2.1 introduces a process execution model for kernel agents based on the underlying normal execution model of BPEL. This section extends that model to capture the fault handling execution model of BPEL.

Requirement Group	**Example**
A: Syntactical	"In BPEL4WS, all faults, whether internal or resulting from a service invocation, are identified by a qualified name." [$6.1]
B: Core Semantics	"If no *catch* or *catchall* is selected, the fault is not caught by the current scope and is rethrown to the immediately enclosing scope." [$13.4]
C: Details and Constraints	"The fault variable [of a *catch* clause] is optional because a fault might not have additional data associated with it." [$13.4]
D: Special Cases	"The semantics of a process in which an installed compensation handler is invoked more than once is undefined." [$13.3.2]
E: Interpretation	"Fault handling in a business process can be thought of as a mode switch from the normal processing in a scope." [$13.4]
F: Extensibility	"In the future, BPEL4WS will add input and output parameters to compensation handlers..." [$13.3.1]

Table 5.1: Requirement groups of fault and compensation handling in BPEL

Fault handling in BPEL can be thought of as a mode switch from the normal execution of the process[11]. When a fault occurs in the execution of an activity, the fault is thrown up to the innermost enclosing scope. If the scope handles the fault successfully, it sends an *exited* signal to its parent scope and ends gracefully, but if the fault is re-thrown from the fault handler, or a new fault is occurred during the fault handling procedure, the scope sends a *faulted* signal along with the thrown fault to its parent scope. The fault is thrown up from scopes to parent scopes until a scope handles it successfully. A successful fault handling switches the execution mode back to normal. If a fault reaches the global scope, the process execution terminates. The Coordination Protocol presented in Requirement B11 of FCRL explores this behaviour in more detail (see Appendix A.4).

The normal execution lifecycle of the process execution model presented in Chapter 4 (Figure 4.5) needs to be extended to comprise the fault handling mode of BPEL processes. The occurrence of a fault causes the kernel agent (be it an activity agent or the main process) to leave its normal execution lifecycle and enter a fault handling lifecycle. Figure 5.3 illustrates the extended execution lifecycle of BPEL activities.

When a kernel agent encounters a fault, it leaves its normal execution by switching to the *Execution-Fault* mode. If the kernel agent is neither a scope agent nor a process agent, it should also notify its parent agent of the fault. This transition is performed by the TransitionToExecutionFault rule. For every kernel agent, the dynamic function *faultThrown* (defined on kernel agents) keeps the current fault which is thrown in the execution of the agent. The default value of *faultThrown* is *undef*.

---

fault : (AGENT_FAULTED ∪ FORCED_TERMINATION) → FAULT
faultThrown : KERNEL_AGENT → FAULT

TransitionToExecutionFault(*fault* : FAULT) ≡
  execMode(*self*) := *emExecutionFault*
  faultThrown(*self*) := *fault*
  if *self* ∉ (SCOPE_AGENT ∪ PROCESS) **then**
    trigger *s* : AGENT_FAULTED, *parentAgent*(*self*)
      fault(*s*) := *fault*

From the *Execution-Fault* mode, the execution path of scope agents (and process agents)

[11]See Requiremet E2 in Appendix A.4.

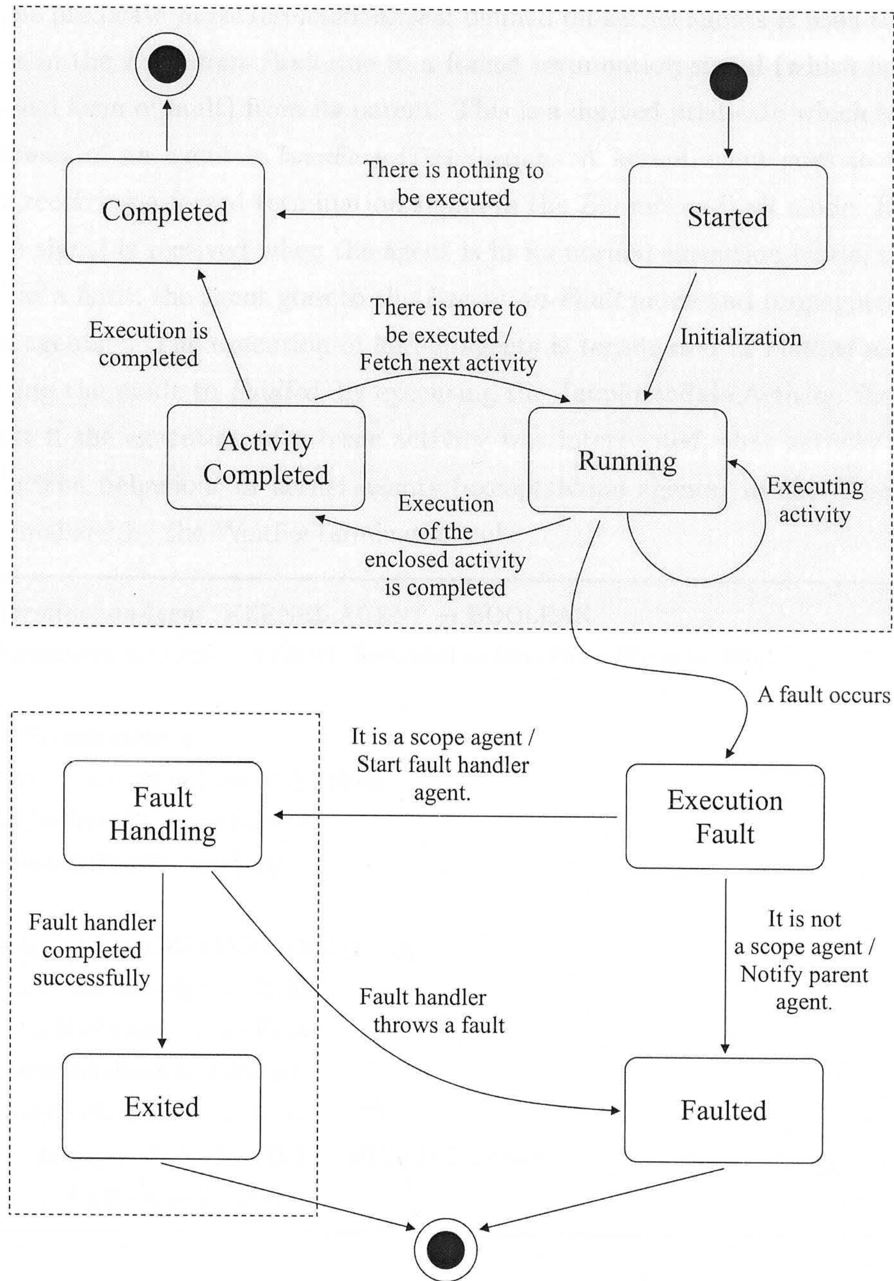


Figure 5.3: Activity execution lifecycle: extended by fault handling

becomes different from other kernel agents. In the *Execution-Fault* mode, any kernel agent which is not a scope or a process, waits to receive a forced termination signal from its parent agent[12]. The predicate *forcedTerminationAgent* defined on kernel agents is used to indicate if the agent is in the *Execution-Fault* due to a forced termination signal (which is considered to be a special form of fault) from its parent. This is a derived predicate which is true when the *faultThrown* of an agent is *bpwsForcedTermination*. A kernel agent goes to the *Faulted* mode after receiving a forced termination signal in the *Execution-Fault* mode. If the forced termination signal is received when the agent is in its normal execution mode, the signal is considered as a fault, the agent goes to the *Execution-Fault* mode and propagates the signal to its child agents[13]. The execution of kernel agents is terminated in *Faulted* mode. Along with changing the mode to *Faulted*, by executing the TerminateBasicActivity, the agent also ensures that if the execution of a basic activity was interrupted, that activity is finalized properly[14]. The behaviour of kernel agents (except scope agents) in the *Execution-Fault* mode is formalized by the WaitForTermination rule.

---

forcedTerminationAgent : KERNEL_AGENT → BOOLEAN
forcedTerminationAgent(a) ≡ (*faultThrown(a)* = *bpwsForcedTermination*)

WaitForTermination ≡
  if *forcedTerminationAgent(self)* **then**
    execMode(self) := *emFaulted*
    TerminateBasicActivity(*self*)
  else
    onsignal *s* : FORCED_TERMINATION
      faultThrown(self) := *fault(s)*
      execMode(self) := *emFaulted*
      TerminateBasicActivity(*self*)
      forall *child* **in** *childAgents(self)*
        trigger *s′* : FORCED_TERMINATION, *child*
          fault(s′) := *fault(s)*

As mentioned earlier, scope agents (and process agents as global scopes) have a different

[12]This is to comply with Requirement B16 of FCRL.

[13]This complies with the Requirements B16 and B11 (vi and vii) of FCRL in Appendix A.4.

[14]See Requirement B16 of FCRL.

behaviour in the *Execution-Fault* mode. A scope agent in this mode should switch to the *Fault-Handling* mode, trying to handle the fault by executing its fault handler. If the fault handles throws a fault (or rethrows the original fault) the scope agent switches to the *Faulted* mode and terminates. If the fault handling procedure is successful, the scope agent terminates in the *Exited* mode. Section 5.2.5 explains this behaviour in more details.

The extended program of sequence agents is presented in Spec 5.8 as an example on how kernel agents (except scope agents) are extended to observe fault handling requirements. The *core* version of the sequence program is presented in Section 4.2.2. The sequence program of the *core*, henceforth referred to as $\mathsf{SequenceProgram}_{core}$, is extended without changing its original behaviour. The $\mathsf{SequenceProgram}_{core}$ is guarded in the *Running* mode by the *normalExecution* predicate which was equal to true in the *core*. In this extension a new predicate, *faultExtensionSignal*, is defined to be true if the agent receives a signal that is related to the *fault and compensation extension*. The predicate *normalExecution* is then refined to be the negation of *faultExtensionSignal* which prevents the execution of $\mathsf{SequenceProgram}_{core}$ when the agent has to deal with faults.

$faultExtensionSignal : \text{KERNEL_AGENT} \rightarrow \text{BOOLEAN}$

$faultExtensionSignal \equiv$

$\quad \exists s (s \in signalSet(rootProcess(self)) \wedge signalSource(s) = self \wedge$

$\quad\quad s \in (\text{AGENT_EXITED} \cup \text{AGENT_FAULTED} \cup \text{FORCED_TERMINATION})$

$normalExecution(a : \text{KERNEL_AGENT}) \equiv \neg faultExtensionSignal(a)$

An *agent-exited* signal from a child agent indicates an unsuccessful completion of the child agent. But according to Requirement B11-ii-B of FCRL (see Appendix A.4), this is not considered a faulted completion. Thus, a sequence agent goes to the *Activity-Completed* mode upon receiving an agent-exited signal. An *agent-faulted* signal from a child agent indicates that the child agent finished in *Faulted* mode. The parent agent then switches to the *Execution-Fault* mode using the TransitionToExecutionFault rule, after receiving an agent-faulted signal. A *forced-termination* signal sent from a parent agent to a child agent indicates that the execution of the child agent must be terminated. This signal is also treated as a fault by the recipient agent which changes the execution mode of the agent to the *Execution-Fault* mode. The agent then propagates the forced-termination signal to its child agents in order to terminate all its enclosed activities.

SequenceProgram ≡
  SequenceProgram$_{core}$
  case *execMode*(*self*) **of**
    emRunning → HandleExeceptionsInRunningMode
    emExecutionFault → WaitForTermination
    emFaulted → **stop** *self*

HandleExceptionsInRunningMode ≡
  if *faultExtensionSignal*(*self*) **then**
    onsignal *s* : AGENT_EXITED
      execMode(*self*) := *emActivityCompleted*
    otherwise
      onsignal *s* : AGENT_FAULTED
        TransitionToExecutionFault(*fault*(*s*))
      otherwise
        onsignal *s* : FORCED_TERMINATION
          faultThrown(*self*) := *fault*(*s*)
          forall *child* **in** *childAgents*(*self*)
            trigger *s'* : FORCED_TERMINATION, *child*
              fault(*s'*) := *fault*(*s*)
          execMode(*self*) := *emExecutionFault*

Spec 5.8: Sequence program: extended by fault and compensation behaviour

Although the presented program in this section is a sequence program, the fault and compensation handling extension is designed in such a way that all other kernel agents, except scopes and processes, share the same extended behaviour in the *Running* mode, using the HandleExceptionsInRunningMode rule. The complete formalization including the initial state, rules and programs is provided in Appendix D.

5.2.4 Throwing Faults

A business process in BPEL can throw a fault internally using the *throw* activity. The *throw* activity gets a fault name and an optional fault variable that holds additional information about the fault, and throws an internal fault at its execution point.

In the BPEL Abstract Machine, the semantics of *throw* is captured by the ExecuteThrow rule, which basically uses the TransitionToExecutionFault rule, the same rule that other activities use when they encounter a fault in their execution. The TransitionToExecutionFault

rule (see Section 5.2.3) takes care of the transition of the kernel agent to the *Execution-Fault* mode. The ExecuteThrow rule also uses the Synchronization rule to handle activity synchronization issues [43].

ExecuteThrow(*activity* : THROW) ≡
  TransitionToExecutionFault(*activityFault*(*activity*))
  Synchronization(*activity*)

5.2.5 Scope Agent: Refined

Scopes in BPEL are the core of fault and compensation handling behaviour. Fault handlers and compensation handlers are both defined for local scopes and the main process which is considered a global scope. In this section, the behaviour of *scope* presented in Section 5.1.6 is extended to cover fault and compensation handling.

The Running Mode

The scope program is refined using the same approach as the refinement of the sequence program (see Section 5.2.3). In the *Running* mode, receiving an agent-exited signal indicates that the execution of the child agent is completed (with a fault which is handled and is not thrown upwards). So, upon receiving such a signal, the scope agent goes to the *Activity-Completed* mode which will eventually lead to a successful completion of the execution of the agent (complies to the protocol presented in B11 of FCRL).

A scope agent behaves differently from other activity agents in the sense that it always tries to handle a fault thrown in its scope of execution. The scope agent treats agent-faulted signals and forced-termination signals in the same way, changing its execution mode to *Execution-Fault* and set *faultThrown*(*self*) to the fault that is associated with the signal[15]. This extended behaviour of the scope in the *Running* mode is guarded by the *faultExtensionSignal* which is described in Section 5.2.3. The ScopeAgentRunningExtended rule is presented below:

[15] In case of a forced-termination signal, this fault is always equal to the distinguished value of *bpwsForcedTermination*.

ScopeAgentRunningExtended $\equiv$
  if *faultExtensionSignal(self)* **then**
    onsignal s : AGENT_EXITED
      execMode(self) := *emActivityCompleted*
    otherwise
      onsignal s : AGENT_FAULTED
        execMode(self) := *emExecutionFault*
        faultThrown(self) := *fault(s)*
      otherwise
        onsignal s : FORCED_TERMINATION
          execMode(self) := *emExecutionFault*
          faultThrown(self) := *fault(s)*
          // The scope agent resends the forced termination signal
          // in its execution-fault mode.

The Execution-fault Mode

To comply with Requirements B12 and B16 of FCRL (see Appendix A.4), two tasks must be accomplished by a scope agent in the *Execution-Fault* mode: (1) terminating all activities directly enclosed within the scope, and (2) executing the fault handler rule and changing the execution mode to *Fault-Handling.* TerminateBasicActivity ensures that if the enclosed activity of the scope is a basic activity, it is terminated properly without leaving any trace (e.g., no input descriptor is left waiting for a message). A forced-termination signal is also sent to any child activity agent if the enclosing activity of the scope is a structured activity. Meanwhile, an instance of the fault handler agent is created and is initialized with proper properties to handle the thrown fault. The ScopeAgentExecutionFault rule presented below, specifies the behaviour of scope agents in their *Execution-Fault* mode:

---

handlerScope : FAULT_HANDLER_AGENT → SCOPE

ScopeAgentExecutionFault ≡
  TerminateBasicActivity(*self*)
  forall *child* **in** *childAgents*(*self*)
    trigger *s* : FORCED_TERMINATION, *child*
      fault(*s*) := *bpwsForcedTermination*
  new *handler* : FAULT_HANDLER_AGENT
    parentAgent(*handler*) := *self*
    handlerScope(*handler*) := *baseActivity*(*self*)
    faultThrown(*handler*) := *faultThrown*(*self*)
  execMode(*self*) := *emFaultHandling*

It is optional for scopes to have fault handlers[16], but there is always a default fault handling procedure that is performed when there is no fault handler defined for a scope. This default fault-handling procedure is documented in Requirement B8 as follows [4]:

1. Run all available compensation handlers for immediately enclosed scopes in the reverse order of completion of the corresponding scopes.

2. Rethrow the fault to the next enclosing scope.

To provide this default fault handling, we assume that in the pre-processing phase (see Section 3.3), a fault handler is attached to all the scopes without a fault handler, with a *catchall* clause that includes two activities: `<compensate/>` and `<rethrow/>`. According to the Requirement B7 of FCRL[17], the `<compensate/>` activity can be used to perform the first task. The *rethrow* activity is not defined in the LRM and is introduced later in BPEL (see Section 6.1.5) to rethrow the fault which is caught in a *catchall* clause.

The Fault-handling Mode

The fault-handling mode for a scope agent is basically a waiting mode. When a scope is in fault-handling mode, it is basically waiting for its fault handler to complete its execution. There are three possible signals that can be received by scope in this mode:

---

[16]See Requirement B12 of FCRL, Appendix A.4.

[17]See Appendix A.4.

- An *agent-completed* signal in this mode indicates that the fault handler completed its execution successfully. This will result in completion of the scope program in an *Exited* mode (see Requirement B11).
- An *agent-faulted* signal indicates that the fault handler encountered an internal fault and abnormally terminated. The scope agent also terminates by switching to the *Faulted* mode and throwing the fault thrown by the fault handler to its parent agent[18].
- Upon receiving of a *forced-termination* signal, the scope agent propagates the signal to its child agent(s) and terminates its execution by going to the *Faulted* mode. The response of a fault handler to a forced-termination signal is discussed in Section 5.2.6.

