

Technical Report

# Complete Abstract Operational Semantics for the Web Service Business Process Execution Language

Dirk Fahland  
fahland@informatik.hu-berlin.de

September 12, 2005

Humboldt-Universität zu Berlin, Institut für Informatik



## Abstract

In this technical report we present an abstract operational semantics for the *Business Process Execution Language for Web Services*, or BPEL for short. In effect, the semantics defined herein are a variation and an extension of the semantics published first in [FGV04a] and [Far04] defined by the group of Uwe Glässer the Simon Fraser University, Vancouver, Canada. We namely add semantics for *correlation handling*, *dead path elimination* and *event handling*; we define the *data handling* on a finer level; we slightly alter the basic framework of how activities are formalized in [FGV04a] in order to achieve greater robustness against changes of the informal specification.

Furthermore this technical report serves as a base for a joint work with the group of Simon Fraser University.

# Contents

<b>1</b>	<b>Introduction</b>	<b>11</b>
1.1	Introduction to WS-BPEL . . . . .	12
1.2	Existing Work . . . . .	13
1.3	Aim of this Work . . . . .	14
<b>2</b>	<b>The ASM model</b>	<b>16</b>
2.1	Ground Models . . . . .	16
2.2	Abstract-State Machines . . . . .	16
2.2.1	Derived Functions and Abstract Functions . . . . .	17
2.2.2	Creating New Elements . . . . .	18
2.2.3	Agents . . . . .	18
2.3	How to Read this Document . . . . .	19
<b>3</b>	<b>Overall Architecture and Approach to Formalization</b>	<b>22</b>
3.1	Basic Thoughts about the Architecture of BPEL . . . . .	22
3.1.1	Activities . . . . .	22
3.1.2	Messages and Instantiation . . . . .	24
3.1.3	Extensions to BPEL . . . . .	24
3.2	Formalizing the Architecture of BPEL . . . . .	24
3.2.1	Formalizing Activities . . . . .	24
3.2.2	Formalizing the Inbox Manager . . . . .	26
3.2.3	Formalizing Extensions of BPEL . . . . .	26
<b>4</b>	<b>Process Definition</b>	<b>27</b>
4.1	Translating a Process Definition into an Initial State . . . . .	27
4.2	Definition of Process Structure . . . . .	29
4.2.1	The Process . . . . .	29
4.2.2	Activity Tree . . . . .	29
4.2.3	Links . . . . .	30
4.2.4	Variables . . . . .	31
<b>5</b>	<b>Control Flow</b>	<b>33</b>
5.1	Process Instances . . . . .	33
5.1.1	Instance of an Activity . . . . .	33
5.1.2	Executing an Instance of an Activity . . . . .	33
5.1.3	The Subinstance Tree . . . . .	34
5.2	Agents execute Activities . . . . .	35
5.3	The Internal States of an Activity . . . . .	36

---

5.4	Dynamics of the Control Flow . . . . .	38
5.4.1	The signal concept . . . . .	38
5.4.2	Dynamics of Links . . . . .	40
5.4.3	Starting Activities . . . . .	42
5.4.4	Completing Activities . . . . .	42
5.4.5	Hierarchy of Running Instances of Activities . . . . .	43
5.5	Faults and their Propagation . . . . .	43
5.5.1	Throwing Faults . . . . .	44
5.5.2	Hierarchially Propagating Faults . . . . .	44
5.5.3	The Stop Concept . . . . .	45
5.5.4	Stopping Activities . . . . .	46
5.6	Dead Path Elimination . . . . .	46
5.7	Termination of a Process Instance . . . . .	47
5.8	Time . . . . .	48
5.9	Pattern for the ASM rules for Activities . . . . .	49
5.10	Summary & Conclusion . . . . .	50
<b>6</b>	<b>Communication</b>	<b>51</b>
6.1	Communication Structure . . . . .	51
6.1.1	External Interface . . . . .	51
6.1.2	Internal Interface . . . . .	52
6.2	Messages . . . . .	53
6.3	Values of Variables . . . . .	54
6.4	Correlating Messages to Process Instances . . . . .	55
6.4.1	Message Properties . . . . .	55
6.4.2	Correlation Handling . . . . .	56
6.4.3	Services and Endpoint References . . . . .	57
6.4.4	Message Addressing with WS-Addressing . . . . .	58
6.5	Communication Endpoints . . . . .	59
6.5.1	Linking Internal and External Interface . . . . .	60
6.5.2	Communication Endpoints for Activities . . . . .	60
6.5.3	ASM-Rules to Send and Receive Messages . . . . .	61
	Receiving Messages . . . . .	61
	Sending Messages . . . . .	62
6.6	Inbox-, Outbox- and Instance-Manager . . . . .	63
6.6.1	Inbox-Manager . . . . .	64
6.6.2	Outbox-Manager . . . . .	65
6.6.3	Instance-Manager . . . . .	66
<b>7</b>	<b>Basic Activities</b>	<b>68</b>
7.1	Basic Activity Rule . . . . .	68
7.1.1	Starting, Completing and Stopping Basic Activities . . . . .	69
7.1.2	Terminating Process Instances: Basic Activities . . . . .	69
7.1.3	Basic Activity Rule – Final Refinement . . . . .	69
7.2	Empty . . . . .	70

7.3	Wait . . . . .	71
7.3.1	Starting Wait . . . . .	71
7.3.2	Running Wait . . . . .	72
7.3.3	Wait – Final Refinement . . . . .	72
7.4	Throw . . . . .	73
7.4.1	Running Throw . . . . .	73
7.4.2	Completing Throw . . . . .	74
7.4.3	Throw – Final Refinement . . . . .	74
7.5	Terminate . . . . .	75
7.5.1	Running Terminate . . . . .	75
7.5.2	Terminate – Final Refinement . . . . .	75
7.6	Receive . . . . .	75
7.6.1	Running Receive . . . . .	76
7.6.2	Receive – Final Refinement . . . . .	77
7.7	Reply . . . . .	77
7.7.1	Running Reply . . . . .	78
7.7.2	Receive – Final Refinement . . . . .	80
7.8	Invoke . . . . .	80
7.8.1	Running Invoke . . . . .	82
	Invoking a remote Operation . . . . .	82
	Receiving the Reply of a remote Operation . . . . .	83
7.8.2	Invoke – Final Refinement . . . . .	84
7.9	Assign . . . . .	84
7.9.1	Running Assign . . . . .	85
7.9.2	Assign – Final Refinement . . . . .	86
<b>8</b>	<b>Structured Activities</b>	<b>87</b>
8.1	Structured Activity Rule . . . . .	87
8.1.1	Starting a Structured Activity . . . . .	88
8.1.2	Running a Structured Activity . . . . .	88
8.1.3	Completing a Structured Activity . . . . .	89
8.1.4	Stopping a Structured Activity . . . . .	89
8.1.5	Terminating Process Instances: Structured Activities . . . . .	90
8.1.6	Structured Activity Rule – Final Refinement . . . . .	90
8.2	Flow . . . . .	91
8.2.1	Starting Flow . . . . .	91
8.2.2	Running Flow . . . . .	92
8.2.3	Flow – Final Refinement . . . . .	92
8.3	Sequence . . . . .	92
8.3.1	Starting Sequence . . . . .	93
8.3.2	Running Sequence . . . . .	94
8.3.3	Sequence – Final Refinement . . . . .	95
8.4	Switch . . . . .	95
8.4.1	Starting Switch . . . . .	96
8.4.2	Running Switch . . . . .	97

---

8.4.3	Switch – Final Refinement . . . . .	97
8.5	Pick . . . . .	97
8.5.1	Starting Pick . . . . .	98
8.5.2	Running Pick . . . . .	100
	Pick – Select the first Event . . . . .	100
	Pick – Run the chosen Branch . . . . .	102
8.5.3	Pick – Final Refinement . . . . .	103
8.6	While . . . . .	103
8.6.1	Starting While . . . . .	103
8.6.2	Running While . . . . .	103
	Starting an Execution of the Body . . . . .	104
	Completing the Execution of the Body . . . . .	105
8.6.3	While – Final Refinement . . . . .	105
<b>9</b>	<b>The Scope, Fault Handling &amp; Event Handling</b>	<b>106</b>
9.1	Scope – General Architecture . . . . .	106
9.1.1	Handling <i>signalStop</i> . . . . .	109
9.1.2	Organizing Flow of Faults . . . . .	110
9.2	Scope – Positive Control-Flow . . . . .	111
9.2.1	Initiate Stopping of a Scope in mode <i>positive</i> . . . . .	112
9.2.2	Starting a Scope in mode <i>positive</i> . . . . .	112
9.2.3	Running a Scope in mode <i>positive</i> . . . . .	113
9.2.4	Stopping a Scope in mode <i>positive</i> . . . . .	113
9.2.5	Completing a Scope in mode <i>positive</i> . . . . .	114
9.2.6	Scope in mode <i>positive</i> – Final Refinement . . . . .	116
9.3	Fault Handler . . . . .	117
9.3.1	Starting a Fault Handler . . . . .	117
9.3.2	Running a Fault Handler . . . . .	118
	Catching a Fault . . . . .	118
	Stopping Positive Control Flow of the FaultHandler’s Scope . . . . .	119
	Handling the Caught Fault . . . . .	120
9.3.3	Fault Handler – Final Refinement . . . . .	121
9.4	Scope – Negative Control-Flow . . . . .	122
9.4.1	Starting a Scope in mode <i>negative</i> . . . . .	123
9.4.2	Running a Scope in mode <i>negative</i> . . . . .	123
9.4.3	Completing a Scope in mode <i>negative</i> . . . . .	123
9.4.4	Stopping a Scope in mode <i>negative</i> . . . . .	124
9.5	Event Handler . . . . .	124
9.5.1	Dead Path Elimination and Event Handlers . . . . .	125
9.5.2	Executing an Event Handler . . . . .	125
9.6	OnAlarm Event Handler . . . . .	126
9.6.1	Starting an OnAlarm Event Handler . . . . .	126
9.6.2	Running an OnAlarm Event Handler . . . . .	126
	Handling <i>onAlarm</i> events . . . . .	127
	Finishing the Handling of <i>onAlarm</i> events . . . . .	128

---

9.6.3	OnAlarm Event Handler – Final Refinement . . . . .	128
9.7	OnMessage Event Handler . . . . .	128
9.7.1	Starting an OnMessage Event Handler . . . . .	128
9.7.2	Running an OnMessage Event Handler . . . . .	129
	Handling <i>onMsg</i> events . . . . .	129
	Finishing the Handling of <i>onAlarm</i> events . . . . .	130
9.7.3	Completing an OnMessage Event Handler . . . . .	131
9.7.4	OnMessage Event Handler – Final Refinement . . . . .	131
<b>10</b>	<b>Compensation Handling</b>	<b>132</b>
10.1	Abstract Semantics for Compensation Handling . . . . .	132
10.1.1	Required Shared Structures . . . . .	133
10.1.2	Compensation Handling in Scopes . . . . .	133
	Preparing a Scope for Compensation . . . . .	133
	Behaviour in the Case of Compensation . . . . .	134
10.1.3	Compensation Handler . . . . .	136
10.1.4	Compensate . . . . .	136
	Starting Compensate . . . . .	137
	Running Compensate . . . . .	138
	Completing Compensate . . . . .	139
	Stopping Compensate . . . . .	139
10.2	Structures for Compensation Handling . . . . .	139
10.2.1	Activity-to-Activity Communication for Compensation Handling . . . . .	139
10.2.2	The Compensation Stack . . . . .	141
10.3	Compensation Handling in Scopes . . . . .	141
10.3.1	Preparing a Scope for Compensation . . . . .	141
10.3.2	Behaviour in the Case of Compensation . . . . .	142
10.3.3	Integrating Compensation Handling into the Scope . . . . .	143
10.4	Compensate . . . . .	144
10.4.1	The Private Compensation Stack . . . . .	144
10.4.2	Starting Compensate . . . . .	146
10.4.3	Running Compensate . . . . .	146
10.4.4	Compensate – Final Refinement . . . . .	147
10.5	Compensation Handling at the Process Level by <code>enableInstanceCompensation="yes"</code> 148	
<b>11</b>	<b>Summary &amp; Conclusion</b>	<b>152</b>
<b>A</b>	<b>Abstract-State Machines</b>	<b>156</b>
A.1	Specific Rules for the BPEL semantics . . . . .	156
A.1.1	Sets . . . . .	156
A.1.2	State Transitions (of finite state machines) . . . . .	156
A.1.3	Asynchronous message passing via determined channel . . . . .	156

---

<b>B</b>	<b>ASM-Rules for BPEL</b>	<b>158</b>
B.1	Rules and Functions for the Static Structure of a BPEL-Process . . . . .	158
B.1.1	Activity Tree . . . . .	158
B.2	Rules and Functions for the Dynamic Structure of a BPEL-Process . . . . .	159
B.2.1	Process Managers' ASM Rules . . . . .	159
B.2.2	Activities' ASM Rule . . . . .	160
B.2.3	The Subinstance Tree . . . . .	162
B.2.4	Links . . . . .	162
B.3	Rules and Functions for Communication . . . . .	163
B.3.1	Correlation Handling . . . . .	163
B.3.2	Receiving and Sending Messages . . . . .	165
B.4	Rules and Functions for Basic Activities . . . . .	166
B.4.1	Empty . . . . .	166
B.4.2	Wait . . . . .	166
B.4.3	Throw . . . . .	167
B.4.4	Terminate . . . . .	167
B.4.5	Receive . . . . .	168
B.4.6	Reply . . . . .	169
B.4.7	Invoke . . . . .	171
B.4.8	Assign . . . . .	175
B.5	Rules and Functions for Structured Activities . . . . .	179
B.5.1	Structured Activities . . . . .	179
B.5.2	Flow . . . . .	179
B.5.3	Sequence . . . . .	180
B.5.4	Switch . . . . .	181
B.5.5	Pick . . . . .	183
B.5.6	While . . . . .	186
B.6	Rules and Functions for the Scope and the Handlers . . . . .	188
B.6.1	Scope . . . . .	188
	Positive Control Flow . . . . .	188
	Negative Control Flow . . . . .	191
B.6.2	Fault Handler . . . . .	192
B.6.3	Event Handler . . . . .	196
B.6.4	OnAlarm Event Handler . . . . .	196
B.6.5	OnMessage Event Handler . . . . .	198
B.7	Rules and Functions for the Compensation Handling Mechanism . . . . .	201
B.7.1	Scope supporting Compensation Handling . . . . .	201
	Preparing Scope for Compensation . . . . .	201
	Behaviour in the Case of Compensation . . . . .	201
B.7.2	Compensation Handler . . . . .	203
B.7.3	Compensate . . . . .	203
	<b>Bibliography</b>	<b>207</b>

---



# 1 Introduction

In this report, we define abstract operational semantics for the *Web Service Business Process Execution Language* (WS-BPEL or BPEL for short) using the *Abstract State Machines* (ASM) formalism.

BPEL (in its version 1.1) is an XML based web service orchestration language proposed as an industry standard by BEA, IBM Microsoft, SAP and Siebel. The language is designed to define abstract and executable business processes, each defining an order on communicative interaction with some partners. Interaction follows the approach of *Service Oriented Architecture* [EAA<sup>+</sup>04] and the *Web Service* paradigm [Kre01]. Consequently, BPEL constitutes an emerged layer of a technology stack implementing this kind of software architecture. By the time this report is written, BPEL is in the process of standardization by an OASIS technical committee [WT04].

The architecture of BPEL is a result of combining Microsoft's *XLANG* [Tha01] and IBM's *Web Service Flow Language* (WSFL) [Ley01] and adding the concepts of fault handling and compensation for encapsulated parts of a BPEL process. Given a precise XML based syntax, BPEL's semantics is specified using natural language in a working draft of approx. 130 pages to which we refer as "informal specification" [CGK<sup>+</sup>03]. The semantics described therein consists of behavioral descriptions of specific entities of a process combined with general requirements on large parts of, or even the entire, process. The use of natural language gives rise to ambiguous interpretations. The intermixed description of behaviour and requirements cannot prevent an inconsistent definition of BPEL's semantics and makes the informal specification hard to read. The WSBPEL technical committee states that

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.<sup>1</sup>

The aim of this work is to provide a complete formal model of BPEL's architecture and semantics using a mathematically founded formalism. Applying the mathematical precision in defining structures and behaviour aides in finding and resolving ambiguous, inconsistent or even invalid aspects of the language's semantics. It makes the semantics available for formal mathematical deduction. An implementation of the language can be checked for conformance with the specification by the help of model based testing methods [FGV05].

We propose to use the ASM formalism to solve this problem. The formalism has successfully been applied in giving programming languages formal semantics. The most prominent works are on SDL-2000 [EGG<sup>+</sup>01], on VHDL'93 [BGM95], and on Java and the Java Virtual Machine [SSB01]. The crucial requirement of the given task is that the formal model we seek to create has to represent the informal specification *as is*. We need to define structures and behaviour on the level of abstraction as they are described in the informal specification.

---

<sup>1</sup>WS BPEL issue list, issue # 42 [WT04].

This way, one can easily check whether the formal model of BPEL conforms the informal specification by comparatively reading both. Such a first formal model, is called an ASM *ground model* [Bo95]. It may serve as the starting point for refinement towards an implementation as well as for verification or experimental validation using an executable version of ASM like AsmL [Asmb] or AsmGofer [ASMa].

### 1.1 Introduction to WS-BPEL

A BPEL process describes the interaction of one web service with its partners from its point of view. From the technical point of view, each executable business process is a stateful web service which presents itself to other web services as an abstract WSDL service.

More detailed, a BPEL *process* declares a number of *partners* to communicate with, an *activity* which defines the logic behind the interaction with its partners and *variables* that are required to model a stateful interaction. Activities are the building blocks of the behaviour of a BPEL process. They are used to describe control flow and data flow. An activity may be either *basic* or *structured*.

A basic activity describes a work item which cannot sensibly be subdivided: **receive** a message; **reply** to a previously received message; synchronously or asynchronously **invoke** the exchange of messages with a partner; **wait**; manipulate data (**assign**); initiate fault handling by explicitly **throwing** a fault; **terminate** the execution of the process; and do nothing (**empty**).

A structured activity puts work items into a specific order. A structured activity *encloses* at least one *inner* activity which may be structured again. Thus they are used to define the control flow: sequential execution of the inner activities (**sequence**); concurrent execution of the inner activities (**flow**), where the order of execution within a flow may be restricted by **links**; conditional, repeated execution of an activity (**while**); conditional branching of the control flow (**switch**); waiting for an event (receiving a message or satisfying a time-out condition) and proceeding based on the event (**pick**); and *encapsulating* a part of the process regarding control and data flow (**scope**). The scope allows the definition of special structured activities, called *handlers* implementing mechanisms to handle faults (**faultHandlers**), to undo the work previously performed in a part of the process (**compensationHandler**), and to react on events (messages or time-outs) which might occur concurrently to the execution of the encapsulated activities (**eventHandlers**).

A BPEL process is executed in *instances*. The declaration of the elements of the process serves as a template for each instance. An instance is always created due to an inbound message for which no running instance is existing yet. The instances are clearly separated by their states which is (among others) constituted by the state of the control flow and the values of the declared variables. Since a BPEL process is a stateless WSDL service from the point of view of its environment, inbound messages are associated to the correct process instance by the help of BPEL's genuine *correlation handling mechanism* and the use of *endpoint references*<sup>2</sup>.

Manipulating data on a finer level than WSDL-specific datatypes, though being required for an executable process, is not considered to be part of the language. Similarly, com-

---

<sup>2</sup>see WS-Addressing, [CB<sup>+</sup>03, 2]

munication is expressed in terms of an WSDL-interface, abstracting from any operational behaviour. Instead, a BPEL implementation is assumed to work on top of a technology stack of which the underlying layers provide the required functionality. [EAA<sup>+</sup>04, Chapter 5]

## 1.2 Existing Work

Providing formal semantics for BPEL is subject of research in many works. There are four major formalism which were successfully applied: finite state machines (FSM), process algebras (PA), Petri nets (PN), and Abstract State Machines (ASM). A great number of works therein aims on verifying specific properties of a BPEL process.

The principle approach is to translate a given BPEL process into a FSM, a process algebraic expression, or a Petri net considering just the semantical elements which are relevant for the property to be verified. Then, analysis techniques can be applied to the formal representation of the process.

Koshkina and v. Breugel translate a given BPEL process into a process algebraic expression in order to verify its control flow. The work abstracts from data-related aspects [KB03]. Based on this work, they provide an in-depth-analysis of BPEL's Dead-Path-Elimination by formal means [BK05].

Ferrara follows a similar approach by translating to LOTOS, but is more detailed in his formalization. The work skips the aspects related to the instantiation of BPEL processes, namely correlation handling and endpoint references [Fer04a], [Fer04b].

Butler et al focus on the analysis of transactional behaviour of BPEL processes, specifically the compensation handling mechanism. A given process is translated into an expression of a special process algebra designed to suit the problem [BFN05].

Fu et al verify the composition of a number of web services written in BPEL by translating each of them into a guarded finite state machine. Upon formal composition, the entire system can be model checked for relevant properties. This approach includes the translation of control flow, communication and the translation of data flow at an abstract level. Advanced mechanisms like event handling or compensation handling are not considered [FBS04].

The work of Stahl proposes a pattern-based translation of activities into Petri nets. He provides a (parameterized) pattern for each BPEL activity. A given BPEL process is translated into a large Petri net by translating each activity to the according pattern. The semantics are complete regarding the control flow. But the Petri net approach fails to model the handling of data at the level which is given by the informal specification. The approach was first published in [VBS04], the full semantics are given in [Sta05].

The works of Ouyang et al follow the approach of Stahl and define Petri net patterns, the composition of which yields a sound Petri net. The model captures control-flow but likewise skips a detailed definition of manipulation of data. [OAB<sup>+</sup>05].

The Petri net being the result of the translation can then be checked for relevant properties by static analysis on the structure of the net as well as by model checking techniques [SS04]. Both Petri net approaches lack the ability to describe the instantiation of a BPEL process or the concurrent handling of several occurrences of the same on message event.

The list of works mentioned so far is not complete but shows the principle approach in

defining formal semantics for BPEL: the domain of the translation is an expression in the chosen formalism. The formally defined process shall exhibit the same behaviour as the BPEL process as it is described by the informal specification.

Farahbod, Glässer, and Vajihollahi at Simon Fraser University (SFU), Canada, take a different approach. The behaviour of each construct of the language is formally represented in an ASM, a specific BPEL process is defined in the initial state of the ASM. Since the ASM formalism allows to describe structures at any level of detail, their semantics captures the modular architecture of the language consisting of the core and extensions for data handling and fault and compensation handling. In [FGV05] the formal model captures control-flow and data-flow, and expresses instantiation and communication at an abstract level. If one provides a formal interpretation of the few informally specified functions<sup>3</sup> in the model, it is executable. The semantics are incomplete as Dead-Path-Elimination, Correlation Handling, and Event Handling are not covered completely. Furthermore, the informal specification of BPEL allows for a more detailed definition of the data-handling than given in said semantics. The principle approach for an ASM ground model for BPEL was published first in [FGV04b].

### 1.3 Aim of this Work

In this work we propose a slightly different architecture of an ASM model for BPEL and we will formalize the open aspects of the formal ASM model for BPEL from SFU. For the first time, we will provide a clear formal representation of BPEL's static structures in relation to their dynamic properties and to the behaviour of a process instance. We follow and extend the approach of Farahbod et al in order to address the unspecified aspects of the language.

By the clear distinction of static and dynamic aspects of the language, we propose a robust model of BPEL. This robustness against changes and extensions of the language itself is necessary since BPEL is still in the process of standardization and thus subject of change.

The model which we propose in this report does not provide all means for an executable model as there are aspects in the informal specification of BPEL which do not cover any operational behaviour which is required for this aim. We hope that in combination with the model of SFU, together we can come up with a complete, and executable formal model of BPEL. Thus this report serves as basis for a joint work with the group of SFU.

Additionally, this work provides a completely new approach in writing down semantics of BPEL by putting the informal description of the behaviour and their formal definition side by side. We claim that in this way, the structure and the semantics of the language are easy to understand whilst having mathematically reliable formal definition at hand. Furthermore, we claim that this document provides the first complete, formal operational semantics of BPEL related to its activities.

This report is organized as follows. In Chapter 2 we give a brief introduction to the ASM modelling paradigm and the ASM model we are using. Readers familiar with these topics may skip to Section 2.3 where we explain how to read this report. In the subsequent

---

<sup>3</sup>The informal specification of WS-BPEL v1.1 does not allow for a full formal definition of each function as these describe the usage of technologies which are not part of the language.

three chapters we explain and formalize the general architecture of BPEL and the behaviour which is employed in more than one occasion: In Chapter 4 we formalize general aspects of a BPEL process definition. The behaviour of a BPEL process in terms of the control flow, together with BPEL's mechanisms which influence the control flow of each activity is formally defined in Chapter 5. All aspects of communication are defined in Chapter 6. The last four chapters serve for the detailed definition of the structures and behaviour of BPEL's activities. Chapter 7 covers the basic activities. In Chapter 8 the structured activities are defined. Chapter 9 is dedicated to the scope, while we define the compensation handling mechanism in Chapter 10. A conclusion and a summary is given in Chapter 11.

## 2 The ASM model

In this chapter we explain the ASM modelling paradigm and its application to our problem of creating formal semantics for BPEL. We will shed some light on the nature of our problem and informally recall the Abstract-State Machine formalism. Readers new to the formalism, or readers who interested in a complete formal introduction to the ASMs are referred to [Gur95] or [Gur97], and [BS03].

### 2.1 Ground Models

The difficulty of our problem lies in the fact that we are providing the very first step from an informal description of a system towards its formal model. This step has to bridge the gap between ambiguously formulated requirements and possibly inconsistent sentences in a natural language, and an expression in a mathematical formalism which must be consistent and unambiguous. Additionally, the formal model and the informal description must describe the same system to the full extent that can be achieved.