The behaviour of scope agents in the fault handling mode is presented here:

ScopeAgentFaultHandling $\equiv$
  onsignal s : AGENT_COMPLETED
    execMode(*self*) := *emExited*
    trigger s' : AGENT_EXITED, *parentAgent*(*self*)
  otherwise
    onsignal s : AGENT_FAULTED
      faultThrown(*self*) := *fault*(s)
      trigger s' : AGENT_FAULTED, *parentAgent*(*self*)
        fault(s') := *fault*(s)
    otherwise
      onsignal s : FORCED_TERMINATION
        execMode(*self*) := *emFaulted*
        faultThrown(*self*) := *fault*(s)
        forall *child* **in** *childAgents*(*self*)
          trigger s' : FORCED_TERMINATION, *child*
            fault(s') := *fault*(s)

The program of scope agents is presented in Spec 5.9. The behaviour of scope agents in the *Activity-Completed* mode is extended by the InstallCompensationHandler rule which installs a compensation handler for the scope at the time of completion. This rule is described in detail in Section 5.2.7. For the complete list of definitions, rules and programs related to

[18] See Requirement B11 of FCRL, Appendix A.4.

ScopeProgram ≡
  ScopeProgram$_{data}$
  case *execMode*(*self*) **of**
    emRunning → ScopeAgentRunningExtended

    emActivityCompleted → InstallCompensationHandler

    emExecutionFault → ScopeAgentExecutionFault

    emFaultHandling → ScopeAgentFaultHandling

    emExited → **stop** *self*

    emFaulted → **stop** *self*

Spec 5.9: Extended specification of the *scope* activity in fault and compensation handling

scope agents, see Appendix D.

5.2.6 Fault Handling

The normal behaviour of a fault handler starts with selecting a *catch* clause that matches the fault that is being handled. The function *faultHandlerCatchSet* is defined to provide the set of *catch* clauses in the fault handler of a *scope* activity. The abstract predicate *matchingCatch* defined on catch clauses is used to find the matching *catch* clause of a fault. The chosen *catch* clause is then stored in *executingCatch* for further processing.

domain CATCH_CLAUSE
faultHandlerCatchSet : SCOPE → CATCH_CLAUSE-set
executingCatch : FAULT_HANDLER_AGENT → CATCH_CLAUSE
matchingCatch : CATCH_CLAUSE × FAULT → BOOLEAN

FaultHandlerStarted ⊟
  execMode(*self*) := *emRunning*
  choose $c \in$ *faultHandlerCatchSet*(*handlerScope*(*self*))
      with *matchingCatch*(*c*, *faultThrown*(*self*))
    executingCatch(*self*) := *c*

To model the main behaviour of fault handlers (fault handler agents in the *Running* mode), analogous to structured activities in BPEL, we can separate their behaviour into two parts: normal execution and fault-handling extended execution. The normal behaviour of a fault handler in the *Running* mode is similar to other structured activities (see *scope* in Section 5.1.6). If it receives an agent-completed signal, it goes to the *Activity-Completed* mode and finishes its execution; otherwise, it executes the selected *catch* clause. However, according to Requirement B13 of FCRL[19], if no *catch* clause is selected, the fault is rethrown. This is done by executing a pre-defined *catch* clause, called *rethrowCatchClause*.

---

rethrowCatchClause :→ CATCH_CLAUSE
// a constant function referring to a catch clause with a `<rethrow/>` activity.

FaultHandlerRunningNormal ≡
  if *normalExecution*(*self*) **then**
    onsignal *s* : AGENT_COMPLETED
      execMode(*self*) := *emActivityCompleted*
    otherwise
      if *executingCatch*(*self*) = *undef* **then**
        executingCatch(*self*) := *rethrowCatchClause*
      else
        ExecuteActivity(*catchActivity*(*executingCatch*(*self*)))

When it comes to process fault handling signals, fault handler agents only listen to two signals: agent-faulted and agent-completed. According to the LRM[20], "if the scope has already experienced an internal fault and invoked a fault handler, then [...] the forced termination has no effect." Thus, fault handler agents do not process forced-termination signals while they are in the *Running* mode.

There are some ambiguities in the LRM regarding the forced termination signal and the behaviour of scopes and fault handlers, which are discussed in Section 6.1.1.

[19]See Appendix A.4.

[20]See Requirement B15 of FCRL, Appendix A.4.

FaultHandlerRunningExtended ≡
  if *faultExtensionSignal(self)* **then**
    onsignal *s* : AGENT_EXITED
      execMode(self) := *emActivityCompleted*
    otherwise
      onsignal *s* : AGENT_FAULTED
        TransitionToExecutionFault(*fault(s)*)

Occurrence of an internal fault in the execution of a fault handler changes the execution mode of the fault handler to *Execution-Fault.* In this mode, according to the Requirement B17 of FCRL, the fault handler must terminate its execution prematurely. The FaultHandlerExecutionFault, presented below, models this behaviour by finalizing the execution of the basic activity (if any), changing the execution mode to *Faulted*, and propagating a forced-termination signal to its subordinate agent(s).

FaultHandlerExecutionFault ≡
  TerminateBasicActivity(*self*)
  execMode(self) := *emFaulted*
  forall *child* **in** *childAgents(self)*
    trigger s' : FORCED_TERMINATION, *child*
      $fault(s')$:= *bpwsForcedTermination*

The program of fault handler agents is presented in Spec 5.10. The complete specification of fault handler agents is provided in Appendix D.

5.2.7 Compensation Behaviour

Compensation behaviour is a fairly complex issue in BPEL. Originally, there were many open issues on compensation on the issue list of the OASIS WSBPEL Technical Committee [35]. Many of them are now resolved, but there are still a number of open issues yet to be resolved. Some of these issues deal with fundamental topics of compensation behaviour, like the Issue #3 of the WSBPEL Issue List which is about the "current state influence in compensation handlers". This issue changes the way a compensation handler interacts with the current state of its enclosing process as is reflected in Requirement B1 of FCRL (See Appendix A.4).

FaultHandlerProgram ≡
  case *execMode*(*self*) **of**
    emStarted → FaultHandlerStarted

    emRunning →
      FaultHandlerRunningNormal
      FaultHandlerRunningExtended

    emActivityCompleted → FinalizeKernelAgent

    emCompleted → **stop** *self*

    emExecutionFault → FaultHandlerExecutionFault

    emFaulted → **stop** *self*

Spec 5.10: Program of fault handler agents

This issue will be discussed in more detail below.

The Compensate Activity

The *compensate* activity can be used in two forms: (1) compensating a specific scope, and (2) default-order invocation of compensation handlers for completed scopes directly nested within the scope for which the fault or compensation handler is being executed[21].

There are several issues to be taken into account for compensating a scope activity. In this section we specifically focus on two major issues:

1. compensating scopes that are completed more than once, and

2. dealing with the local state view of compensation handlers.

Whenever a scope is completed successfully, a compensation handler is installed for that scope. If a scope is completed more than once, the compensation of that scope involves executing all the installed compensation handlers of that scope in their reverse order of completion; i.e., the last completion of the scope is compensated first[22]. Figure 5.4 illustrates this reverse invocation of compensation handlers.

[21]See Requirements A2 and B7 of FCRL in Appendix A.4.

[22]See Requirement B4 of FCRL, Appendix A.4.

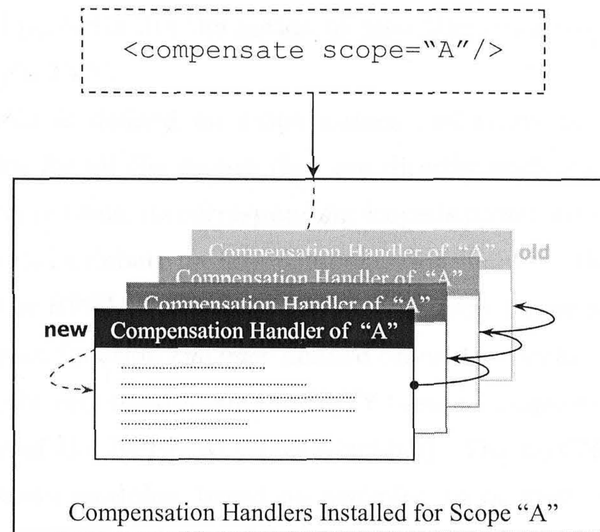


Figure 5.4: Compensation handlers are invoked in their reverse order of completion

A compensation handler for a scope defines how the work of the scope can be reversed. To accomplish this task, the compensation handler needs to see a snapshot of the local state of the scope exactly as it was when the scope was completed; all local variables should have the same value as they had at the time of completion of the scope[23].

The notion of installing a compensation handler for every successfully completed scope provides the required means to deal with the reverse invocation behaviour of the *compensate* activity and the local state view of compensation handlers. To model this notion, we define *compensation modules* representing the installed compensation handlers. Each compensation module identifies the following information:

- the scope for which the compensation handler is installed,
- a snapshot of the local state of the corresponding scope at the time of completion,
- the completion time of the scope, which helps in ordering the execution of compensation handlers.

While the behaviour of the compensation handler of scope S is unique for all instances of scope S, a compensation module basically represents a frozen local state of one instance of a complete execution of scope S, which can later be used to reverse the work of that scope

[23]See Requirement B1 of FCRL, Appendix A.4.

instance. This model perfectly fits the notion of installing compensation handlers provided by the LRM [4, Section 13.3].

The function *cmSet* is defined on scope names and refers to the set of all installed compensation modules for all the scopes that are directly enclosed inside a specific scope. For each compensation module, its corresponding scope is presented by the *cmScope* function. To this date, there is still a debate on how compensation handlers should be ordered in non-trivial cases [35]. In the BPEL Abstract Machine, an abstract representation of this order is provided by the abstract function *cmOrder* defined on compensation modules. This function assigns an element of the ordered domain PRIORITY to each compensation module (PRIORITY is defined in the *core* of the BPEL Abstract Machine). The *topCMOrder* predicate is then defined on compensation modules, based on *cmOrder*, to be true if the the compensation module has the highest order to be executed.

---

domain COMPENSATION_MODULE

cmSet : SCOPE_NAME → COMPENSATION_MODULE-set
cmScope : COMPENSATION_MODULE → SCOPE
cmOrder : COMPENSATION_MODULE → ORDER

cmScopeName : COMPENSATION_MODULE → SCOPE_NAME
cmScopeName(*cm*) := *scopeName*(*cmScope*(*cm*))

cmExecuted : COMPENSATION_MODULE → BOOLEAN
// true, if the execution of the compensation module is started

chosenCM : COMPENSATE_AGENT → COMPENSATION_MODULE
// The (current) compensation module which is being processed by the agent

topCMOrder : COMPENSATION_MODULE → BOOLEAN
topCMOrder(*cm*) ≡

$\forall cm'(cm' \in \text{COMPENSATION_MODULE} \wedge$
$(cmScopeName(cm') = cmScopeName(cm)) \rightarrow (cmOrder(cm') \leq cmOrder(cm))$

The behaviour of the *compensate* activity is abstractly modelled as follows:

a. If there is at least one installed compensation module that matches the

CompensateProgram ≡
  case *execMode(self)* **of**
    emStarted → ChooseNextCM

    emRunning → CompensateAgentRunning

      emActivityCompleted → ChooseNextCM

      emCompleted → **stop** *self*

      emExecutionFault → WaitForTermination

      emFaulted → **stop** *self*

Spec 5.11: Program of compensate agents

specified scope, then:

- choose one matching compensation module,
- remove the module from the set, and
- execute the module.

b. Terminate if there is no more matching module left or if there is a fault.

c. Repeat a.

A new kernel agent, COMPENSATE_AGENT, is introduced to model the behaviour of the *compensate* activity. The program of compensate agents is provided in Spec 5.11. To choose a compensation module, the compensate agent uses the ChooseNextCM rule defined in Spec 5.12.

If there is a matching module available (a module that belongs to the specified scope and has the highest execution order among other modules), ChooseNextCM chooses that module, assigns it as the value of *chosenCM(self)* for future execution, removes it from the set of installed modules, and changes the execution mode to *Running* so that the agent executes the chosen module. To find a matching compensation module, the predicate *matchingCM* is defined for a compensation module and holds if,

- the scope of the compensation module matches the scope specified in the *compensate* activity **or** there is no such scope specified for the *compensate* activity,

ChooseNextCM $\equiv$
  if *thereIsAtLeastOneModule* **then**
    choose *cm* **in** *cmSet(parentScopeName(self))* **with** *mathingCM(cm)*
      chosenCM(self) := *cm*
      execMode(self) := *emRunning*
      remove *cm* **from** *cmSet(parentScopeName(self))*
  else
    FinalizeKernelAgent
where
  thereIsAtLeastOneModule ⊟
    $\exists x(x \in cmSet(parentScopeName(self)) \wedge matchingCM(x))$

  $matchingCM(cm) \equiv$
    $[targetScope(baseActivity(self)) = undef$
      $\vee\ cmScopeName(cm) = targetScope(baseActivity(self))]$
    $\wedge\ topCMOrder(cm)$

Spec 5.12: The ChooseNextCM rule is performed by compensate agents

- **and** the compensation module is the first compensation module of the set which should be executed (this is modelled by the abstract predicate *topCMOrder*).

In the *Running* mode, if there is no fault signal, a compensate agent executes the selected compensation module that is stored in the *chosenCM(self)*. The response of compensate agents to agent-faulted signals and forced-compensation signals is similar to other structured activities. If there is an agent-faulted signal, the agent switches to the *Execution-Fault* mode by executing the TransitionToExecutionFault. If there is a forced-termination signal, it is propagated to the child agents and the compensate agents switches to the *Execution-Fault* mode. Spec 5.13 presents the CompensateAgentRunning rule.

The ExecuteCM defines how compensate agents execute a compensation module. As we will see later, the behaviour of compensation handlers is modelled by agents of type COMPENSATION_HANDLER. A compensation module is assigned to each compensation handler agent. The compensation handler agent, when executed, will use the assigned compensation module to set back values of local variables to what they were at the time of completion of the scope. The compensation activity of a scope (the activity inside the compensation handler of a scope) is specified by the *compensationActivity* function. If there is no compensation handler defined for a scope, this function refers to a `<compensate/>` activity which has the

```
compensationActivity : SCOPE → ACTIVITY
// The activity inside the compensation handler of this scope.
// If there is no compensation handler, the value is <compensate/>.

CompensateAgentRunning ≡
  if normalExecution(self) then
    onsignal s : AGENT_COMPLETED
      execMode(self) := emActivityCompleted
    otherwise
      ExecuteCM(chosenCM(self))
  if faultExtensionSignal(self) then
    onsignal s : AGENT_FAULTED
      TransitionToExecutionFault(fault(s))
    otherwise
      onsignal s : FORCED_TERMINATION
        faultThrown(self) := fault(s)
        forall child in childAgents(self)
          trigger s' : FORCED_TERMINATION, child
            fault(s') := fault(s)
        execMode(self) := emExecutionFault


ExecuteCM(cm : COMPENSATION_MODULE) ⊟
  if ¬cmExecuted(cm) then
    new cma : COMPENSATION_HANDLER_AGENT
      Initialize(cma, compensationActivity(cmScope(cm)))
      cmExecuted(cm) := true
      compHandlerModule(cma) := cm
```

Spec 5.13: The behaviour of compensate agents in the *Running* mode

same behaviour as the default compensation handler[24].

Installing Compensation Modules

The scope agent program presented in Spec 5.9 uses the rule InstallCompensationHandler to install a compensation handler; i.e., to create an appropriate compensation module and add it to the *cmSet* of its parent scope. As mentioned earlier, a compensation module identifies three properties:

- the corresponding scope for which the compensation handler is installed (identified by

[24]See Requirement B7 of FCRL, Appendix A.4.

functions *cmScope* and *cmScopeName*),

- a snapshot of the local state of the corresponding scope at the time of completion (identified by *localSnapshot*),
- and the completion time of the scope (identified by *scopeCompletionTime*).

The InstallCompensationHandler rule creates a compensation module, sets the associated values, and adds the compensation module to the *cmSet* of its parent scope. This rule is defined as follows:

InstallCompensationHandler $\equiv$
  extend COMPENSATION_MODULE **with** *cm*
    scopeCompletionTime(*cm*) := *now*
    cmScope(*cm*) := *baseActivity*(*self*)
    RegisterLocalSnapshot(*cm*, *baseActivity*(*self*))
    add *cm* **to** *cmSet*(*parentScopeName*(*self*))

To model a snapshot of local state variables, *local snapshots* are introduced by defining the domain LOCAL_SNAPSHOT. The function *snapshotVariableValue*(s, v) is defined to hold the value of variable v according to the snapshot s. The set of all the variables of a snapshot s is identified by the function *snapshotVariableSet*. A snapshot of local state variables assigned to a compensation module *cm* is identified by the *localSnapshot*(*cm*) function. The signatures of these functions are presented below:

---

snapshotVariableValue : LOCAL_SNAPSHOT $\times$ VARIABLE $\rightarrow$ VALUE
snapshotVariableSet : LOCAL_SNAPSHOT $\rightarrow$ VARIABLE-set
localSnapshot : COMPENSATION_MODULE $\rightarrow$ LOCAL_SNAPSHOT

For a specific scope, the RegisterLocalSnapshot rule creates a new local snapshot and for every local variable of the scope (addressed by the set *scopeVariables*) adds a pair of variable-value to the *snapshotVariableSet* of the local snapshot. It then assigns this snapshot to the specified compensation module. This rule is presented below.