The correctness of this very first step can be verified only by comparing the formal model and the informal description – assuming that the informal description holds the correct information in case of doubt. Verification of such a nature is simplest if the objects which are described informally and formally are just the same. That is, if the level of abstraction of the formal model is just the same as the level of abstraction provided by the informal description. A model defined for such a purpose and having these properties is what we call a *ground model*. [Bo95].

The ground model includes functional requirements which can be derived directly from the informal description as well as design decision made within the space of informality that is provided by the original description of the system. The whole model is considered to be valid if it exhibits behaviour that is allowed by the informal description only. The model is complete if it captures all behaviour which is described informally. Design decisions have to be made such that the validity of the model can be verified by objective criteria such as falsifiability and testability [FGV05].

### 2.2 Abstract-State Machines

The ASM formalism provides the means to create a ground model with the properties mentioned in the previous section [Bo95]. In this section we will informally recall the essential aspects of ASM and some specialities of which we will make use. Readers which are new to ASM should consider to be introduced to the formalism by [HW02], [Gur97], or [BS03].

At a first glance, any Abstract-State Machine specifies a pseudo-code program of which the variables and functions are built of *inductively defined terms*. But this notation has a rigorous mathematical definition: Each ASM describes a transition system, where a *state* of the system is a  $\Sigma$ -algebra.

By using a  $\Sigma$ -*algebra* as a state, its functions denote the structural relations of the system in a given moment of time. This first-order structure imposes no limitations upon the definition of functions, having a continuous domain or a discrete, being computable or not. Hence, we are free to choose the level of detail of the structures in our system, we need and desire.

Where functions are purely semantical objects, the algebra's signature  $\Sigma$  is used to denote them syntactically: A term  $f(t_1, \dots, t_n)$  of the program is inductively built over the state's signature  $\Sigma$ , and is *interpreted* in a state. We interpret a symbol of the term by mapping it to a function or a constant of the state. Through interpretation we can inductively compute the value of any term in a program. By convention, the possible values in a state are typed by a partition of the carrier of the state.

Yet, "value" is an abstract thing within the ASM formalism. We are not interested in specific values, but whether any two terms are interpreted by the same value. We formalize such a property syntactically using formulas in first-order logic ( $f(t_1, \dots, t_n) = g(s_1, \dots, s_m)$ ). Therefore, we reduce questions about structural relations to the question whether two terms are interpreted equally. From thereon, we describe changes in the state of the system syntactically by *assignments* to terms  $f(t_1, \dots, t_n) := g(s_1, \dots, s_m)$ . Assignments and their parallel, conditional, and quantified composition are denoted in *ASM rules*.

But behind each symbol of a term, there is a rigorously defined mathematical function which precisely formalizes the structural relations in a state. Each assignment denotes an equally precise re-definition of the function for a given argument. By using interpretable terms, ASMs put a focus on which objects of the system are related to which objects and how their relations change.

Upon firing an *ASM rule* its assignments are evaluated such that the corresponding functions of the state are redefined. This constitutes the *step* to the successor state. The *run* of ASM is a sequence of states, where any two subsequent states are related by a step and the first state being a designated *initial* state.

For an initial state of an ASM, a valid interpretation of all symbols of the signature  $\Sigma$  must be given. Symbols of which the interpretation never changes in any step denote *static* functions. Symbols where the interpretation may change from one state to another denote *dynamic* functions. Relations are denoted as boolean-valued functions. By convention, in case the interpretation of a term is not known it is interpreted equally to the dedicated nullary function symbol *undef*.

In the course of this report, we will make use of three further concepts of the ASM formalism. We explain each of them briefly.

### 2.2.1 Derived Functions and Abstract Functions

In many cases, complex structural relations in a state can regularly be expressed by composing functions which formalize simple relations of the system. Formally, we introduce a

new function symbol, and we provide an algebraic specification of its interpretation using the symbols of the simpler functions. The new symbol can be interpreted by interpreting previously defined symbols. We call such a complex function a *derived* function.

Up to now, we considered the interpretation of a symbol to be a rigorously defined mathematical object. In some cases of a real system, such a rigorous interpretation cannot be given. This is mostly the case when specifying interfaces, behaviour of the system's environment, or non-functional requirements. In either of these cases we cannot or we don't want to provide the definition of a function and its changes in a run. Still said aspects are present and need a formal representation.

The ASM formalism knows the concept of the *abstract* function. An abstract function is a first-class citizen of a state and hence the interpretation of a symbol which may be used in any term of the ASM. But we provide an abstract, informal description of its definition only. Evaluation of a logical formula which includes the symbol of an abstract function is subject to the informal description of its definition. We allow dynamic, abstract functions where the redefinition of the function takes place due to a step performed by the environment.

### 2.2.2 Creating New Elements

Given an interpretation of the signature of an ASM in an initial state, we cannot reach a new element of the carrier by the means introduced so far; i.e. we cannot denote a term  $t$  for which the equation  $t = t'$  evaluates to *false* for any term  $t'$ .

But reaching new elements is necessary to introduce unique new objects to the system (like a new message that was never sent before). Formally, the **new** operator serves this purpose. For any type  $Type$  provided by the carrier, and for any term  $t$  over the ASM's signature  $\Sigma$ , the equation  $t = \mathbf{new}(Type)$  evaluates to *false*. Hence by an assignment  $t := \mathbf{new}(Type)$ , we get access to a completely new element.

### 2.2.3 Agents

Programs written in BPEL model distributed, reactive systems. The distributivity implies the existence of autonomously acting "sites" or "nodes", which are connected in some way. We will formalize these nodes as *agents*. From the reactivity we may conclude that each agent must be able to adapt its behaviour to the current state of its environment.

There is an informal characterization of agents for distributed ASMs which suffices our requirements. An agent, informally speaking, constitutes an independent, autonomously acting system. An *ASM agent* is an agent with a unique name  $a$  and an ASM rule  $R = \mathit{agentRule}(a)$  where  $R$  may contain an uninterpreted symbol **self** which is not part of the ASMs signature.

We write *Agents* for the set of all agent names. A *distributed ASM* has a signature  $\Sigma$ , a set of initial states and an initially non-empty, finite set of ASM agents. The set of agents in a distributed ASM is dynamic: agents may be created or removed from the ASM by defining  $\mathit{agentRule}(\cdot)$  accordingly.

All agents share the same global state of the distributed ASM they exist in. Whenever an agent with name  $a$  "decides" to make a step, it fires its rule  $\mathit{agentRule}(a)$  in the current

global state, where the symbol **self** in the rule is interpreted as  $a$ . We call a step of an agent a *move* [Gur95].

By the notion of global state, a distributed ASM makes a *step*, when a subset of its agents makes a *move*, where the rules of the involved agents are fired in parallel. Then, informally characterized, a sequence of states  $S_0 S_1 \dots$  is a *run* of a distributed ASM if  $S_0$  is an initial state of the ASM and for any state  $S_i$  its successor state  $S_{i+1}$  in this sequence is reached by a step of the ASM and the set of all moves in the run suffices the requirements of a *partially ordered run* [Gur95].

Introducing the name of an agent into its rule allows us to use functions with domains involving the name of the agents. In effect, this leads to some sort of “local knowledge” of the agent about the system’s state: an agent can alter just those locations of a state which can be reached by interpreting terms over function symbols of the ASMs signature and the dedicated symbol **self**.

Furthermore, the definition of the distributed ASM allows two different agents in the system which have the same rule  $R$ . Due to the different interpretation of the symbol **self** upon firing  $R$ , each agent may modify a different part of the global state. Yet, both agents execute the same behaviour.

## 2.3 How to Read this Document

In the following chapters we will make use of some notational conventions.

The ASM approach represents types as distinct subsets of the carrier of a state. Each such subset gets a dedicated symbol to identify types by the element relation  $\in$ . We denote the declaration of a new type, i.e. the declaration of a symbol which is to be interpreted as a subset of the carrier disjoint from any other type by

**Universe:** *Type*.

Function symbols which will be used throughout the ASM rules are declared together with the symbols of their types. In any interpretation of this symbol, its arguments and its values must be of the domain of the respective type symbols. We use  $\mathcal{P}(Set)$  to denote the powerset of *Set*.

static functions (2.3.0-1)

$$\begin{aligned} \mathit{function} & : Type_1 \times \dots \times Type_n \rightarrow Type'_1 \times \dots \times Type'_m \\ \mathit{setFunction} & : Type_1 \times \dots \times Type_n \rightarrow \mathcal{P}(Type'_1 \times \dots \times Type'_m) \end{aligned}$$

Symbols which are part of the signature of the ASM are written in italics (*function*). Symbols of variables or parameters of ASM rules which need to be bound to an element of the carrier are written sans serif (**variable**). We will write  $\mathcal{A}$  in front of the declaration of an abstract function and  $\mathcal{D}$  in front of a derived function, respectively (c.f. section 2.2.1).

In addition to the declaration of a function symbol, we may require some restrictions to the allowed interpretations in the initial state of the ASM. Such a requirement is denoted by an expression in first order logic.

requirement for the initial state (2.3.0-1)

$$\forall \text{var} \in \text{Type} : \dots$$

Operators to construct ASM rules, such as **if** are written in bold roman font. An ASM rule which describes a logically corresponding block of behaviour gets a name. The rule may have parameters which must be properly evaluated if the ASM rule is referenced. The definition of an ASM rule will be written as follows:

$$\begin{array}{l} \underline{\text{RULENAME}} \\ (\text{param}_1 \in \text{Type}_1, \dots, \text{param}_n \in \text{Type}_n) \equiv \\ \text{rule body} \end{array} \quad (\text{R2.3.0-1})$$

referenced by RULENAMEB (R2.3.0-2)

The list of the free parameters of the rule body is given in parentheses right after the underlined rule name. In the body of the ASM rule, free parameters are conceived as free variables and may be part of any term defined therein.

$$\text{function}(\text{param}_1, \dots)$$

Each ASM rule gets a unique number which will be printed occasionally when referring to a specific ASM rule, e.g. RULENAME (R2.3.0-1), (R *Chapter.Section.Subsection-# of Rule in Subsection*). This number can be used to find the place of definition of an ASM rule in this document. The electronic .pdf version will supply a hyperlink at these locations which link to the definition of the rule. The same naming scheme is applied for the declaration of functions and for requirements for the initial state.

Sometimes an ASM rule which is already defined might be quoted, mostly for the purpose of refining this rule:

$$\begin{array}{l} \underline{\text{RULENAME}} \\ (\text{param}_1 \in \text{Type}_1, \dots, \text{param}_n \in \text{Type}_n) \equiv \\ \text{rule body} \end{array} \quad (\text{R2.3.0-1})$$

For composed ASM rules which have parameterized parts within, we use an inline-parameterized notation, where the name of the ASM rule and its parameters are connected by “keywords” written in bold roman fonts, see Appendix A.

An ASM rule which was defined as just described, may be referenced in any other ASM rule by writing its name, and, if applicable, by providing the corresponding parameters.

$$\begin{array}{l} \underline{\text{RULENAMEB}} \\ (\text{param}_1 \in \text{Type}_1, \dots, \text{param}_n \in \text{Type}_n) \equiv \\ \dots \\ \text{RULENAME}(\text{param}_1, \dots, \text{param}_n) \end{array} \quad (\text{R2.3.0-2})$$

referencing RULENAME (R2.3.0-1)

Please note that typing of the parameters is for the reader's convenience only. A type-mismatch in any of the rule's arguments has no semantical consequences but might point to an error in the formalization.

In BPEL the behaviour of several different constructs just differs in minor details. Throughout the following chapters, we will define the most abstract behaviour at first and present specific refinement steps to more detailed ASM rules. In most cases, it is not necessary to give a full definition of the refined ASM rule at place. Instead, its definition is placed in the Appendix.

In the Appendix, *all* ASM rules of this document are either defined for the first time, or their definition is quoted. Hence, the experienced reader may read the entire semantics by just looking at the Appendix.

## 3 Overall Architecture and Approach to Formalization

In this chapter we present a rough sketch of the overall architecture of our model of BPEL and explain our approach of defining the complete semantics of the language.<sup>1</sup>

### 3.1 Basic Thoughts about the Architecture of BPEL

#### 3.1.1 Activities

To justify our approach of formalizing BPEL, we present some observations about the architecture of the language. We aim on representing this architecture in our model as well.

The activities of BPEL which we have introduced in section 1.1 describe concepts of communication, control-flow and manipulation of data. Each concept describes the structures and the inherent behaviour that an activity of a certain type exhibits in any BPEL process. We will call these concepts *activity concepts*. The BPEL specification separates each of these concepts regarding their functionality. The structure and the behaviour of an activity concept is defined independently of the other concepts. We will sustain this distinction in the formal model.

Obviously, the concepts cannot describe isolated entities since a BPEL process consists of several activities, where the execution of one activity depends on the execution of other activities (e.g. concurrent execution due to a **flow** activity). From a conceptual point of view, this interaction follows a general pattern which we will conceive as an *interface* that is provided by each activity concept, and that is accessible by each activity concept. Activities communicate via these interfaces using a common protocol. This abstract view on BPEL's conceptual structure is illustrated in figure 3.1.

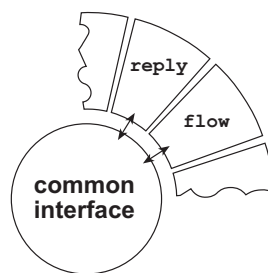


Figure 3.1: Conceptual structure of BPEL

<sup>1</sup>Many thanks to Roozbeh Farahbod and the group at SFU for pointing me on the lack of this chapter.

In the informal specification [CGK<sup>+</sup>03], the activity concepts are defined in a way such that they describe the behaviour of the corresponding activity in any possible situation. When defining a BPEL process, we do not use activity concepts to build the definition, but we use *concrete occurrences* of them. A concrete occurrence of the concept in a process is what we usually refer to as *activity* and it is always given in a concrete process definition. An activity is distinct from its concept in the way that it has specific properties: the **reply** concept just describes the structures by which the recipient of a message and the contents of the message are characterized. A **reply** activity in a concrete BPEL process defines *who* receives the message and *where* to get the contents from. BPEL processes are programs, the concepts define the structures and rules by which a process is executed!

To actually execute the process, runtime properties, such as variable values, are required. Allowing multiple concurrent executions, a BPEL process is executed in a process *instance* which basically consists of *instances of those activities* given by the process definition: The **reply** activity just knows where it generates the contents of its message from, the instance of the **reply** activity has actual values which constitute the actual message. Just like there may be several activities of the same concept in one process definition, we may see several instances of the same activity in one instance of this process (which is due to the behaviour of the on message event handler).

Any two instances of the same activity share the same structural properties. Any two occurrences of the same activity concept exhibit the same principle behaviour. All possible behaviour of an instance of an activity is given by its concept. Its actual behavior is determined by its properties given by the process definition and its runtime values. Figure 3.2 illustrates this.

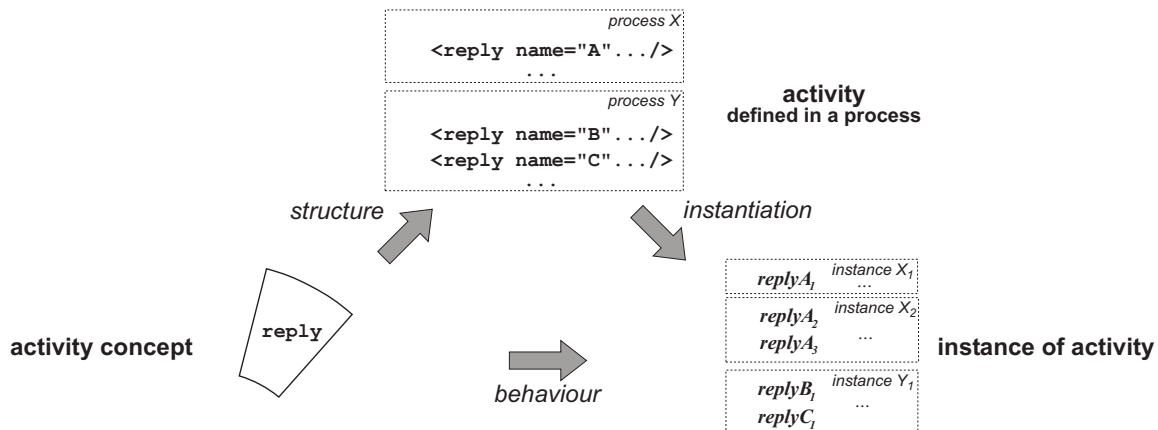


Figure 3.2: Relation of an activity, activity concept and instance of activity

It is exactly this constellation we aim to capture in our formal semantics. Hence we will distinguish concepts of activities, activities and instances of activities formally.

### 3.1.2 Messages and Instantiation

In the previous section we just explained the architecture of BPEL with respect to activities. We did not consider how process instances are created and how messages reach and leave a process. The activity concepts do not describe either of it. This immediately implies the existence of another mechanism in BPEL.

The execution of a process commences if an activity which is capable of receiving a message at first hand (i.e. `receive` and `pick`) and which has the designated property `createInstance` set to `true` receives a message where that message does not match to any other running process instance. [CGK<sup>+</sup>03, Section 6.4]. This definition is circular as it presupposes that a part of the process instance (the instance of the receiving activity) is running to cause the instantiation of the process.

To break the circularity, we have to assume the existence of an entity that is capable of accepting messages even if no process instance is running and which causes the creation of a new instance if it is required. We will call this entity “inbox manager” to conform with the model of Farahbod et al. [FGV05].

### 3.1.3 Extensions to BPEL

The BPEL’s informal specification [CGK<sup>+</sup>03] introduces core features and extensions. The core features contain the definition of activity concepts and other entities which are necessary “for expressing abstract and executable business processes.” [CGK<sup>+</sup>03, Section 6.3]. Extensions are defined to provide further functionality which might not be necessary in every BPEL process.<sup>2</sup>

In BPEL v1.1 [CGK<sup>+</sup>03] two “standard” extensions are included: “extensions for executable business processes” and “extensions for business protocols”. These extensions add minor additional requirements to give more precise semantics to the language where the core’s definition are insufficient for either purpose.

## 3.2 Formalizing the Architecture of BPEL

We will now sketch our approach of formalizing BPEL while sustaining its architecture. We will take a straight forward approach and formalize each activity concept on its own. Furthermore, we need some sort of “inbox manager” which deals with the requirements of section 3.1.2. Extensions, as they are proposed by the BPEL specification do not deserve their own architectural entity in our formal model. Instead, a refinement from the formal model of the “core” to the model including the extensions is the proposed choice.

### 3.2.1 Formalizing Activities

To formalize activities we have to formalize three aspects: 1. activity concepts, 2. activities in a process definition, 3. instances of activities.

---

<sup>2</sup>The usage of the term “core” here differs from the usage in [FGV05].

**Activity Concepts.** To formalize each activity concept on its own, we have to identify the common interface and the common protocol which connects the concepts first (cf. sec. 3.1.1). Using this common base, we can independently formalize each activity concept that is given by the informal specification.

Our approach to do so, is top-down. At first we present the general structures which all activity concepts have in common. Similarly, we will examine behaviour based on just the common structures. We formalize either by using function symbols and ASM rules, respectively. This provides us with an abstract definition of an activity concept – both in structures and behaviour. Chapters 4 and 5 serve this purpose.

From thereon, we will refine structures and extend them for each activity concept that is given according to the informal specification. Having refined structures at hand, we present refinement steps from the abstract definition of an activity concepts towards the detailed formal definition of a chosen concept. Our level of detail here will represent the definitions of the “core”. This is done in chapters 7 to 10.

Please note that the definition of function symbols and ASM rules is invariably fixed with respect to the informal specification. It does in no way depend on the concrete definition of a process and just contains a formal representation of the information that is given in [CGK<sup>+</sup>03].

**Activities in a Process Definition.** The ASM rules and the function symbols which formalize the activity concepts are just one part of a formal ASM model of BPEL. Both were defined to suit any activity in any process definition. The formal representation of an activity is always subject to a concrete process definition.

To formalize a BPEL process definition, we will translate a concrete BPEL process definition into functions given in the initial state of the ASM. That is, we provide a reasonable interpretation for every function symbol that has been defined for the activity concepts. In order to be able to deal with any well-formed BPEL process, we present a scheme by which one can translate a process definition given in XML into an algebraic specification of the functions in the initial state of the ASM. This scheme is defined in section 4.1.

Thus, the set of all suitable initial states is defined by the set of all well-formed BPEL process definitions.

**Instances of Activities.** An instance of an activity is a thing which exists only during the execution of a process. Hence its formal representation may exist only during a run of the ASM: Activity concepts define the behaviour of activities and therefore make use of function symbols to denote instances of activities. The interpretation of these symbols will change depending on the ASM rules and depending on the run of the system, e.g. the arrival of messages which cause the creation of a new process instance.

We formalize the active behaviour that is due to instances of activities by assigning an ASM agent to each instance of an activity. The ASM agent executes the ASM rule of its activity’s concept and modifies the state of the process instance according to it’s activities definition.

Instances of activities and their relation to concepts of activities are formalized in section 5.1.1.

Thus, our formal model defines a set of distributed multi-agent ASMs, each having the same set of ASM rules but differing in the set of initial states, depending on the chosen process definition. The domain of our formalization are the possible runs of all ASMs.

#### 3.2.2 Formalizing the Inbox Manager

Unlike the activity concepts, the semantics of the inbox manager are vaguely described in the informal specification. Although its principle required behaviour is obvious, the operational steps remain unknown. Farahbod et al defined a straight-forward model which comprises the necessary behaviour to receive messages from an external source without knowing its designated process instance, and to either assign this message to the corresponding process instance, or to create a new process instance whenever necessary. [FGV05].

We will skip a full formal definition of the inbox manager in this document, despite the fact that such a definition is required for a complete formal semantics of the entire language. Our reasons are manifold, among them the fact that the description of its behaviour is merely vague: Unlike for the activity concepts, we cannot propose a ground model for the inbox manager that can be verified by comparing formal model and informal description. Nonetheless we present an abstract definition of its behaviour in section 6.6 which is a simple abstraction of the model that has been defined by Farahbod et al.

#### 3.2.3 Formalizing Extensions of BPEL

As stated in the introduction of this section, extensions of BPEL are formalized by refinement. In this document, we will confine ourselves to the “core” semantics and defer a formal definition of the extensions to a later version of this report.

## 4 Process Definition

Before we can specify any behavioral aspects of BPEL's activities, we need to express the underlying static constructs. Using Abstract-State Machines, we formalize the entire process definition by the help of terms over function- and relation-symbols.

As outlined Chapter 3 and in [Fah04] the semantic domain of our formalization are the runs of an Abstract-State Machine. This includes the process definition of a specific process, which will be given by static functions in the initial state: We do not translate a specific BPEL process into a specific set of ASM rules. Instead, the ASM rules we present in this report are designed to act upon the static functions which encode the process definition.

ASM rules are written, using function symbols which are interpreted in a given  $\Sigma$ -algebra  $S$ , a state of an ASM. If we want to fire an ASM rule in  $S$ , we have to be sure that the function symbols used in the rule have a proper interpretation in  $S$ . For any correct process definition, the ASM rules are meant to act upon any state which includes this process definition encoded by functions.

This essentially means, that we need to be able to choose our initial states such that the interpretation of the function symbols in the ASM rules yields a correct process definition. We may define the set of suitable initial states  $S \in I$  syntactically by the help of an algebraic specification  $p_{initial}$ :  $S \in I$  iff  $S$  is a model of  $p_{initial}$ .

### 4.1 Translating a Process Definition into an Initial State

There is a simple mechanism to translate a given XML document into an algebraic specification  $p_{initial}$  in first order logic which may be interpreted in a  $\Sigma$ -algebra  $S$ . Therefore we may translate any syntactically correct BPEL process definition into an algebraic specification defining the set of initial states. In [Fah04] we have shown the principles of expressing a BPEL process definition by the help of typed function symbols. We recall these principles here briefly.

A BPEL process definition is written in XML and is well-formed if it satisfies the according XML schema, i.e. it's grammar. The nature of XML languages allows to decompose complex structures using XML elements, parent-child-relations and referencing by properties of XML elements.

Usually, the denotation of an *XML element* in a document implies the existence of a unique entity that is of the type of the element's *XML tag*. Formalized in an ASM, the XML tag requires a new type symbol in the signature. For (almost) every XML element that is defined in an XML document, the existence of a unique element in the initial state of the ASM is required.

We use the existential quantifier and predicate-logic to guarantee pairwise distinct elements in our initial state that corresponds to the process definition. Let  $e_1 \equiv \langle \mathbf{tag1} \dots / \rangle$ ,  $\dots$ ,  $e_n \equiv \langle \mathbf{tagn} \dots / \rangle$  be the XML elements of the XML document and let

$Type_1, \dots, Type_n$  be the type symbols for the respective XML tags. Then

$$\exists t_1 \in Type_1, \dots, \exists t_n \in Type_n : \bigwedge_{i \neq j} t_i \neq t_j$$

denotes an algebraic specification that assures the existence of the required number of unique elements.

The *parent-child-relations* which are induced by the tree-structure relate two XML elements to each other. Formally, where the grammar allows an XML element of a type to be child of an XML element of another type, we need to declare a typed function symbol which expresses the possibility of this relation. In a concrete XML document, (almost) every parent-child-relation of XML tags implies a function definition of the corresponding functions, where the element of the carrier corresponding to the parent maps to corresponding element of the child. For any relation

```
<tagi ...>
  <tagj .../>
</tagi>
```

the algebraic specification which has declared  $t_i \in Type_i$  and  $t_j \in Type_j$ ,  $t_i \neq t_j$  requires an equation

$$relation_{i,j}(t_i) = t_j$$

where the function symbol  $relation_{i,j}$  denotes the relations between XML elements having **tagi** and those having **tagj**.

Where parent-child-relations of XML elements are given implicitly, explicit relations are expressed using XML's naming scheme and *attributes of XML elements*. An attribute of an XML element might either refer to another XML element of the same document by its name or it might denote some constant value. If the XML schema allows an XML element to carry an attribute, we declare a function symbol mapping from the type of the element's tag to the corresponding domain. In a concrete XML document, the function has to be defined accordingly.

The mechanism is similar to the formalization of parent-child-relations, except that the value of the function for the corresponding argument is the element which represents the XML element, and not the name. To formalize

```
<tagi name="name1" ... />
<tagj attribute1="name1" attribute2="17" .../>
```

the algebraic specification which has declared  $t_i \in Type_i$  and  $t_j \in Type_j$ ,  $t_i \neq t_j$  requires equations

$$attribute_{j,1}(t_j) = t_i \wedge attribute_{j,2}(t_j) = 17$$

where the function symbols  $relation_{j,1}$  and  $relation_{j,2}$  denote the relations due to the attributes of **tagj** respectively.

The complete algebraic specification which formalizes an XML document is of the form

$$\exists t_1 \in Type_1, \dots, \exists t_n \in Type_n : \bigwedge_{i \neq j} t_i \neq t_j \wedge p(t_1, \dots, t_n)$$

where  $p(t_1, \dots, t_n)$  is the conjunction of all equations which formalize either parent-child-relations or attributes of XML elements.

Since every BPEL process definition is an XML document, the sketched mechanism can be applied to generate an algebraic specification which defines the set of suitable initial states that include the definition of this process. In the course of this report we will relate XML elements, attributes and parent-child-relations to the corresponding type symbols and function symbols.

## 4.2 Definition of Process Structure

BPEL provides a large number of entities and relations which need to be expressed formally. To maintain understandability of this formalization, we will formalize an entity or a relation by introducing a dedicated function symbol the first time we need its formal representation. We will also relate the formalization to the corresponding XML code at place. Thus we will give no compact representation of the entire static structure of BPEL processes.

Regardless of this mode of conduct, there is are some general structure in each BPEL process which we need from the very beginning. These structures defines the hierarchical relations between the process' activities. This hierarchy induces the control flow of the process. Therefore we will define it and examine its properties in the following. Similarly, large parts of BPEL's semantics rely on the manipulation of variables. Consequently, we present their static aspect: the declaration.

### 4.2.1 The Process

The entire definition of a BPEL process is enclosed in `process` tags. Everything defined therein belongs to a specific process and needs a formal representation. Additionally, a number of entities of a process is also associated to the very process directly. Hence, we formalize the process.

**Universe:** *Process* (abstract definition of a process, `<process .../>`)

### 4.2.2 Activity Tree

It can easily be seen that the structured XML code of a BPEL process definition defines a directed tree over all activity elements (we will equally use the term *activity* for the rest of this document).

It is obvious that each activity of the process definition requires a unique element in the initial state:

**Universe:** *Activity*.

For each type of activity, we provide its own universe, where all of them are pairwise disjoint, see Appendix B.1. Each activity is part of a process which it needs to know.

$$activityProcess : Activity \rightarrow Process$$

The tree of activities defines a hierarchy: an activity  $A$  is the *parent activity* of an activity  $B$  if  $A$  is the next (syntactically) enclosing activity of  $B$ . Then  $B$  is a *child activity* of  $A$  if  $A$  is the parent activity of  $B$ .  $A$  may have several child activities.

BPEL's basic activities, which do not enclose any further activities, constitute the leaves of this tree. The root of the tree is the process element itself which (transitively) encloses all activities of the process. The process element has the operational semantics of a scope, which is an activity:

**Universe:**  $Scope \subset Activity$ .

$$processScope : Process \rightarrow Scope$$

The *activity tree* is the directed tree induced by the process definition, where the nodes are the set of all activities of the process, and there is a directed edge from activity  $A$  to activity  $B$  iff  $A$  is parent activity of  $B$ .

In the rest of this report, we will use the activity tree to explain global properties of a process. The operational semantics use the local parent-child-relations, only. Therefore we define the following function symbols.

$$\begin{aligned} parentActivity & : Activity \rightarrow Activity_{structured} \\ childActivities & : Activity \rightarrow \mathcal{P}(Activity) \end{aligned}$$

By convention, if  $act$  is the root of the activity tree, then  $parentActivity(act) = act$  holds. The hierarchy of activities, as expressed in the activity tree, induces the control flow of the process. In general, a child activity  $B$  of activity  $A$  may only be executed if  $A$  allows it. In turn,  $A$  may finish its execution only after  $B$  has finished. We will examine and formalize this behaviour in the next chapter.

### 4.2.3 Links

The strict hierarchical parent-child-relations of the activities of a process are not sufficient to define causal dependencies between activities which are located in different subtrees of the activity tree.

Therefore the link concept is defined in BPEL. A flow activity defines concurrent execution of its (transitive) child activities. A *link* from activity  $B$  to activity  $B'$  in the subtree under a flow defines a causal dependency:  $B'$  may start its execution only, after  $B$  has finished. Hence a link is declared at a flow which encloses  $B$  and  $B'$  and it is defined by its endpoints.

**Universe:**  $Link$

$$\begin{aligned} flowLinks & : Flow \rightarrow \mathcal{P}(Link) \\ linkSource & : Link \rightarrow Activity \\ linkTarget & : Link \rightarrow Activity \end{aligned}$$

#### 4.2.4 Variables

A BPEL process is capable of handling data by the use of *variables*. It is sufficient to conceive a variable of BPEL process like a variable in any other imperative programming language: values may be assigned, read and may be manipulated in parts. A variable must be declared, either globally at the `process` or locally at a `scope`.<sup>1</sup>

```
<scope name="...">
  <variables>?
    <variable name="ncname" messageType="qname"?
              type="qname"? element="qname"?/>+
  </variables>
</scope>
```

The `scope` at which the variable is declared, owns the variable, i.e. any dynamic properties of the variable are subject to the dynamic properties of its scope. This relation gives rise to the following declarations:

**Universe:** *Variable*

static functions (4.2.4-1)

$$\begin{aligned} \text{scopeVariables} & : \text{Scope} \rightarrow \mathcal{P}(\text{Variable}) \\ \text{variableScope} & : \text{Variable} \rightarrow \text{Scope} \end{aligned}$$

Each variable has a `name` to reference it, and a `type`, being either a WSDL message type (`messageType`), or a XML schema simple type (`type`) or a XML schema element (`element`). For our formalization we do not distinguish XML schema simple type and XML schema element: The manipulation of the inner structure of a variable with such a type is not subject of BPEL. This does not hold for the WSDL message type. We know its internal structure and for the correlation handling mechanism (c.f. sec. 6.4.2) we need to access its internal structure. Hence the following formal definitions.

**Universe:** *Variable*

**Universe:** *MessageType*

**Universe:** *XMLschemaType*

$$\begin{aligned} \text{variableMessageType} & : \text{Variable} \rightarrow \text{MessageType} \text{ (messageType)} \\ \text{variableXMLschemaType} & : \text{Variable} \rightarrow \text{XMLschemaType} \text{ (type and element)} \end{aligned}$$

requirement for the initial state (4.2.4-1)

$$\begin{aligned} \forall \text{var} \in \text{Variable} : \\ \neg(\text{variableMessageType}(\text{var}) \neq \text{undef}) \\ \wedge \text{variableXMLschemaType}(\text{var}) \neq \text{undef} \end{aligned}$$

<sup>1</sup>As stated in section 4.2.2, a `process` tag implicitly defines a *Scope* as root of the activity tree.

We do not formalize the name of a *Variable* to be a property. Instead we assume that each **variable** has a unique name and hence the name and the variable are identical. We formalize values of variables in the chapter 6.

## 5 Control Flow

Chapter 4 defined those function symbols where the interpretation encodes the process definition. We focussed on two special structures, the activity tree and links, since these induce the process' control flow. In this chapter we will examine, how both structures are used to define the control flow of a process. We will also provide an overview of how a process is executed.

### 5.1 Process Instances

The BPEL process definition just provides the static framework for the execution of the process. It defines all the structural entities and their relations which constitute the process. Their runtime properties like variable values, messages, etc are subject to a specific execution depending on its input.

To execute a BPEL process, an *instance* of it has to be created. Then this process instance is executed. Both, the creation and the execution of a process instance, is broken down to the creation and execution of *instances of activities*. There may be more than just one instance of an activity per process instance. Any two instances of the same activity are executed concurrently. Each instance of an activity belongs to exactly one process instance. This behaviour meets the requirements of distributed business processes.

#### 5.1.1 Instance of an Activity

In the formal model, we introduce a symbolic identifier to distinguish instances of activities:

**Universe:** *SubInstance*

The instance of an activity *act* is a pair  $(sI, act) \in SubInstance \times Activity$ , *sI* is the symbolic identifier to discriminate  $(sI, act)$  from other instances of *act*. An activity's instance has *dynamic properties*. Formally, a dynamic property is a mapping from an activity's instance to some value. The value of a dynamic property is modified during the execution of the activity's instance. The dynamic properties vary among the types of the activities. The *state* of an activity's instance is given by its dynamic properties.

#### 5.1.2 Executing an Instance of an Activity

Executing an instance of a basic activity means to execute a simple operation like sending a message or modifying a variable.

Executing an instance of a structured activity means to execute instances of its child activities: Depending on the semantics of the activity and the activity's definition, it may

have to execute its child activities in a particular order, execute all of its child activities concurrently or repeatedly execute a child activity, to name a few examples.

An instance of a structured activity executes one of its child activities by creating an instances of the child activity, and by executing this instance. The instance of a structured activity awaits the successful completion of all instances of child activities it created, before completing itself.

An instance of an activity is executed at most once. For each execution of a child activity, a structured activity must generate a fresh instance. Hence, the instances of activities form a tree. Its root is an instance of the BPEL process under consideration.

The root's execution is triggered by the inbox manager which we have introduced in section 3.1.2 and which we will formalize after the following subsection 6.6. The execution of a process instance completes successfully if its instance of the root activity completes successfully.

### 5.1.3 The Subinstance Tree

We will now formalize the relations between instances of activities. We already introduced the activity tree via which instances of parent and child activities are related and we will now deal with the symbolic identifiers *SubInstance* which discriminate instances ( $sl, act$ ) of an activity *act*.

It is not necessary to introduce a new symbolic identifier for each instance of an activity. In general, a structured activity executes a child activity at most once – hence no need to discriminate several instances of the same child activity. There are just two exceptions: one is the while-activity (**Universe:** *While*) which iterates the execution of its child activity, the other one is the event handler (**Universe:** *EventHandler*) which we will introduce in Section 9.5.

Therefore we will treat the special behaviour of repeated instantiation like an exception in our model: We may continue to use the same symbolic identifier upon the creation of new instances of child activities by a structured activity  $act \in Activity \setminus (While \cup EventHandler)$ . The usage of the same symbolic identifier propagates downwards the activity tree. Should we require a new symbolic identifier, we obtain one using ASM's **new** operator. We relate the new identifier to the identifier of the creating instance of its parent activity by the help of dynamic functions.

**Universe:** *SubInstance*

dynamic functions (F5.1.3-1)

$$\begin{aligned} parentSubInstance & : SubInstance \rightarrow SubInstance \\ childSubInstances & : SubInstance \rightarrow \mathcal{P}(SubInstance) \end{aligned}$$

requirement for the initial state (5.1.3-1)

$$\begin{aligned} \forall sl \in SubInstance : \\ & ( sl \in ProcessInstance \wedge parentSubInstance(sl) = sl) \\ & \vee ( sl \notin ProcessInstance \wedge parentSubInstance(sl) = undef) \\ & \wedge childSubInstances(sl) = \emptyset \end{aligned}$$

Starting the execution of a BPEL process at an instance  $(\mathbf{pl}, \mathbf{act}_{root})$  of its root activity, we successively define *parentSubInstance* and *childSubInstances*. This yields a tree of symbolic identifiers from *SubInstance*, induced by the activity tree, having  $\mathbf{pl}$  as root. We call this tree the *subinstance tree of  $\mathbf{pl}$* .

The ASM rule to extend the subinstance tree is defined straight forward:

**EXTENDSUBINSTANCETREE** (R5.1.3-3)

$(\mathbf{sl} \in \mathit{SubInstance}, \mathbf{sl}_{child} \in \mathit{SubInstance}) \equiv$

**add  $\mathbf{sl}_{child}$  to  $\mathit{childSubInstances}(\mathbf{sl})$**   
 $\mathit{parentSubInstance}(\mathbf{sl}_{child}) := \mathbf{sl}$

referenced in WHILESTARTNEWBODY (R8.6.2-57),

Any maximal set of instances activities which belong to the same process instance and which share the same symbolic identifier or one in the subtree of the subinstance tree below their identifier is a *subinstance* of the process instance. Consequently, we read the root  $\mathbf{pl}$  of a subinstance tree as the identifier of a process instance.

**Universe:**  $\mathit{ProcessInstance} \subset \mathit{SubInstance}$

Very similar to the activity tree, we will use the subinstance tree whenever we need to talk about global properties of a process instance. Any operations and rules on the instances of activities will be defined locally, using *parentSubInstance* and *childSubInstances* only.

As we define the semantics we will eventually need to evaluate the identifier  $\mathbf{sl}$  of instance of an activity  $I_1 = (\mathbf{sl}, \mathbf{act})$  from the point of view of  $I_2 = (\mathbf{sl}', \mathbf{act}')$  which is in the subinstance of  $I_1$ , i.e. where  $\mathbf{sl}'$  is a node under  $\mathbf{sl}$  in the subinstance tree and  $\mathbf{act}'$  is a node under  $\mathbf{act}$  in the activity tree. The derived function  $\mathit{currentSubInstance}_{fromInner}$  will serve our needs: for the mentioned scenario  $\mathit{currentSubInstance}_{fromInner}(\mathbf{sl}', \mathbf{act}', \mathbf{act})$  will evaluate to  $\mathbf{sl}$ . The definition ( $\mathcal{D}$  B.2.3-1) is given in the Appendix B.2.3.

$\mathcal{D}$   $\mathit{currentSubInstance}_{fromInner} : \mathit{SubInstance} \times \mathit{Activity} \times \mathit{Activity} \rightarrow \mathit{SubInstance}$

## 5.2 Agents execute Activities

We conceive a BPEL process instance as a distributed system where each instance of an activity is executed concurrently to any other instance of an activity. Our formal definition of this behaviour results in the use of ASM agents as defined in section 2.2.3.

By a choice of simplicity, we introduce a unique ASM agent for each activity and subinstance this activity is executed in. Hence each agent has to have the knowledge about its activity and about its current state. This results in the following function symbols with obvious interpretation.

$\mathit{myActivity} : \mathit{Agent} \rightarrow \mathit{Activity}$   
 $\mathit{mySubInstance} : \mathit{Agent} \rightarrow \mathit{SubInstance}$

All ASM agents execute the same ASM rule `RUNBPELACTIVITY` (RB.2.2-104), explicitly defined in Appendix B.2.2. An abstract version of this rule which does not discriminate types of activities is `RUNBPELACTIVITYpattern`.<sup>1</sup>

---

$\frac{\text{RUNBPELACTIVITY}_{pattern}}{(\text{self})} \equiv$	(R5.2.0-4)
---	------------

`EXECUTEACTIVITYpattern(mySubInstance(self), myActivity(self))`

references `EXECUTEACTIVITYpattern` (R5.9.0-18)

`EXECUTEACTIVITYpattern` will be defined in the following. Any further ASM rule defined in this report is a refinement of `EXECUTEACTIVITYpattern` or is referenced in it. Exploiting the macro concept of ASM rules, each rule is written down in a context where the symbol `self` has a determined interpretation upon firing of this rule: namely the name of the agent executing the said activity.

**Important note:** Simplifying the denotation and understanding of ASM rules, we will write and read them from the perspective of the firing agent, or more conveniently from the perspective of the executed instance of activity performing an operational step.

Section 5.4.1 will make use of this perspective to define communication between activities. In advance, we need to learn about the basic internal structure of an activity.

### 5.3 The Internal States of an Activity

With the subinstance tree of a process instance, we may now properly define the state and the execution of an activity. Altogether, they form the “activity life-cycle” of an activity. In conjunction with the activity tree, this builds the base for the actual control flow and hence the execution of a process instance.

The principle behaviour of an activity, as outlined in section 5.1.2 is captured best by the help of a finite state machine over *internal states* of this activity.

Each internal state reflects the current overall state of an activity’s execution. Depending on its internal state, the activity modifies dynamic properties (including its internal state). Initially, each activity is *disabled*. It may be set to *enabled* to start its execution; *running* serving as state during the activity’s execution. Finishing its execution, it enters the terminal state *completed* which expresses a *successful completion* of its execution. We say that an activity is *active* after becoming *enabled* but before reaching a terminal, internal state.

An activity which throws a fault enters *faulted*; *stopping* is required for a premature termination in case of an error, *completing* and *stopped* are final states of a successfully or unsuccessfully completed execution respectively. Hence, an activity may be in the following internal states:

**Universe:**  $ActivityStates = \{disabled, enabled, running, faulted, stopping, stopped,$

---

<sup>1</sup>The first full definition of an ASM rule will be marked with a vertical bar on the left side. A repeatedly denoted ASM rule that was defined somewhere else in the text will have no such bar.

*completing, completed* }

Then the internal state of an activity is a runtime property:

dynamic functions (F5.3.0-2)

$$activityState : SubInstance \times Activity \rightarrow ActivityState$$

We require any activity to be inactive at the start of process instance, that is all activities have its internal states *disabled* in any subinstance:

requirement for the initial state (5.3.0-2)

$$\forall sl \in SubInstance, act \in Activity : activityState(sl, act) = disabled$$

A finite state machine describes the possible changes of an activity's internal state. Figure 5.1 shows the graphical representation of this finite state machine. We will formalize it in the ASM rules of the activities later on.

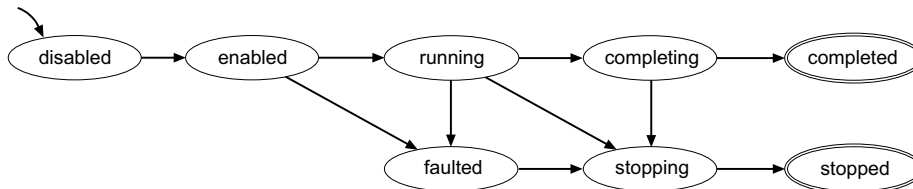


Figure 5.1: Finite state machine of the internal states of a BPEL activity

Since all activities use the same model of internal states, we may define a first ASM rule which abstractly defines the finite state machine of the internal states.

ACTIVITYSTATEMACHINE

(R5.3.0-5)

$(sl \in SubInstance, act \in Activity) \equiv$

**if**  $activityState(sl, act) = enabled$  **then**  
 STARTACTIVITY( $sl, act$ )  
**if**  $activityState(sl, act) = running$  **then**  
 RUNACTIVITY( $sl, act$ )  
**if**  $activityState(sl, act) = completing$  **then**  
 COMPLETEACTIVITY( $sl, act$ )  
**if**  $activityState(sl, act) = stopping$  **then**  
 STOPACTIVITY( $sl, act$ )

referenced in EXECUTEACTIVITY<sub>pattern</sub> (R5.9.0-18)

We will define such a rule (e.g. SEQUENCESTATEMACHINE) for each type of activity, including the corresponding referenced rules (e.g. STARTSEQUENCE).

If during the execution of a process instance, an activity's internal state finally equals *completed*, we say that the activity *completed successfully*. If the activity's internal state equals *stopped*, we say that the activity *completed unsuccessfully*. In any other case, the activity didn't complete (yet).

## 5.4 Dynamics of the Control Flow

What can't be seen in Fig. 5.1 and ACTIVITYSTATEMACHINE, but will be defined in the ASM rules is, that the transition from *disabled* to *enabled* requires an external event. The same holds for the transitions to *stopping*. Together with condition (5.3.0-2) we may conclude that no activity can leave the *disabled* state on its own.

In section 5.1 we roughly described the dynamics of the control flow as hierarchical initialization of execution. We will now define a concept that formalizes the (hierarchical) interaction of activities according to the control flow, including the transition of the internal state from *disabled* to *enabled*.

### 5.4.1 The signal concept

We conceive a BPEL process as a distributed system where each activity is executed concurrently to the rest of the process. Furthermore, we assume that no activity is allowed to read or alter any runtime properties of another activity, including its internal state. Yet, the steps of each activity respect the hierarchical definition of the entire process. Avoiding the need of a central coordinating entity, we let activities communicate directly to each other using messages. To discriminate from messages exchanged between processes, we use the term *signal*.

A signal carries no further information except its type which corresponds to its intrinsic semantics. A signal may only be sent along the structure of the activity tree from the parent to the child or vice versa (see section 4.2.2).

We discriminate two types of signals. *DownSignals* are sent downwards the activity tree. We may read them as orders from the parent to the child.

**Universe:**  $DownSignals =_{def} \{signalEnable, signalStop, signalComplete, signalNegateLinks, signalTerminate\}$

Signal	semantics
<i>signalEnable</i>	start a child activity's execution
<i>signalStop</i>	interrupt a child activity's execution
<i>signalComplete</i>	notify a child activity to finish its execution (used for eventHandlers)
<i>signalNegateLinks</i>	let a child activity execute the DPE
<i>signalTerminate</i>	interrupt the receiving activity's execution in case of a process termination

*UpSignals* are sent upwards the activity tree. We may read them as acknowledgements for orders.

**Universe:**  $UpSignals =_{def} \{signalCompleted, signalStopped, signalTerminate\}$

Signal	semantics
<i>signalCompleted</i>	tells that the sender successfully completed its execution
<i>signalStopped</i>	tells that the sender interrupted its execution
<i>signalTerminate</i>	interrupt the receiving activity's execution in case of a process termination

The distributed setting requires a signal storing channel between the sender and the receiver of a signal. We conceive the exchange of signals as a purely local interaction between a parent and a child activity via *signalChannel<sub>down</sub>* and *signalChannel<sub>up</sub>*. Thus, a channel is determined by its endpoints and the *SubInstance* of the child activity: We don't need the information of the parent activity's *SubInstance* as it is always known from the subinstance tree (see section 5.1.3).

$$\begin{aligned} signalChannel_{down} & : SubInstance \times Activity \times Activity \rightarrow \mathcal{P}(DownSignals) \\ signalChannel_{up} & : SubInstance \times Activity \times Activity \rightarrow \mathcal{P}(UpSignals) \end{aligned}$$

Now we have the infrastructure to formally define the interaction between activities. Sending a signal *sig* from parent activity *A* to a child activity *B* means adding *sig* to *signalChannel<sub>down</sub>(sI, A, B)*, where *sI* is the subinstance of *B* which is meant to receive the signal. Receiving a signal means taking an element out of this set. The channel for the opposite direction is *signalChannel<sub>up</sub>(sI, B, A)*. We will use the ASM rule

**signal sig to targetActivity in subInstance via channel**

to model the sending of a signal from the perspective of the firing activity. Firing the ASM rule

**onSignal sig from sourceActivity in subInstance via channel do R otherwise R'**

from the perspective of the receiving activity means that in case the signal *sig* is found in the specified channel, *R* will be fired, otherwise *R'*. The detailed definition can be found in Appendix A.1.3.

An activity (being in its initial internal state *disabled*) may set itself to *enabled* upon receiving the *signalEnable* from its parent activity. The following fragment of an ASM rule is our formal definition of this behaviour.

**onSignal** *signalEnable* **from** *parentActivity*(act) **in** sl **via** *signalChannel*<sub>down</sub> **do**  
**set** *activityState* **from** *disabled* **to** *enabled*

The introduction of signal channels is a design choice to meet the requirement of communication among instances of activities that can be inferred from the informal specification. By defining distinct signals channels between any two instances of activities that might communicate with each other, we construct an interface to this activity. This interface, which can be considered as part of the instance of the activity, is the only way to influence an activity's behaviour. Hence each instance of an activity is encapsulated by the inbound and outbound channels.

The concept of communication among agents as defined by Farahbod et al provides exactly the same runs like the channel concept when considering the states of instances of activities only. Both models differ in the presence of an architectural aspect which might be of interest for an actual implementation of BPEL.

#### 5.4.2 Dynamics of Links

We already defined the general static structure of links in 4.2.3. We will now extend the definitions and define their semantics in a subinstance.

In a process instance, a link may hold a boolean value, the “Link Status”. Having a direction, a link establishes some kind of one way communication channel being orthogonal to the activity tree. Initially a link holds no value (*undef*).

$$linkStatus : SubInstance \times Link \rightarrow \{true, false\}$$

requirement for the initial state (5.4.2-1)

$$\forall sl \in SubInstance \forall link \in Link \ linkStatus(sl, link) = undef$$

The value of a link is set by its source activity, upon reaching the internal state *completed* (see section 5.3). The set value is computed by the “Transition Condition” (formally: *linkSourceTransitionCondition*) which is a *BooleanExpression* over any runtime properties of the process instance. Once set, the Link Status must not be changed. We may read this behaviour as sending a single boolean message via the link.

If an activity is target of several links, it computes a single boolean value from the values of all incoming links. The associated boolean function is called “Join Condition” (formally: *linkTargetJoinCondition*) and written down as *BooleanLinkExpression*.

**Universe:** *BooleanLinkExpression*

**Universe:** *BooleanExpression* =<sub>def</sub> *BooleanLinkExpression*  $\cup$  ...

$$\begin{aligned} \textit{linkSourceTransitionCondition} & : \textit{Link} \rightarrow \textit{Expression} \\ \textit{linkTargetJoinCondition} & : \textit{Activity} \rightarrow \textit{BooleanLinkExpression} \end{aligned}$$

In BPEL, boolean conditions are strings with syntax and semantics outside the scope of the language. Thus we assume a unique object of the corresponding universe in the initial state. To evaluate this object, we require an abstract function

$$\mathcal{A} \textit{evaluateBooleanExpression} : \textit{SubInstance} \times \textit{BooleanExpression} \rightarrow \{\textit{true}, \textit{false}\}$$

which returns *true* iff the given boolean expression evaluates to *true* in the given subinstance. We may now define the ASM rules and functions for the operational steps of the dynamics of links.