RegisterLocalSnapshot(cm : COMPENSATION_MODULE, $scope$: SCOPE) $\equiv$
  extend LOCAL_SNAPSHOT **with** *snapshot*
    forall v **in** *scopeVariables*(*scope*)
      snapshotVariableValue(*snapshot*, v) := *variableValue*(v, *rootProcess*(*self*))
      add v **to** *snapshotVariableSet*(*snapshot*)
    localSnapshot(*cm*) := *snapshot*

Compensation Handlers

The behaviour of a compensation handler is modelled by an agent of the domain COMPENSATION_HANDLER. A compensation handler agent starts by restoring the values of local variables using the local snapshot assigned to its compensation modules. This is performed by the RestoreLocalVariables. It then goes to the *Running* mode, to execute its enclosed activity, identified by *innerActivity*(*baseActivity*(*self*)). In the *Running* more, this agent uses the previously mentioned rule HandleExceptionInRunningMode which defines its behaviour in case of receiving a fault handling signal. When the execution of its enclosed activity is completed, a compensation handler agent finalizes its execution using the FinalizeKernelAgent (in the *Activity-Completed* mode). The behaviour of this agent in the *Completed* mode, the *Execution-Fault* mode, and the *Faulted* mode is similar to the sequence agent presented in Section 5.2.3. The compensation handler program is presented in Spec 5.14.

```
CompensationHandlerProgram ≡
  case execMode(self) of
    emStarted →
      RestoreLocalVariables
      execMode(self) := emRunning

    emRunning →
      if normalExecution(self) then
        onsignal s : AGENT_COMPLETED
          execMode(self) := emActivityCompleted
        otherwise
          ExecuteActivity(baseActivity(self))
      HandleExceptionInRunningMode

      emActivityCompleted → FinalizeKernelAgent

      emCompleted → stop self

      emExecutionFault → WaitForTermination

      emFaulted → stop self


RestoreLocalVariables ≡
  let snapshot = localSnapshot(compensationModule(self))
    forall v in snapshotVariableSet(snapshot)
      value(v, rootProcess(self)) := snapshotVariableValue(snapshot, v)
```

Spec 5.14: The program of compensation handler agents

Chapter 6

Conclusion

This thesis presents the most comprehensive formal semantics model for the Business Process Execution Language for Web Services known so far. Our model provides a robust mathematical framework in the form of a distributed real-time ASM as a well defined technical foundation for establishing the key language attributes of BPEL. More specifically, this model, in combination with the LRM, is meant to serve as precise documentation providing a reliable basis for implementations and enhancing further development of the language. To avoid a gap between the informal language definition and the formal semantics, we model the language definition *as is* without making any compromises. Furthermore, the dynamic nature of industrial standardization also demands flexibility and robustness of the formalization approach. To this end, we feel that the ASM modelling paradigm offers a good compromise between practical relevance and mathematical elegance (which has already been approved in other standardization contexts [22]).

Clearly, BPEL is a fairly complex and inherently complicated language. This is partly due to the fact that this language originates from merging two other Web services composition languages, namely XLANG and WSFL, and includes artifacts from both languages [37]. Considering the goal of the BPEL, i.e., providing "a language for the formal specification of business processes and business interaction protocols" [4], it would be appealing to systematically validate and verify the key properties of the language. However, a well-defined specification is a prerequisite for verification; otherwise, one would have to face a gap between the vagueness of the natural language used in the informal documentation and the mathematically precise language that is ultimately required for verification. On the other hand, depending on the verification method one chooses, a transformation of our model to

another formal language in the verification domain may be required. This transformation, however, then is a transformation from one formal (mathematical) language to another formal language, which does not suffer from the ambiguities and vagueness of a natural language description.

Apparently, there is no simple way to prove that the resulting formalization provides a faithful model of the language. However, constructing a ground model — an abstract, complete, precise and yet understandable mathematical model — by carefully analyzing and eliciting requirements indeed seems to be the best one can do in the overall attempt toward making the semantic model as sound and complete as possible [11, 5]. To achieve a high degree of reliability in establishing the requirements, we make our model as close as possible to the informal documentation. This is attained by

- choosing a natural level of abstraction that reflects how the LRM views the language,
- providing a mathematical image of the language semantics defined by the LRM through a direct mapping from the constructs of the language, their properties, and their relations to the elements of our model, and
- adopting the terminology that is used in the LRM.

This leads to a direct and concise representation of the informal documentation, which allows conceptual justification of the model and also provides a basis for experimental validation of the design through simulation and testing.

Constructing such a formal specification demonstrates the practicability of the formalization approach. Although this formal model does not yet address some minor aspects of the language definition, it provides a comprehensive semantic framework, and clearly those aspects that are not yet captured do not introduce any new challenges in modelling. As far as a comprehensive semantics specification is concerned, to the best of our knowledge, the result of our work can be considered the frontier of semantic modelling of BPEL.

In the application domain of e-Business, it is highly desirable to build on sufficiently reliable standards. An important conclusion that we draw from the work presented here is that in order to establish and maintain a reliable standard for BPEL, we need a proper formalization of the fundamental semantic issues. The presented approach will allow us to "*reason about the current specification and related issues*" and to "*uncover issues that would otherwise go unnoticed*" [35, Issue #40]. There are unclear details in the specification of

fundamental aspects of BPEL (see Section 6.1 and the issues listed in [35]) that need to be clarified and cannot be left to the language implementations.

The following section provides some examples on how a formal specification can support validation of the language definition effectively revealing inconsistencies, loose ends, and ambiguities.

6.1 Validation

The OASIS WSBPEL Technical Committee has been working on the LRM since April 2003, basically to eliminate weak points in the language definition and to continue the work on specifying the common concepts for a business process execution language. To this date, this committee has listed 130 issues of which 81 are considered to be resolved [35].

As a result of building this ground model for BPEL, we actually have discovered a number of weak points in the LRM which will be exemplified in this section. We also proposed a new activity for BPEL to provide synchronous request-respond services to business partners which is currently handled in the language through two separate activities (i.e., *receive* and *reply*). Having two separate activities to provide a single synchronized service has caused a number of difficulties and ambiguities to the language (see Issues #26, #49, #50, #120, and #123[1]). This proposal is discussed in Appendix F.

6.1.1 Termination Due to a Fault

The LRM does not specify exactly how activity termination (due to a fault) takes place. It states that when a fault occurs in a *scope*, the fault handler begins by implicitly terminating all activities inside the *scope*. Further, in Section 20.1 on standard faults, the LRM states that *forcedTermination* is used by a scope to terminate its enclosing activities. However, it is not clear how the *forcedTermination* fault is used to terminate enclosing activities. The LRM does not state whether the faulted activities should wait for the *forceTermination* fault when they encounter a fault, or they should terminate automatically.

For instance, assume that there is a *flow* activity inside a *scope* S that has two concurrent scope branches A and B. If a fault occurs in branch B (scope B) that cannot be handled, should this branch terminate before receiving a *forcedTermination* fault from *scope* S? If

[1]At the time of producing this document, Issues #26, #120, and #123 are still open.

branch B terminates without waiting for a *forcedTermination* fault, then should the *flow* also terminate (which means A should also terminate)? If that is the case, then why should *scope* S send a *forcedTermination* fault at all?

There are also ambiguities on how fault handlers deal with a *forcedTermination* fault. According to the LRM[2], if a *forcedTermination* fault comes for a scope that is already in fault handling mode, the fault handler is not interrupted and it is allowed to finish. It is not clear what happens if the fault handler in this situation encounters an internal fault. Given that the handler had already received a *forcedTermination* fault, should it wait for another forced termination fault to propagate it to its enclosing activities?

6.1.2 Clarification on Activity Termination

About terminating the *assign* activity, the LRM indicates that

1. When a fault occurs in the execution of an assignment activity (*assign*), the destination variables are left unchanged as they were at the start of the execution of the activity (Requirement C12 of FCRL).

2. In response to a *forcedTermination* fault, which is by the LRM considered an internal fault, the assign activities are allowed to complete rather than being interrupted (Requirement B16 of FCRL).

Although the second requirement is reasonable, it conceptually contradicts the first requirement. To resolve this issue, the first requirement should be restricted to all faults except the *forcedTermination* fault.

6.1.3 Faults and the Compensate Activity

The LRM is not specific about what happens when a compensation handler encounters a fault. The LRM indicates that if a compensation handler encounters a fault and the fault is not handled in a scope inside the compensation handler, *"it is rethrown to the* parent scope*"*[3]. There are two issues regarding this statement:

1. There is no precise definition of a *parent scope* in the LRM. From the context where this term is used, one can assume that it has the same meaning as the terms *immediately*

[2]See Requirement B15 of FCRL in Appendix A.4.

[3]Item (v) of Requirement B11 of FCRL in Appendix A.4

enclosing scope or *enclosing scope* of an activity which are used widely in the LRM. On the other hand, a compensation handler of a scope is invoked (indirectly) either by a *compensate* activity or through a default fault handler or a compensation handler of a higher-level enclosing scope. Thus, the parent scope of a compensation handler is not always the immediately enclosing scope of that handler (a compensation handler may be invoked through a hierarchy of higher-level compensation handlers). To this end, the parent scope of a compensation handler needs a precise definition.

2. A *compensate* activity may invoke a number of compensation handlers installed for a specific scope. The LRM is not precise about the behaviour of a *compensate* activity in which one of the installed compensation handlers encounters a fault. One could assume that the compensate activity should terminate prematurely, but this aspect is not specified precisely in the LRM.

6.1.4 Invoking Compensation Handlers

In [4, Sections 13.3.2 and 14.2], on invoking compensation handlers, the LRM specifies that

1. "Invoking a compensation handler that has not been installed is equivalent to the empty activity (it is a no-op[4])-this ensures that fault handlers do not have to rely on state to determine which nested scopes have completed successfully."[5]

2. "If an installed compensation handler is invoked more than once during the execution of a process instance, a compliant implementation MUST throw the standard bpws:repeatedCompensation fault."[6]

It seems that the LRM tries to explicitly separate the compensation behaviour of two types of scopes: (1) scopes that are not completed yet, and (2) scopes that have been compensated before but have not been completed again to be compensated for the second time. There are two consideration regarding this case:

1. A BPEL program does not actually invoke an *installed* compensation handler of a scope. Instead, the general compensation handler of a scope is invoked, which then

[4]A 'no-op' operation is an operation that does nothing.

[5]See Requirement B3 of FCRL in Appendix A.

[6]See Requirement C16 of FCRL in Appendix A.

leads to invocation of its corresponding installed compensation handlers (e.g., invoking compensation handler of scope A through `<compensate scope='A'>`). Thus, the wording of the second requirement needs to be changed to something like: "If the compensation handler of a scope is invoked, for which all the previously installed compensation handlers are already invoked before and there is no *new* installed compensation handler, a compliant implementation MUST throw the standard bpws:repeatedCompensation fault."

2. Even considering a revised version of the second requirement, when all the previously installed compensation handlers of a scope are already invoked and there is no new installed compensation handler for that scope, in accordance with the first case, compensating that scope should be equivalent to an empty activity.

6.1.5 Rethrowing a Fault

A fault handler may include a number of *catch* clauses along with at most one *catchall* clause. While for a *catch* clause at least a fault name or a fault variable must be specified, a *catchall* clause has no parameters; i.e., a *catchall* clause has no information about the original fault that is thrown and is caught by this clause. This prevents a *catchall* clause to rethrow a fault to the parent scope. We encountered this problem while we were modelling the default fault handling behaviour of scopes. We identified the need for a special activity to allow a *catchall* clause to rethrow its original fault to the parent scope. At the same time, this issue was addressed by the OASIS WSBPEL Technical Committee and was resolved using a similar approach by introducing a `<rethrow/>` construct that rethrows the original fault in a *catch* or *catchall* clause[7].

---

[7]See Requirement A7 of FCRL in Appendix A.4.

Appendix A

Requirements Lists

A.1 Receive-RL

Requirements list of the *receive* activity:

1. "The <receive> construct allows the business process to do a blocking wait for a matching message to arrive." [$6.2]

2. Receive activity is one of the start activities that can cause process instantiation. "This is done by setting the *createInstance* attribute of such an activity to 'yes'. When a message is received by such an activity, an instance of the business process is created if it does not already exist (see 11.4. Providing Web Service Operations and 12.4. Pick)." [$6.4]

3. "The combination of a <receive> and a <reply> forms a request-response operation on the WSDL *portType* for the process." [$6.2]

4. "If more than one start activity is enabled concurrently, then all such activities must use at least one correlation set and must use the same correlation sets (see 10. Correlation and the 16.3. Multiple Start Activities example). If exactly one start activity is expected to instantiate the process, the use of correlation sets is unconstrained. This includes a *pick* with multiple *onMessage* branches; each such branch can use different correlation sets or no correlation sets." [$6.4] (also in Correlation-RL-2)

5. "Variables associated with message types can be specified as input or output variables

for *invoke*, *receive*, and *reply* activities (see 11.3. Invoking Web Service Operations and 11.4. Providing Web Service Operations)." [$9.2]

6. "In addition, it [receive activity] may specify a variable that is to be used to receive the message data received. However, this attribute is syntactically optional since it is absolutely required only in executable processes." [$11.4]

7. "A receive activity annotated in this way [with *createInstance=yes*] MUST be an initial activity in the process, that is, the only other basic activities may potentially be performed prior to or simultaneously with such a *receive* activity MUST be similarly annotated receive activities." [$11.4]

8. "It is permissible to have the createInstance attribute set to 'yes' for a set of concurrent initial activities." but "All such receive activities MUST use the same correlation sets (see 10. Correlation)." [$11.4]

9. "Compliant implementations MUST ensure that only one of the inbound messages carrying the same correlation set tokens actually instantiates the business process (usually the first one to arrive, but this is implementation dependent)." [$11.4]

10. "A business process instance MUST NOT simultaneously enable two or more *receive* activities for the same *partnerLink*, *portType*, operation and correlation set(s). ... For the purposes of this constraint, an *onMessage* clause in a *pick* and an *onMessage* event handler are equivalent to a *receive* (see 12.4. Pick and 13.5.1. Message Events)." [$11.4]

11. "If during the execution of a business process instance, two or more *receive* activities for the same partner link, *portType*, operation and correlation set(s) are in fact simultaneously enabled, then the standard fault *bpws:conflictingReceive* MUST be thrown by a compliant implementation." [$14.5]

12. " *conflictingReceive* is thrown when more than one *receive* activity or equivalent (currently, *onMessage* branch in a *pick* activity) are enabled simultaneously for the same partner link, port type, operation and correlation set(s)." (similar to the #11) [$20.1]

13. "*correlationViolation* is thrown when the contents of the messages that are processed in an *invoke*, *receive*, or *reply* activity do not match specified correlation information." [$20.1]

14. In case of termination, each "*wait*, *receive*, *reply* and *invoke* activity is interrupted and terminated prematurely." [$13.4.2]

A.2 Reply-RL

Requirements list of the *reply* activity:

1. "The `<reply>` construct allows the business process to send a message in reply to a message that was received through a `<receive>`. The combination of a `<receive>` and a `<reply>` forms a request-response operation on the WSDL *portType* for the process." [$6.2] "A *reply* activity is used to send a response to a request previously accepted through a *receive* activity. Such responses are only meaningful for synchronous interactions." [$11.4]

2. "The correlation between a request and the corresponding reply is based on the constraint that more than one outstanding synchronous request from a specific partner link for a particular *portType*, operation and correlation set(s) MUST NOT be outstanding simultaneously." [$11.4]

3. "For the purposes of this constraint [Reply-RL-3], an *onMessage* clause in a *pick* is equivalent to a *receive* (see 12.4. Pick)." [$11.4]

4. "Moreover, a reply activity must always be preceded by a receive activity for the same partner link, *portType* and (request/response) operation, such that no reply has been sent for that receive activity." [$11.4]

5. "Note that the `<reply>` activity corresponding to a given request has two potential forms. If the response to the request is normal, the *faultName* attribute is not used and the variable attribute, when present, will indicate a variable of the normal response message type. If, on the other hand, the response indicates a fault, the *faultName* attribute is used and the variable attribute, when present, will indicate a variable of the message type for the corresponding fault." [$11.4]

6. A reply activity MAY specify a variable that contains the message data to be sent [$11.4].

7. "Variables associated with message types can be specified as input or output variables for *invoke*, *receive*, and *reply* activities (see 11.3. Invoking Web Service Operations and 11.4. Providing Web Service Operations)." [$9.2]

8. "If the variable reference is omitted for an outgoing message, then any properties of the message are considered to have been initialized through opaque assignment..." [$15.1]

9. "If a reply activity is being carried out during the execution of a business process instance and no synchronous request is outstanding for the specified *partnerLink*, *portType*, operation and correlation set(s), then the standard fault *bpws:invalidReply* MUST be thrown by a compliant implementation." [$14.5]

10. "*correlationViolation* is thrown when the contents of the messages that are processed in an *invoke*, *receive*, or *reply* activity do not match specified correlation information." [$20.1]

11. "*invalidReply* is thrown when a reply is sent on a partner link, *portType* and operation for which the corresponding *receive* with the same correlation has not been carried out." (similar to #9) [$20.1]

12. "In case of activity termination, the activities *wait*, *reply* and *invoke* are added to *receive* as being instantly terminated rather than being allowed to finish." [$4.3]

A.3 Data-RL

Requirements list of the variables and data handling in BPEL:

1. "The type of each variable may be a WSDL message type, an XML Schema simple type or an XML Schema element." [$9.2]

2. "The name of a variable should be unique within its own scope." [$9.2]

3. "If a local variable has the same name and same *messageType/type/element* as a variable defined in an enclosing scope, the local variable will be used in local assignments and/or *getVariableProperty* functions." [$9.2]

4. "It is not permitted to have variables with same name but different *messageType/ type/element* within an enclosing scope hierarchy. The behavior of such variables is not defined." [$9.2]

5. "Variables associated with message types can be specified as input or output variables for *invoke*, *receive*, and *reply* activities." [$9.2]

6. "When an invoke operation returns a fault message, this causes a fault in the current scope. The fault variable in the corresponding fault handler is initialized with the fault message received." [$9.2]

7. "Each variable is visible only in the scope in which it is defined and in all scopes nested within the scope it belongs to." [$9.2]

8. "A global variable is in an uninitialized state at the beginning of a process. A local variable is in an uninitialized state at the start of the scope it belongs to." [$9.2]

9. In Executable Processes, "An attempt during process execution to use any part of a variable before it is initialized MUST result in the standard *bpws:uninitializedVariable* fault." [$14.2]

10. In Executable Processes, "the *inputVariable* attribute for *invoke* and the variable attribute for *receive* and reply activities are not optional in executable processes. In addition, the *outputVariable*] attribute is not optional for invoke when the operation concerned is a request/response operation." [$14.5]

11. In Executable Processes, "the *inputVariable* attribute for *omMessage* handlers is not optional in executable processes. In addition, the *outputVariable* attribute is not optional for invoke when the operation concerned is a request/response operation." [$14.8]

12. "Unlike executable processes, variables in abstract processes do not need to be fully initialized before being used since some computation is left implicit in abstract processes. However, since message properties are meant to represent 'transparent', i.e., protocol relevant data, BPEL4WS requires that all message properties in a message must be initialized before the message can be used, for example before the variable of the message is used as the *inputVariable* in a Web Service operation invocation." [$15.1]

13. "...it is permissible, in abstract processes, to omit the variable reference attributes from the `<invoke/>`, `<receive/>`, and `<reply/>` activities. The meaning of such an omission must be stated clearly." [$15.1]