The following functions simplify handling and evaluating link values, especially for sets of (incoming or outgoing) links.

$$\begin{aligned} \mathcal{D} \textit{activityTargetLinks} & : \textit{Activity} \rightarrow \mathcal{P}(\textit{Link}) \text{ and} \\ \mathcal{D} \textit{activitySourceLinks} & : \textit{Activity} \rightarrow \mathcal{P}(\textit{Link}) \end{aligned}$$

return the set of links that start/end at the given activity (see Appendix B.2.4).

derived functions ( $\mathcal{D}$  5.4.2-1)

$$\mathcal{D} \textit{allLinksSet} : \textit{SubInstance} \times \textit{Activity} \rightarrow \{\textit{true}, \textit{false}\}$$

$$(\textit{sl}, \textit{act}) \mapsto \forall \textit{link} \in \textit{activityTargetLinks}(\textit{act}) \textit{linkStatus}(\textit{sl}, \textit{link}) \neq \textit{undef}$$

determines whether all incoming links have a valid value, which happens only if the source activities of the incoming links reached *completing*.

$$\mathcal{D} \textit{joinConditionSatisfied} : \textit{SubInstance} \times \textit{Activity} \rightarrow \{\textit{true}, \textit{false}\}$$

$$(\textit{sl}, \textit{act}) \mapsto \begin{cases} \textit{true}, & \textit{allLinksSet}(\textit{sl}, \textit{act}) \\ & \wedge (\textit{activityTargetLinks}(\textit{act}) \neq \emptyset \\ & \Rightarrow (\textit{cond} = \textit{linkTargetJoinCondition}(\textit{act}) \\ & \wedge \textit{evaluateExpression}(\textit{sl}, \textit{cond}) = \textit{true}) \\ & ) \\ \textit{false}, & \textit{else} \end{cases}$$

returns *true* iff *linkTargetJoinCondition* could be evaluated to *true*. Please note that *joinConditionSatisfied* evaluates to *true* if the activity in the argument is not the target of any *Link*.

To set all outgoing links of an activity according to the *linkSourceTransitionCondition*, fire EVALUATEOUTGOINGLINKS (R5.4.2-6). To set all outgoing links to *false*, fire DISABLEOUTGOINGLINKS (R5.4.2-7).

EVALUATEOUTGOINGLINKS (R5.4.2-6)

$(sl \in SubInstance, act \in Activity) \equiv$

**forall** link  $\in$  *activitySourceLinks*(act) **do**  
**let** transCond = *linkSourceTransitionCondition*(link) **in**  
*linkStatus*(sl, link) := *evaluateExpression*(sl, transCond)

referenced in ...

DISABLEOUTGOINGLINKS (R5.4.2-7)

$(sl \in SubInstance, act \in Activity) \equiv$

**forall** link  $\in$  *activitySourceLinks*(act) **where** *linkStatus*(sl, link) = *undef* **do**  
*linkStatus*(sl, link) := *false*

referenced in PROPAGATEDPE (R5.6.0-15)

### 5.4.3 Starting Activities

Any activity which is enabled by its parent activity has to delay its execution until its *linkTargetJoinCondition* is satisfied. To start the execution of an activity, check if its incoming links can be evaluated to *true*. If so, set it to *running*.

STARTACTIVITY (R5.4.3-8)

$(sl \in SubInstance, act \in Activity) \equiv$

**if** *allLinksSet*(sl, act)  $\wedge$  *joinConditionSatisfied*(sl, act) = *true* **then**  
**set** *activityState* **from** *enabled* **to** *running*

referenced in: ACTIVITYSTATEMACHINE (R5.3.0-5)

Each BPEL activity uses either this ASM rule or a refinement of it to start its execution.

### 5.4.4 Completing Activities

To successfully complete the execution of an activity, evaluate all outgoing links by their *linkSourceTransitionCondition*, set the internal state to *completed* and send *signalCompleted* to its parent activity.

COMPLETEACTIVITY (R5.4.4-9)

$(sl \in SubInstance, act \in Activity) \equiv$

**set** *activityState* **from** *completing* **to** *completed*  
**signal** *signalCompleted* **to** *parentActivity*(act) **in** sl **via** *signalChannel<sub>up</sub>*  
EVALUATEOUTGOINGLINKS(sl, act)

■ referenced in `ACTIVITYSTATEMACHINE` (R5.3.0-5)

Each BPEL activity uses either this ASM rule or a refinement of it to complete its execution.

### 5.4.5 Hierarchy of Running Instances of Activities

In the subsequent sections, we will present some mechanisms of BPEL which work similarly for all activities. For structured activities, these mechanisms are defined to exchange signals or faults with their *running* instances of their child activities.

Since a structured activity is the only entity of a process instance which can cause the execution of its child activities, it is capable of knowing *all* running instances of its child activities. This observation gives rise to the following function.

dynamic functions (F5.4.5-1)

$$activityRunningChilds : SubInstance \times Activity \rightarrow \mathcal{P}(SubInstance \times Activity)$$

requirement for the initial state (5.4.5-1)

$$\begin{aligned} \forall sl \in SubInstance \quad \forall act \in Activity \\ activityRunningChilds(sl, act) = \emptyset \end{aligned}$$

In chapters 8, 9 and 10 where we define the semantics of structured activities, this function will be properly updated such that it always contains just the instances of child activities which are currently being executed.

Please note that the function is defined for basic activities as well. But the value of *activityRunningChilds* will never change for a basic activity. This technicality is introduced to keep the ASM rules simple. We will make use of these properties in the definition of the mentioned mechanisms.

## 5.5 Faults and their Propagation

The previous sections defined the basic concepts of how activities of a BPEL process instance communicate with each other and how control flow in principle can be defined. Without writing it, we assumed the case of the *positive control flow* where no (expected or unexpected) errors occurred.<sup>2</sup>

In BPEL, errors occur whenever an activity wants to operate on an undefined state of its process instance (e.g. read an invalid variable value). Since the behaviour of the activity would be undefined, it delegates the responsibility for this error to a part of the process that has a defined behaviour in such a state. The error is delegated as a *fault* which is basically a message describing the error. The enclosing scope of the faulting activity is responsible for any faults that occur inside of it.<sup>3</sup> The faulting activity delegates the error by sending the fault to its enclosing scope.

<sup>2</sup>We consider errors on the level of BPEL, only.

<sup>3</sup>Since at least the process itself is a scope, there is always an enclosing scope.

Due to the distribution of the activities of a BPEL process, the faulting activity just knows its parent activity, which is either the enclosing scope or has the same enclosing scope. By passing a fault to the parent activity along the activity tree, the fault always reaches the enclosing scope, which has a defined behaviour for the current state of the process. We call this mechanism *fault propagation*. We say that the fault is *thrown* when it is initially created at the faulting activity.

Since the principles of the fault propagation rely on the same basics as the signal concept (see sec. 5.4.1), we add a third channel  $faultChannel_{up}$ , upwards along the activity tree. Its purpose is obvious from the fault propagation. This channel may hold faults, each being a message with a structured content, which is why we don't mix it with  $signalChannel_{up}$ .

The minimal structure of a fault is its name ( $FaultName$ ) describing the type of error. Any further structure depends on the specific fault and will be formalized when we need it.

**Universe:**  $Fault$

**Universe:**  $FaultName$

$$\begin{aligned} faultName & : & Fault & \rightarrow & FaultName \\ faultChannel_{up} & : & SubInstance \times Activity \times Activity & \rightarrow & \mathcal{P}(Fault) \end{aligned}$$

### 5.5.1 Throwing Faults

Throwing a  $Fault$  is operationally executed by creating it and sending it via  $signalChannel_{fault}$  to its parent activity, see THROW (R5.5.1-10). An activity which has thrown a  $Fault$  may no longer succeed in its execution and enters the internal state *faulted*.

<p><u>THROW</u></p> <p>(sl <math>\in</math> <math>SubInstance</math>, act <math>\in</math> <math>Activity</math>, fName <math>\in</math> <math>FaultName</math>) <math>\equiv</math></p> <p><b>let</b> fault = <b>new</b>(<math>Fault</math>) <b>in</b>  <math>faultName</math>(fault) := fName  <b>signal</b> fault <b>to</b> <math>parentActivity</math>(act) <b>in</b> sl <b>via</b> <math>faultChannel_{up}</math>  <b>set</b> <math>activityState</math> <b>from</b> <i>enabled, running, completing</i> <b>to</b> <i>faulted</i></p>	(R5.5.1-10)
--	-------------

### 5.5.2 Hierarchially Propagating Faults

Using the signal concept, propagating faults means checking the incoming  $signalChannel_{fault}$  of all running child activities. If any fault is found, it is moved to the  $signalChannel_{fault}$  directed towards the own parent activity. By definition, the propagation of faults is applied by structured activities only. Hence the set of running child activities is given by  $activityRunningChilds$  (F5.4.5-1) (see section 5.4.5).

PROPAGATEFAULTSACTIVITY (R5.5.2-11)

( $sl \in SubInstance, act \in Activity_{structured}$ )  $\equiv$   
**forall** ( $sl_{child}, child$ )  $\in activityRunningChilds(sl, act)$  **do**  
**forall**  $fault \in faultChannel_{up}(sl_{child}, child, act)$  **do**  
**add**  $fault$  **to**  $faultChannel_{up}(sl, act, parentActivity(act))$   
**remove**  $fault$  **from**  $faultChannel_{up}(sl_{child}, child, act)$

referenced in EXECUTEACTIVITY<sub>pattern</sub> (R5.9.0-18)

### 5.5.3 The Stop Concept

A *Fault* finally reaches the enclosing scope which is responsible for handling it. In BPEL a scope has always defined semantics, regardless which *Fault* was thrown. In order to handle the fault, the scope needs to be certain that it has full control over the part of process it is responsible for. Therefore any activity which is still *running* when the scope catches the fault needs to be stopped.

Similar to activating activities, *stopping* them is done using signals. We defined *signalStop* to initiate the termination of an activity's execution. *signalStopped* acknowledges the termination.

Hence, the scope has to apply a rule similar to

**signal** *signalStop* **to**  $childActivity(scope)$  **in**  $sl$  **via**  $signalChannel_{down}$ .

Its child activity reacts on *signalStop* by applying

**onSignal** *signalStop* **from**  $parentActivity(act)$  **in**  $sI$  **via**  $signalChannel_{down}$  **do**  
 ACTIVITYSETTOSTOPPING( $sl, act$ )

This rule is not defined within ACTIVITYSTATEMACHINE (R5.3.0-5) but outside of it since any activity has to react on *signalStop* regardless of its current internal state.

ACTIVITYSETTOSTOPPING (R5.5.3-12)

( $sl \in SubInstance, act \in Activity$ )  $\equiv$   
**set**  $activityState$  **from** *enabled, running, completing, faulted* **to** *stopping*

referenced in EXECUTEACTIVITY<sub>pattern</sub> (R5.9.0-18)

Applying ACTIVITYSETTOSTOPPING (R5.5.3-12) leads to the internal state *stopping*. In some cases, the rule needs to be refined. Being in *stopping*, a basic activity simply ends its execution and confirms with *signalStopped* sent to its parent activity. A structured activity has to propagate *signalStop* to each of its child activities which are running. Then it has to receive *signalStopped* from each of these childs before it may confirm its termination.

Since instances of activities are executed concurrently, it may happen that a parent activity sends *signalStop* to one of its child activities while the child activity just completes

its execution successfully. This means that the child activity doesn't need to preemptively terminate its execution. Therefore the parent activity will consider *signalCompleted* which is sent by the child activity as a confirmation for *signalStop* likewise.

#### 5.5.4 Stopping Activities

To finally stop the execution of an activity, set all outgoing links to *false*, i.e. let the link targets know that they won't be activated, set the activity's internal state to *stopped* and send *signalStopped* to its parent activity.

<u>STOPACTIVITY</u>	(R5.5.4-13)
$(sl \in SubInstance, act \in Activity_{basic}) \equiv$ <b>set</b> <i>activityState</i> <b>from</b> <i>stopping</i> <b>to</b> <i>stopped</i> <b>signal</b> <i>signalStopped</i> <b>to</b> <i>parentActivity(act)</i> <b>in</b> <i>sl</i> <b>via</b> <i>signalChannel<sub>up</sub></i> DISABLEOUTGOINGLINKS( <i>sl</i> , <i>act</i> )	

referenced in ACTIVITYSTATEMACHINE (R5.3.0-5), EXECUTEACTIVITY<sub>pattern,basic</sub> (R7.1.3-28)

This behaviour implements the link dynamics and the stop-concept.

## 5.6 Dead Path Elimination

Having defined the link dynamics (see sec. 5.4.2) in terms of function symbols and ASM rules, we silently disregarded a quite common case. As long as we have an enclosing flow, we may connect any two activities *A* and *B* with a link (assuming we do not create a cycle). From the link dynamics we can easily deduce that *A* has to reach *completing*, before *B* may enter *running*.

Now let *A* be in a branch of a switch activity *S*<sup>4</sup> and let *B* not be enclosed by *S*. If during the execution of *S* the branch with *A* inside is not chosen, *A* will never be executed, but *B* is waiting for a proper *linkStatus*. The mechanism to overcome this scenario is called *Dead Path Elimination (DPE)*.

Upon choosing its branch, *S* signals all of its other child activities which will *not* be executed to set the status of their outgoing links to *false*. This signal (*signalNegateLinks*) is propagated down the activity tree inside each branch, disregarding the internal state of the activities. The propagation stops when reaching while activities or event handlers, as links cannot cross these activities and hence there can be no *enabled* activity waiting for the determination of the statuses of its incoming links.

<u>INITDPE</u>	(R5.6.0-14)
$(sl \in SubInstance, act \in Activity, child \in Activity) \equiv$ <b>signal</b> <i>signalNegateLinks</i> <b>to</b> <i>child</i> <b>in</b> <i>sl</i> <b>via</b> <i>signalChannel<sub>down</sub></i>	

<sup>4</sup>Following the intuition, a branch is the subtree of the activity tree with its root being a child activity of the switch.

referenced in PROPAGATEDPE (R5.6.0-15)

Now, assuming  $B$  has valid values for each incoming link, it might happen that the  $linkTargetJoinCondition$  evaluates to  $false$ . Since an activity  $B$  may enter *running* only if the condition evaluates to  $true$  (see STARTACTIVITY (R5.4.3-8)), any activity which receives a link from  $B$  might get stuck as well. The BPEL specification provides two possible behaviors: To either throw a fault  $joinFailure \in FaultName$  or to set the status of its outgoing links to  $false$  as written above. Both cases are discriminated by a static property ( $activitySuppressJoinFailure$ ) of the concerned activity.<sup>5</sup>

$$activitySuppressJoinFailure : Activity \rightarrow \{true, false\}$$

A *Link* is not allowed to cross the boundaries of a while activity or of an event handler. This directly implies that the DPE is applicable only for instances of activities which share the same identifier  $sl \in SubInstance$ . Hence the following ASM rule.

PROPAGATEDPE (R5.6.0-15)  
 $(sl \in SubInstance, act \in Activity) \equiv$

```

onSignal  $signalNegateLinks$  from  $parentActivity(act)$ 
  in  $sl$  via  $signalChannel_{down}$  do
    DISABLEOUTGOINGLINKS( $sl, act$ )
    if  $act \in Activity_{structured} \setminus \{While, EventHandler\}$  then
      forall  $child \in childActivities(act)$  do
        INITDPE( $sl, act, child$ )
    otherwise
      if  $allLinksSet(sl, act) \wedge joinCondition.Satisfied(sl, act) = false$  then
        if  $activitySuppressJoinFailure(act) = true$  then
          DISABLEOUTGOINGLINKS( $sl, act$ )
        else
          THROW( $sl, act, joinFailure$ )

```

referenced in EXECUTEACTIVITY<sub>pattern</sub> (R5.9.0-18),

references INITDPE (R5.6.0-14)

## 5.7 Termination of a Process Instance

Apart from a proper handling of faults during the execution of a process instance by scopes, it is allowed to terminate an entire process instance – generally caused by the *Terminate* activity. This behaviour is not intended to be captured by a specific part of the process, it simply ends its execution. ‘Ending the execution’ is broken down to the activities: a process instance is terminated iff all of its activities are either *completed* or *stopped*.

<sup>5</sup>In case this property is not defined for an activity in the process definition, it inherits the value from the next enclosing activity with a defined value for this *suppressJoinFailure* property.

TERMINATEACTIVITY (R5.7.0-16)

( $sl \in SubInstance, act \in Activity$ )  $\equiv$

**set** *activityState* **from** *disabled, enabled, running, stopping, completing* **to** *stopped*

referenced in PROPAGATETERMINATE (R5.7.0-17)

The BPEL specification states that this should happen as fast as possible. In our distributed setting, we may realize this by implementing the “flooding” of the echo algorithm using the signal channels.<sup>6</sup> The dedicated signal is *signalTerminate* and can be sent upwards and downwards along the activity tree.

PROPAGATETERMINATE (R5.7.0-17)

( $sl \in SubInstance, act \in Activity$ )  $\equiv$

**onSignal** *signalTerminate* **from** *parentActivity(act)*

**in** *sl* **via** *signalChannel<sub>down</sub>* **do**

TERMINATEACTIVITY(*sl, act*)

**forall** ( $sl_{child}, child$ )  $\in$  *activityRunningChilds(sl, act)* **do**

**signal** *signalTerminate* **to** *child* **in**  $sl_{child}$  **via** *signalChannel<sub>down</sub>*

**select** ( $sl_{child}, child$ )  $\in$  *activityRunningChilds(sl, act)*

**where**  $signalTerminate \in signalChannel_{up}(sl_{child}, child, act)$  **in**

**onSignal** *signalTerminate* **from** *child* **in**  $sl_{child}$  **via** *signalChannel<sub>up</sub>* **do**

TERMINATEACTIVITY(*sl, act*)

**forall** ( $sl'_{child}, child'$ )  $\in$  *activityRunningChilds(sl, act)*

**where** ( $sl'_{child}, child'$ )  $\neq$  ( $sl_{child}, child$ ) **do**

**signal** *signalTerminate* **toAll** *child'* **in**  $sl'_{child}$  **via** *signalChannel<sub>down</sub>*

**signal** *signalTerminate* **to** *parentActivity(act)* **in** *sl* **via** *signalChannel<sub>up</sub>*

referenced in EXECUTEACTIVITY<sub>pattern</sub> (R5.9.0-18)

Relying on the correct definition of *runningChildActivities* (F5.4.5-1) for basic activities, this rule also suites the leaves of the activity tree.

To initiate the termination of a process instance, a basic activity has to send *signalTerminate* to its parent, see section 7.5 and RUNTERMINATE (R7.5.1-33).

## 5.8 Time

In the forthcoming chapters, we will require some notion of time to order incoming messages by their arrival or events by their occurrence. For the semantics of BPEL it suffices to consider *Time* as a total ordered set as this effectively the only relation we need for this

<sup>6</sup>We don't need the “echo” of the echo algorithm.

type.

**Universe:**  $Time$

Let  $(Time, +_{Time})$  be a semigroup with neutral element  $0_{Time} \in Time$  and let  $\leq_{Time}$  be a partial order on  $Time$  induced by  $+_{Time}$ .<sup>7</sup>

Our model cannot express the advancement of time, and it is also not required to do so. Instead we assume that the environment can tell us what the current time is ( $getCurrentTime$ ).

Similar to the link conditions met in section 5.4.2, we might want to compare moments in time given as strings ( $TimeExpression$ ) which have to be evaluated in the current subinstance ( $evaluateTimeExpression$ ). Again, we assume that the environment has the knowledge to do so.

**Universe:**  $TimeExpression$

**external functions**

$$\begin{aligned} getCurrentTime & : && ProcessInstance & \rightarrow & Time \\ evaluateTimeExpression & : && SubInstance \times TimeExpression & \rightarrow & Time \end{aligned}$$

## 5.9 Pattern for the ASM rules for Activities

The previous section of this chapter introduced all general concepts of activity-activity interaction: each BPEL activity makes use of these and all activities behave equally on these concepts. We define a general pattern which assembles all concepts according to the semantics of BPEL. It is a pattern since the concrete ASM rules of each activity are a refinement of this pattern. There will be no activity executing this pattern itself. Yet, it will complete our view on the general structure of BPEL's activities.

$\frac{\text{EXECUTEACTIVITY}_{pattern}}{(sl \in SubInstance, act \in Activity) \equiv}$ <p style="margin-left: 20px;">PROPAGATEDPE(<math>sl, act</math>)</p> <p style="margin-left: 20px;">PROPAGATETERMINATE(<math>sl, act</math>)</p> <p style="margin-left: 20px;"><b>onSignal</b> <math>signalStop</math> <b>from</b> <math>parentActivity(act)</math>  <span style="margin-left: 40px;"><b>in</b> <math>sI</math> <b>via</b> <math>signalChannel_{down}</math> <b>do</b></span>  <span style="margin-left: 40px;">ACTIVITYSETTOSTOPPING(<math>sl, act</math>)</span></p> <p style="margin-left: 20px;"><b>otherwise</b></p> <p style="margin-left: 40px;"><b>onSignal</b> <math>signalEnable</math> <b>from</b> <math>parentActivity(act)</math></p>	(R5.9.0-18)
--	-------------

<sup>7</sup>A suitable refinement for  $Time$  would be the natural numbers with the corresponding operation and relation.

```
    in sl via signalChanneldown do
    set activityState from disabled to enabled
PROPAGATEFAULTSACTIVITY(sl, act)
ACTIVITYSTATEMACHINE(sl, act)
```

referenced in RUNBPELACTIVITY<sub>pattern</sub> (R5.2.0-4)

references PROPAGATEDPE (R5.6.0-15), PROPAGATETERMINATE (R5.7.0-17),  
ACTIVITYSETTOSTOPPING (R5.5.3-12), PROPAGATEFAULTSACTIVITY (R5.5.2-11),  
ACTIVITYSTATEMACHINE (R5.3.0-5)

## 5.10 Summary & Conclusion

In the preceding sections of this chapter we defined structures and ASM rules to formalize the dynamics of the control-flow in BPEL. The definitions are based on the structures we identified in chapter 4, and on the informal specification [CGK<sup>+</sup>03].

We identified instances of activities to be the building blocks of a process instance. We furthermore have shown that we can properly represent an instance of an activity by using two simple structures, the activity tree and the subinstance tree. The actual control flow of a process is constituted by internal states of instances of activities and asynchronous communication between neighbored instances of activities.

The possible control flow is modular. Negative control flow is a conservative extension of the positive control flow. Additional functionalities like dead-path elimination or termination of a process instance are simple conservative extensions as well. We achieve this by a strict separation of static structures and dynamic properties. The use of ASM agents to execute instances of activities is necessary to provide an executable specification of the system. Nonetheless, the ASM agents are not a part of BPEL and may be replaced by another mechanism where this is applicable.

Thus, we defined a framework for executing distributed, reactive systems of which the key-components have modular functionalities. It is fairly easy to remove some behaviour or extend the model for a new kind of control flow. Although the definitions we have given in this chapter were derived from the informal description of BPEL, they are sufficiently independent of BPEL to be reused for specifying another distributed, reactive system.

In the following chapters of this report will define the functions and ASM rules which are specific for BPEL. We will denote the semantics of BPEL's activities as a refinement of the framework defined in this chapter.

## 6 Communication

The purpose of a BPEL process is modelling communication with other web services, which may be other BPEL processes, but don't have to be. Everything related to communication relies on existing technologies. Interfaces are defined using WSDL specifications. Messages are exchanged via SOAP using WS-Addressing. The operational BPEL semantics have to use the interfaces to these technologies.

### 6.1 Communication Structure

This section describes two types of interfaces used for communication with other web services. The first one is the external WSDL interface. The WSDL interface must be known to any other web service which wants to initiate a communication with the specified BPEL process. The second one is the internal interface. It specifies which communicating operations of the process logically correspond to each other. Furthermore, it provides symbols to abstract from a concrete remote web service with a concrete address, identifying each by its purpose for the business logic of the process.

#### 6.1.1 External Interface

The external interface is merely a data structure written in WSDL. It defines the syntax to be used in messages, that shall be received by the web service (or in our case BPEL process) which implements this interface.

A message is processed by an *operation* (**Universe:** *Operation*). Logically corresponding operations are grouped to *port types* (**Universe:** *PortType*, *pToperations*). The port types are associated with the process publishing the interface.

**Universe:** *PortType* (port type of a web service)

**Universe:** *Operation* (operation of a Web Service)

$$\begin{aligned} processPortTypes & : Process \rightarrow \mathcal{P}(PortType) \\ pTOperations & : PortType \rightarrow \mathcal{P}(Operations) \end{aligned}$$

A WSDL operation is specified by the types of messages it accepts and returns (*operationMessageCategories*). BPEL allows the implementation of two types of operations:

- to either just receive a message (*input* message) without returning an answer, or
- to receive (*input*) and reply with an answer (*output* message). In the latter case, it is allowed to return a fault message instead of the “normal” answer. For each type of fault message that might occur, a *FaultName* is given.

Additionally, each operation restricts the type of the message (**Universe:**  $MessageType$ ) it sends and receives ( $operationMessageType$ ), which of course may be different for each direction and fault.

**Universe:**  $MessageCategory = \{input, output\} \cup FaultName$  (kinds of message exchange supported by an operation)

**Universe:**  $MessageType$  (structured types for messages)

$$\begin{aligned} operationMessageCategories & : & Operation & \rightarrow \mathcal{P}(MessageCategory) \\ operationMessageType & : & Operation \times MessageCategory & \rightarrow \mathcal{P}(MessageType) \end{aligned}$$

Obviously, for each operation defined in this interface, the possessing BPEL process has to have activities implementing it.

### 6.1.2 Internal Interface

When talking about communication, there is always a sender and a receiver involved. Neither is (or can be) expressed by the types defined in the previous subsection, only. In BPEL sender and receiver of a message are web services.

Just as a BPEL process publishes its external WSDL interface for communication, each BPEL process knows the external interface of the remote web services it intends to communicate with. In other words: any remote web service a given (“local”) BPEL process wants to communicate with has to implement the interface declaration known to the “local” BPEL process.

Using this paradigm, BPEL distinguishes remote web services in terms of their functionality. The functionality is expressed in terms of communicative capabilities: the business logic requires a partner to offer functionality via specific WSDL operations and to use “local” functionality via specific WSDL operations.

Hence, we obtain a data type relating parts of interfaces: a *partner link type* is a pair of port types, where each component has a name describing the relation of this link. The name is called *role* putting a perspective on this relation. One role is played by the “local” process, the other role is played by the remote web service. The partner link type does not define which of the roles is played by whom.

**Universe:**  $Role$  (symbol to logically abstract from a concrete  $portType$ )

**Universe:**  $RoleType =_{def} \{myRole, partnerRole\}$  (distinguish the local and the remote process)

**Universe:**  $PartnerLinkType$  (data type relating two port types)

$$\begin{aligned} partnerLinkTypeRoles & : & PartnerLinkType & \rightarrow (Role \times Role) \\ rolePortType & : & Role & \rightarrow PortType \end{aligned}$$

A *partner link* is a typed symbol (of type  $PartnerLinkType$ ). It has two purposes: Firstly, a partner link defines which role is played by the “local” process and which one by the remote web service. Secondly, during the execution of process instance, a partner link is bound to a concrete remote web service fulfilling the requirement of the partner link

type: i.e. the remote web service calls the specified local interface and provides the required remote operations. The binding is formalized in section 6.4.3.

**Universe:** *PartnerLink* (symbol to connect two logically corresponding *Roles*)

$$\begin{aligned} \textit{partnerLinkRole} & : \textit{PartnerLink} \times \textit{RoleType} \rightarrow \textit{Role} \\ \textit{processPartnerLinks} & : \textit{Process} \rightarrow \mathcal{P}(\textit{PartnerLink}) \end{aligned}$$

A partner link may leave its definition for one *RoleType* undefined. This means that remote web service is either not required to call any local operations, or the other way around that the “local” process doesn’t call any remote operations.

Hence a partner link is a symbol for the logical abstraction of concrete remote web services and a BPEL process can send and receive messages in terms of partner links instead of addresses. Obviously, the information of partner links is neither required nor useful outside the scope of the very process. We need a translating mechanism which is formalized in section 6.4.4.

Sometimes, the characterization of remote web services by pairs of port types doesn’t suffice as business logic may involve a number of different port types. A *partner* is a set of partner links. Its definition requires that if any of these partner links is bound to a concrete web service, then all of them are bound to the same: The remote web service has to fulfil all requirements.

**Universe:** *Partner* (symbol to logically abstract from a concrete partner in terms of its interface)

$$\begin{aligned} \textit{partnerPLinks} & : \textit{Partner} \rightarrow \mathcal{P}(\textit{PartnerLink}) \\ \textit{processPartners} & : \textit{Process} \rightarrow \mathcal{P}(\textit{Partner}) \end{aligned}$$

## 6.2 Messages

We already defined the static parts of the communication in section 6.1.1 and 6.1.2. To actually exchange messages we need a real message. A message is generated during runtime. Each message is unique. Hence for each message which will ever be sent or received during the execution a BPEL process, we need a unique object from an infinite universe.

**Universe:** *BPELMsg* (concrete message)

The very existence of a message, doesn’t define its content. As mentioned previously, the content of a message is typed by a *MessageType*. A *MessageType* is a non-trivial, structured type itself. Each *MessageType* defines *message type parts*, each being a simple or complex structured type again.

**Universe:** *MessageType* (structured type of a message)

**Universe:** *MessageTypePart* (substructure of the *MessageType*)

$$\begin{aligned} msgMsgType & : \quad \text{BPELMsg} \rightarrow \text{MessageType} \\ messageTypeParts & : \quad \text{MessageType} \rightarrow \mathcal{P}(\text{MessageTypePart}) \end{aligned}$$

In BPEL the level of *MessageTypePart* is the finest level explicitly defining and referencing structured types. Any further substructures are defined by XML-attributes (read: strings) referencing XSD-declarations. We consider these strings on an abstract level whenever we need to, defining abstract functions similar to the boolean conditions of *Links* (cf. section 5.4.2).

Hence, the contents of a message cannot be expressed in atomic values like integers or strings. For the operational semantics of BPEL it suffices to use abstract *values*. Each value, atomic or structured is expressed by a unique element of an infinite **Universe**: *Value*. Terms interpreted as values are interpreted equally if and only if they are interpreted as the same element in *Value*.

**Universe**: *Value* (abstraction from values)

$$msgPartValue : \text{BPELMsg} \times \text{MessageTypePart} \rightarrow \text{Value}$$

Once again, if we need to explore the substructure of a value, we use abstract functions. These abstract functions constitute the interface to some technology which defines the correct semantics in the corresponding domain.

*Faults* as defined in section 5.5 may be exchanged between two web services, as one can already guess from the definition of *operationMessageCategories*. Thus a *Fault* is a special BPELMsg.

**Universe**: *Fault*  $\subset$  BPELMsg

### 6.3 Values of Variables

In section 4.2.4 we already formalized the declaration of a *Variable*. Now we define the *value* of a *Variable*.

Although a *Variable* constitutes a structural entity on its own, the value of a *Variable* *var* is nothing else but a public runtime property of the *Scope* *sc* at which it is declared. That is, the value of *var* at a given instance (*sl*, *sc*) of its scope is public to any activity which belongs to the same subinstance and which is enclosed by *sc*. The value of *var* in this subinstance must be function of *sl*.

Furthermore, we need to respect the knowledge about the structure of the variable's values as they are given by its type. We may limit our formalization to those types (and their substructures) that are explicitly known in BPEL: We formalized two different kinds of types, *MessageType* and *XMLschemaType*. For the *MessageType* we know that it consists of a set of *MessageTypeParts* (see previous section 6.2). For an *XMLschemaType*, we do not know anything. Hence the explicitly known structures are

**Universe**:  $\text{VariableTypePart} = \text{MessageType} \cup \text{MessageTypePart} \cup \text{XMLschemaType}$ .

The value of a variable is then given by the values of all of its type's explicitly known substructures. These mappings are formalized by the following function:

$$\text{variablePartValue} : \text{SubInstance} \times \text{Variable} \times \text{VariableTypePart} \rightarrow \text{Value}$$

Considering abstract *Values* only, we must assume that in any state, for a variable of type *MessageType*, its *variablePartValue* for the *MessageType* and the *variablePartValues* for each of its *MessageTypeParts* are consistent, i.e. a modification of the value of one the parts yields a different value for the entire structure.

Consider an activity ( $sl', act'$ ) which wants to access a variable *var*'s value at its scope ( $sl, sc$ ) where  $sl \neq sl'$ . Following the distributed approach, the identifier *sl* must be evaluated from ( $sl', act'$ ) just by knowing *var* and  $sc = \text{variableScope}(\text{var})$  in turn. The derived function *currentSubInstanceFromInner* (D B.2.3-1) will do so and we will apply this function wherever necessary.

## 6.4 Correlating Messages to Process Instances

The external interface of a BPEL process is defined by a WSDL document. A process is execute in instances, hence messages are sent and received by instances. Yet the interface doesn't provide any structural information on how to address an instance.

Instead, in the field of BPEL this issue is handled by two mechanisms: the *correlation handling* defined within the language itself and an extension of a message addressing specification *WS-Addressing* outside of BPEL. In both cases the process needs to map an incoming message to a process instance, hence the language has to handle both of them. We formalize the correlation handling in the next two subsections followed by a formalization of WS-Addressing and its relation to BPEL.

### 6.4.1 Message Properties

In section 6.1.2 we formalized how a communication partner is defined in terms of the underlying business logic. To associate messages by its contents to a concrete process instance, BPEL defines a similar approach to declare a number of types which are required to create an actual mapping in an execution.

The contents of a message is typed by a structured data-type (cf. sec. 6.2). Some field of such a data-type may be sufficient to identify a message as corresponding to a specific instance, similar to the primary-key concept in relational databases. A unique customer-ID might server as an example.

*Message Properties* are named data-types, annotating the pure structure with a name of the business logic of the process.

```
<bpws:property name="ncname" type="qname"/>
```

**Universe:** *MessageProperty* (name for a typed substructure of a *MessageType*)

*Property Aliases* relate message properties and substructures of a *MessageType*, i.e. it gives a specific substructure of a data type a name within the business logic. This

substructure needn't to be a *MessageTypePart* but can be a substructure of it. As these finer data types are not defined in BPEL, but referenced by a string describing a path in the structured data type, we introduce a new **Universe**: *QueryPath* for each such string. The remaining properties are translated straight forward.

```
<bps:propertyAlias propertyName="qname" messageType="qname"
    part="ncname" query="queryString"/>
```

**Universe**: *PropertyAlias* (link abstracting name and definition of substructure in a *MessageType*)

**Universe**: *QueryPath* (path identifying the substructure)

$$\begin{aligned} \text{aliasMessageType} & : \text{PropertyAlias} \rightarrow \text{MessageType} \\ \text{aliasMessagePart} & : \text{PropertyAlias} \rightarrow \text{MessageTypePart} \\ \text{aliasQueryPath} & : \text{PropertyAlias} \rightarrow \text{QueryPath} \\ \text{aliasMessageProperty} & : \text{PropertyAlias} \rightarrow \text{MessageProperty} \end{aligned}$$

By a *PropertyAlias*, we may identify a specific substructure of an incoming or outgoing message in terms of the business logic.

#### 6.4.2 Correlation Handling

The *Correlation Handling* is a mechanism which constrains the contents of messages which may be sent or received by a process instance. If no two process instances of a BPEL process define the same constraints, a message can unambiguously be mapped to a process instance by the help of these constraints.

The constraints are expressed as sets of *PropertyAliases*. Each such set is called a *correlation set* (using such a set, a message is correlated to a process instance).

**Universe**: *CorrelationSet* (set of *PropertyAliases*)

$$\text{correlationProperties} : \text{CorrelationSet} \rightarrow \mathcal{P}(\text{PropertyAlias})$$

Correlation sets are defined within scopes. A correlation set is defined for exactly one scope.<sup>1</sup> The scope's structure is "enriched" with the correlation properties being referenced in the correlation set: For each property, the scope may hold a value in each *SubInstance* it is executed (*correlationPropertyValue*).

$$\begin{aligned} \text{scopeCorrelationSets} & : \text{Scope} \rightarrow \mathcal{P}(\text{CorrelationSet}) \\ \text{correlationSetScope} & : \text{CorrelationSet} \rightarrow \text{Scope} \end{aligned}$$

$$\text{correlationPropertyValue} : \text{SubInstance} \times \text{Scope} \times \text{PropertyAlias} \rightarrow \text{Value}$$

The following abstract function *messagePartSubValue* allows us to explore subvalues of a value typed with a structured data type. Assuming that the first parameter has the type of the second parameter and that the third parameter specifies a valid substructure within the type, the function returns the corresponding subvalue within the given value.

<sup>1</sup>We abstract from BPEL's overloading mechanism for correlation sets with equal names, assuming an unambiguous representation in the initial state.

$A \text{ messagePartSubValue} : Values \times Message\text{TypePart} \times Query\text{Part} \rightarrow Value$

The correctness of the values of a correlation set is always checked by an activity  $A$  inside the scope they are defined at.<sup>2</sup> Executing  $A$ , we always have a fixed  $SubInstance$   $sI$  for which the correctness needs to be checked. From  $A$  and  $sI$  upwards we can easily determine the scope and its correct  $SubInstance$   $sI_{scope}$  to obtain the current valid value: The scope is given by  $correlationSetScope$ ;  $sI_{scope}$  lies on the  $sI$ - $processInstance(sI)$ -path in the sub instance tree (see 5.1.3). Hence, we may define the following functions and rules.

$\mathcal{D} \text{ correlationSatisfied} : SubInstance \times Scope \times PropertyAlias \times Value \rightarrow \{true, false\}$

returns *true* iff the given property alias has the given value in the given sub instance and scope where the property is referenced from (see Appendix B.3.1).

$\mathcal{D} \text{ correlationSatisfied}_{msg} : SubInstance \times BPELMsg \times \mathcal{P}(CorrelationSet) \rightarrow \{true, false\}$

returns *true* iff the contents of the given message correlates in the given sub instance on the given correlation sets (see Appendix B.3.1).

$\mathcal{D} \text{ correlationSatisfied}_{var} : SubInstance \times Variable \times \mathcal{P}(CorrelationSet) \rightarrow \{true, false\}$

returns *true* iff the contents of the given variable correlates in the given sub instance on the given correlation sets (see Appendix B.3.1).

We use two rules to initialize the correlation properties of an instance: CORRELATEINSTANCETOMESSAGE (RB.3.1-108) with arguments ( $sl \in SubInstance, act \in Activity, msg \in BPELMsg, CS \in \mathcal{P}(CorrelationSet), CS_{init} \in \mathcal{P}(CorrelationSet)$ ) sets all uninitialized properties that are defined in a correlation set within  $CS_{init}$  in the possessing scope and the corresponding  $SubInstance$  (determinable by  $sl$ ) to the corresponding values of  $msg$ , if  $msg$  correlates with  $sl$  on  $CS$ . CORRELATEINSTANCETOVARIABLE (RB.3.1-109) does the same for a given variable (see Appendix B.3.1).

### 6.4.3 Services and Endpoint References

(BPEL:7.4) and (WS-Addressing)

This section shows how a BPEL process instance is linked to its environment and to other web services and their (possible) instances.

With regards to communicative aspects, processes are being encapsulated by services which have a physical address. Thus messages are exchanged via ports which are unambiguously determined by the address, the port type and the operation.

**Universe:** Service (name of a service)

**Universe:** Address (abstract from addresses)

$$\begin{aligned} processService & : Process \rightarrow Service \\ serviceAddress & : Service \rightarrow Address \end{aligned}$$

<sup>2</sup>Otherwise, we wouldn't have the definition of the correlation set available at this activity.

Besides the Correlation Handling, instances of web service may communicate directly if they can identify their partner of communication.<sup>3</sup> The *Address* is not sufficient, since it identifies the process, but not its instances. Current developments of web service technologies employs another mechanism: each process instance gains a globally unique identifier, the *EndpointIdentification*. To exchange these and to relate them to a physical address, they are stored within an *EndpointReference*.

**Universe:** *EndpointReference*

**Universe:** *EndpointIdentification*

$$\begin{aligned} \textit{endpointAddress} & : \textit{EndpointReference} \rightarrow \textit{Address} \\ \textit{endpointService} & : \textit{EndpointReference} \rightarrow \textit{Service} \\ \textit{endpointInstanceID} & : \textit{EndpointReference} \rightarrow \textit{EndpointIdentification} \end{aligned}$$

The *EndpointIdentification* is a generalized term for BPEL's *ProcessInstance* and we will read it as such.<sup>4</sup>

A BPEL process has a symbol for each possible communication involving the current process and a remote process. This symbol is bound to an *EndpointReference* upon the first communication between both. Using the symbol, a process instance can refer to and communicate with its opposite without having to know its physical address or instance. However, this mapping has to be stored inside the process instance:

$$\begin{aligned} \textit{partnerLinkRoleEndpoint} & : \\ & \textit{ProcessInstance} \times \textit{PartnerLink} \times \textit{Role} \rightarrow \textit{EndpointReference} \end{aligned}$$

#### 6.4.4 Message Addressing with WS-Addressing

Although BPEL talks about communication and partners in terms of symbols, it is meant to execute it within existing technologies. For this purpose, the WS-Addressing standard provides the required definitions to bridge the gap. We will define the structures required for our purposes and explain shortly how they correspond to the structures in BPEL.

Each message is assigned a sender in the form of an *EndpointReference*. Additionally WS-Addressing supports replying and sending of faults to different processes. BPEL makes no use of this feature, but we have to fill the entire structure.

$$\begin{aligned} \textit{msgSourceReference} & : \textit{BPELMsg} \rightarrow \textit{EndpointReference} \\ \textit{msgReplyReference} & : \textit{BPELMsg} \rightarrow \textit{EndpointReference} \\ \textit{msgFaultReference} & : \textit{BPELMsg} \rightarrow \textit{EndpointReference} \end{aligned}$$

The specification of a receiver of the message is not encapsulated in an *EndpointReference*. This becomes useful when speaking about assigning a message

---

<sup>3</sup>A web service doesn't require a specific technology to be implemented. Hence it doesn't have to define a mechanism similar to BPEL's correlation handling. Yet instance-to-instance communication might be necessary.

<sup>4</sup>A proper usage of both in the interaction of WS-Addressing and BPEL is not defined yet.

to a concrete process instance. First of all, a message is delivered to an *Address*.

$$\text{msgDestination} : \text{BPELMsg} \rightarrow \text{Address}$$

Furthermore, a message also describes which operation and portType is meant to receive it. WS-Addressing provides some variable fields to so: The following functions together specify the `<action .../>` tag of the SOAP message identifying the addressed *PortType* and *Operation*.

$$\begin{aligned} \text{msgActionPortType} &: \text{BPELMsg} \rightarrow \text{PortType} \\ \text{msgActionOperation} &: \text{BPELMsg} \rightarrow \text{Operation} \end{aligned}$$

Messages may be sent in relation to previous messages or operations. BPEL uses just one type of relation in WS-Addressing to identify a message as a reply to a previously sent message. The reason is, that a reply is not received via an operation of the receiver, but is the result of an operation of the partner.

**Universe:** *RelationShip*

*replyRelation*  $\in$  *RelationShip* identifies the “unspecified”-URI for reply messages.

$$\text{msgRelationShip} : \text{BPELMsg} \rightarrow \mathcal{P}(\text{RelationShip})$$

The specification of the receiving instance is left open in WS-Addressing. Although using the *EndpointIdentification*, we cannot assume that the object (or string) which globally identifies an process instance, is also used locally. Hence the translation depends on the specific web service. WS-Addressing allows optional tags for this purpose. The following function is a logical consequence of the extensibility of the WS-Addressing mechanism and the requirements of BPEL: The SOAP message needs a tag identifying the addressed *ProcessInstance* whenever the sender intends this. As mentioned, we read *ProcessInstances* as *EndpointIdentifications*.

$$\text{msgDestinationInstance} : \text{BPELMsg} \rightarrow \text{EndpointIdentification}$$

As shown, BPEL offers a number of views on its communication endpoints and communication channels. The design of BPEL and WS-Addressing allows us to combine all of them in a layered model. On top of the model the activities access logical endpoints which unambiguously identify the partner the process instances communicates with. The model is founded on the detailed addressing and referencing as defined in WS-Addressing, specifying addresses and ports.

## 6.5 Communication Endpoints

The previous sections of this chapter formalized the aspects of identifying sender and receiver of messages. We will now formalize the structures which allow us to send and receive messages, i.e. we need the site that has the address the message is sent to.

Knowing that we can send any message between processes using existing technology, we can abstract from these mechanisms and assume logical communication endpoints which are accessible to BPEL activities.

### 6.5.1 Linking Internal and External Interface

The description of the external interface (cf. sec. 6.1.1) and of the internal interface (cf. sec. 6.1.2) are independent. When it comes to the exchange of messages, both need to be put together in order to properly specify in which way messages are exchanged.

In BPEL, this is done at the activity which executes this communication: each communicating activity declares a partner link, a port type and an operation to link internal interface and external interface. Additionally, each such activity references a variable which stores the data of the message within the process. And this activity references the correlation sets which the messages need to satisfy.

This description altogether suffices to specify the way a message is received or sent. Therefore we introduce a new structural element, the *port descriptor* which joins this description and which gives us an object at hand to just talk about the communication and nothing else.

**Universe:** *PortDescriptor* (symbol to subsume the description of the external and the internal interface required for communication)

<i>portPartnerLink</i>	:	<i>PortDescriptor</i>	→	<i>PartnerLink</i>
<i>portPortType</i>	:	<i>PortDescriptor</i>	→	<i>PortType</i>
<i>portOperation</i>	:	<i>PortDescriptor</i>	→	<i>Operation</i>
<i>portVariable</i>	:	<i>PortDescriptor</i>	→	<i>Variable</i>
<i>portCorrelationSets</i>	:	<i>PortDescriptor</i>	→	$\mathcal{P}(\text{CorrelationSet})$
<i>portInitCorrelationSet</i>	:	<i>PortDescriptor</i> × <i>CorrelationSet</i>	→	{ <i>true</i> , <i>false</i> }

### 6.5.2 Communication Endpoints for Activities

From an activity's point of view, a message is passed via a concrete *PartnerLink* over a port which has a *PortType* and an *Operation* and which is bound to a specific *ProcessInstance*.

The *localPort\** functions describe these ports "locally" provided by the activity's process.

<i>localPort<sub>in</sub></i>	:	<i>ProcessInstance</i> × <i>PartnerLink</i> × <i>PortType</i> × <i>Operation</i>	→	$\mathcal{P}(\text{BPELMsg})$
<i>localPort<sub>out</sub></i>	:	<i>ProcessInstance</i> × <i>PartnerLink</i> × <i>PortType</i> × <i>Operation</i>	→	$\mathcal{P}(\text{BPELMsg})$
<i>localPort<sub>fault</sub></i>	:	<i>ProcessInstance</i> × <i>PartnerLink</i> × <i>PortType</i> × <i>Operation</i>	→	$\mathcal{P}(\text{Fault})$

To send messages to a remote partner, we have to send messages to *its* port. Reply-ing messages is done using *our* ports, i.e. by putting a message into the corresponding *localPort<sub>out</sub>* or *localPort<sub>fault</sub>*. Invoking messages is different as the invoking, "local" process doesn't provide the ports. Instead, the ports are provided by the partner. To clearly distinguish the process' ports and the partner's ports, we define the *remotePort\** functions.

<i>remotePort<sub>in</sub></i>	:	<i>ProcessInstance</i> × <i>PartnerLink</i> × <i>PortType</i> × <i>Operation</i>	→	$\mathcal{P}(\text{BPELMsg})$
<i>remotePort<sub>out</sub></i>	:	<i>ProcessInstance</i> × <i>PartnerLink</i> × <i>PortType</i> × <i>Operation</i>	→	$\mathcal{P}(\text{BPELMsg})$
<i>remotePort<sub>fault</sub></i>	:	<i>ProcessInstance</i> × <i>PartnerLink</i> × <i>PortType</i> × <i>Operation</i>	→	$\mathcal{P}(\text{Fault})$

From the activity's point of view, we assume that any message that is put in a *remotePort<sub>in</sub>* is transferred to the determined endpoint, and any message that is replied from said endpoint finally arrives in *remotePort<sub>out</sub>* or *remotePort<sub>fault</sub>*. See the definition of the inbox and outbox managers in section 6.6.

### 6.5.3 ASM-Rules to Send and Receive Messages

Any activity which receives a message or which sends a message executes some principle operational steps on the structures of the communication endpoints and the port descriptors. In this section we will present these principle operational steps in ASM rules. Any activity which supports communication will use a refined version of the rules presented herein.

Consider the dynamic function

$$port : ProcessInstance \times PartnerLink \times PortType \times Operation \rightarrow \mathcal{P}(BPELMsg)$$

to be a general in- and outbox for any kind of message exchange. It will be replaced by the proper functions upon refining the abstract ASM rules for sending and receiving messages.

#### Receiving Messages

A receiving activity implements an operation of the BPEL process. It waits for a message to arrive at its local communication endpoint *port* (see above), specified by a port descriptor. If a message is found in that local port it is received. In case the message satisfies the correlation properties, its contents is copied to a variable, otherwise a fault is thrown. In case a correlation property referenced by the port descriptor hasn't been set for this process instance yet, the properties are set by the contents of the message (cf. sec. 6.4.2 (Correlation Handling)).

The following rule describes the most abstract behaviour of receiving a message, based on the structures we have defined so far.

$$\frac{\text{AWAITANDRECEIVECORRELATINGMESSAGE}_{pattern}}{(sl \in SubInstance, act \in Activity, portdescr \in PortDescriptor)} \equiv \quad (R6.5.3-19)$$

```

let partnerL = portPartnerLink(portdescr),
    pT = portPortType(portdescr),
    op = portOperation(portdescr),
    pl = processInstance(sl) in
select msg ∈ port(pl, partnerL, pT, op) in
    let CS = portCorrelationSet(portdescr) in
        if correlationSatisfied(sl, msg, CS) then
            remove msg from port(pl, partnerL, pT, op)
            RECEIVEMESSAGEpattern(sl, act, portdescr, msg)
        else
            THROW(sl, act, correlationViolation)

```

references RECEIVEMESSAGE<sub>pattern</sub> (R6.5.3-20), THROW (R5.5.1-10)

By refining AWAITANDRECEIVEMESSAGE<sub>pattern</sub> for each receiving activity we will also replace *port* by the correct functions defined in Section 6.5.2.