14. "If no variable is specified for an incoming message, then the abstract process may not refer subsequently to the message or its properties (if any)." [$15.1]

15. "If the variable reference is omitted for an outgoing message, then any properties of the message are considered to have been initialized through opaque assignment..." [$15.1]

16. When variable references are omitted, correlation set references may be interpreted as follows:

    (a) "For an incoming message which initializes a correlation set (initiator case), the correlation set is deemed to be initialized." [$15.1]

    (b) "For an outgoing message which initializes a correlation set (initiator case), the correlation tokens (which are message properties) are initialized through implicit opaque assignment..." [$15.1]

    (c) "For an outgoing message which references but does not initialize a correlation set (follower case), the proper initialization of the message properties is implicit. In this case, the already initialized correlation set itself provides the token values for the outgoing message." [$15.1]

17. "*uninitializedVariable* is thrown when there is an attempt to access the value of an uninitialized part in a message variable." [$20.1]

18. "If a correlation set is initialized by rule 1 or 2 above [16a and 16b], then outgoing messages in the same correlated exchange must also refrain from referencing a message variable. This restriction applies because it is not possible to initialize the properties of the outgoing messages from the correlation set alone." [$15.1]

A.4 FC-RL

Requirements list of Fault and Compensation Handling in BPEL:

Group A: Syntactical

1. A scope can provide fault handlers and one compensation handler. [$13]
2. "The compensation handler can be invoked by using the *compensate* activity, which names the scope for which the compensation is to be performed, that is, the scope whose compensation handler is to be invoked." [$13.3.2]
3. "This activity [*compensate*] can be used only in the following parts of a business process:
   - In a fault handler of the scope that immediately encloses the scope for which compensation is to be performed.
   - In the compensation handler of the scope that immediately encloses the scope for which compensation is to be performed. " [$13.3.2]
4. "Note that in case an invoke activity has a compensation handler defined inline, the name of the activity is the name of the scope to be used in the compensate activity." [$13.3.2]
5. "In BPEL4WS, all faults, whether internal or resulting from a service invocation, are identified by a qualified name." [$6.1]
6. "BPEL4WS does not require fault names to be defined prior to their use in a throw element." [$11.6]
7. All custom fault handlers can rethrow the original fault with the syntax `<rethrow/>` that has no attributes. [35, Issue #95]

Group B: Core Semantics

1. **Revised by OASIS**: "Compensation handlers always interact with the current state of the process, specifically the state of variables declared in their associated scope and all enclosing scopes. [...] The current state of the process consists of the current local state of all scopes that have been started. This includes scopes that have completed but for which the associated compensation handler has not been invoked. For completed uncompensated scopes their current local state is the state as it was at the time of completion." [OASIS Issue #3]

Original: "BPEL4WS semantics state that the compensation handler, if invoked, will see a frozen snapshot of all variables, as they were when the scope being compensated was completed." [$13.3.1]

2. "...compensation handlers cannot update live data in the variables that the business process is using." "A compensation handler, once installed, can be thought of as a completely self-contained action that is not affected by, and does not affect, the global state of the business process instance. It can only affect external entities." [$13.3.1]

3. "A compensation handler for a scope is available for invocation only when the scope completes normally. Invoking a compensation handler that has not been installed is equivalent to the empty activity (it is a no-op)-this ensures that fault handlers do not have to rely on state to determine which nested scopes have completed successfully." [$13.3.2]

4. "If a scope being compensated by name was nested in a loop, the instances of the compensation handlers in the successive iterations are invoked in reverse order." [$13.3.2]

5. "It is no longer possible to have other than depth first order of control-flow-dictated completion since we ban circular dependencies via links between reversible scopes." [Reversible and Permeable Scopes, in resolution to OASIS issue #10]

6. "If the compensation handler for a scope is absent, the default compensation handler invokes the compensation handlers for the immediately enclosed scopes in the reverse order of the completion of those scopes." [$13.3.2]

7. "Note that the `<compensate/>` activity in a fault or compensation handler attached to scope `S` causes the default-order invocation of compensation handlers for completed scopes directly nested within `S`. The use of this activity can be mixed with any other user-specified behaviour except the explicit invocation of `<compensate scope='Sx'/>` for scope `Sx` nested directly within `S`. Explicit invocation of compensation for such a scope nested within `S` disables the availability of default-order compensation, as expected." [$13.3.2]

8. In Section 13.4.1, the LRM indicates that: "Whenever a fault handler (for any fault) or the compensation handler is missing for any given scope, they

are implicitly created with the following behavior:

Fault handler:

- Run all available compensation handlers for immediately enclosed scopes in the reverse order of completion of the corresponding scopes.
- Rethrow the fault to the next enclosing scope.

Compensation handler:

- Run all available compensation handlers for immediately enclosed scopes in the reverse order of completion of the corresponding scopes."

9. "A business process instance is terminated ... When a fault reaches the process scope, and is either handled or not handled. In this case the termination is considered abnormal even if the fault is handled and the fault handler does not rethrow any fault. A compensation handler is never installed for a scope that terminates abnormally." [$6.4]

10. "The throw activity can be used when a business process needs to signal an internal fault explicitly. Every fault is required to have a globally unique `QName`. The throw activity is required to provide such a name for the fault and can optionally provide a variable of data that provides further information about the fault. A fault handler can use such data to analyze and handle the fault and also to populate any fault messages that need to be sent to other services." [$11.6]

11. Quoted from [$20.3.1]: Coordination Protocol for BPEL4WS Scope

    i. "A nested scope may complete successfully. In this case a compensation handler is installed for the nested scope. This is modeled with a Completed signal from the nested scope to its parent scope.

    ii. A nested scope may encounter a fault internally. In this case the scope always terminates unsuccessfully.

        A. If the fault handler rethrows a fault to its enclosing scope, this is modeled as a Faulted signal from the nested scope to its parent scope.

        B. If the fault is handled and not rethrown, the scope exits gracefully

from the work of its parent scope. This is modeled as an Exited signal from the nested scope to its parent scope.

iii. After a nested scope has completed, (a fault or compensation handler for) the parent scope may ask it to compensate itself by invoking its compensation handler. The compensate action is modeled with a Compensate signal from the parent scope to the nested scope.

iv. Upon successful completion of the compensation, the nested scope sends the Compensated signal to its parent scope.

v. The compensation handler may itself fault internally. In this case

  A. If the fault is not handled by a scope within the compensation handler, it is rethrown to the parent scope. This is modeled as a Faulted signal from the nested scope to its parent scope.

  B. If the fault is handled and not rethrown, we assume that the compensation was able to complete successfully. In this case the nested scope sends the Compensated signal to its parent scope.

vi. If there is a fault in the parent scope independent of the work of the nested scope, the parent scope will ask the nested scope to prematurely abandon its work by sending a Cancel signal.

vii. The nested scope, upon receiving the cancel signal, will interrupt and terminate its behavior (as though there were an internal fault), and return a Canceled signal to the parent.

viii. Finally, when a parent scope decides that the compensation for a completed nested scope is not needed any more it sends a Close signal to the nested scope. After discarding the compensation handler the nested scope responds with a Closed signal.

ix. In case there is a race between the Completed signal from the nested scope and the Cancel signal from the parent scope, the Completed signal wins, i.e., the nested scope is deemed to have completed and the Cancel signal is ignored.

x. In case a Cancel signal is sent to a nested scope that has already faulted internally, the Cancel signal is ignored and the scope will eventually send either a Faulted or an Exited signal to the parent."

12. "The optional fault handlers attached to a scope provide a way to define a set of custom fault-handling activities, syntactically defined as catch activities...If the fault name is missing, then the catch will intercept all faults with the right type of fault data. ...A catchAll clause can be added to catch any fault not caught by a more specific catch handler." [$13.4]

13. "If no *catch* or *catchall* is selected, the fault is not caught by the current scope and is rethrown to the immediately enclosing scope." [$13.4]

14. "If the fault occurs in (or is rethrown to) the global process scope, and there is no matching fault handler for the fault at the global level, the process terminates abnormally, as though a terminate activity had been performed." [$13.4](see B9)

15. "Scopes provide the ability to control the semantics of forced termination to some degree. When the activity being terminated is in fact a scope, the behavior of the scope is interrupted and the fault handler for the standard *bpws:forcedTermination* fault is run. Note that this applies only if the scope is in normal processing mode. If the scope has already experienced an internal fault and invoked a fault handler, then as stated above, all other fault handlers including the handler for *bpws:forcedTermination* are uninstalled, and the forced termination has no effect. The already active fault handler is allowed to complete." [$13.4.2]

16. In [$13.4.2] it says, the behaviour of a normal fault handler "begins by implicitly (recursively) terminating all activities directly enclosed within its associated scope that are currently active.

    - The assign activities are sufficiently short-lived that they are allowed to complete rather than being interrupted when termination is forced. The evaluation of expressions when already started is also allowed to complete. Each wait, receive, reply and invoke activity is interrupted and terminated prematurely. When a synchronous invoke activity (corresponding to a request/reply operation) is interrupted and terminated prematurely, the response (if received) for such a terminated activity is silently discarded. The notion of termination does not apply to empty, terminate, and throw.
    - All structured activity behavior is interrupted. The iteration of while is

interrupted and termination is applied to the loop body activity. If switch has selected a branch, then the termination is applied to the activity of the selected branch. The same applies to pick. If either of these activities has not yet selected a branch, then the switch and the pick are terminated immediately. The sequence and flow constructs are terminated by terminating their behavior and applying termination to all nested activities currently active within them."

17. "If a fault occurs in a fault handler E for a scope C, the fault can be caught through the use of a scope within E. If the fault is not caught by a scope within E, it is immediately thrown to the parent scope of C and the behavior of E terminates prematurely. In effect, no distinction is made between faults that E rethrows deliberately and faults that occur as undesired faults in E." [$13.4.2]

Group C: Details and Constraints

1. "If a compensation handler is specified for the business process as a whole (see 13.3. Compensation Handlers), a business process instance can be compensated after normal completion by platform-specific means. This functionality is enabled by setting the enableInstanceCompensation attribute of the process to 'yes'." [$6.4]
2. "The variable provided as the value of the faultVariable attribute in a catch handler to hold fault data is now scoped to the fault handler itself rather than being inherited from the associated scope." [$4.3]
3. "This attribute [suppressJoinFailure, in Process] determines whether the joinFailure fault will be suppressed for all activities in the process. The effect of the attribute at the process level can be overridden by an activity using a different value for the attribute. The default for this attribute is 'no'." [$6.2]
4. "When an invoke operation returns a fault message, this causes a fault in the current scope. The fault variable in the corresponding fault handler is initialized with the fault message received (see 13. Scopes and 13.4. Fault Handlers)." [$9.2]
5. "Semantically, [for the invoke activity] the specification of local fault and/or

compensation handlers is equivalent to the presence of an implicit scope immediately enclosing the activity [invoke] and providing those handlers. The name of such an implicit scope is always the same as the name of the activity it encloses." [$11.2]

6. "If, on the other hand, the response indicates a fault, the faultName attribute is used and the variable attribute, when present, will indicate a variable of the message type for the corresponding fault." [$11.4]

7. "The fault variable [of a *catch* clause] is optional because a fault might not have additional data associated with it." [$13.4]

8. "The following rules are used to select the catch activity that will process a fault:

   i. If the fault has no associated fault data, a catch activity that specifies a matching faultName value will be selected if present. Otherwise, the default catchAll handler is selected if present.

   ii. If the fault has associated fault data, a catch activity specifying a matching faultName value and a faultVariable whose type (WSDL message type) matches the type of the fault's data will be selected if present. Otherwise, a catch activity with no specified faultName and with a faultVariable whose type matches the type of the fault data will be selected if present. Otherwise, the default catchAll handler is selected if present." [$13.4]

9. "When a fault handler for scope S handles a fault that occurred in S without rethrowing, links that have S as the source will be subject to regular evaluation of status after the fault has been handled, because processing in the enclosing scope is meant to be continued." [$13.4]

10. "The fault handler for the *bpws:forcedTermination* fault is designed like other fault handlers, but this fault handler cannot rethrow any fault. Even if an uncaught fault occurs during its behavior, it is not rethrown to the next enclosing scope. This is because the enclosing scope has already faulted, which is what is causing the forced termination of the nested scope." [$13.4.2]

11. "The use of error handling features in a serializable scope is governed by the following rules:

i. The fault handlers for a serializable scope share the serializability domain of the associated scope, that is, in case a fault occurs in a serializable scope, the behavior of the fault handler is considered part of the serializable behavior (in commonly used implementation terms, locks are not released when making the transition to the fault handler). This is because the repair of the fault needs a shared isolation environment to provide predictable behavior.

ii. The compensation handler for a serializable scope does not share the serializability domain of the associated scope.

iii. For a serializable scope with a compensation handler, the creation of the state snapshot for compensation is part of the serializable behavior. In other words, it is always possible to reorder behavior steps as if the scope had sufficiently exclusive access to the shared variables all the way to completion, including the creation of the snapshot." [$13.6]

12. "If there is any fault during the execution of an assignment activity, the destination variables are left unchanged as they were at the start of the activity." [$14.3]

13. "After a correlation set is initiated, the values of the properties for a correlation set must be identical for all the messages in all the operations that carry the correlation set and occur within the corresponding scope until its completion. If at execution time this constraint is violated, the standard fault *bpws:correlationViolation* MUST be thrown by a compliant implementation. The same fault MUST be thrown if an activity with the initiate attribute set to no attempts to use a correlation set that has not been previously initiated." [$14.4]

14. "If during the execution of a business process instance, two or more receive activities for the same partner link, portType, operation and correlation set(s) are in fact simultaneously enabled, then the standard fault *bpws:conflictingReceive* MUST be thrown by a compliant implementation." [$14.5]

15. "If more than one outstanding synchronous request on a specific partner link for a particular portType, operation and correlation set(s) is outstanding simultaneously during the execution of a business process instance, then the

standard fault *bpws:conflictingRequest* MUST be thrown by a compliant implementation. Note that this is semantically different from the *bpws:conflictingReceive*, because it is possible to create the conflictingRequest by consecutively receiving the same request on a specific partner link for a particular portType, operation and correlation set(s). If a reply activity is being carried out during the execution of a business process instance and no synchronous request is outstanding for the specified partnerLink, portType, operation and correlation set(s), then the standard fault *bpws:invalidReply* MUST be thrown by a compliant implementation." [$14.5]

16. "If an installed compensation handler is invoked more than once during the execution of a process instance, a compliant implementation MUST throw the standard *bpws:repeatedCompensation* fault." [$14.7]

17. Standard Faults [$20.1] (the complete list is presented in Appendix A

    bpws:conflictingReceive Thrown when more than one receive activity or equivalent (currently, onMessage branch in a pick activity) are enabled simultaneously for the same partner link, port type, operation and correlation set(s).

    bpws:conflictingRequest Thrown when more than one synchronous inbound request on the same partner link for a particular port type, operation and correlation set(s) are active.

    bpws:forcedTermination Thrown as the result of a fault in an enclosing scope.

    bpws:correlationViolation Thrown when the contents of the messages that are processed in an invoke, receive, or reply activity do not match specified correlation information.

    bpws:repeatedCompensation Thrown when an installed compensation handler is invoked more than once.

    bpws:invalidReply Thrown when a reply is sent on a partner link, portType and operation for which the corresponding receive with the same correlation has not been carried out.

Group D: Special Cases

1. "The semantics of a process in which an installed compensation handler is invoked more than once is undefined." [$13.3.2]
2. "The first extension [for Executable Processes] defines a standard fault for errorneous use of the XPath 1.0 function defined for extracting global property values from variables." [$14.1]
3. "An attempt during process execution to use any part of a variable before it is initialized MUST result in the standard *bpws:uninitializedVariable* fault." [$14.2]
4. "The second extension defines a standard fault for violation of type matching constraints. If any of the matching constraints defined in the section 9.3.1. Type Compatibility in Assignment is violated during execution, the standard fault *bpws:mismatchedAssignmentFailure* MUST be thrown by a compliant implementation." [$14.3]

Group E: Interpretation

1. "...it is important to note that BPEL4WS uses two standard internal faults for its core control semantics, namely, *bpws:forceTermination* and *bpws:joinFailure*. These are the only two standard faults that play a role in the core concepts of BPEL4WS." [$5]
2. "Fault handling in a business process can be thought of as a mode switch from the normal processing in a scope." [$13.4]

Group F: Extensibility

1. "In the future, BPEL4WS will add input and output parameters to compensation handlers..." [$13.3.1]

Appendix B

The Revised Core

B.1 Initial Definitions

// Agents
domain PROCESS
domain INBOX_MANAGER
domain OUTBOX_MANAGER

// Activity Agents
domain SEQUENCE_AGENT
domain WHILE_AGENT
domain FLOW_AGENT
domain FLOW_THREAD_AGENT// sub agents of a flow agent
domain PICK_AGENT
domain PICK_ALARM_AGENT
domain PICK_MESSAGE_AGENT
domain SWITCH_AGENT

ACTIVITY_AGENT ≡
SEQUENCE_AGENT
∪ WHILE_AGENT
∪ FLOW_AGENT
∪ PICK_AGENT
∪ SWITCH_AGENT
∪ FLOW_THREAD_AGENT
∪ PICK_ALARM_AGENT
∪ PICK_MESSAGE_AGENT

SUBPROCESS_AGENT ≡ ACTIVITY_AGENT
// Specifies the set of agents that are sub agents of a process.
// Other elements may be added to this set, later.