Having received a message, its contents needs to be stored and the correlation handling mechanism needs to be run. The most general behaviour of this kind is defined in the following ASM rule.

$$\frac{\text{RECEIVEMESSAGE}_{\text{pattern}}}{(\text{sl} \in \text{SubInstance}, \text{act} \in \text{Activity}, \text{portdescr} \in \text{PortDescriptor}, \text{msg} \in \text{BPELMsg})} \equiv \quad (\text{R6.5.3-20})$$

```

let var = portVariable(portdescr) in
  COPYMSGTOVARIABLE(sl, msg, var)
let CS = portCorrelationSet(portdescr),
  CSinit = {cs ∈ CS | portInitCorrelationSet(portdescr, cs) = true} in
  CORRELATEINSTANCETOMESSAGE(sl, act, msg, CS, CSinit)

```

referenced in AWAITANDRECEIVECORRELATINGMESSAGE<sub>pattern</sub> (R6.5.3-19),  
 AWAITANDRECEIVECORRELATINGMESSAGE<sub>pick</sub> (RB.5.5-135),  
 references COPYMSGTOVARIABLE (RB.3.2-110), CORRELATEINSTANCETOMESSAGE  
 (RB.3.1-108)

### Sending Messages

A sending activity creates a new message. The contents of the message is copied from the *Reply*'s variable. The contents of the variable has to satisfy the correlation properties of the process instance, otherwise a fault is thrown. If the correlation properties are satisfied, the message is sent. In case a correlation property referenced by the sending activity hasn't been set for this process instance yet, the properties are set by the contents of the variable (cf. sec. 6.4.2 (Correlation Handling)).

The following rule describes the most abstract behaviour of sending a message, based on the structures we have defined so far.

$$\frac{\text{GENERATEANDSENDCORRELATINGMESSAGE}_{\text{pattern}}}{(\text{sl} \in \text{SubInstance}, \text{act} \in \text{Activity}, \text{portdescr} \in \text{PortDescriptor})} \equiv \quad (\text{R6.5.3-21})$$

```

let var = portVariable(portdescr),
  CS = portCorrelationSets(portdescr) in
  if correlationSatisfiedvar(sl, var, CS) = true then
    GENERATEANDSENDMESSAGEpattern(sl, act, portdescr)
  else
    THROW(sl, act, correlationViolation)

```

references GENERATEANDSENDMESSAGE<sub>pattern</sub> (R6.5.3-22)

Having verified that the data that is about to be sent satisfies the correlation sets, a new message is generated. The contents of the variable is copied to the new message and the correlation handling is run. Finally, the message is sent to its destination.

---

$\frac{\text{GENERATEANDSENDMESSAGE}_{pattern}}{\quad}$  (R6.5.3-22)

$(sl \in SubInstance, act \in Activity, portdescr \in PortDescriptor) \equiv$

```

let message = new(BPELMsg) in
  let var = portVariable(portdescr) in
    COPYVARIABLETOMSG(sl, var, message)
  let CS = portCorrelationSets(portdescr),
      CSinit = {cs ∈ CS | portInitCorrelationSet(portdescr, cs) = true} in
    CORRELATEINSTANCETOVARIABLE(sl, act, var, CS, CSinit)
  SENDMESSAGEpattern(sl, portdescr, message)

```

referenced in GENERATEANDSENDCORRELATINGMESSAGE<sub>pattern</sub> (R6.5.3-21),  
 references COPYVARIABLETOMSG (RB.3.2-111), CORRELATEINSTANCETOVARIABLE  
 (RB.3.1-109), SENDMESSAGE<sub>pattern</sub> (R6.5.3-23)

Sending a message is simply done by putting it into the appropriate port.

$\frac{\text{SENDMESSAGE}_{pattern}}{\quad}$  (R6.5.3-23)

$(sl \in SubInstance, portdescr \in PortDescriptor, msg \in BPELMsg) \equiv$

```

let partnerL = portPartnerLink(portdescr),
    pT = portPortType(portdescr),
    op = portOperation(portdescr),
    pl = processInstance(sl) in
  add msg to port(pl, partnerL, pT, op)

```

referenced in GENERATEANDSENDMESSAGE<sub>pattern</sub> (R6.5.3-22)

## 6.6 Inbox-, Outbox- and Instance-Manager

The following section includes an abstract definition of the *inbox-manager* which receives messages, and matches them with running instances, and causes the creation of a new process instance whenever required. Further we define an *outbox-manager* of a process which collects the messages of all running instances and forwards them to the network. Both definitions are an adaption of the semantics in [FGV05] to suit the semantics defined in this report. Finally, we define an *instance-manager* whose sole responsibility is to handle signals sent from any instance of the process' root activity.

Each of these managers is an entity which operates at the level of the process and not at the level of a process instance. The effects of these managers is to make the existence of process instances transparent to the environment of the process.

**Universe:** *ProcessManager* (representing an entity capable of performing actions on behalf of a set of process instances)

The behaviour of a *ProcessManager* is triggered by an ASM agent.

$$myManager : Agent \rightarrow ProcessManager$$

### 6.6.1 Inbox-Manager

At first, we require the existence of some entity in the initial state of the ASM which is responsible for forwarding message on the behalf of a BPEL process.

$$\begin{aligned} managerProcess & : ProcessManager \rightarrow Process \\ processInboxManager & : Process \rightarrow ProcessManager \\ processInbox & : Process \rightarrow \mathcal{P}(BPELMsg) \end{aligned}$$

requirement for the initial state (6.6.1-1)

$$\begin{aligned} \forall process \in Process \exists inboxmanager \in ProcessManager : \\ & processInboxManager(process) = inboxmanager \\ & \wedge managerProcess(inboxmanager) = process \\ & \wedge processInbox(process) = \emptyset \end{aligned}$$

Obeying the vague description of the inbox manager's behaviour, we introduce an abstract function which includes all information to solve the message matching of BPEL.

$$A msgMatchesInstance : BPELMsg \times Process \times ProcessInstance \rightarrow \{true, false\}$$

$msgMatchesInstance(msg, process, pl)$  returns *true* iff the message *msg* matches the process instance *pl* of *process* by the semantics of WS-Addressing and correlation handling.

Each message gets a timestamp (see Time, sec. 5.8) upon receipt (see Pick activity, sec. 8.5).

$$messageReceiveTime : BPELMsg \rightarrow Time$$

The inbox manager checks the inbox of the process for messages and assigns a message to the appropriate process instance. If no appropriate process instance is found, the message is assigned to a fresh instance. Following the description of the requirements of instantiation, we agree to Farahbod et al. and assume that the system holds a fresh process instance to receive the message which "causes" its creation. Additionally, there must be a formal representation of the set of running process instances.

$$\begin{aligned} processWaitingInstance & : Process \rightarrow ProcessInstance \\ processRunningInstances & : Process \rightarrow \mathcal{P}(ProcessInstance) \end{aligned}$$

requirement for the initial state (6.6.1-2)

$$\begin{aligned} \forall process \in Process : \\ & processRunningInstances(process) = \emptyset \\ & \wedge \exists pl \in ProcessInstance : processWaitingInstance(process) = pl \end{aligned}$$

Upon assigning the message to the waiting instance, we need to create a new instance and initialize its behaviour. This is done by adding *signalEnable* to its incoming *signalChannel<sub>down</sub>*.

RUNINBOXMANAGER (R6.6.1-24)  
 (self)  $\equiv$

```

let msgManager = myManager(self),
    process = managerProcess(self) in
select msg  $\in$  processInbox(process) in
  select pl  $\in$  processRunningInstances(process)
    where msgMatchesInstance(msg, process, pl) = true in
      messageReceiveTime(msg) := getCurrentTime(pl)
      ASSIGNMESSAGE TO INSTANCE(msg, process, pl)
  ifnone
    let pl = processWaitingInstance(process) in
      messageReceiveTime(msg) := getCurrentTime(pl)
      ASSIGNMESSAGE TO INSTANCE(msg, process, pl)
      add pl to processRunningInstances(process)

    let plnew = new(ProcessInstance) in
      processWaitingInstance(process) := plnew
      let scope = processScope(process) in
        add signalEnable to signalChanneldown(plnew, scope, scope)
  
```

references ASSIGNMESSAGE TO INSTANCE

We assume that ASSIGNMESSAGE TO INSTANCE distributes the chosen message to the appropriate *localPort<sub>in</sub>* if this message is sent to an operation provided by the process. In case the message is a reply to a previously sent message, it shall be passed to the appropriate *remotePort<sub>out</sub>* or *remotePort<sub>fault</sub>* (c.f. sec. 6.5.2). The distribution is subject to correlation handling (cf. sec. 6.4.2) and WS-Addressing (cf. sec. 6.4.4).

The set of running instances also needs to be update if a process instance completes its execution. The necessary ASM rules are defined at the scope in section 9.2.5.

Please note that this formalization strictly requires that the next message which causes the creation of a new instance may be received only, after the control flow of the waiting instance has reached a state in which it cannot proceed without receiving an instantiating message.

### 6.6.2 Outbox-Manager

Similar to the inbox manager, we define an abstract version of the outbox manager by the help of the following abstract function

*A availablePorts* :  
 $Process \rightarrow \mathcal{P}(ProcessInstance \times PartnerLink \times PortType \times Operation)$

where  $availablePorts(process)$  is the set of all tuples  $\mathbf{p} = (p_l, p_L, p_T, o_p)$  such that  $localPort_{out}(\mathbf{p})$ , or  $localPort_{fault}(\mathbf{p})$ , or  $remotePort_{in}(\mathbf{p})$  are used by an activity of  $process$  to send a message. Then the behaviour of the outbox manager is straight forward.

$$\begin{aligned} processOutbox & : Process \rightarrow \mathcal{P}(BPELMsg) \\ processOutboxManager & : Process \rightarrow ProcessManager \end{aligned}$$

requirement for the initial state (6.6.2-1)

$$\begin{aligned} \forall process \in Process \exists outboxmanager \in SystemManager : \\ & processOutboxManager(process) = outboxmanager \\ & \wedge managerProcess(outboxmanager) = process \\ & \wedge processOutbox(process) = \emptyset \end{aligned}$$

RUNOUTBOXMANAGER

(R6.6.2-25)

(self)  $\equiv$

```

let msgManager = myManager(self),
    process = managerProcess(self) in
for all p  $\in$  availablePorts(process) do
  for all msg  $\in$  localPortout(p) do
    remove msg from localPortout(p)
    add msg to processOutbox(process)

  for all msg  $\in$  localPortfault(p) do
    remove msg from localPortfault(p)
    add msg to processOutbox(process)

  for all msg  $\in$  remotePortin(p) do
    remove msg from remotePortin(p)
    add msg to processOutbox(process)

```

references ASSIGNMESSAGE TOINSTANCE

We assume that prior to the access of the outbox manager, the message has been treated according to the semantics of WS-Addressing (cf. sec. 6.4.4). Assuming proper semantics of  $msgMatchesInstance$ , one could identify  $localPort_{in}$  and  $remotePort_{in}$ ,  $localPort_{out}$  and  $remotePort_{out}$ , and  $localPort_{fault}$  and  $remotePort_{fault}$ , respectively for two communicating BPEL processes.

### 6.6.3 Instance-Manager

We introduce the process manager at this point to provide an entity of the process that does the “clean-up” after a process instance has completed its execution.

$$processInstanceManager : Process \rightarrow ProcessManager$$

requirement for the initial state (6.6.3-1)

$$\begin{aligned} \forall \text{ process} \in \text{Process} \exists \text{instancemanager} \in \text{ProcessManager} : \\ \text{processInstanceManager}(\text{process}) = \text{instancemanager} \\ \wedge \text{managerProcess}(\text{instancemanager}) = \text{process} \end{aligned}$$

Following the framework which we have defined so far, the execution of a process instance completes successfully if *signalCompleted* is found the upward signal channel of the *processScope*, cf. sections 4.2.2, 5.1.2, 5.4.1 and 5.4.4. If a process instance completes, we may remove its identifier from *processRunningInstances*. Similarly, *signalTerminate* notifies about a preemptive termination of a process instance (cf. sec. 5.7).

<p><u>RUNINSTANCEMANAGER</u>          (self) <math>\equiv</math>  <b>let</b> instanceManager = <i>myManager</i>(self),          process = <i>managerProcess</i>(self),          scope = <i>processScope</i>(scope) <b>in</b>  <b>forall</b> pl <math>\in</math> <i>processRunningInstances</i>(process) <b>do</b>            <b>if</b> <i>signalComplete</i> <math>\in</math> <i>signalChannel<sub>up</sub></i>(pl, scope, scope) <b>then</b>              <b>remove</b> <i>signalComplete</i> <b>from</b> <i>signalChannel<sub>up</sub></i>(pl, scope, scope)              <b>remove</b> pl <b>from</b> <i>processRunningInstances</i>(process)            <b>if</b> <i>signalTerminate</i> <math>\in</math> <i>signalChannel<sub>up</sub></i>(pl, scope, scope) <b>then</b>              <b>remove</b> <i>signalTerminate</i> <b>from</b> <i>signalChannel<sub>up</sub></i>(pl, scope, scope)              <b>remove</b> pl <b>from</b> <i>processRunningInstances</i>(process)</p>	<p>(R6.6.3-26)</p>
---	--------------------

The behaviour of the instance-manager can be extended accordingly to meet requirements for process instances which end their execution by throwing a fault. In section 10.5 we present a refinement of the semantics given above to allow the compensation of a process instance due to `enableInstanceCompensation="yes"`.

## 7 Basic Activities

This chapter formalizes all structures and behaviour associated with BPEL's basic activities. Basing on EXECUTEACTIVITY<sub>pattern</sub> (R5.9.0-18), we will present two refinement steps. The first one restricts the behaviour of the general pattern to basic activities. The second refinement step will be performed for each basic activity resulting in the full formal definition of the activity's behaviour by an ASM rule.

### 7.1 Basic Activity Rule

First, let us recall EXECUTEACTIVITY<sub>pattern</sub> (R5.9.0-18) including the ACTIVITYSTATEMACHINE (R5.3.0-5).

$$\frac{\text{EXECUTEACTIVITY}_{\text{pattern}}}{(\text{sl} \in \text{SubInstance}, \text{act} \in \text{Activity})} \equiv \text{(R5.9.0-18)}$$

PROPAGATEDPE(*sl*, *act*)  
PROPAGATETERMINATE(*sl*, *act*)  
**onSignal** *signalStop* **from** *parentActivity*(*act*)  
    **in** *sl* **via** *signalChannel*<sub>down</sub> **do**  
        ACTIVITYSETTOSTOPPING(*sl*, *act*)  
**otherwise**  
    **onSignal** *signalEnable* **from** *parentActivity*(*act*)  
        **in** *sl* **via** *signalChannel*<sub>down</sub> **do**  
            **set** *activityState* **from** *disabled* **to** *enabled*  
PROPAGATEFAULTSACTIVITY(*sl*, *act*)  
**if** *activityState*(*sl*, *act*) = *enabled* **then**  
    STARTACTIVITY(*sl*, *act*)  
**if** *activityState*(*sl*, *act*) = *running* **then**  
    RUNACTIVITY(*sl*, *act*)  
**if** *activityState*(*sl*, *act*) = *completing* **then**  
    COMPLETEACTIVITY(*sl*, *act*)  
**if** *activityState*(*sl*, *act*) = *stopping* **then**  
    STOPACTIVITY(*sl*, *act*)

Control flow is much simpler for basic activities since they don't have to coordinate with child activities. This allows us to refine PROPAGATETERMINATE and to adapt the stop concept for basic activities.

### 7.1.1 Starting, Completing and Stopping Basic Activities

In principle, the rules to start (STARTACTIVITY (R5.4.3-8)), complete (COMPLETEACTIVITY (R5.4.4-9)) or preemptively terminate (STOPACTIVITY (R5.5.4-13)) the execution of a basic activity have already been defined in Chapter 5. In case the behaviour of a specific activity is different, we will present a refinement of these rules.

### 7.1.2 Terminating Process Instances: Basic Activities

As written in section 5.7, a basic activity takes part in the termination of a process instance by simply stopping its execution. Therefore listens for an incoming *signalTerminate* and then performs the local termination.

$\frac{\text{PROPAGATE\_TERMINATE}_{basic}}{(sl \in \text{SubInstance}, act \in \text{Activity}_{basic})} \equiv$ <p style="margin-left: 20px;"><b>onSignal</b> <i>signalTerminate</i> <b>from</b> <i>parentActivity</i>(act)  <b>in</b> sl <b>via</b> <i>signalChannel<sub>down</sub></i> <b>do</b>  TERMINATEACTIVITY(sl, act)</p>	(R7.1.2-27)
--	-------------

referenced in EXECUTEACTIVITY<sub>pattern,basic</sub> (R7.1.3-28)  
referencing TERMINATEACTIVITY (R5.7.0-16)

### 7.1.3 Basic Activity Rule – Final Refinement

With the rule definitions of the previous sections and chapters, the refinement of EXECUTEACTIVITY<sub>pattern</sub> (R5.9.0-18) reads as follows:

- Replace PROPAGATETERMINATE by PROPAGATETERMINATE<sub>basic</sub> (R7.1.2-27), which just listens to an incoming *signalTerminate* and terminates the basic activity without propagating the signal.
- Remove

**if** *activityState*(sl, act) = *stopping* **then**  
STOPACTIVITY(sl, act)

and replace ACTIVITYSETTOSTOPPING(sl, act) by STOPACTIVITY(sl, act), which makes basic activities immediately stop upon receiving *signalStop*.

This results in EXECUTEACTIVITY<sub>pattern,basic</sub> (R7.1.3-28).

---

(R7.1.3-28)

```

EXECUTEACTIVITYpattern,basic
  (sl ∈ SubInstance, act ∈ Activitybasic) ≡
  PROPAGATEDPE(sl, act)
  PROPAGATETERMINATEbasic(sl, act)
  onSignal signalStop from parentActivity(act)
    in sI via signalChanneldown do
      STOPACTIVITY(sl, act)
  otherwise
    onSignal signalEnable from parentActivity(act)
      in sI via signalChanneldown do
        set activityState from disabled to enabled
    if activityState(sl, act) = enabled then
      STARTACTIVITY(sl, act)
    if activityState(sl, act) = running then
      RUNACTIVITY(sl, act)
    if activityState(sl, act) = completing then
      COMPLETEACTIVITY(sl, act)

```

refines EXECUTEACTIVITY<sub>pattern</sub> (R5.9.0-18)

The behaviour of any basic activity in BPEL can be defined as a refinement of the above rule. Hence this rule will be the starting point for the refinement of each basic activity for the rest of this chapter.

## 7.2 Empty

```

<empty standard-attributes>
  standard-elements
</empty>

```

**Universe:** *Empty*

*Empty* has no further properties. *Empty* simply switches to *completed* once reaching *running*.

---

(R7.2.0-29)

```

RUNEMPTY
  (sl ∈ SubInstance, empty ∈ Empty) ≡
  set activityState from running to completing

```

referenced in EXECUTEEMPTY (RB.4.1-112)

To obtain EXECUTEEMPTY (RB.4.1-112) from EXECUTEACTIVITY<sub>pattern,basic</sub> (R7.1.3-28) apply the following refinement:

- Replace RUNACTIVITY by RUNEMPTY (R7.2.0-29), which implements positive control flow for *Empty*.

The final rule EXECUTEEMPTY (RB.4.1-112) can be found in the appendix B.4.1

## 7.3 Wait

```
<wait (for="duration-expr" | until="deadline-expr")
  standard-attributes>
  standard-elements
</wait>
```

**static functions** (see also *Time*, sec. 5.8)

**Universe:** *Wait*

**Universe:** *WaitType* = {*for*, *until*}

**Universe:** *TimeExpression*

$$\begin{aligned} \text{waitType} & : \text{Wait} \rightarrow \text{WaitType} \\ \text{waitTimeExpression} & : \text{Wait} \rightarrow \text{TimeExpression} \end{aligned}$$

**dynamic functions**

$$\text{waitEndTime} : \text{SubInstance} \times \text{Wait} \rightarrow \text{Time}$$

### 7.3.1 Starting Wait

A *Wait* is started by evaluating its *waitTimeExpression* and setting its internal state to *running*. Depending on the *waitType*, the activity's semantics are to wait until a certain moment in time has been reached (any wait-for-condition can be translated into an until-condition knowing the current time). This final point is determined upon activation of the activity. Hence STARTACTIVITY (R5.4.3-8) needs to be refined:

- in parallel with the change of internal state the following rules have to be applied:
  - if  $\text{then} \in \text{Time}$  is the moment in time expressed by the wait-for-expression of  $\text{wait} \in \text{Wait}$ 

$$\text{waitEndTime}(\text{sl}, \text{wait}) := \text{getCurrentTime}(\text{sl}) +_{\text{Time}} \text{then}$$
 determines the waiting-interval,
  - if  $\text{then} \in \text{Time}$  is the moment in time expressed by the wait-until-expression of  $\text{wait} \in \text{Wait}$  then

$waitEndTime(sl, wait) :=$  then  
determines the waiting-interval.

This refinement results in the following rule

<u>STARTWAIT</u>	(R7.3.1-30)
$(sl \in SubInstance, wait \in Wait) \equiv$ <b>if</b> $allLinksSet(sl, act) \wedge joinConditionSatisfied(sl, act) = true$ <b>then</b> <b>let</b> $then = evaluateTimeExpression(processInstance(sl), waitTimeExpression(wait))$ <b>in</b> <b>if</b> $waitType(wait) = for$ <b>then</b> $waitEndTime(sl, wait) := getCurrentTime(sl) +_{Time} then$ <b>then</b> <b>if</b> $waitType(wait) = until$ <b>then</b> $waitEndTime(sl, wait) := then$ <b>set activityState from enabled to running</b>	

referenced in EXECUTEWAIT (RB.4.2-113)  
refining STARTACTIVITY (R5.4.3-8)

### 7.3.2 Running Wait

Assuming  $waitEndTime$  has been set by STARTWAIT (R7.3.1-30) and being in *running*, a *Wait* activity may enter *completing* if the current time has reached the determined  $waitEndTime$ . Otherwise the activity has to stay in *running*. This is formalized by the following rule

<u>RUNWAIT</u>	(R7.3.2-31)
$(sl \in SubInstance, wait \in Wait) \equiv$ <b>let</b> $now = getCurrentTime(processInstance(sl))$ <b>in</b> <b>if</b> $now \leq_{Time} waitEndTime(sl, wait)$ <b>then</b> <b>set activityState from running to completing</b>	

referenced in EXECUTEWAIT (RB.4.2-113)

### 7.3.3 Wait – Final Refinement

To obtain EXECUTEWAIT (RB.4.2-113) from EXECUTEACTIVITY<sub>pattern,basic</sub> (R7.1.3-28) apply the following refinement:

- Replace STARTACTIVITY (R5.4.3-8) by STARTWAIT (R7.3.1-30), which additionally defines  $waitEndTime(sl, wait)$  for  $wait \in Wait$  in its current subinstance  $sl$  depending on the process definition.

- Replace RUNACTIVITY by RUNWAIT (R7.3.2-31), which implements positive control flow for *Wait*, to wait until the current time of the process instance has reached *waitEndTime*(sl, wait).

The final rule EXECUTEWAIT (RB.4.2-113) can be found in the appendix B.4.2

## 7.4 Throw

```
<throw faultName="qname"
    faultVariable="ncname"?
    standard-attributes>
  standard-elements
</throw>
```

**static functions** (see also *Fault*, sec. 5.5)

**Universe:** *Throw*

$$\begin{aligned} \text{throwFaultName} &: \text{Throw} \rightarrow \text{FaultName} \\ \text{throwFaultVariable} &: \text{Throw} \rightarrow \text{Variable} \end{aligned}$$

### 7.4.1 Running Throw

Executing a *Throw* means throwing a *Fault* specified by *throwFaultName* and *throwFaultVariable*. A fault generated by *Throw* may contain further contents written in a variable. Since faults are messages, said contents can be assigned to the fault using COPYVARIABLETOMSG (RB.3.2-111).

Hence we obtain the rule to run throw by refining THROW (R5.5.1-10):

THROW (R5.5.1-10)

```
(sl ∈ SubInstance, act ∈ Activity, fName ∈ FaultName) ≡
let fault = new(Fault) in
  faultName(fault) := fName
  signal fault to parentActivity(act) in sl via signalChannelfault
set activityState from enabled, running, completing to faulted
```

- In parallel with sending the fault to the parent activity apply

```
if throwFaultVariable(fault) ≠ undef then
  COPYVARIABLETOMSG(sl, throwFaultVariable(fault), fault)
```

to assign the contents of the *throwFaultVariable* to *fault* if a variable is specified.

- Replace

```
set activityState from enabled, running, completing to faulted
```

by

**set** *activityState* **from** *running* **to** *faulted*

to strengthen the condition for the transition to the internal state *faulted*, since this rule may be applied in *running* only.

- replace the reference *fName* from the rule's header with *throwFaultName(throw)*, which is always fixed for this activity.

This refinement results in

RUNTHROW (R7.4.1-32)  
 $(sl \in SubInstance, throw \in Throw) \equiv$

```

let fault = new(Fault) in
  faultName(fault) := throwFaultName(throw)
  signal fault to parentActivity(throw) in sl via signalChannelfault
  if throwFaultVariable(fault)  $\neq$  undef then
    COPYVARIABLETOMSG(sl, throwFaultVariable(fault), fault)
  set activityState from running to faulted

```

refines THROW (R5.5.1-10),  
 references COPYVARIABLETOMSG (RB.3.2-111)

### 7.4.2 Completing Throw

A *Throw* which successfully executed its behaviour by throwing a *Fault* does not complete since throwing a *Fault* implies to enter the state *faulted*. Hence, the notion of completion does not apply to *Throw*.

### 7.4.3 Throw – Final Refinement

To obtain EXECUTETHROW (RB.4.3-114) from EXECUTEACTIVITY<sub>pattern,basic</sub> (R7.1.3-28) apply the following refinement:

- Replace RUNACTIVITY by RUNTHROW (R7.4.1-32), which implements positive control flow for *Throw*, throwing a fault with (optional) contents.
- Remove

```

if activityState(sl, throw) = completing then
  COMPLETEACTIVITY(sl, throw)

```

because the notion of completion does not apply to *Throw*.

The final rule EXECUTETHROW (RB.4.3-114) can be found in the appendix B.4.3.