KERNEL_AGENT ≡ PROCESS ∪ SUBPROCESS_AGENT

AGENT ≡ KERNEL_AGENT ∪ INBOX_MANAGER ∪ OUTBOX_MANAGER

// Events
domain ONMESSAGE// OnMessageEvents of Pick activity
domain ONALARM// OnAlarmEvents of Pick activity
domain EVENT_DESCRIPTOR

EVENT ≡ ONMESSAGE ∪ ONALARM

// Activities
domain REPLY
domain RECEIVE
domain INVOKE
domain WAIT
domain TERMINATE
domain EMPTY
domain SEQUENCE
domain WHILE
domain SWITCH
domain PICK
domain FLOW

ACTIVITY ≡ REPLY ∪ RECEIVE ∪ SEQUENCE ∪ INVOKE
∪ WHILE ∪ SWITCH ∪ PICK ∪ FLOW
∪ WAIT ∪ TERMINATE ∪ EMPTY ∪ ASSIGN
∪ SCOPE ∪ COMPENSATE

IN_OPERATION ≡ RECEIVE ∪ INVOKE ∪ ONMESSAGE
OUT_OPERATION ≡ REPLY ∪ INVOKE
INOUT_OPERATION ≡ IN_OPERATION ∪ OUT_OPERATION

// In-Out Descriptors
domain OUTPUT_DESCRIPTOR
domain INPUT_DESCRIPTOR
INOUT_DESCRIPTOR ≡ INPUT_DESCRIPTOR ∪ OUTPUT_DESCRIPTOR

// Activity dependents
domain LINK
// Represents the link between activities in a parallel execution (flow).

domain SWCASE
// CASE elements of a switch, it include CASEs and OTHERWISE
// OTHERWISE is a special case with an always-true condition

domain PRIORITY
// an ordered domain with a least element called *leastPriority*
$leastPriority :\rightarrow \text{PRIORITY}$

// MESSAGE
domain MESSAGE

// Different Execution Modes of Running Agents
EXECUTION_MODE $\equiv$
$\{emStarted, emRunning, emActivityCompleted, emCompleted\}$

// ——————— Kernel Agent ———————
$execMode : \text{KERNEL_AGENT} \rightarrow \text{EXECUTION_MODE}$
// initial value: *emStarted*

$rootProcess : \text{KERNEL_AGENT} \rightarrow \text{PROCESS}$
// Returns the process agent to which this running agent belongs

$$rootProcess(a) \equiv \begin{cases} a, & \text{if } a \in \text{PROCESS}; \\ rootProcess(parentAgent(a)), & \text{otherwise.} \end{cases}$$

$receiveMode : \text{KERNEL_AGENT} \rightarrow \text{BOOELAN}$
// initial value : false

$eventOccured : \text{KERNEL_AGENT} \rightarrow \text{BOOLEAN}$
$eventOccured(a : \text{KERNEL_AGENT}) \equiv$
$\exists e(e \in occuredEvents(parentAgent(a)))$
$\wedge\, parentAgent(a) \in \text{PICK_AGENT}$

$normalExecution : \text{KERNEL_AGENT} \rightarrow \text{BOOLEAN}$
$normalExecution(a) \equiv true$

// ——————— Process ———————

$mainActivity$: PROCESS $\rightarrow$ ACTIVITY

// This is the activity which the process should execute

$waitingSetForInput$: PROCESS $\rightarrow$ OUTPUT_DESCRIPTOR-set

// For each process there is a set indicating the input operations

// that are watiting for their messages to arrive.

$waitingSetForOutput$: PROCESS $\rightarrow$ OUTPUT_DESCRIPTOR-set

// For each process there is a set indicating the output operations

// that are watiting for their messages to be sent out.

$waitingForInput$: PROCESS $\rightarrow$ BOOLEAN

$waitingForInput(p : \text{PROCESS}) \equiv (waitingSetForInput(p) \neq \emptyset)$

// indicates if the process is waiting for any message or not

$waitingForOutput$: PROCESS $\rightarrow$ BOOLEAN

$waitingForOutput(p : \text{PROCESS}) \equiv (waitingSetForOutput(p) \neq \emptyset)$

// indicates if the process is waiting for any message to be sent, or not

$completedInOperations$: PROCESS $\rightarrow$ INPUT_DESCRIPTOR-set

$completedOutOperations$: PROCESS $\rightarrow$ OUTPUT_DESCRIPTOR-set

$subordinateAgentSet$: PROCESS $\rightarrow$ SUBPROCESS_AGENT-set

// Returns the set of subprocess agents that have been created

// and work under control of this process

$subordinateAgentSet(p : \text{PROCESS}) \equiv$

$\{a \mid a \in \text{ACTIVITY_AGENT} \wedge rootProcess(a) = p\}$

// ———————— Subprocess Agent ————————

$parentAgent$: SUBPROCESS_AGENT $\rightarrow$ KERNEL_AGENT

// Parent Agent (one layer above in the creation tree) of an agent

$waitingOnInput$: (SUBPROCESS_AGENT $\times$ PROCESS) $\rightarrow$ BOOLEAN

$waitingOnInput(a, p) \equiv$

$\exists d(d \in waitingSetForInput(p) \wedge dscAgent(d) = a)$

$\wedge \forall d'(d' \in completedInOperations(p) \rightarrow dscAgent(d') \neq a)$

// ———————— Inbox Manager ————————

$inboxSpace$: INBOX_MANAGER $\rightarrow$ MESSAGE-set

// It is used as an interface to the environment.

// Messages come from the environment to this set.

// InboxManager processes these messages

// and assign them to appropriate process instances.

$match$: (PROCESS $\times$ IN_OPERATION $\times$ MESSAGE) $\rightarrow$ BOOLEAN

// Tells whether a messages matches a specific input operation of

// a process instance or not.

// ———————— Outbox Manager ————————

$outboxSpace$: OUTBOX_MANAGER $\rightarrow$ MESSAGE-set

// It is used as an interface to the environment.

// Outbox manager puts messages in this set and

// they are received by the environment.

// ———————— Activity Agent ————————

$baseActivity$: ACTIVITY_AGENT $\rightarrow$ ACTIVITY

// The activity for which this ACTIVITY_AGENT is responsible;

// i.e. the PICK, SWITCH, SEQUENCE or FLOW,...

// ———————— Activity ————————

$assignedAgent$: ACTIVITY $\rightarrow$ ACTIVITY_AGENT

// Assumes that every activity that is read is unique.

// For each structured activity returns the Activity Agent that is executing it.

// Returns undef if no agent is assigned or the activity is a basic activity.

// Ro there is no need for agents to eliminate this relationship when they end.

$sourceLinkSet$: ACTIVITY $\rightarrow$ LINK-set

// An activity can be the source of a set of links; returns this set

$targetLinkSet$: ACTIVITY $\rightarrow$ LINK-set

// An activity can be the target of a set of links; returns this set

$activityJoinCondition$: ACTIVITY $\rightarrow$ BOOLEAN

// Default joinCondition: The logical OR of the link status of all

// incoming links of the activity.

// Returns true if the joinCondition of the activity is satisfied.

// ———————— In Operation ————————

$initiateCorrelation$: IN_OPERATION $\rightarrow$ BOOLEAN

// Indicates whether an input operation initiates a new

// correlation set or not

// ———————— Out Operation ————————

$opaqueMessage$(OUT_OPERATION) $\rightarrow$ MESSAGE

// External function, assigns an opaque message to an operation.

// ———————— Invoke Activity ————————

$synchronous$: INOVKE $\rightarrow$ BOOLEAN

// Returns true if the invoke activity contains synchronous interactions;

// i.e. request/response

// ——————— Wait Activity ———————

$completionTime$: WAIT $\rightarrow$ TIME

// CompletionTime returns the time when a wait activity is completed.

// In case of 'until' its trivial, but in case of 'for' it needs the starting time,

// which is accessible through the activity itself.

// ——————— While Activity ———————

$waCondition$: WHILE $\rightarrow$ BOOLEAN

// It is assumed that this function also evaluates the condition expression.

$innerActivity$: WHILE $\rightarrow$ ACTIVITY

// Returns the activity that is defined inside a while

// ——————— Switch Activity ———————

$swCaseSet$: SWITCH $\rightarrow$ SWCASE-set

// the list of case elements of the switch plus OTHERWISE.

$swCaseCondition$: SWCASE $\rightarrow$ BOOLEAN

// External function.

// the value of the conditional expression of a switch case element

// For 'otherwise', it always returns true

$swCaseActivity$: SWCASE $\rightarrow$ ACTIVITY

// The activity associated with a case element or otherwise.

$swPrority$: SWCASE $\rightarrow$ PRIORITY

// Each switch case is assigned a priority, resembling the order between cases

// the lowest priority is assigned to otherwise

// ——————— Pick Activity ———————

$onMessageSet$: PICK $\rightarrow$ ONMESSAGE-set

// Set of the onMessage events of the pick activity

$onAlarmSet$: PICK $\rightarrow$ ONALARM-set

// Set of the onAlarm events of pick activity

// ————————— Event: OnMessage and OnAlaram —————————

$onEventActivity$: EVENT $\rightarrow$ ACTIVITY

// the activity associated with a specific event.

$triggerTime$: ONALARM $\times$ TIME $\rightarrow$ TIME

// the trigger time of an onAlarm activity.

// If onAlarm is defined by a For it will use the starting time of the alarm agent

// to determine the trigger time.

// ————————— Flow Activity —————————

$flowActivitySet$: FLOW $\rightarrow$ ACTIVITY-set

// Set of concurrent activities in a FLOW activity

$flowAgentSet$: FLOW_AGENT $\rightarrow$ FLOW_THREAD_AGENT-set

// initial value: $\emptyset$

// The set of alive thread agents that are working under a flow agent.

// ————————— Common Properties —————————

$startTime$: WAIT $\cup$ PICK_ALARM_AGENT $\rightarrow$ TIME

// initial value: *undef*

// startTime is needed in wait activity. It keeps the starting point of a wait

// activity and is needed in case of waiting for a duration.

// Pick alarm agent also keeps the starting time.

// ————————— Sequence Agent —————————

$sequenceCounter$: SEQUENCE_AGENT $\rightarrow$ ACTIVITY

// the next activity in the sequence

// If there is no more activities in the sequence, returns undef

$currentActivity$: SEQUENCE_AGENT $\rightarrow$ ACTIVITY

// initial value: *undef*

// Keeps track of the current activity which is being executed

// ———————— Switch Agent ————————
$foundBranch$: SWITCH_AGENT $\rightarrow$ ACTIVITY
// initial value: *undef*
// The activity that is associated with the branch which is chosen
// by switch to be executed.

// ———————— Pick Agent ————————
$triggeredEvents$: PICK_AGENT $\rightarrow$ EVENT_DESCRIPTOR-set
// The set of events that happened
// initial value: $\emptyset$

$chosenActivity$: PICK_AGENT $\rightarrow$ ACTIVITY
// initial value: *undef*
// The activity that is chosen by the pick agent to be executed.

// ———————— Link ————————
$linkTransitionCondition$: LINK $\rightarrow$ BOOLEAN
// Evaluates the transition condition of a link.

$linkStatus$: LINK $\rightarrow$ $\{lsPositive, lsNegative, lsNotDefined\}$
// initial value: lsNotDefined
// Returns the status of a link.

// ———————— In/Out Descriptor ————————
$dscAgent$: INOUT_DESCRIPTOR $\rightarrow$ KERNEL_AGENT
// Assigns to each descriptor, a running agent that created that descriptor.

$dscOperation$: INOUT_DESCRIPTOR $\rightarrow$ INOUT_OPERATION
// There is an Input/Output operation that is bound to every inout descriptor.

$dscCompletedTime$: INOUT_DESCRIPTOR $\rightarrow$ TIME
// If not undef, contains the completition time of the input/output operation.

// ———————— Event Descriptor ————————

edscEvent : EVENT_DESCRIPTOR → EVENT

// There is an event bound to every event descriptor.

edscCompletedTime : EVENT_DESCRIPTOR → TIME

// If not undef, contains the completition time of the event

B.2 Programs

```
// ——————————— Inbox Manager ———— ————
InboxManagerProgram ≡
  if inboxSpace(self) ≠ ∅ then
    choose p ∈ PROCESS, m ∈ inboxSpace(self),
        descriptor ∈ waitingSetForInput(p) with
        waitingOnIO(dscAgent(descriptor), p) ∧ match(p, operation, m)
      AssignMessage(p, descriptor, m)

      if p = dummyProcess then
        new newDummy : PROCESS
          dummyProcess := newDummy
where
    operation ≡ dscOperation(descriptor)

AssignMessage(p : PROCESS, descriptor : INPUT_DESCRIPTOR, m : MESSAGE) ≡
  if initiateCorrelation(op) then
    InitiateCorrelation(p, descriptor, m)
  dscCompletedTime(descriptor) := now
  add descriptor to completedInOperations(p)
  remove m from inboxSpace(self)
  remove descriptor from waitingSetForInput(p)
where
    op ≡ dscOperation(descriptor),
    agent ≡ dscAgent(descriptor)

// ——————————— Outbox Manager ——— ————————
OutboxManagerProgram ≡
  choose p ∈ PROCESS, descriptor ∈ waitingSetForOutput(p)
    DeliverMessage(p, descriptor)
```

```
DeliverMessage(p : PROCESS, descriptor : OUTPUT_DESCRIPTOR) ≡
  if variable(opr) = undef then
    add opaqueMessage(opr) to outboxSpace(self)
    if initiateCorrelation(opr) then
      InitiateCorrelation(p, descriptor, opaqueMessage(opr))
  dscCompletedTime(descriptor) := now
  add descriptor to completedInOperations(p)
  remove descriptor from waitingSetForOutput(p)
where
    opr ≡ dscOperation(descriptor)

// ——————————— Process ———————————
ProcessProgram ≡
  case execMode(self) of
    emStarted → execMode(self) := emRunning
    emRunning →
    if normalExecution(self) then
      onsignal s : AGENT_COMPLETED
        execMode(self) := emActivityCompleted
      otherwise
        ExecuteActivity(mainActivity(self))
    emActivityCompleted → execMode(self) := emCompleted
    emCompleted → stop self


// ——————————— Helper Rules ———————————
FinalizeActivity(activity : ACTIVITY) ≡
  execMode(self) :=
emActivityCompleted
  Synchronization(activity)

FinalizeKernelAgent ≡
  execMode(self) := emCompleted
  trigger s : AGENT_COMPLETED, parentAgent(self)
  Synchronization(baseActivity(self))
```

GenerateInputDescriptor(*operation* : INPUT_OPERATION) ≡
  extend INPUT_DESCRIPTOR **with** *descriptor*
    SetInOutDescriptor(*operation*, *descriptor*)
    add *descriptor* **to** *waitingSetForInput*(*rootProcess*(*self*))

GenerateOutputDescriptor(*operation* : OUTPUT_OPERATION) ≡
  extend OUTPUT_DESCRIPTOR **with** *descriptor*
    SetInOutDescriptor(*operation*, *descriptor*)
    add *descriptor* **to** *waitingSetForOutput*(*rootProcess*(*self*))

SetInOutDescriptor(*operation* : OUTPUT_OPERATION, *dsc* : INOUT_DESCRIPTOR) ≡
  dscAgent(*dsc*) := *self*
  dscOperation(*dsc*) := *operation*

Synchronization(*activity* : ACTIVITY) ≡
  forall *link* ∈ *sourceLinkSet*(*activity*)
    if *linkTransitionCondition*(*link*) **then**
      linkStatus(*link*) := *lsPositive*
    else
      linkStatus(*link*) := *lsNegative*

// in the following rule, the predicates linkStatusDefined and joinCondition state
// synchronization dependencies between concurrent activities. Their definition is
// captured in the complete formal model by checking certain conditions before
// executing an activity. For brevity, these conditions are left abstract here.
// suspended is set to true before entering this module

ExecuteActivity(*activitiy* : ACTIVITY) ≡
  if $\forall x(x \in targetLinkSet(activity) \wedge (linkStatus(x) \neq lsNotDefined))$ **then**
    if *activityJoinCondition*(*activity*) **then**
      ExecuteBasicActivity(*activity*)
      ExecuteStructuredActivity(*activity*)

ExecuteBasicActivity($activity$: ACTIVITY) $\equiv$
  if $activity \in$ RECEIVE **then**
    ExecuteReceive($activity$)
  if $activity \in$ REPLY **then**
    ExecuteReply($activity$)
  if $activity \in$ INVOKE **then**
    ExecuteInvoke($activity$)
  if $activity \in$ WAIT **then**
    ExecuteWait($activity$)
  if $activity \in$ EMPTY **then**
    ExecuteEmpty($activity$)

ExecuteStructuredActivity($activity$: ACTIVITY) $\equiv$
  if $assignedAgent(activity) = undef$ **then**
    if $activity \in$ SEQUENCE **then**
      new s : SEQUENCE_AGENT
        Initialize($s, activity$)
    if $activity \in$ WHILE **then**
      new w : WHILE_AGENT
        Initialize($w, activity$)
    if $activity \in$ SWITCH **then**
      new sw : SWITCH_AGENT
        Initialize($sw, activity$)
    if $activity \in$ PICK **then**
      new p : PICK_AGENT
        Initialize($p, activity$)
    if $activity \in$ FLOW **then**
      new f : FLOW_AGENT
        Initialize($f, activity$)