## 7.5 Terminate

```
<terminate standard-attributes>
  standard-elements
</terminate>
```

**Universe:** *Terminate*

*Terminate* has no further properties.

### 7.5.1 Running Terminate

*Terminate* simply sends the *signalTerminate* to its parent activity and terminates itself via TERMINATEACTIVITY (R5.7.0-16). This initiates the termination of the current process instance (cf. 5.7). The initiation is formalized by the following rule

$\frac{\text{RUNTERMINATE}}{(\text{sl} \in \text{SubInstance}, \text{terminate} \in \text{Terminate}) \equiv}$ <p style="margin-left: 20px;"> <b>signal</b> <i>signalTerminate</i> <b>to</b> <i>parentActivity</i>(<i>terminate</i>) <b>in</b> <i>sl</i> <b>via</b> <i>signalChannel<sub>up</sub></i>            TERMINATEACTIVITY(<i>sl</i>, <i>terminate</i>)         </p>	(R7.5.1-33)
--	-------------

referenced in EXECUTETERMINATE (RB.4.4-115)

### 7.5.2 Terminate – Final Refinement

To obtain EXECUTETERMINATE (RB.4.4-115) from EXECUTEACTIVITY<sub>pattern,basic</sub> (R7.1.3-28) apply the following refinement:

- Replace RUNACTIVITY by RUNTERMINATE (R7.5.1-33), which implements positive control flow for *Terminate*, sending a *signalTerminate* to the parent activity.

The final rule EXECUTETERMINATE (RB.4.4-115) can be found in the appendix B.4.4.

## 7.6 Receive

```
<receive partnerLink="ncname" portType="qname" operation="ncname"
  variable="ncname"? createInstance="yes|no"? standard-attributes>
  standard-elements
  <correlations>?
    <correlation set="ncname" initiate="yes|no"?>+
  </correlations>
</receive>
```

**static functions** (see also *PortDescriptor*, sec. 6.5.1)

**Universe:** *Receive*

<i>receivePartnerLink</i>	:	<i>Receive</i>	→	<i>PartnerLink</i>
<i>receivePortType</i>	:	<i>Receive</i>	→	<i>PortType</i>
<i>receiveOperation</i>	:	<i>Receive</i>	→	<i>Operation</i>
<i>receiveVariable</i>	:	<i>Receive</i>	→	<i>Variable</i>
<i>receiveCorrelationSets</i>	:	<i>Receive</i>	→	$\mathcal{P}(\text{CorrelationSet})$
<i>receiveInitCorrelationSet</i>	:	<i>Receive</i> × <i>CorrelationSet</i>	→	$\mathcal{P}(\text{CorrelationSet})$
<i>receivePortDescriptor</i>	:	<i>Receive</i>	→	<i>PortDescriptor</i>

The port descriptor of a receive can be canonically defined from the properties of the `<receive .../>` element and the `<correlations .../>` element.

requirement for the initial state (7.6.0-1)

$$\begin{aligned}
&\forall \text{receive} \in \text{Receive} \exists \text{pd} \in \text{PortDescriptor} \\
&\quad \text{receivePortDescriptor}(\text{receive}) = \text{pd} \\
&\quad \wedge \text{portPartnerLink}(\text{pd}) = \text{receivePartnerLink}(\text{receive}) \\
&\quad \wedge \text{portPortType}(\text{pd}) = \text{receivePortType}(\text{receive}) \\
&\quad \wedge \text{portOperation}(\text{pd}) = \text{receiveOperation}(\text{receive}) \\
&\quad \wedge \text{portVariable}(\text{pd}) = \text{receiveVariable}(\text{receive}) \\
&\quad \wedge \text{portCorrelationSets}(\text{pd}) = \text{receiveCorrelationSets}(\text{receive}) \\
&\quad \wedge \forall \text{cs} \in \text{portCorrelationSets}(\text{pd}) \\
&\quad \quad \text{portInitCorrelationSets}(\text{pd}, \text{cs}) = \text{receiveInitCorrelationSet}(\text{receive}, \text{cs})
\end{aligned}$$

We ignore the `createInstance` attribute as its semantics is related to the creation of a new process instance which we consider in ...

### 7.6.1 Running Receive

A *Receive* waits for a message to arrive at its local communication endpoint

$$\text{localPort}_{in} : \text{ProcessInstance} \times \text{PartnerLink} \times \text{PortType} \times \text{Operation} \rightarrow \mathcal{P}(\text{BPELMsg})$$

(see in chapter 6 sections 6.1.2 (Internal Interface) and 6.5 (Communication Endpoints)). After receiving a message, the activity completes. The operational steps for a *Receive* are a refinement of the abstract behaviour of `AWAITANDRECEIVECORRELATINGMESSAGEpattern` (R6.5.3-19) presented in Section 6.5.3.

- Replace *port* by *localPort<sub>in</sub>*,  
to receive a message at a communication endpoint provided by the activity's BPEL process;
- In parallel with

RECEIVEMESSAGE<sub>pattern</sub>(sl, receive, portdescr, msg)

apply

**set** *activityState* **from** *running* **to** *completing*

to finish the execution of the *Reply* activity.

The full formal definition of AWAITANDRECEIVECORRELATINGMESSAGE<sub>receive</sub> (RB.4.5-117) can be found in the Appendix B.4.5. A *Receive* activity is run by waiting for and then receiving a message:

RUNRECEIVE (R7.6.1-34)

(sl ∈ *SubInstance*, receive ∈ *Receive*) ≡

**let** portdescr = *receivePortDescriptor*(receive) **in**

AWAITANDRECEIVECORRELATINGMESSAGE<sub>receive</sub>(sl, receive, portdescr)

referenced in EXECUTERECEIVE (RB.4.5-116),

referencing AWAITANDRECEIVECORRELATINGMESSAGE<sub>receive</sub> (RB.4.5-117)

## 7.6.2 Receive – Final Refinement

To obtain EXECUTERECEIVE (RB.4.5-116) from EXECUTEACTIVITY<sub>pattern,basic</sub> (R7.1.3-28) apply the following refinement:

- Replace RUNACTIVITY by RUNRECEIVE (R7.6.1-34), which implements positive control flow for *Receive*, receiving a message from the logical partner of the *Receive*.

The final rule EXECUTERECEIVE (RB.4.5-116) can be found in the appendix B.4.5.

## 7.7 Reply

```
<reply partnerLink="ncname" portType="qname" operation="ncname"
  variable="ncname"? faultName="qname"? standard-attributes>
  standard-elements
  <correlations>?
    <correlation set="ncname" initiate="yes|no"?>+
  </correlations>
</reply>
```

**static functions** (see also *PortDescriptor*, sec. 6.5.1; *Fault*, *FaultName*, sec. 5.5)

**Universe:** *Reply*

<i>replyPartnerLink</i>	:	<i>Reply</i>	$\rightarrow$	<i>PartnerLink</i>
<i>replyPortType</i>	:	<i>Reply</i>	$\rightarrow$	<i>PortType</i>
<i>replyOperation</i>	:	<i>Reply</i>	$\rightarrow$	<i>Operation</i>
<i>replyVariable</i>	:	<i>Reply</i>	$\rightarrow$	<i>Variable</i>
<i>replyCorrelationSets</i>	:	<i>Reply</i>	$\rightarrow$	$\mathcal{P}(\textit{CorrelationSet})$
<i>replyInitCorrelationSet</i>	:	<i>Reply</i> $\times$ <i>CorrelationSet</i>	$\rightarrow$	$\mathcal{P}(\textit{CorrelationSet})$
<i>replyPortDescriptor</i>	:	<i>Reply</i>	$\rightarrow$	<i>PortDescriptor</i>
<i>replyFaultName</i>	:	<i>Reply</i>	$\rightarrow$	<i>FaultName</i>

The port descriptor of a reply can be canonically defined from the properties of the `<reply .../>` element and the `<correlations .../>` element, see requirement (7.6.0-1).

### 7.7.1 Running Reply

An operation of a BPEL process may require to return a result of that operation. A *Reply* activity implements to return a message for such an operation. The sent message is interpreted as a reply to a previous message.

Since sending a reply means sending a message, the general structures and rules for sending a message of Section 6.5.3 apply. After sending the reply, the *Reply* completes its execution. We derive the concrete operational steps for *Reply* from the general rules of Section 6.5.3 via refinement.

A reply may be a normal *BPELMsg* or a *Fault*. The *Reply* activity supports both. In each case, a new message is generated and sent. To send a message, i.e. to place it at the *Reply*'s local communication endpoint, it is formally added to the set *localPort<sub>out</sub>* or *localPort<sub>fault</sub>*, respectively. The rules to do so are  $\text{SENDMESSAGE}_{\textit{reply}}$  (RB.4.6-121),  $\text{SENDMESSAGE}_{\textit{fault}}$  (RB.4.6-122). Both are obtained by a simple data refinement of  $\text{SENDMESSAGE}_{\textit{pattern}}$  (R6.5.3-23), properly replacing the abstract set *port*. The ASM rules can be found in the Appendix B.4.6.

Now, generating and sending the reply message is straight forward from  $\text{GENERATEANDSENDMESSAGE}_{\textit{pattern}}$  (R6.5.3-22):

- Replace  $\text{SENDMESSAGE}_{\textit{pattern}}$  by  $\text{SENDMESSAGE}_{\textit{reply}}$ , to send a reply message.

The refinement results in  $\text{GENERATEANDSENDMESSAGE}_{\textit{reply}}$  (RB.4.6-119), given in Appendix B.4.6.

Sending a fault message involves one further operation: the new *Fault* needs a name to classify it. We obtain the full specification by refining  $\text{GENERATEANDSENDMESSAGE}_{\textit{pattern}}$  (R6.5.3-22):

- In **new**(*BPELMsg*) replace *BPELMsg* by *Fault*, to specifically create a new *Fault*;
- Replace `SENDMESSAGEpattern` by `SENDMESSAGEfault`, to send a fault message;
- In parallel with

`SENDMESSAGEfault(sl, portdescr, message)`

apply

`faultFName(fault) := replyFaultName(reply)`

to assign the specified name to the new fault message.

The refinement results in `GENERATEANDSENDMESSAGEfault` (RB.4.6-120), given in Appendix B.4.6.

A *Reply* is executed by either sending a fault, or a normal reply message and completing its execution. The choice between a fault and a normal message is given by *replyFaultName*: if it is undefined a normal message is replied, otherwise a fault is sent. Both is subject to the satisfaction of the correlation properties. This gives rise to the ASM rule `GENERATEANDSENDCORRELATINGMESSAGEreply` which is refined from `GENERATEANDSENDCORRELATINGMESSAGEpattern` (R6.5.3-21) by the following refinement:

- Replace

`GENERATEANDSENDMESSAGEpattern(sl, act, portdescr)`

by

```

if replyFaultName(reply) = undef then
  GENERATEANDSENDMESSAGEreply(sl, reply, portdescr)
else
  GENERATEANDSENDMESSAGEfault(sl, reply, portdescr)
set activityState from running to completing

```

to choose between sending a fault or a normal reply and to complete the execution of a *Reply*.

$$\frac{\text{GENERATEANDSENDCORRELATINGMESSAGE}_{reply}}{(\text{sl} \in \text{SubInstance}, \text{act} \in \text{Activity}, \text{portdescr} \in \text{PortDescriptor})} \equiv \quad (\text{R7.7.1-35})$$

```

let var = portVariable(portdescr),
    CS = portCorrelationSets(portdescr) in
if correlationSatisfiedvar(sl, var, CS) = true then
  if replyFaultName(reply) = undef then
    GENERATEANDSENDMESSAGEreply(sl, reply, portdescr)
  else
    GENERATEANDSENDMESSAGEfault(sl, reply, portdescr)
  set activityState from running to completing
else
  THROW(sl, act, correlationViolation)

```

refines GENERATEANDSENDCORRELATINGMESSAGE<sub>pattern</sub> (R6.5.3-21),  
 referenced in RUNREPLY (R7.7.1-36),  
 references GENERATEANDSENDMESSAGE<sub>reply</sub> (RB.4.6-119), GENERATEANDSENDMESSAGE<sub>fault</sub>  
 (RB.4.6-120)

Having defined how a reply is sent, the ASM rule which defines the behaviour of a running *Reply* is quite simple.

$$\frac{\text{RUNREPLY}}{(\text{sl} \in \text{SubInstance}, \text{reply} \in \text{Reply})} \equiv \quad (\text{R7.7.1-36})$$

```

let portdescr = replyPortDescriptor(reply) in
GENERATEANDSENDCORRELATINGMESSAGEreply(sl, reply, portdescr)

```

referenced in EXECUTEREPLY (RB.4.6-118),  
 references GENERATEANDSENDCORRELATINGMESSAGE<sub>reply</sub> (R7.7.1-35)

### 7.7.2 Receive – Final Refinement

To obtain EXECUTEREPLY (RB.4.6-118) from EXECUTEACTIVITY<sub>pattern,basic</sub> (R7.1.3-28) apply the following refinement:

- Replace RUNACTIVITY by RUNREPLY (R7.7.1-36), which implements positive control flow for *Reply*, sending a reply message to the logical partner of the *Reply*.

The final rule EXECUTEREPLY (RB.4.6-118) can be found in the appendix B.4.6.

## 7.8 Invoke

The full syntax of *Invoke* reads as follows

```

<invoke partnerLink="ncname" portType="qname" operation="ncname"
  inputVariable="ncname"? outputVariable="ncname"?
  standard-attributes>

  standard-elements

  <correlations>?
    <correlation set="ncname" initiate="yes|no"?
      pattern="in|out|out-in"/>+
  </correlations>

  <catch faultName="qname" faultVariable="ncname"?>*
    activity
  </catch>
  <catchAll>?
    activity
  </catchAll>
  <compensationHandler>?
    activity
  </compensationHandler>
</invoke>

```

The latter three elements `catch`, `catchAll` and `compensationHandler` or a shortened notation of a scope (cf. sec. 9.1) which is always implicitly given for any *Invoke* that defines an `outputVariable`. We assume that each BPEL process is written in normal form, where these elements are defined in a *Scope* directly enclosing the *Invoke*.

**static functions** (see also *PortDescriptor*, sec. 6.5.1)

**Universe:** *Invoke*

<i>invokePartnerLink</i>	:	<i>Invoke</i>	→	<i>PartnerLink</i>
<i>invokePortType</i>	:	<i>Invoke</i>	→	<i>PortType</i>
<i>invokeOperation</i>	:	<i>Invoke</i>	→	<i>Operation</i>
<i>invokeInputVariable</i>	:	<i>Invoke</i>	→	<i>Variable</i>
<i>invokeOutputVariable</i>	:	<i>Invoke</i>	→	<i>Variable</i>
<i>invokeCorrelationSets</i>	:	<i>Invoke</i>	→	$\mathcal{P}(\textit{CorrelationSet})$
<i>invokeCorrelationSetDirection</i>	:	<i>Invoke</i> × <i>CorrelationSet</i>	→	$\mathcal{P}(\{in, out\})$
<i>invokeInitCorrelationSet</i>	:	<i>Invoke</i> × <i>CorrelationSet</i>	→	$\mathcal{P}(\textit{CorrelationSet})$
<i>invokePortDescriptor<sub>send</sub></i>	:	<i>Invoke</i>	→	<i>PortDescriptor</i>
<i>invokePortDescriptor<sub>receive</sub></i>	:	<i>Invoke</i>	→	<i>PortDescriptor</i>

An *Invoke* requires two different port descriptors to define both directions of communication. Their definition follows from the properties of the `<invoke .../>` element and its `<correlations .../>` element by the requirement (B.4.7-1), which is given in the appendix B.4.7.

### 7.8.1 Running Invoke

The activity *Invoke* calls operations of remote web services. Such an operation may describe a simple request, without a reply. Then the *Invoke* is called *asynchronous*. Or an *Invoke* calls an operation that returns a reply. Then the *Invoke* has to wait for it and it is called *synchronous*. In the latter case, the *Invoke* is executed in two stages.

#### dynamic functions

$$invokeState : SubInstance \times Invoke \rightarrow \{sending, receiving\}$$

In any case, first the outgoing message is sent to the remote web services, to invoke the remote operation (*sending*). If the operation involves a reply, *Invoke* has to wait until the reply arrives (*receiving*).

$$\begin{array}{l} \underline{RUNINVOKE} \\ (sl \in SubInstance, invoke \in Invoke) \equiv \\ \text{if } invokeState(sl, invoke) = sending \text{ then} \\ \quad RUNINVOKE_{sending}(sl, invoke) \\ \text{if } invokeState(sl, invoke) = receiving \text{ then} \\ \quad RUNINVOKE_{receiving}(sl, invoke) \end{array} \quad (R7.8.1-37)$$

referenced in EXECUTEINVOKE (RB.4.7-123),  
references RUNINVOKE<sub>sending</sub> (R7.8.1-38), RUNINVOKE<sub>receiving</sub> (R7.8.1-40)

#### Invoking a remote Operation

Invoking the operation of a remote web services is done by sending an invoking message to the appropriate communication endpoints.

$$\begin{array}{l} \underline{RUNINVOKE_{sending}} \\ (sl \in SubInstance, invoke \in Invoke) \equiv \\ \text{let } portdescr_{send} = invokePortDescriptor_{send}(invoke) \text{ in} \\ \quad GENERATEANDSENDCORRELATINGMESSAGE_{invoke}(sl, invoke, portdescr_{send}) \end{array} \quad (R7.8.1-38)$$

referenced in RUNINVOKE (R7.8.1-37),  
references GENERATEANDSENDCORRELATINGMESSAGE<sub>invoke</sub> (R7.8.1-39)

We refine the general rules of Section 6.5.3 to formalize this behaviour.

Firstly, we refine GENERATEANDSENDCORRELATINGMESSAGE<sub>pattern</sub> (R6.5.3-21) to GENERATEANDSENDCORRELATINGMESSAGE<sub>invoke</sub> (R7.8.1-39) by the help of the following refinement mapping:

- Replace  $\text{GENERATEANDSENDMESSAGE}_{pattern}$  by  $\text{GENERATEANDSENDMESSAGE}_{invoke}$ , to address the correct communication endpoint;
- In parallel with  $\text{GENERATEANDSENDMESSAGE}_{invoke}$ , apply

```

if  $invokePortDescriptor_{receive}(invoke) \neq undef$  then
   $invokeState(sl, invoke) := receiving$ 
else
  set  $activityState$  from  $running$  to  $completing$ 

```

to prepare this *Invoke* to wait for the reply of the remote web service in case of a synchronous *Invoke*, or to complete the execution of this *Invoke* in case of an asynchronous *Invoke*.

$\text{GENERATEANDSENDCORRELATINGMESSAGE}_{invoke}$  (R7.8.1-39)

$(sl \in SubInstance, invoke \in Invoke, portdescr \in PortDescriptor) \equiv$

```

let  $var = portVariable(portdescr)$ ,
   $CS = portCorrelationSets(portdescr)$  in
if  $correlationSatisfied_{var}(sl, var, CS) = true$  then
   $\text{GENERATEANDSENDMESSAGE}_{invoke}(sl, invoke, portdescr)$ 
  if  $invokePortDescriptor_{receive}(invoke) \neq undef$  then
     $invokeState(sl, invoke) := receiving$ 
  else
    set  $activityState$  from  $running$  to  $completing$ 
  else
     $THROW(sl, act, correlationViolation)$ 

```

refines  $\text{GENERATEANDSENDCORRELATINGMESSAGE}_{pattern}$  (R6.5.3-21),

references  $\text{GENERATEANDSENDMESSAGE}_{invoke}$  (RB.4.7-124),  $THROW$  (R5.5.1-10)

The remaining rules for sending an invoking message  $\text{GENERATEANDSENDMESSAGE}_{invoke}$  (RB.4.7-124) and  $\text{SENDMESSAGE}_{invoke}$  (RB.4.7-125) are the result of a straight-forward refinement of  $\text{SENDMESSAGE}_{pattern}$  (R6.5.3-23). This refinement replaces *port* by  $remotePort_{in}$ , the local mirror of the remote web service's communication interface (see Section 6.5). The definition of both rules is given in the Appendix B.4.7.

### Receiving the Reply of a remote Operation

Receiving a reply of a remote operation is done by the means described in Section 6.5.3. The refinement of the ASM rules defined in that section is straight-forward: replace *port* by  $remotePort_{out}$ , the local mirror of the remote web service's communication endpoint, and change the internal state of this activity to *completing* once a message is received (see also *Receive*, sec. 7.6). The resulting rule  $\text{AWAITANDRECEIVECORRELATINGMESSAGE}_{invoke}$  (RB.4.7-126) is given in the Appendix B.4.7.

Yet, the remote web service might reply with a *Fault* instead of a *BPELMsg*. In such a case, this *Fault* is treated as if it occurred locally during the execution of the local *Invoke* activity. The occurrence of such a *Fault* is prioritized over receiving a regular reply message and the *Fault* doesn't need to satisfy the correlation properties. This gives rise to the following behaviour of an *Invoke* in the stage *receiving*.

$$\frac{\text{RUNINVOKE}_{\text{receiving}}}{(\text{sl} \in \text{SubInstance}, \text{invoke} \in \text{Invoke})} \equiv \quad (\text{R7.8.1-40})$$

```

let portdescrreceive = invokePortDescriptorreceive(invoke) in
  let partnerL = portPartnerLink(portdescr),
      pT = portPortType(portdescr),
      op = portOperation(portdescr),
      pl = processInstance(sl) in
    select fault ∈ remotePortfault(pl, partnerL, pT, op) in
      signal fault to parentActivity(invoke) in sl via signalChannelfault
      remove fault from remotePortfault(pl, partnerL, pT, op)
    ifnone
      AWAITANDRECEIVECORRELATINGMESSAGEinvoke(sl, invoke, portdescrreceive)

```

referenced in RUNINVOKE (R7.8.1-37),

references AWAITANDRECEIVECORRELATINGMESSAGE<sub>invoke</sub> (RB.4.7-126)

## 7.8.2 Invoke – Final Refinement

To obtain EXECUTEINVOKE (RB.4.7-123) from EXECUTEACTIVITY<sub>pattern,basic</sub> (R7.1.3-28) apply the following refinement:

- Replace RUNACTIVITY by RUNINVOKE (R7.8.1-37), which implements positive and negative control flow for *Invoke* for invoking an operation of a remote web service and in case of a synchronous invoke, for additionally awaiting the reply.

The final rule EXECUTEINVOKE (RB.4.7-123) can be found in the appendix B.4.7.

## 7.9 Assign

```

<assign standard-attributes>
  standard-elements
  <copy>+
    from-spec
    to-spec
  </copy>
</assign>

```

Both *from-spec* and *to-spec* are non-terminal symbols of BPEL's grammar. We consider them in detail in Appendix B.4.8.

**static functions**

**Universe:**  $AssignSpec$  (abstractly describing from-spec/to-spec)

$$assignSpecs : Assign \rightarrow \mathcal{P}(AssignSpec \times AssignSpec)$$

By convention, the first component of a an  $AssignSpec$ -pair is the **from-spec**, the second component is the **to-spec**.

**derived functions**

We define the following derived functions to distinguish the first and the second component of an  $AssignSpec$ -pair by names

$$\mathcal{D} assignFromSpec : AssignSpec \times AssignSpec \rightarrow AssignSpec$$

$$(from, to) \mapsto from$$

$$\mathcal{D} assignToSpec : AssignSpec \times AssignSpec \rightarrow AssignSpec$$

$$(from, to) \mapsto to$$

**7.9.1 Running Assign**

The purpose of  $Assign$  is to manipulate data.  $Assign$  operates on different data structures, which include  $Variables$ . Each pair  $AssignSpec \times AssignSpec$  describes a single assignment of values. The  $assignFromSpec$  describes from where to read the values, the  $assignToSpec$  describes to where to write the values.  $Assign$  performs all assignments together. The following rule formalizes the behaviour.

RUNASSIGN (R7.9.1-41)

$(sl \in SubInstance, assign \in Assign) \equiv$

**forall** copySpec  $\in assignSpecs(assign)$  **do**

**let** value =  $assignSpecValue(sl, assignFromSpec(copySpec))$  **in**

SETVALUEBYSPEC( $sl, value, assignToSpec(copySpec)$ )

referenced in EXECUTEASSIGN (RB.4.8-127),

referencing SETVALUEBYSPEC (RB.4.8-128)

The function  $assignSpecValue : SubInstance \times AssignSpec \rightarrow Value$  returns the current value of the specified data structure in the current subinstance. SETVALUEBYSPEC (RB.4.8-128) assigns a value to a specified data structure in the current subinstance. Details for both are given in Appendix B.4.8.

### 7.9.2 Assign – Final Refinement

To obtain EXECUTEASSIGN (RB.4.8-127) from EXECUTEACTIVITY<sub>pattern,basic</sub> (R7.1.3-28) apply the following refinement:

- Replace RUNACTIVITY by RUNASSIGN (R7.9.1-41), which implements positive control flow of *Assign*, manipulating data by assignments.

The final rule EXECUTEASSIGN (RB.4.8-127) can be found in the appendix B.4.8.

## 8 Structured Activities

In this chapter we formalize the structures and behaviour of BPEL's structured activities. Like in the previous chapter 7 we refine EXECUTEACTIVITY<sub>pattern</sub> (R5.9.0-18). This time we cannot apply a refinement which is common for all structured activities and which restricts the behaviour given in EXECUTEACTIVITY<sub>pattern</sub> (R5.9.0-18).

Therefore we will provide a new refinement step for each structured activity to achieve the full formal definition of its behaviour.

### 8.1 Structured Activity Rule

In this section we verify our claim given above, that there is no common behaviour of BPEL's structured activity which is more special than EXECUTEACTIVITY<sub>pattern</sub> (R5.9.0-18).