Initialize($agent$: ACTIVITY_AGENT, $activity$: ACTIVITY) $\equiv$
  $assignedAgent(activity) := agent$
  $parentAgent(agent) := self$
  $baseActivity(agent) := activity$

```
// ——————————— Execute Receive ———————————
ExecuteReceive(activity : RECEIVE) ≡
  if ¬receiveMode(self) ∧ ¬outstandingReceiveConflict(activity) then
    receiveMode(self) := true// The running agent waits to receive a message
    GenerateInputDescriptor(activity)

  if receiveMode(self) then
    choose descriptor ∈ completedInOperations(self)
        with dscAgent(descriptor) = self ∧ dscOperation(descriptor) = activity
      receiveMode(self) := false
      FinalizeActivity(activity)

// ——————————— Execute Reply ———————————
ExecuteReply(activity : REPLY) ≡
  if requestResponseConditionSatisfied(activity) then
    if ¬replyMode(self) then
      replyMode(self) := true
      GenerateOutputDescriptor(activity)
    if replyMode(self) then
      choose descriptor ∈ completedOutOperations(self) with
          dscAgent(descriptor) = self ∧ dscOperation(descriptor) = activity
        replyMode := false
        FinalizeActivity(activity)
// Where requestResponseConditionSatisfied(activity) deals with
// Requirements #2, #3, and #4 of the requirements list of the reply activity.
```

```
// ——————————— Execute Invoke ———————————
ExecuteInvoke(activity : INVOKE) ≡
  if ¬replyMode(self) ∧ ¬recevieMode(self) then // i.e. if it is the first step
    replyMode(self) := true
    GenerateOutputDescriptor(activity)

  if replyMode(self) ∧ ¬receiveMode(self) then
    choose descriptor ∈ completedOutOperations(self)
         with dscAgent(descriptor) = self ∧ dscOperation(descriptor) = activity
      replyMode(self) := false

      if synchronous(activity) then
        // The running agent waits to receive a message
        receiveMode(self) := true
        GenerateInputDescriptor(activity)
      else
        FinalizeActivity(activity)

  if ¬replyMode(self) ∧ receiveMode(self) then
    choose descriptor ∈ completedInOperations(self)
         with dscAgent(descriptor) = self ∧ dscOperation(descriptor) = activity
      receiveMode(self) := false
      FinalizeActivity(activity)

// ——————————— Execute Wait ———————————
ExecuteWait(activity : WAIT) ≡
  if startTime(self) = undef then
    startTime(self) := now
  else
    if completionTime(activity, startTime) = now then
      startTime(self) := undef
      FinalizeActivity(activity)
```

```
// ————————— Execute Empty —————————
ExecuteEmpty(activity : EMPTY) ≡
  FinalizeActivity(activity)


// ————————— Sequence Agent —————————
SequenceProgram ≡
  case execMode(self) of
    emStarted →
      currentActivity(self) := sequenceCounter(self)
      execMode(self) := emRunning

    emRunning →
      if normalExecution(self) then
        onsignal s : AGENT_COMPLETED
          execMode(self) := emActivityCompleted
        otherwise
          ExecuteActivity(currentActivity(self))

    emActivityCompleted →
      currentActivity(self) := sequenceCounter(self)
      if currentActivity(self) = undef then
        FinalizeKernelAgent
      else
        execMode(self) := emRunning

    emCompleted → stop self
```

```
// ——————————— While Agent ———————————
WhileProgram ≡
  case execMode(self) of
    emStarted →
      if waCondition(baseActivity(self)) then
        execMode(self) := emRunning
      else
        FinalizeKernelAgent

    emRunning →
      if normalExecution(self) then
        onsignal s : AGENT_COMPLETED
          execMode(self) := emActivityCompleted
        otherwise
          ExecuteActivity(innerActivity(baseActivity(self)))

    emActivityCompleted →
      if waCondition(baseActivity(self)) then
        execMode(self) := emRunning
      else
        FinalizeKernelAgent

    emCompleted → stop self
```

```
// ——————————— Switch Agent ———————————
SwitchProgram ≡
  case execMode(self) of
    emStarted →
      let caseSet = swCaseSet(baseActivity(self)) in
        choose c ∈ caseSet with
            swCaseCondition(c) = true ∧
            (∀x(x ∈ caseSet ∧ swCaseCondition(x) = true) →
                (swPriority(c) = swPriority(x)))
        // choosing the first [with highest priority] branch with a true condition
        // choose is always successful, because we have a default OTHERWISE
        foundBranch(self) := swCaseActivity(c)
        execMode(self) := emRunning

    emRunning →
      if normalExecution(self) then
        onsignal s : AGENT_COMPLETED
          execMode(self) := emActivityCompleted
        otherwise
          ExecuteActivity(foundBranch(self))

    emActivityCompleted → FinalizeKernelAgent

    emCompleted → stop self
```

```
// ——————————— Pick Agent ———————————
PickProgram ≡
  case execMode(self) of
    emStarted →
      new a : PICK_ALARM_AGENT
        Initialize(a, activity(self))
      new m : PICK_MESSAGE_AGENT
        Initialize(m, activity(self))
      execMode(self) := emRunning

    emRunning →
      if normalExecution(self) then
        onsignal s : AGENT_COMPLETED
          execMode(self) := emActivityCompleted
        otherwise
          if chosenAct(self) = undef then
            choose dsc ∈ occurredEvents(self) with MinTime(dsc)
              chosenAct(self) := onEventAct(edscEvent(dsc))
          else
            ExecuteActivity(chosenAct(self)))

    emActivityCompleted → FinalizeKernelAgent

    emCompleted → stop self
where
  MinTime(dsc) ≡ ∀d(d ∈ occurredEvents(self) ⇒ edscTime(dsc) ≤ edscTime(d))

// ——————————— Pick Message Agent ———————————
PickMessageProgram ≡
  case execMode(self) of
    emStarted → PickMessageAgentStarted
    emRunning → PickMessageAgentRunning
    emActivityCompleted →
      FinalizePickMessageAgent
      execMode(self) := emCompleted
    emCompleted → stop self
```

PickMessageAgentStarted $\equiv$
  if $eventOccured(self)$ **then**
    $execMode(self) := emCompleted$
  else
    $execMode(self) := emRunning$
    forall $event \in onMessageEventSet(activity(self))$
      GenerateInputDescriptor(event)
      // creates the input descriptor & adds it to the waiting set

PickMessageAgentRunning $\equiv$
  if *normalExecution(self)* **then**
    if $eventOccured(self)$ **then**
      $execMode(self) := emActivityCompleted$
    else
      choose $d \in completedMsgEvents$
        GenerateEventDescriptor($dscOpr(d), dscTime(d)$)
        // creates the event descriptor & adds it to *occuredEvents*
        $execMode(self) := emActivityCompleted$
where
  $completedMsgEvents \equiv$
    $\{d \mid d \in completedInOprs(rootProcess(self))$
    $\wedge\ dscOpr(d) \in onMessageEventSet(activity(self))\}$

FinalizePickMessageAgent $\equiv$
  forall $dscr \in waitingSet$ **with** $dscAgent(dscr) = self$
    remove $dscr$ **from** $waitingSet$
where
  $waitingSet \equiv waitingSetForInput(rootProcess(self))$

// ——————— Pick Alarm Agent ——— —— ———

PickAlarmProgram $\equiv$
  case *execMode(self)* **of**
    emStarted $\rightarrow$
      if *eventOccured(self)* **then**
        execMode(self) := *emCompleted*
      else
        startTime(self) := *now*
        execMode(self) := *emRunning*
    emRunning $\rightarrow$ PickAlarmAgentRunning
    emActivityCompleted $\rightarrow$ *execMode(self)* := *emCompleted*
    emCompleted $\rightarrow$ **stop** *self*

PickAlarmAgentRunning $\equiv$
  if *normalExecution(self)* **then**
    if *eventOccured(self)* **then**
      execMode(self) := *emActivityCompleted*
    else
      forall $e \in$ *triggeredAlarms*
        GenerateEventNotification(e, *triggerTime*(e, *startTime(self)*))
        // creates the event descriptor & adds it to *occuredEvents*
        execMode(self) := *emActivityCompleted*
where
  triggeredAlarms $\equiv$
    $\{e \mid e \in onAlarmEventSet(activity(self))$
    $\wedge\ triggerTime(e, startTime(self)) \leq now\}$

```
// ——————————— Flow Agent ———————————
FlowProgram ≡
  case execMode(self) of
    emStarted →
      execMode(self) := emRunning
      // creates threads to concurrently execute enclosed activities
      forall activity ∈ flowActivitySet(self)
        new fThread : FLOW_THREAD_AGENT
          Initialize(fThread, activity)
          add fThread to flowAgentSet(self)

    emRunning →
      if normalExecution(self) then
        onsignal s : AGENT_COMPLETED
          remove signalSource(s) from flowAgentSet(self)
        if flowAgentSet(self) = ∅ then
          // All threads are done, flow activity is completed.
          execMode(self) := emActivityCompleted

    emActivityCompleted → FinalizeKernelAgent

    emCompleted → stop self
```

// ——————————— Flow Thread Agent ——— ——— ———

FlowThreadProgram $\equiv$

case *execMode(self)* **of**

emStarted $\rightarrow$ *execMode(self)* := *emRunning*

emRunning $\rightarrow$

if *normalExecution(self)* **then**

onsignal s : AGENT_COMPLETED

execMode(self) := *emActivityCompleted*

otherwise

ExecuteActivity(*baseActivity(self)*)

emActivityCompleted $\rightarrow$

FinalizeKernelAgent

emCompleted $\rightarrow$ **stop** *self*

Appendix C

Data Handling Extension

C.1 Initial Definitions

domain EXPRESSION
domain VALUE
domain VARIABLE
domain MESSAGE_TYPE
domain XML_TYPE
domain XML_ELEMENT
domain MESSAGE

domain FROM_ELEMENT
domain TO_ELEMENT

domain ASSIGN
domain SCOPE
domain SCOPE_AGENT

$\text{ACTIVITY} \equiv \text{ACTIVITY}_{\text{core}} \cup \text{SCOPE} \cup \text{ASSIGN}$
$\text{ACTIVITY_AGENT} \equiv \text{ACTIVITY_AGENT}_{\text{core}} \cup \text{SCOPE_AGENT}$

// ———————— Types and Values ————————

$varType$: VARIABLE → MESSAGE_TYPE ∪ XML_TYPE ∪ XML_ELEMENT

$varValue$: VARIABLE × PROCESS → VALUE

$expValue$: EXPRESSION × PROCESS → VALUE

$value$: (EXPRESSION ∪ VARIABLE) × PROCESS → VALUE

$$value(x,p) \equiv \begin{cases} varValue(x,p), & \text{if } x \in \text{VARIABLE;} \\ expValue(x,p), & \text{if } x \in \text{EXPRESSION.} \end{cases}$$

// ———————— In/Out Operation ————————

$variable$: INOUT_OPERATION → VARIABLE

// Every input/output operation needs a variable into which

// to store the received message,

// or from which to send the message.

// ———————— In/Out Descriptor ————————

$dscVariableValue$: INOUT_DESCRIPTOR → VALUE

// The snapshot value of the activity variable in the descriptor, on the time

// the descriptor is created.

// ———————— Message Values ————————

$messageValue$: VALUE → MESSAGE

// the MESSAGE of a message-type VALUE

// ———————— Scope ————————

$scopeVariables$: SCOPE → VARIABLE-set

// The set of all the local variables of this scope.

$innerActivity$: WHILE ∪ SCOPE → ACTIVITY

// The activity defined inside a while or scope

C.2 Programs

```
// ——————— Inbox Program : AssignMessage Extended ———————
AssignMessage(p : PROCESS,
              descriptor : INPUT_DESCRIPTOR, m : MESSAGE) ≡
  AssignMessage_core(p, descriptor, m)
  AssignMessage_data(p, descriptor, m)

AssignMessage_data(p : PROCESS,
              descriptor : INPUT_DESCRIPTOR, m : MESSAGE) ≡
  if variable(dscOperation(descriptor)) ≠ undef then
    AssignMessageValueToVariable(p, descriptor, m)

AssignMessageValueToVariable(p : PROCESS,
              descriptor : INPUT_DESCRIPTOR, m : MESSAGE) ≡
  value(variable(dscOperation(descriptor)), p) := m

// ——————— Outbox Program : DelvierMessage Extended ———————
DeliverMessage(p : PROCESS, descriptor : OUTPUT_DESCRIPTOR) ≡
  DeliverMessage_core(p, descriptor)
  DeliverMessage_data(p, descriptor)

DeliverMessage_data(p : PROCESS, descriptor : OUTPUT_DESCRIPTOR) ≡
  if variable(operation) ≠ undef then    // variable should contain a message
    AssignValueToMessage(descriptor)
where
  operation = dscOperation(descriptor)

AssignValueToMessage(descriptor : OUTPUT_DESCRIPTOR) ≡
  if correlationSatisfied(descriptor) then
    add messageValue(dscVariableValue(descriptor)) to outboxSpace(self)
```

// ———————— SetInOutDescriptor Extended ————————

SetInOutDescriptor(*descriptor* : INOUT_DESCRIPTOR,
operation : INOUT_OPERATION, *agent* : KERNEL_AGENT) ≡
SetInOutDescriptor$_{core}$(*descriptor*, *operation*, *agent*)
SetInOutDescriptor$_{data}$(*descriptor*, *operation*, *agent*)

SetInOutDescriptor$_{data}$(*descriptor* : INOUT_DESCRIPTOR,
operation : INOUT_OPERATION, *agent* : KERNEL_AGENT) ≡
if $operation \in$ IN_OPERATION $\wedge$ $variable(operation) \neq undef$ **then**
SetDescriptorValue(*descriptor*)
where
$operation = dscOperation(descriptor)$

SetDescriptorValue(*descriptor* : INOUT_DESCRIPTOR) ⊟
$dscVariableValue(descriptor) :=$
$value(variable(dscOperation(descriptor)), rootProcess(self))$

// ———————— Execute Assign ————————

ExecuteAssign(*activity* : ASSIGN) ≡
forall c **in** $copyElements(activity)$
ExecuteCopy($fromSpec(c)$, $toSpec(c)$)
FinalizeActivity(*activity*)

// ———————— ExecuteBasicActivity: Extended ————————

ExecuteBasicActivity(*activity* : ACTIVITY) ≡
ExecuteBasicActivity$_{core}$(*activity*)
if $activity \in$ ASSIGN **then**
ExecuteAssign(*activity*)

// ——————— Scope Agent ———————

ScopeProgram ≡
  case *execMode*(*self*) **of**
    emStarted →
      execMode(*self*) := *emRunning*
      InitializeLocalVariables(*baseActivity*(*self*))

    emRunning →
      if *normalExecution*(*self*) **then**
        onsignal s : AGENT_COMPLETED
          execMode(*self*) := *emActivityCompleted*
        otherwise
          ExecuteActivity(*innerActivity*(*baseActivity*(*self*)))

    emActivityCompleted → FinalizeKernelAgent

    emCompleted → **stop** *self*

InitializeLocalVariables(*scope* : SCOPE) ≡
  forall v **in** *scopeVariables*(*scope*)
    variableValue(v, *rootProcess*(*self*)) := *uninitializedVariableValue*

// ——————— ExecuteStructuredActivity: Extended ———————

ExecuteStructuredActivity(*activity*) ≡
  $\text{ExecuteStructuredActivity}_{core}$(*activity*)
  if *assignedAgent*(*activity*) = *undef* **then**
    if *activity* ∈ SCOPE **then**
      new s : SCOPE_AGENT
        Initialize(s, *activity*)

Appendix D

Fault and Compensate Extension

D.1 Initial Definitions

domain SCOPE
domain COMPENSATE
domain THROW

ACTIVITY $\equiv$ ACTIVITY$_{data}$ $\cup$ SCOPE $\cup$ COMPENSATE $\cup$ THROW

domain COMPENSATE_AGENT
domain COMPENSATION_HANDLER_AGENT
domain FAULT_HANDLER_AGENT

ACTIVITY_AGENT $\equiv$ ACTIVITY_AGENT$_{data}$ $\cup$ COMPENSATE_AGENT
HANDLER_AGENT $\equiv$
    COMPENSATION_HANDLER_AGENT $\cup$ FAULT_HANDLER_AGENT
SUBPROCESS_AGENT $\equiv$ SUBPROCESS_AGENT$_{core}$ $\cup$ HANDLER_AGENT
EXECUTION_MODE $\equiv$ EXECUTION_MODE$_{core}$ $\cup$
    {*emExecutionFault*, *emFaulted*, *emFaultHandling*, *emExited*}

domain FAULT
domain SCOPE_NAME
domain COMPENSATION_MODULE
domain LOCAL_SNAPSHOT
domain CATCH_CLAUSE

domain AGENT_FAULTED
domain AGENT_EXITED
domain FORCED_TERMINATION
$\text{SIGNAL} \equiv \text{SIGNAL}_{core} \cup$
$\quad \text{AGENT_FAULTED} \cup \text{AGENT_EXITED} \cup \text{FORCED_TERMINATION}$

// ——————— Fault Extension Signal ———————
$\textit{faultExtensionSignal} : \text{KERNEL_AGENT} \rightarrow \text{BOOLEAN}$
$\textit{faultExtensionSignal} \equiv$
$\quad \exists s(s \in \textit{signalSet}(\textit{rootProcess}(\textit{self})) \wedge \textit{signalSource}(s) = \textit{self} \wedge$
$\quad\quad s \in (\text{AGENT_EXITED} \cup \text{AGENT_FAULTED} \cup \text{FORCED_TERMINATION})$

$\textit{normalExecution}(a : \text{KERNEL_AGENT}) \equiv \neg \textit{faultExtensionSignal}(a)$

// ——————— baseActivity refinement ———————
$\textit{baseActivity} : \text{KERNEL_AGENT} \rightarrow \text{ACTIVITY}$

$$\textit{baseActivity}(a) \equiv \begin{cases} \textit{baseActivity}_{core}(a), & \text{if } a \in \text{ACTIVITY_AGENT}; \\ \textit{globalScope}, & \text{if } a \in \text{PROCESS}. \end{cases}$$

// ——————— Fault Handling ———————
$activityFault$: THROW → FAULT

$faultVariable$: FAULT → VARIABLE

$fault$: (AGENT_FAULTED ∪ FORCED_TERMINATION) → FAULT

$faultThrown$: KERNEL_AGENT → FAULT

$forcedTerminationAgent$: KERNEL_AGENT → BOOLEAN
$forcedTerminationAgent(a) \equiv (faultThrown(a) = bpwsForcedTermination)$

$handlerScope$: FAULT_HANDLER_AGENT → SCOPE

$faultHandlerCatchSet$: SCOPE → CATCH_CLAUSE-set

$catchActivity$: CATCH_CLAUSE → ACTIVITY
// indicates the activity specified in a catch clause

// ——————— Compensation Handling ———————
$scopeName$: SCOPE → SCOPE_NAME
$compensationActivity$: SCOPE → ACTIVITY
// The activity inside the compensation handler of this scope.
// if there is no compensation handler, the value is `<compensate/>`.

$targetScope$: COMPENSATE → SCOPE_NAME
// The scope parameter of a compensate activity
// Note that it is just a name, not an instance of a scope activity.

$cmSet$: (SCOPE_NAME) → COMPENSATION_MODULE-set
$cmScope$: COMPENSATION_MODULE → SCOPE

$cmScopeName$: COMPENSATION_MODULE → SCOPE_NAME
$cmScopeName(cm) := scopeName(cmScope(cm))$

$cmOrder$: COMPENSATION_MODULE $\rightarrow$ PRIORITY
// To this time, the order in which the completed scopes must be compensated
// is not completely decided yet. This function, abstractly
// orders compensation modules for execution.

$topCMOrder$: COMPENSATION_MODULE $\rightarrow$ BOOLEAN
// An abstract predicate which is true if the module is the first module in
// the execution order of compensation modules.

$compHandlerModule$: COMPENSATION_HANDLER $\rightarrow$ COMPENSATION_MODULE

$cmExecuted$: COMPENSATION_MODULE $\rightarrow$ BOOLEAN
$scopeCompletionTime$: COMPENSATION_MODULE $\rightarrow$ TIME
$localSnapshot$: COMPENSATION_MODULE $\rightarrow$ LOCAL_SNAPSHOT

$snapshotVariableSet$: LOCAL_SNAPSHOT $\rightarrow$ VARIABLE-set
$snapshotVariableValue$: (LOCAL_SNAPSHOT $\times$ VARIABLE) $\rightarrow$ VALUE

$chosenCM$: COMPENSATE_AGENT $\rightarrow$ COMPENSATION_MODULE

$parentScopeName$: KERNEL_AGENT $\rightarrow$ SCOPE_NAME
$parentScopeName(a) \equiv$

$$\begin{cases} cmScopeName(compHandlerModule(a)), & \text{if } a \in \text{COMPENSATION_HANDLER;} \\ scopeName(baseActivity(parentAgent(a))), & \text{if } parentAgent(a) \in \text{SCOPE_AGENT;} \\ undef, & \text{if } a \in \text{PROCESS} \\ & \quad \vee\ parentAgent(a) \in \text{PROCESS;} \\ parentScopeName(parentAgent(a)), & \text{otherwise.} \end{cases}$$

// ——————— New Derived Function ——— — —

$activity$: KERNEL_AGENT $\rightarrow$ ACTIVITY

$$activity(a) \equiv \begin{cases} currentActivity(a), & \text{if } a \in \text{SEQUENCE_AGENT;} \\ chosenActivity(a), & \text{if } a \in \text{PICK_AGENT;} \\ foundBranch(a), & \text{if } a \in \text{SWITCH_AGENT;} \\ innerActivity(a), & \text{if } a \in \text{WHILE_AGENT} \cup \text{SCOPE_AGENT;} \\ catchActivity(executingCatch(a)), & \text{if } a \in \text{FAULT_HANDLER;} \\ baseActivity(a), & \text{otherwise.} \end{cases}$$

D.2 Programs

// ——————— Throw Activity ———————

ExecuteThrow(*activity* : THROW) ≡
  TransitionToExecutionFault(*activityFault*(*activity*))
  Synchronization(*activity*)

TransitionToExecutionFault(*fault* : FAULT) ≡
  execMode(*self*) := *emExecutionFault*
  faultThrown(*self*) := *fault*
  InformFaultToParent(*fault*)

InformFaultToParent(*fault* : FAULT) ≡
  if *self* $\notin$ (SCOPE_AGENT $\cup$ PROCESS) **then**
    trigger *s* : AGENT_FAULTED, *parentAgent*(*self*)
      fault(*s*) := *fault*

// ——————— Scope ———————

ScopeProgram ≡
  $\text{ScopeProgram}_{data}$
  case *execMode*(*self*) **of**
    emRunning → ScopeAgentRunningExtended

    emActivityCompleted → InstallCompensationHandler

    emExecutionFault → ScopeAgentExecutionFault

    emFaultHandling → ScopeAgentFaultHandling

    emExited → **stop** *self*

    emFaulted → **stop** *self*

InstallCompensationHandler $\equiv$
  extend COMPENSATION_MODULE **with** *cm*
    scopeCompletionTime(*cm*) := *now*
    cmScope(*cm*) := *baseActivity*(*self*)
    RegisterLocalSnapshot(*cm*, *baseActivity*(*self*))
    add *cm* **to** *cmSet*(*parentScopeName*(*self*))

RegisterLocalSnapshot(*cm* : COMPENSATION_MODULE, *scope* : SCOPE) $\equiv$
  extend LOCAL_SNAPSHOT **with** *snapshot*
    forall *v* **in** *scopeVariables*(*scope*)
      snapshotVariableValue(*snapshot*, *v*) := *variableValue*(*v*, *rootProcess*(*self*))
      add *v* **to** *snapshotVariableSet*(*snapshot*)
    localSnapshot(*cm*) := *snapshot*

ScopeAgentRunningExtended $\equiv$
  if *faultExtensionSignal*(*self*) **then**
    onsignal *s* : AGENT_EXITED
      execMode(*self*) := *emActivityCompleted*
    otherwise
      onsignal *s* : AGENT_FAULTED
        execMode(*self*) := *emExecutionFault*
        faultThrown(*self*) := *fault*(*s*)
      otherwise
        onsignal *s* : FORCED_TERMINATION
          execMode(*self*) := *emExecutionFault*
          faultThrown(*self*) := *fault*(*s*)
          // The scope agent resends the forced termination signal
          // in its execution-fault mode.

ScopeAgentExecutionFault $\equiv$
  TerminateBasicActivity(*self*)
  InitiateForcedTermination
  CreateFaultHandler
  execMode(*self*) := *emFaultHandling*

InitiateForcedTermination $\equiv$
  forall *child* **in** *childAgents(self)*
    trigger s : FORCED_TERMINATION, *child*
      $fault(s) := bpwsForcedTermination$

CreateFaultHandler $\equiv$
  new *handler* : FAULT_HANDLER_AGENT
    parentAgent(handler) := *self*
    handlerScope(handler) := *baseActivity(self)*
    faultThrown(handler) := *faultThrown(self)*

ScopeAgentFaultHandling $\equiv$
  onsignal s : AGENT_COMPLETED
    execMode(self) := *emExited*
    PassExitedToParent
  otherwise
    onsignal s : AGENT_FAULTED
      faultThrown(self) := *fault(s)*
      PassFaultedToParent(*fault(s)*)
    otherwise
      onsignal s : FORCED_TERMINATION
        execMode(self) := *emFaulted*
        faultThrown(self) := *fault(s)*
        PassForcedTerminationToChildren(*fault(s)*)

PassExitedToParent $\equiv$
  trigger s' : AGENT_EXITED, *parentAgent(self)*

PassFaultedToParent($fault$: FAULT) $\equiv$
  trigger s' : AGENT_FAULTED, *parentAgent(self)*
    $fault(s') := fault$

PassForcedTerminationToChildren($fault$: FAULT) $\equiv$
  forall *child* **in** *childAgents(self)*
    trigger s' : FORCED_TERMINATION, *child*
    $fault(s') := fault$

TerminateBasicActivity($agent$: KERNEL_AGENT) $\equiv$
let $activity = activity(agent)$
case $activity$ **of**
RECEIVE $\rightarrow$
RemoveDscrFromInputWaitingSet($agent, activity$)
REPLY $\rightarrow$
RemoveDscrFromOutputWaitingSet($agent, activity$)
INVOKE $\rightarrow$
RemoveDscrFromInputWaitingSet($agent, activity$)
RemoveDscrFromOutputWaitingSet($agent, activity$)

RemoveDscrFromInputWaitingSet($agent$: KERNEL_AGENT,
$activity$: ACTIVITY) $\equiv$
choose d **in** $waitingSetForInput$ **with**
$dscAgent(d) = agent \wedge dscOperation(d) = activity$
remove d **from** $waitingSetForInput$
where
$waitingSetForInput \equiv waitingSetForInput(rootProcess(agent))$

RemoveDscrFromOutputWaitingSet($agent$: KERNEL_AGENT,
$activity$: ACTIVITY) $\equiv$
choose d **in** $waitingSetForOutput$ **with**
$dscAgent(d) = agent \wedge dscOperation(d) = activity$
remove d **from** $waitingSetForOutput$
where
$waitingSetForOutput \equiv waitingSetForOutput(rootProcess(agent))$

```
// ——————— Fault Handler ———————
FaultHandlerProgram ≡
  case execMode(self) of
    emStarted → FaultHandlerStarted

    emRunning →
      FaultHandlerRunningNormal
      FaultHandlerRunningExtended

    emActivityCompleted → FinalizeKernelAgent

    emCompleted → stop self

    emExecutionFault → FaultHandlerExecutionFault

    emFaulted → stop self

FaultHandlerStarted ≡
  execMode(self) := emRunning
  ChooseMatchingCatchClause
  // executingCatch must be set to the matching catch

ChooseMatchingCatchClause ≡
  choose c ∈ faultHandlerCatchSet(handlerScope(self))
      with matchingCatch(c, faultThrown(self))
    executingCatch(self) := c

FaultHandlerRunningNormal ≡
  if normalExecution(self) then
    onsignal s : AGENT_COMPLETED
      execMode(self) := emActivityCompleted
    otherwise
      if executingCatch(self) = undef then
        PickRethtowCatchClause
      else
        ExecuteCatchActivity
```

PickRethrowCatchClause $\equiv$
  executingCatch(*self*) := *rethrowCatchClause*

ExecuteCatchActivity $\equiv$
  ExecuteActivity(*catchActivity*(*executingCatch*(*self*)))

FaultHandlerRunningExtended $\equiv$
  if *faultExtensionSignal*(*self*) **then**
    onsignal s : AGENT_EXITED
      execMode(*self*) := *emActivityCompleted*
    otherwise
      onsignal s : AGENT_FAULTED
        TransitionToExecutionFault($fault(s)$)

FaultHandlerExecutionFault $\equiv$
  TerminateBasicActivity(*self*)
  execMode(*self*) := *emFaulted*
  InitiateForcedTermination

// ——————— Compensate ———————
CompensateProgram $\equiv$
  case *execMode*(*self*) **of**
    emStarted $\rightarrow$ ChooseNextCM

    emRunning $\rightarrow$ CompensateAgentRunning

    emActivityCompleted $\rightarrow$ ChooseNextCM

    emCompleted $\rightarrow$ **stop** *self*

    emExecutionFault $\rightarrow$ WaitForTermination

    emFaulted $\rightarrow$ **stop** *self*

ChooseNextCM ≡
  if *thereIsAtLeastOneModule* **then**
    ChooseMatchingCompensationModule
    execMode(self) := *emRunning*
  else
    FinalizeKernelAgent

ChooseMatchingCompensationModule ≡
  choose *cm* **in** *cMSet(parentScopeName(self))* **with** *mathingCM(cm)*
    chosenCM(self) := *cm*
    remove *cm* **from** *cmSet(parentScopeName(self))*

thereIsAtLeastOneModule ≡
  $\exists x(x \in cMSet(parentScopeName(self)) \wedge matchingCM(x)$

matchingCM(cm) ≡
    $[targetScope(baseActivity(self)) = undef$
      $\vee\ cMScopeName(cm) = targetScope(baseActivity(self))]$
    $\wedge\ topCMOrder(cm)$

CompensateAgentRunning ≡
  if *normalExecution(self)* **then**
    onsignal *s* : AGENT_COMPLETED
      execMode(self) := *emActivityCompleted*
    otherwise
      ExecuteChosenCompensationModule
  if *faultExtensionSignal(self)* **then**
    onsignal *s* : AGENT_FAULTED
      TransitionToExecutionFault(*fault(s)*)
    otherwise
      onsignal *s* : FORCED_TERMINATION
        faultThrown(self) := *fault(s)*
        PassForcedTerminationToChildren(*fault(s)*)
        execMode(self) := *emExecutionFault*

ExecuteChosenCompensationModule ≡
  let *cm* = *chosenCM(self)*
    if ¬*cmExecuted*(*cm*) **then**
      CreateCompensationHandler(*cm*)

CreateCompensationHandler(*cm* : COMPENSATION_MODULE) ≡
  new *cma* : COMPENSATION_HANDLER_AGENT
    Initialize(*cma*, *compensationActivity*(*cmScope*(*cm*)))
    cmExecuted(*cm*) := *true*
    compHandlerModule(*cma*) := *cm*

// ———————— Compensation Handler ————————
CompensationHandlerProgram ≡
  case *execMode*(*self*) **of**
    emStarted →
      RestoreLocalVariables
      execMode(*self*) := *emRunning*

    emRunning →
      if *normalExecution*(*self*) **then**
        onsignal *s* : AGENT_COMPLETED
          execMode(*self*) := *emActivityCompleted*
        otherwise
          ExecuteActivity(*baseActivity*(*self*))
      HandleExceptionInRunningMode

      emActivityCompleted → FinalizeKernelAgent

      emCompleted → **stop** *self*

      emExecutionFault → WaitForTermination

      emFaulted → **stop** *self*

RestoreLocalVariables ≡
  let *snapshot* = *localSnapshot*(*compensationModule*(*self*))
    forall *v* **in** *snapshotVariableSet*(*snapshot*)
      value(*v*, *rootProcess*(*self*)) := *snapshotVariableValue*(*snapshot*, *v*)

// ———————— Process ————————
// Note: A process instance works similar to a scope in case of a fault
// The process behavior is the same as the scope agent in the execution fault mode
ProcessProgram ≡
  $\text{ProcessProgram}_{core}$
  case *execMode*(*self*) **of**
    emRunning → ProcessAgentRunningExtended

    emExecutionFault → ScopeAgentExecutionFault

    emFaultHandling → ProcessAgentFaultHandling

    emExited → **stop** *self*

    emFaulted → **stop** *self*

ProcessAgentRunningExtended ≡
  if *faultExtensionSignal*(*self*) **then**
    onsignal *s* : AGENT_EXITED
      execMode(*self*) := *emActivityCompleted*
    otherwise
      onsignal *s* : AGENT_FAULTED
        execMode(*self*) := *emExecutionFault*
        faultThrown(*self*) := *fault*(*s*)
  // No forced termination signal

ProcessAgentFaultHandling ≡
  onsignal s : AGENT_COMPLETED
    execMode(*self*) := *emExited*
  otherwise
    onsignal s : AGENT_FAULTED
      execMode(*self*) := *emFaulted*
// If a fault occurs inside the fault handler of a process, the process terminates

// ———————— Sequence Agent ————————
SequenceProgram ≡
  $\mathsf{SequenceProgram}_{core}$
  case *execMode*(*self*) **of**
    emRunning → HandleExeceptionsInRunningMode
    emExecutionFault → WaitForTermination
    emFaulted → **stop** *self*

HandleExceptionsInRunningMode ≡
  if *faultExtensionSignal*(*self*) **then**
    onsignal s : AGENT_EXITED
      execMode(*self*) := *emActivityCompleted*
    otherwise
      onsignal s : AGENT_FAULTED
        TransitionToExecutionFault($fault(s)$)
      otherwise
        onsignal s : FORCED_TERMINATION
          faultThrown(*self*) := *fault*(s)
          PassForcedTerminationToChildren(*fault*(s))
          execMode(*self*) := *emExecutionFault*

WaitForTermination ≡
  if *forcedTerminationAgent*(*self*) **then**
    execMode(*self*) := *emFaulted*
    TerminateBasicActivity(*self*)
  else
    onsignal s : FORCED_TERMINATION
      faultThrown(*self*) := *fault*(s)
      execMode(*self*) := *emFaulted*
      TerminateBasicActivity(*self*)
      PassForcedTerminationToChildren(*fault*(s))

// ————— While Agent —————
WhileProgram ≡
  $\text{WhileProgram}_{core}$
  case *execMode*(*self*) **of**
    emRunning → HandleExeceptionsInRunningMode
    emExecutionFault → WaitForTermination
    emFaulted → **stop** *self*

// ————— Switch Agent —————
SwitchProgram ≡
  $\text{SwitchProgram}_{core}$
  case *execMode*(*self*) **of**
    emRunning → HandleExeceptionsInRunningMode
    emExecutionFault → WaitForTermination
    emFaulted → **stop** *self*

// ————— Pick Agent —————
PickProgram ≡
  $\text{PickProgram}_{core}$
  case *execMode*(*self*) **of**
    emRunning → HandleExeceptionsInRunningMode
    emExecutionFault → WaitForTermination
    emFaulted → **stop** *self*

// ——————— Pick Message Agent ———————

PickMessageProgram ≡
  $\mathsf{PickMessageProgram}_{core}$
  case *execMode*(*self*) **of**
    emRunning →
      if *faultExtensionSignal*(*self*) **then**
        onsignal s : FORCED_TERMINATION
          faultThrown(*self*) := *fault*(s)
          execMode(*self*) := *emExecutionFault*

    emExecutionFault →
      // A fault occurs only due to a forced termination
      FinalizePickMessageAgent
      execMode(*self*) := *emFaulted*

    emFaulted → **stop** *self*

// ——————— Pick Alarm Agent ———————

PickAlarmProgram ≡
  $\mathsf{PickAlarmProgram}_{core}$
  case *execMode*(*self*) **of**
    emRunning →
      if *faultExtensionSignal*(*self*) **then**
        onsignal s : FORCED_TERMINATION
          faultThrown(*self*) := *fault*(s)
          execMode(*self*) := *emExecutionFault*

    emExecutionFault →
      // A fault occurs only due to a forced termination
      execMode(*self*) := *emFaulted*

    emFaulted → **stop** *self*

```
// ——————— Flow Agent ———————
FlowProgram ≡
  FlowProgram_core
  case execMode(self) of
    emRunning → HandleFlowExeceptionsInRunningMode
    emExecutionFault → WaitForTermination
    emFaulted → stop self


HandleFlowExceptionsInRunningMode ≡
  if faultExtensionSignal(self) then
    onsignal s : AGENT_EXITED
      UpdateFlowAgentSet(s)
    otherwise
      onsignal s : AGENT_FAULTED
        TransitionToExecutionFault(fault(s))
      otherwise
        onsignal s : FORCED_TERMINATION
          faultThrown(self) := fault(s)
          PassForcedTerminationToParent(fault(s))
          execMode(self) := emExecutionFault

UpdateFlowAgentSet(s : SIGNAL) ⊟
  remove sourceSignal(s) from flowAgentSet(self)

// ——————— Flow Thread Agent ———————
FlowThreadProgram ⊟
  case execMode(self) of
    emRunning → HandleExeceptionsInRunningMode
    emExecutionFault → WaitForTermination
    emFaulted → stop self
```

Appendix E

Signaling

E.1 Introduction

domain AGENT_COMPLETED
domain AGENT_FAULTED
domain AGENT_EXITED
domain FAULT_HANDLER_COMPLETED
domain FAULT_HANDLER_FAULTED
domain FORCED_TERMINATION

SIGNAL ≡
AGENT_COMPLETED
∪ AGENT_FAULTED
∪ AGENT_EXITED
∪ FAULT_HANDLER_COMPLETED
∪ FAULT_HANDLER_FAULTED
∪ FORCED_TERMINATION

$signalSource$: SIGNAL $\rightarrow$ KERNEL_AGENT
$signalTarget$: SIGNAL $\rightarrow$ KERNEL_AGENT

$signalSet$: PROCESS $\rightarrow$ SIGNAL-set

// ——————— New Keywords ———————

trigger s : SIGNAL_DOMAIN, agent
  Rule

$\equiv$

extend SIGNAL_DOMAIN **with** s
  $signalSource(s) := self$
  $signalTarget(s) :=$ agent
  add s **to** $signalSet(rootProcess(self))$
  Rule

onsignal s : SIGNAL_DOMAIN
  $Rule_1$
otherwise
  $Rule_2$

$\equiv$

if $\exists s(s \in signalSet(rootProcess(self)) \wedge$
    $signalSource(s) = self \;\wedge\; s \in$ SIGNAL_DOMAIN)
  choose $s \in signalSet(rootProcess(self))$ **with**
      $s \in$ SIGNAL_DOMAIN $\wedge\; signalSource(s) = self$
    remove s **from** $signalSet(rootProcess(self))$
    $Rule_1$
else
  $Rule_2$

Appendix F

A Draft Proposal for Synchronized Request-Respond

The *reply* activity in BPEL is different from other activities as it cannot be used independently. A *reply* activity should always follow a previous *receive* activity[1]. The LRM states that "*... a reply activity must always be preceded by a receive activity for the same partner link, portType and (request/response) operation, such that no reply has been sent for that receive activity*" [$11.4]. This means that all the PPO[2] and correlation parameters of a *reply* activity are redundant[3]; i.e., they should have the same values as those of the corresponding *receive* activity. The reason for these limitations is that *reply* is introduced in BPEL to provide synchronous input/output (request/response) behaviour.

Assigning a *reply* activity to its corresponding *receive* activity seems to be a challenge about which the WSBPEL TC has a number open issues [35].

The *invoke* activity in BPEL handles synchronous output/input operations. As a business process is not supposed to perform any task between a pair of synchronous output and input operations, one activity can handle this task. Furthermore, as all the PPO parameters for the output and input operations in a synchronous communication is identical, this activity identifies only one set of PPO parameters. Hence, there is no further complication

[1]See Requirement #4 of the reply in Appendix A.2.

[2]PartnerLink, PortType, and Operation

[3]"The correlation between a request and the corresponding reply is based on the constraint that more than one outstanding synchronous request from a specific partner link for a particular portType, operation and correlation set(s) MUST NOT be outstanding simultaneously." [$11.4]

```
<synchreceive partnerLink="ncname" portType="qname" operation="ncname"
           inputvariable="ncname"? outputvariable="ncname"?
           createInstance="yes|no"? faultName="qname"?
           standard-attributes>

    <in-std-elements>?
      standard-elements
    </in-std-elements>

    <out-std-elements>?
      standard-elements
    </out-sd-elements>

    <correlations>?
       <correlation set="ncname" initiate="yes|no"?
                    pattern="in|out|out-in"/>+
    </correlations>

    activity

</synchreceive>
```

Spec F.1: Format of a *synchreceive* activity

regarding the assignment of the input and output operations.

Given the advantages of using an *invoke* actiivity for synchronous output/input operations, we propose here a new activity to be defined for synchronous *receive/reply* as described in Spec F.1. The behaviour of this activity is exactly equivalent to the behaviour of a *sequence* starting with a *receive* and ending with a synchronous *reply*, performing the main activity of *synchreceive* in between. To be more precise, the *synchreceive* activity presented in Spec F.2 is equivalent to the *sequence* activity presented in Spec F.3.

In addition, to further elaborate the semantics of *synchreceive*, we address the following issues:

1. standard-attributes are divided into two sets, one for the receive part, and the other for the reply part;

```
<synchreceive partnerLink="pl1" portType="pt1" operation="op1"
           inputvariable="vi" outputvariable="vo"
           createInstance="civalue" faultName="fn"
           faultvariable="fv"
           standard-attributes>
    <correlations>
       <correlation set="cset1" initiate="yes"
                    pattern="in"/>
       <correlation set="cset2" initiate="no"
                    pattern="out"/>
       <correlation set="cset3" initiate="no"
                    pattern="in-out"/>
    </correlations>
    activity
</synchreceive>
```

Spec F.2: An example of using the *synchreceive* activity

2. a new variable parameter *faultVariable* is also introduced to enable sending a fault reply. When a *synchreceive* activity finishes executing its main activity, the decision on sending a fault or a normal message will be made based on the values of this variable. If the variable is not undefined, it indicates that a fault should be sent out as the reply.

```
 <sequence>
   <receive partnerLink="pl1" portType="pt1" operation="op1"
          variable="vo" createInstance="civalue"
          standard-attributes>
     <correlations>
        <correlation set="cset1" initiate="yes">
        <correlation set="cset3" initiate="no">
     </correlations>
   </receive>

   activity

   <reply partnerLink="pl1" portType="pt1" operation="op1"
          variable="vi" faultName="fn" faultvariable="fv"
          standard-attributes>
     <correlations>
        <correlation set="cset2" initiate="no">
        <correlation set="cset3" initiate="no">
     </correlations>
   </reply>
</sequence>
```

Spec F.3: The *sequence* activity equivalent to the *synchreceive* example

Bibliography

[1] J.R. Abrial. *The B-Book: Assigning Programs to Meanings.* Cambridge University Press, 1996.

[2] BBN Technologies, Carnegie-Mellon University, Nokia, Stanford University, SRI International, and Yale University. *DAML-S Specification Version 0.9*, May 2003. www.daml.org/services/daml-s.

[3] BEA Systems Inc., IBM Corp., and Microsoft Corp. *Web Services Addressing (WS-Addressing)*, March 2004. Last cited May 2004, ftp://www6.software.ibm.com/software/developer/library/ws-add200403.pdf.

[4] BEA Systems, International Business Machines Corporation, Microsoft Corporation, SAP AG, Siebel Systems. *Business Process Execution Language for Web Services Version 1.1*, May 2003. Last cited July 2004, `http://www-106.ibm.com/developerworks/webservices/library/ws-bpel`.

[5] A. Benczur, U. Glässer, and T. Lukovszki. Formal Description of a Distributed Location Service for Ad Hoc Mobile Networks. In E. Börger, A. Gargantini, and E. Riccobene, editors, *Abstract State Machines 2003 - Advances in Theory and Practice*, volume 2589, pages 204–217. Springer, 2003.

[6] E. Börger. A Logical Operational Semantics for Full Prolog. Part I: Selection Core and Control. In E. Börger, H. Kleine Büning, M. M. Richter, and W. Schönfeld, editors, *CSL'89. 3rd Workshop on Computer Science Logic*, volume 440 of *LNCS*, pages 36–64. Springer, 1990.

[7] E. Börger. A Logical Operational Semantics of Full Prolog. Part II: Built-in Predicates for Database Manipulation. In B. Rovan, editor, *Mathematical Foundations of Computer Science*, volume 452 of *LNCS*, pages 1–14. Springer, 1990.

[8] E. Börger, U. Glässer, and W. Müller. The Semantics of Behavioral VHDL'93 Descriptions. In *EURO-DAC'94. European Design Automation Conference with EURO-VHDL'94*, pages 500–505, Los Alamitos, California, 1994. IEEE CS Press.

[9] E. Börger, U. Glässer, and W. Müller. Formal Definition of an Abstract VHDL'93 Simulator by EA-Machines. In C. Delgado Kloos and P. T. Breuer, editors, *Formal Semantics for VHDL*, pages 107–139. Kluwer Academic Publishers, 1995.

[10] E. Börger and D. Rosenzweig. The WAM – Definition and Compiler Correctness. In C. Beierle and L. Plümer, editors, *Logic Programming: Formal Methods and Practical Applications*, Studies in Computer Science and Artificial Intelligence, chapter 2, pages 20–90. North-Holland, 1994.

[11] E. Börger and R. Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis.* Springer-Verlag, 2003.

[12] Egon Börger. The ASM Refinement Method. *Formal Aspects of Computing*, pages 237–257, 2003.

[13] Francisco Curbera, William A. Nagy, and Sanjiba Weerawarana. Web Services: Why and how, 2001. cited April 2004, `http://researchweb.watson.ibm.com/people/b/bth/OOWS2001/nagy.pdf`.

[14] John Derrick and Eerke Boiten. *Refinement in Z and Object-Z: Foundations and Advanced Applications.* Formal Approaches to Computing and Information Technology. Springer, May 2001.

[15] E. Börger and J. Schmid. Composition and Submachine Concepts for Sequential ASMs. In P. Clote and H. Schwichtenberg, editors, *Computer Science Logic (Proceedings of CSL 2000)*, volume 1862 of *LNCS*, pages 41–60. Springer-Verlag, 2000.

[16] R. Eschbach, U. Gässer, R. Gotzhein, and A. Prinz. On the Formal Semantics of SDL-2000: A Compilation Approach Based on an Abstract SDL Machine. In Y. Gurevich

and P. Kutter and M. Odersky and L. Thiele, editor, *Abstract State Machines: Theory and Applications*, volume 1912 of *LNCS*, pages 242–265. Springer-Verlag, 2000.

[17] Dirk Fahland. Ein Ansatz einer formalen Semantik der Business Process Execution Language for Web Services mit Abstract State Machines. Technical report, Humboldt-Universität zu Berlin, June 2004.

[18] R. Farahbod, U. Glässer, and M. Vajihollahi. Specification and Validation of the Business Process Execution Language for Web Services. Technical Report SFU-CMPT-TR-2003-06, Simon Fraser University, September 2003.

[19] R. Farahbod, U. Glässer, and M. Vajihollahi. Abstract Operational Semantics of the Business Process Execution Language for Web Services. Technical Report SFU-CMPT-TR-2004-03, Simon Fraser University, April 2004.

[20] R. Farahbod, U. Glässer, and M. Vajihollahi. Specification and Validation of the Business Process Execution Language for Web Services. In *Proc. of the 11th International Workshop on Abstract State Machines (ASM'2004)*. Springer-Verlag, 2004.

[21] D. F. Ferguson, B. Lovering, T. Storey, and J. Shewchuk. Secure, Reliable, Transacted Web Services: Architecture and Composition. Technical report, MSDN Library, September 2003.

[22] U. Glässer, R. Gotzhein, and A. Prinz. The formal semantics of sdl-2000: status and perspectives. *Comput. Networks*, 42(3):343–358, 2003.

[23] U. Glässer, Y. Gurevich, and M. Veanes. An Abstract Communication Architecture for Modeling Distributed Systems. *To appear in IEEE Transactions on Software Engineering*, 2004.

[24] Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.

[25] Y. Gurevich. Sequential Abstract State Machines Capture Sequential Algorithms. *ACM Transactions on Computational Logic*, 1(1):77–111, July 2000.

[26] Y. Gurevich and J. Huggins. The Railroad Crossing Problem: An Experiment with Instantaneous Actions and Immediate Reactions. In *Proceedings of CSL'95 (Computer Science Logic)*, volume 1092 of *LNCS*, pages 266–290. Springer, 1996.

[27] Y. Gurevich and N. Tillmann. Partial Updates: Exploration. *Journal of Universal Computer Science*, 7(11):917–951, 2001.

[28] ITU-T Recommendation Z.100 Annex F (11/00). *SDL Formal Semantics Definition.* International Telecommunication Union, 2001.

[29] M. Koshkina and F. van Breugel. Verification of Business Processes for Web Services. Technical Report CS-2003-11, York University, October 2003.

[30] A. Martens. On usability of web services. *1st Web Services Quality Workshop (WQW 2003)*, 2003.

[31] A. Martens. *Verteilte Geschftsprozesse - Modellierung und Verifikation mit Hilfe von Web Services.* PhD thesis, Humboldt University of Berlin, Berlin, Germany, 2003.

[32] A. Martens. Analysis and re-engineering of web services. *To appear in 6th International Conference on Enterprise Information Systems (ICEIS'04)*, 2004.

[33] Microsoft FSE Group. *The Abstract State Machine Language.* cited June 2003, http://research.microsoft.com/fse/asml/.

[34] Srini Narayanan and Sheila A. McIlraith. Simulation, verification and automated composition of web services. In *Proceedings of the eleventh international conference on World Wide Web*, pages 77–88. ACM Press, 2002.

[35] Organization for the Advancement of Structured Information Standards (OASIS). *WS BPEL issues list*, April 2004. http://www.oasis-open.org.

[36] David O'Riordan. Business Process Standards for Web Services. Technical report, Tect, April 2002. Available from http://www.webservicesarchitect.com/content/articles/oriordan01.asp.

[37] C. Peltz. Web services orchestration and choreography. *Web Services Journal*, 2004. http://www.sys-con.com/webservices/.

[38] Chris Peltz. Web services orchestration and choreography. *Web Services Journal*, 3, july 2003.

[39] R. Eschbach and U. Glässer and R. Gotzhein and M. von Löwis and A. Prinz. Formal Definition of SDL-2000: Compiling and Running SDL Specifications as ASM Models. *Journal of Universal Computer Science*, 7(11):1024–1049, 2001.

[40] Mike Rosen and John Parodi. Architecting Web Services. Technical report, IONA Technologies PLC, December 2001.

[41] J. Ruf, D. Hoffmann, J. Gerlach, T. Kropf, W. Rosenstiehl, and W. Mueller. The simulation semantics of systemc. In *Proceedings of the conference on Design, automation and test in Europe*, pages 64–70. IEEE Press, 2001.

[42] R. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine: Definition, Verification, Validation.* Springer-Verlag, 2001.

[43] M. Vajihollahi. High level specification and validation of the business process execution language for web services. Master's thesis, Simon Fraser University, Burnaby, Canada, April 2004.

[44] W.M.P. van der Aalst, M. Dumas, A.H.M. ter Hofstede, and P. Wohed. Pattern-Based Analysis of BPML (and WSCI). Technical Report FIT-TR-2002-05, Queensland University of Technology, 2002.

[45] W.M.P. van der Aalst, M. Dumas, A.H.M. ter Hofstede, and P. Wohed. Analysis of web services composition languages: The case of bpel4ws. *1st Web Services Quality Workshop (WQW 2003)*, 2003.

[46] J.M. Vidal, P. Buhler, and C. Stahl. Multiagent systems with workflows. *IEEE Internet Computing*, 8(1):76–82, January/February 2004.

[47] W3C. *XML Path Language (XPath) Version 1.0*, November 1999. Last cited June 2004, `http://www.w3.org/TR/1999/REC-xpath-19991116`.

[48] W3C. *XML Schema: Formal Description*, September 2001. Last cited July 2004, `http://www.w3.org/TR/xmlschema-formal/`.

[49] W3C. *Web Service Choreography Interface (WSCI) 1.0*, August 2002. Last cited May 2004, `http://www.w3.org/TR/2002/NOTE-wsci-20020808`.

[50] W3C. *SOAP Version 1.2 Part 0: Primer*, June 2003. Last cited April 2004, http://www.w3.org/TR/2003/REC-soap12-part0-20030624/.

[51] W3C. *Web Services Description Language (WSDL) Version 1.2 Part 1: Core Language*, June 2003. Last cited May 2004, http://www.w3.org/TR/2003/WD-wsdl12-20030303.

[52] W3C. *Extensible Markup Language (XML) 1.0 (Third Edition)*, February 2004. Last cited July 2004, http://www.w3.org/TR/2004/REC-xml-20040204.

[53] Jim Woodcock and Jim Davies. *Using Z: Specification, refinement, and proof.* Prentice Hall Europe, 1996.

[54] World Wide Web Consortium. *Web Services Architecture*, February 2004. Last cited May 2004, http://www.w3.org/TR/ws-arch/.

Index

abstract model, 25, 26
abstract process, 13, 61, 65
abstract state machine, *see* ASM
    distributed, *see* DASM
activity, 13
    assign, 14, 61, 62
    empty, 14
    flow, 15, 33
    invoke, 14
    pick, 15
    receive, 14, 31
    reply, 14
    scope, 19, 61, 72
    sequence, 15
    switch, 15
    throw, 15
    wait, 14
    while, 15
activity agent, 31
    definition, 27
agent faulted, 95
agent interaction model, 3, 48, 49
agent-exited signal, 82
agent-faulted signal, 82, 95
ASM, 1, 2

basic activity, 13
block constructor, 22
BPEL, 1, 12, 13
BPEL Abstract Machine, 1, 20, 24, 25
BPEL4WS, *see* BPEL
business protocol, 13, 65
business token, 19

choose constructor, 22
choreography, 12
compensate
    activity, 74
    agent, 94
compensation
    definition, 19
    handler, 19, 74
    module, 92
conditional constructor, 22
conservative extension, 41
correlation
    definition, 18
correlation set, 19
correlation token, 19

DASM, 20
DASM agents, 21
data expressions, 61
DRL, 62
dummy process, 27, 28

executable business process, 13
execution lifecycle, 46, 47
    extended, 79
    normal, 47, 79
exited signal, 79
extension
    data handling, 60, 65, 69
    fault and compensation, 73, 82, 83

fault handler, 19, 74, 75, 82
faulted signal, 79
flow agent, 33
flow thread agent, 32, 33
forall constructor, 22
forced-termination signal, 82, 95

Freedom of Abstraction, 38

global variables, 61
ground model, 3, 101

horizontal extensions, 36
HTTP, 10
HTTPS, 10

IDL, 8
import constructor, 23
inbox manager, 27, 28, 31
input descriptor, 66
instantiation, 42
interoperability, 8, 12

local snapshot, 97
local variables, 62
Long Running Transactions, 19, 73
LRM, 1, 51
LRT, *see* Long Running Transactions

network abstract machine, 25

OASIS, 1, 13
opaque message, 58
Orchestration, 12
outbox manager, 27, 28
output descriptor, 28

process agent, 30
process execution model, 3, 36, 45, 46, 49, 79
process execution tree, 46
process instance, 27

refinement
- complete, 40
- conservative, 41
- correct, 40
- data, 41
- horizontal, 41, 43
- procedural, 42
- submachine, 42
- vertical, 43

requirements list, 36, 51, 60
- data handling, 62

requirements lists, 53, 106
reverse set, 23

scope, *see* activity, scope
service abstract machine, 25
SMTP, 10
SOAP, 2, 10
structured activity, 13
substitutivity
- principle of, 37

throw, 76

UDDI, 11

variables, 61

Web, 7, 8
Web services, 10, 11
- architecture, 7
- composition, 11, 12
- definition, 7–9

World Wide Web, 1
WS-Addressing, 10
WS-MetadataExchange, 11
WSCI, 12
WSDL, 1, 2, 11

XML, 10
XPath, 62
XSD, 10, 11