We first recall the definition of EXECUTEACTIVITY<sub>pattern</sub> (R5.9.0-18) including the ACTIVITYSTATEMACHINE (R5.3.0-5).

$$\begin{array}{l} \text{EXECUTEACTIVITY}_{\text{pattern}} \\ \hline (\text{sl} \in \text{SubInstance}, \text{act} \in \text{Activity}) \equiv \end{array} \quad (\text{R5.9.0-18})$$

PROPAGATEDPE(sl, act)  
PROPAGATETERMINATE(sl, act)  
**onSignal** *signalStop* **from** *parentActivity*(act)  
    **in** *sl* **via** *signalChannel<sub>down</sub>* **do**  
        ACTIVITYSETTOSTOPPING(sl, act)  
**otherwise**  
    **onSignal** *signalEnable* **from** *parentActivity*(act)  
        **in** *sl* **via** *signalChannel<sub>down</sub>* **do**  
            **set** *activityState* **from** *disabled* **to** *enabled*  
        PROPAGATEFAULTSACTIVITY(sl, act)  
    **if** *activityState*(sl, act) = *enabled* **then**  
        STARTACTIVITY(sl, act)  
    **if** *activityState*(sl, act) = *running* **then**  
        RUNACTIVITY(sl, act)  
    **if** *activityState*(sl, act) = *completing* **then**  
        COMPLETEACTIVITY(sl, act)  
    **if** *activityState*(sl, act) = *stopping* **then**  
        STOPACTIVITY(sl, act)

### 8.1.1 Starting a Structured Activity

To start a structured activity, more operational steps are required than just a transition from *enabled* to *running*. Yet, the common behaviour of all structured activities has already been given in `STARTACTIVITY` (R5.4.3-8), cf. sec. 5.4.3. We will specifically refine this rule for each activity.

### 8.1.2 Running a Structured Activity

Running a structured activity involves executing instances of one or more of its child activities. The structured activity must know which instances of its child activity are currently running:

- A structured activity may complete only, if all of its enabled child activities completed.
- A structured activity stops by stopping all of its running child activities.

The behaviour is a repeating pattern for both, the execution and preemptive termination. Both patterns refer to the set of currently running instances of activities. Recall that we already declared a function for this purpose in section 5.4.5.

$$activityRunningChilids : SubInstance \times Activity \rightarrow \mathcal{P}(SubInstance \times Activity)$$

Using this function, we may define two abstract ASM rules which will be refined and instantiated accordingly by each structured activity. A structured activity being in *running* usually exhibits the following behaviour: the activity waits until all child activities completed their execution, then a specific action `RUNACTIVITYNORUNNINGCHILDS` is performed.

$$\frac{RUNACTIVITY_{pattern}}{(sl \in SubInstance, act \in Activity_{structured}) \equiv} \quad (R8.1.2-42)$$

```

if activityRunningChilids(sl, act) = ∅ then
  RUNACTIVITYNORUNNINGCHILDS(sl, act)
else
  forall (slchild, child) ∈ activityRunningChilids(sl, act) do
    onSignal signalCompleted from child in slchild via signalChannelup do
      remove (slchild, child) from activityRunningChilids(sl, act)

```

In most cases, `RUNACTIVITYNORUNNINGCHILDS` instantiates to a simple transition to the internal state *completing*. The refinement of `RUNACTIVITYpattern` is straight forward and yields `RUNACTIVITYstructured`.

- Instantiate `RUNACTIVITYNORUNNINGCHILDS` with

**set** *activityState* **from** *running* **to** *completing*

The definition is given in the Appendix B.5.1.

### 8.1.3 Completing a Structured Activity

Completing a structured activity does not differ from the behaviour already formalized in COMPLETEACTIVITY (R5.4.4-9) in section 5.4.4.

### 8.1.4 Stopping a Structured Activity

To stop a structured activity, we follow exactly the stop-concept as we formalized it in sec. 5.5.3. To recall the stop-concept, receiving *signalStop* from the parent activity makes the current activity immediately transition to *stopping* (cf. ACTIVITYSETTOSTOPPING (R5.5.3-12)).

Being in *stopping* STOPACTIVITY is applied. Each structured activity requires its specific behaviour in that internal state. Yet, what they have in common is a two-staged behaviour: first send initiate preemptive termination by sending *signalStop* to all *activityRunningChilds*, then await their completion or termination by *signalCompleted* or *signalStopped* respectively.

Thus we introduce the following set and the following dynamic function.

**Universe:**  $StopMode =_{def} \{sendingStop, awaitingStopped\}$

dynamic functions (F8.1.4-1)

$$activityStopMode : SubInstance \times Activity_{structured} \rightarrow StopMode$$

requirement for the initial state (8.1.4-1)

$$\forall sl \in SubInstance \forall act \in Activity_{structured} \\ activityStopMode(sl, act) = sendingStop$$

The preemptive termination of the running activities requires no specific order. The mentioned behaviour is uniform for all structured activities and formally reads follows.

$$\frac{STOPACTIVITY_{structured}}{(sl \in SubInstance, act \in Activity_{structured})} \equiv \begin{array}{l} \mathbf{if} \ activityRunningChilds(sl, act) = \emptyset \ \mathbf{then} \\ \quad STOPACTIVITY(sl, act) \\ \mathbf{else} \\ \quad \mathbf{if} \ activityStopMode(sl, act) = sendingStop \ \mathbf{then} \\ \quad \quad \mathbf{forall} \ (sl_{child}, child) \in activityRunningChilds(sl, act) \ \mathbf{do} \\ \quad \quad \quad \mathbf{signal} \ signalStop \ \mathbf{to} \ child \ \mathbf{in} \ sl_{child} \ \mathbf{via} \ signalChannel_{down} \\ \quad \quad \quad \mathbf{set} \ activityStopMode \ \mathbf{from} \ sendingStop \ \mathbf{to} \ awaitingStopped \\ \quad \quad \mathbf{if} \ activityStopMode(sl, act) = awaitingStopped \ \mathbf{then} \end{array} \quad (R8.1.4-43)$$

```

forall ( $sl_{child}, child$ )  $\in$   $activityRunningChilds(sl, act)$  do
  onSignal  $signalCompleted, signalStopped$  from  $child$ 
    in  $sl_{child}$  via  $signalChannel_{up}$  do
      remove ( $sl_{child}, child$ ) from  $activityRunningChilds(sl, act)$ 

```

referenced in ACTIVITYSTATEMACHINE (R5.3.0-5),  
 references STOPACTIVITY (R5.5.4-13)

### 8.1.5 Terminating Process Instances: Structured Activities

The rules for terminating process instances have been introduced in section 5.7, where we formally defined the behaviour with structured activities in mind.

### 8.1.6 Structured Activity Rule – Final Refinement

By refining EXECUTEACTIVITY<sub>pattern</sub> with

- Replace RUNACTIVITY by RUNACTIVITY<sub>structured</sub>.
- Replace STOPACTIVITY by STOPACTIVITY<sub>structured</sub>.

we obtain the most special behaviour of all structured activities in BPEL. The following ASM rule will be the starting point for the refinement of each structured activity for the rest of this chapter.

<pre> EXECUTEACTIVITY<sub>pattern,structured</sub>   (<math>sl \in SubInstance, act \in Activity</math>) <math>\equiv</math>   PROPAGATEDPE(<math>sl, act</math>)   PROPAGATETERMINATE(<math>sl, act</math>)   <b>onSignal</b> <math>signalStop</math> <b>from</b> <math>parentActivity(act)</math>     <b>in</b> <math>sI</math> <b>via</b> <math>signalChannel_{down}</math> <b>do</b>       ACTIVITYSETTOSTOPPING(<math>sl, act</math>)   <b>otherwise</b>     <b>onSignal</b> <math>signalEnable</math> <b>from</b> <math>parentActivity(act)</math>       <b>in</b> <math>sl</math> <b>via</b> <math>signalChannel_{down}</math> <b>do</b>         <b>set</b> <math>activityState</math> <b>from</b> <math>disabled</math> <b>to</b> <math>enabled</math>       PROPAGATEFAULTSACTIVITY(<math>sl, act</math>)     <b>if</b> <math>activityState(sl, act) = enabled</math> <b>then</b>       STARTACTIVITY(<math>sl, act</math>)     <b>if</b> <math>activityState(sl, act) = running</math> <b>then</b>       RUNACTIVITY<sub>structured</sub>(<math>sl, act</math>)     <b>if</b> <math>activityState(sl, act) = completing</math> <b>then</b> </pre>	(R8.1.6-44)
---	-------------

```

    COMPLETEACTIVITY(sl, act)
    if activityState(sl, act) = stopping then
        STOPACTIVITYstructured(sl, act)

```

refines EXECUTEACTIVITY<sub>pattern</sub> (R5.9.0-18) references RUNACTIVITY<sub>structured</sub> (RB.5.1-129),  
STOPACTIVITY<sub>structured</sub> (R8.1.4-43)

## 8.2 Flow

Executing a *Flow* is done by executing all its child activities concurrently.

```

<flow standard-attributes>
    standard-elements

    <links>?
        <link name="ncname">+
    </links>

    activity+
</flow>

```

### static functions

**Universe:** *Flow*

We already defined the static structure of *Links* in Section 4.2.3 and the static structure of the activity tree in Section 4.2.2. *Flow* has no further structures.

### 8.2.1 Starting Flow

To start a *Flow*, wait until all join conditions are satisfied, then enable all child activities and enter the internal state *running*.

Together with enabling the child activities, we add their instances to *activityRunningChilds* (F5.4.5-1).

Now we may refine STARTACTIVITY (R5.4.3-8) to obtain STARTFLOW (R8.2.1-45). First, let's reconsider STARTACTIVITY (R5.4.3-8):

STARTACTIVITY (R5.4.3-8)

$$(sl \in SubInstance, act \in Activity) \equiv$$

```

if allLinksSet(sl, act)  $\wedge$  joinConditionSatisfied(sl, act) = true then
    set activityState from enabled to running

```

By applying the following refinement

- In parallel with

```

    set activityState from enabled to running

```

apply

```
forall child ∈ childActivities(flow) do
  signal signalEnable to child in sl via signalChanneldown
  add (sl, child) to activityRunningChilds(sl, flow)
```

to enable all of *Flow*'s child activities concurrently and to keep track of all currently running child activities.

we obtain the following rule

<p><u>STARTFLOW</u></p> <p>(sl ∈ <i>SubInstance</i>, flow ∈ <i>Flow</i>) ≡</p> <p>if <i>allLinksSet</i>(sl, act) ∧ <i>joinConditionSatisfied</i>(sl, act) = true then</p> <p>forall child ∈ <i>childActivities</i>(flow) do</p> <p>  signal <i>signalEnable</i> to child in sl via <i>signalChannel</i><sub>down</sub></p> <p>  add (sl, child) to <i>activityRunningChilds</i>(sl, flow)</p> <p>  set <i>activityState</i> from <i>enabled</i> to <i>running</i></p>	<p>(R8.2.1-45)</p>
---	--------------------

refines STARTACTIVITY (R5.4.3-8),  
referenced in EXECUTEFLOW (RB.5.2-130)

### 8.2.2 Running Flow

Running a *Flow* means to wait until all child activities completed their execution. Hence we may use RUNACTIVITY<sub>structured</sub> (RB.5.1-129) directly.

### 8.2.3 Flow – Final Refinement

To obtain EXECUTEFLOW (RB.5.2-130) from EXECUTEACTIVITY<sub>pattern,structured</sub> (R8.1.6-44) apply the following refinement:

- Replace STARTACTIVITY (R5.4.3-8) by STARTFLOW (R8.2.1-45), which additionally activates all child activities of a *Flow*;

The final rule EXECUTEFLOW (RB.5.2-130) can be found in the appendix B.5.2.

## 8.3 Sequence

A **sequence** activity has one or more child activities that are executed sequentially, in the order in which they are listed within the <sequence> element, that is, in lexical order. The **sequence** activity completes when the final activity in the sequence has completed.

```

<sequence standard-attributes>
  standard-elements

  activity+
</sequence>

```

### static functions

**Universe:** *Sequence*

The activity tree (cf. 4.2.2 captures parent-child-relations in the process' structure only. The child activities of a *Sequence* are strictly ordered by their occurrence in the process definition. We introduce two static functions.

$$\begin{aligned}
 \textit{sequenceFirstActivity} & : \textit{Sequence} \rightarrow \textit{Activity} \\
 \textit{sequenceActivityNext} & : \textit{Activity} \rightarrow \textit{Activity}
 \end{aligned}$$

We use *sequenceActivityFirst* to reach the first child activity of a *Sequence*. *sequenceActivityNext* subsequently defines the successor of each child activity of a *Sequence*. *sequenceActivityNext*(act) = *undef* iff act has no successor in a sequence. The definition of both functions in the initial state is obvious from the process definition.

### 8.3.1 Starting Sequence

The execution of a *Sequence* begins with the execution of its first child activity. To execute a *Sequence*'s child activity, send *signalEnable* and remember that this activity is currently executed. The following rule groups both operational steps:

$  \frac{\text{ENABLESEQUENCECHILD}}{(\textit{sl} \in \textit{SubInstance}, \textit{seq} \in \textit{Sequence}, \textit{next} \in \textit{Activity}) \equiv}  $ <p style="margin: 0;"> <b>signal</b> <i>signalEnable</i> <b>to</b> <i>next</i> <b>in</b> <i>sl</i> <b>via</b> <i>signalChannel<sub>down</sub></i>  <b>add</b> (<i>sl</i>, <i>next</i>) <b>to</b> <i>activityRunningChilds</i>(<i>sl</i>, <i>seq</i>)  </p>	(R8.3.1-46)
--	-------------

referenced in STARTSEQUENCE (R8.3.1-47), RUNSEQUENCE (RB.5.3-132)

By applying the following refinement on STARTACTIVITY (R5.4.3-8) we obtain STARTSEQUENCE (R8.3.1-47):

- In parallel with

**set** *activityState* **from** *enabled* **to** *running*

apply

ENABLESEQUENCECHILD(*sl*, *seq*, *sequenceFirstActivity*(*seq*))

to enable the first activity of a *Sequence* and remember that the first activity activity is currently being executed.

**STARTSEQUENCE** (R8.3.1-47)

$(sl \in SubInstance, seq \in Sequence) \equiv$

**if**  $allLinksSet(sl, act) \wedge joinConditionSatisfied(sl, act) = true$  **then**  
 ENABLESEQUENCECHILD( $sl, seq, sequenceFirstActivity(seq)$ )  
**set** *activityState* **from** *enabled* **to** *running*

refines STARTACTIVITY (R5.4.3-8),  
 referenced in EXECUTESEQUENCE (RB.5.3-131),  
 references ENABLESEQUENCECHILD (R8.3.1-46)

### 8.3.2 Running Sequence

An sequence  $(sl, seq) \in SubInstance \times Sequence$  is running iff  $activityRunningChilds(sl, seq) \neq undef$ . To run a *Sequence*, wait until the child activity which is currently being executed, completes successfully. If this child activity has a successor, enable the successor. Otherwise, the *Sequence* completes its execution successfully.

For this behaviour we need to refine  $RUNACTIVITY_{structured}$  (RB.5.1-129):

- In parallel with

**remove**  $(sl_{child}, child)$  **from**  $activityRunningChilds(sl, act)$

apply

**let**  $next = sequenceActivityNext(child)$  **in**  
**if**  $next \neq undef$  **then**  
 ENABLESEQUENCECHILD( $sl, seq, next$ )

to execute the successor activity of the child activity which just completed.

This behaviour yields the ASM rule RUNSEQUENCE (RB.5.3-132), the definition is given in the Appendix B.5.3.

After the application of RUNSEQUENCE (RB.5.3-132), the *Sequence* is either *completing* or  $activityRunningChilds(sl, seq) \neq undef$ . Furthermore it can easily be shown that

$$\forall sl \in SubInstance \forall seq \in Sequence \ |activityRunningChilds(sl, seq)| \leq 1$$

holds in every state.

### 8.3.3 Sequence – Final Refinement

To obtain EXECUTESEQUENCE (RB.5.3-131) from EXECUTEACTIVITY<sub>pattern,structured</sub> (R8.1.6-44) apply the following refinement:

- Replace STARTACTIVITY (R5.4.3-8) by STARTSEQUENCE (R8.3.1-47), which additionally activates the first child activities of a *Sequence*;
- Replace RUNACTIVITY<sub>structured</sub> by RUNSEQUENCE (RB.5.3-132), which defines the sequential execution of a *Sequence*'s child activities and defines the positive control flow of *Sequence*;

The final rule EXECUTESEQUENCE (RB.5.3-131) can be found in the appendix B.5.3.

## 8.4 Switch

The **switch** activity allows to choose exactly one child activity for execution by a boolean condition. Each child activity is associated to a condition, together constituting a *branch*. Upon enabling a **switch** activity, the branches are checked in the order of their appearance. The first branch whose condition is satisfied is enabled. If none is satisfied, the alternative **otherwise** branch is taken. The **otherwise** branch is given at least implicitly. A **switch** activity completes if the activity of the chosen branch completes.

```
<switch standard-attributes>
  standard-elements

  <case>+
    <condition expressionLanguage="anyURI"?>
      ... bool-expr ...
    </condition">+
    activity
  </case>

  <otherwise?>
    activity
  </otherwise>
</switch>
```

**static functions** (see also *BooleanExpression*, cf. 5.4.2)

**Universe:** *Switch*

**Universe:** *SwitchBranch* (defined by case)

**Universe:** *BooleanExpression* (abstracting from condition)

Like the child activities of a *Sequence*, the branches of a *Switch* are ordered by their occurrence. Each *SwitchBranch* has a boolean condition and an activity. Each *Switch* always has the alternative *otherwise-SwitchBranch*.

$$\begin{aligned}
\textit{switchFirstBranch} & : \quad \textit{Switch} \rightarrow \textit{SwitchBranch} \\
\textit{switchBranchNext} & : \quad \textit{SwitchBranch} \rightarrow \textit{SwitchBranch} \\
\textit{switchBranchCondition} & : \quad \textit{SwitchBranch} \rightarrow \textit{BooleanExpression} \cup \{\textit{otherwise}\} \\
\textit{switchBranchActivity} & : \quad \textit{SwitchBranch} \rightarrow \textit{Activity} \\
\textit{switchBranches} & : \quad \textit{Switch} \rightarrow \mathcal{P}(\textit{SwitchBranch})
\end{aligned}$$

By definition, the *otherwise-SwitchBranch* is the last one in the list:

requirement for the initial state (8.4.0-1)

$$\begin{aligned}
& \forall b \in \textit{SwitchBranch} \\
& \quad \textit{switchBranchCondition}(b) = \textit{otherwise} \leftrightarrow \textit{switchBranchNext}(b) = \textit{undef}
\end{aligned}$$

### 8.4.1 Starting Switch

To start a *Switch* select the first *SwitchBranch* with satisfied *switchBranchCondition* and execute its activity.

A child activity *C* of a *Switch* may enclose the source activity *S* of some link *L* (cf. 5.4.2). If *C* is not executed by *Switch*, *S* never becomes enabled. Hence *L*'s target activity never may become enabled. To notify *L*'s target activity about that situation, the dead-path-elimination (cf. 5.6) must be run. Therefore we send *signalNegateLinks* to all child activities which are *not* enabled by *Switch*.

This behaviour formally reads as follows

<pre> SWITCHCHOSEBRANCH   (sl ∈ SubInstance, switch ∈ Switch) ≡   let branch = fulfilledSwitchBranch(sl, switchFirstBranch(switch)),       child = switchBranchActivity(branch) in   add (sl, child) to activityRunningChilds(sl, switch)   signal signalEnable to child in sl via signalChannel<sub>down</sub>   forall skippedChild ∈ childActivities(switch) \ {child} do     INITDPE(sl, switch, skippedChild) </pre>	(R8.4.1-48)
---	-------------

referenced in STARTSWITCH (R8.4.1-49),  
references INITDPE (R5.6.0-14)

The derived function *fulfilledSwitchBranch* ((D B.5.4-1)) recursively determines which *SwitchBranch* is the first branch of which the *switchBranchCondition* is met, see also condition (8.4.0-1). The definition is given in the Appendix B.5.4.

By applying the following refinement on STARTACTIVITY (R5.4.3-8) we obtain STARTSWITCH (R8.4.1-49):

- In parallel with

**set** *activityState* **from** *enabled* **to** *running*

apply

SWITCHCHOSEBRANCH(*sl*, *switch*)

to chose the first *SwitchBranch* whose condition is met and to execute the activity of the chosen *SwitchBranch* of a *Switch*.

STARTSWITCH (R8.4.1-49)

(*sl* ∈ *SubInstance*, *seq* ∈ *Sequence*) ≡

**if** *allLinksSet*(*sl*, *act*) ∧ *joinConditionSatisfied*(*sl*, *act*) = *true* **then**

SWITCHCHOSEBRANCH(*sl*, *switch*)

**set** *activityState* **from** *enabled* **to** *running*

refines STARTACTIVITY (R5.4.3-8),  
referenced in EXECUTESWITCH (RB.5.4-133),  
references SWITCHCHOSEBRANCH (R8.4.1-48)

## 8.4.2 Running Switch

By properly defining *activityRunningChilds* in SWITCHCHOSEBRANCH (R8.4.1-48), no modification of RUNACTIVITY<sub>structured</sub> (RB.5.1-129) is required.

## 8.4.3 Switch – Final Refinement

To obtain EXECUTESWITCH (RB.5.4-133) from EXECUTEACTIVITY<sub>pattern,structured</sub> (R8.1.6-44) apply the following refinement:

- Replace STARTACTIVITY (R5.4.3-8) by STARTSWITCH (R8.4.1-49), which additionally activates the first satisfied branch of a *Switch* and initiates the dead-path-elimination on all other branches.

The final rule EXECUTESWITCH (RB.5.4-133) can be found in the appendix B.5.4.

## 8.5 Pick

A **pick** activity works similar to a **switch**: A **pick** defines a number of events. To each event, an activity is associated. Just the first event of a **pick** that occurs is handled by executing its activity.

We distinguish two types of events. An **onMessage** event waits for an external message to arrive. An **onAlarm** event waits until a specific time condition is met.

```

<pick createInstance="yes|no"? standard-attributes>
  standard-elements

  <onMessage partnerLink="ncname" portType="qname" operation="ncname"
    variable="ncname"?>+
    <correlations?>
      <correlation set="ncname" initiate="yes|no"?>+
    </correlations>
    activity
  </onMessage>

  <onAlarm (for="duration-expr" | until="deadline-expr")>*
    activity
  </onAlarm>
</pick>

```

**static functions** (see also *Wait*, c.f. 7.3; *Receive*, c.f. 7.6; *PortDescriptor*, c.f. 6.5.1)

**Universe:** *Pick*

**Universe:** *EventDescriptor* (for *onMessage* and *onAlarm*)

$$\begin{aligned}
pickEvents & : Pick \rightarrow \mathcal{P}(EventDescriptor) \\
eventType & : EventDescriptor \rightarrow \{onAlarm, onMsg\} \\
eventActivity & : EventDescriptor \rightarrow Activity \\
event_{Msg}PortDescriptor & : EventDescriptor \rightarrow PortDescriptor \\
event_{Alarm}Type & : EventDescriptor \rightarrow \{for, until\} \\
event_{Alarm}Expression & : EventDescriptor \rightarrow TimeExpression
\end{aligned}$$

**dynamic functions** (see *Time*, c.f. 5.8) The dynamics of *Pick* require us to remember which event occurred first, and hence, which child activity of *Pick* we are executing. Similar to *Wait* (c.f. 7.3), we need to initialize *Time* values to evaluate conditions. Therefore we introduce the following functions.

$$\begin{aligned}
event_{Alarm}EndTime & : SubInstance \times EventDescriptor \rightarrow Time \\
pickChosenActivity & : SubInstance \times Pick \rightarrow Activity
\end{aligned}$$

requirement for the initial state (8.5.0-1)

$$\forall sl \in SubInstance \forall pick \in Pick \ pickChosenActivity(sl, pick) = undef$$

### 8.5.1 Starting Pick

To enable a *Pick*, initialize the *Time* conditions of all *onAlarm* events and set internal state to *running*. Initialization of an *onAlarm* event is essentially identical to initializing a *Wait*. Therefore we refine STARTWAIT (R7.3.1-30) to initialize a single *onAlarm* event.

STARTWAIT (R7.3.1-30)

$(sl \in SubInstance, wait \in Wait) \equiv$

```

if  $allLinksSet(sl, act) \wedge joinConditionSatisfied(sl, act) = true$  then
  let  $then = evaluateTimeExpression(processInstance(sl), waitTimeExpression(wait))$  in
    if  $waitType(wait) = for$  then
       $waitEndTime(sl, wait) := getCurrentTime(sl) +_{Time}$  then
    if  $waitType(wait) = until$  then
       $waitEndTime(sl, wait) := then$ 
    set  $activityState$  from  $enabled$  to  $running$ 

```

- Remove the first **if** operator and the last **set** assignment, to reduce the behaviour to the *initialization* of an onAlarm event.
- Replace any occurrence of **wait** by **event**, to initialize an event.
- Replace any occurrence of  $waitType$  by  $event_{AlarmType}$ , to use the structures of *EventDescriptor*.

Applying the above refinement leads to the following ASM rule

INITALARMEVENT (R8.5.1-50)

$(sl \in SubInstance, event \in EventDescriptor) \equiv$

```

let  $then = evaluateTimeExpression(sl, event_{Alarm}Expression(event))$  in
  if  $event_{Alarm}Type(event) = for$  then
     $event_{Alarm}EndTime(sl, event) := getCurrentTime(sl) +_{Time}$  then
  if  $event_{Alarm}Type(event) = until$  then
     $event_{Alarm}EndTime(sl, event) := then$ 

```

refining STARTWAIT (R7.3.1-30),

referenced in STARTPICK (R8.5.1-51), STAREVENTHANDLER<sub>onAlarm</sub> (R9.6.1-77)

Now we may refine STARTACTIVITY (R5.4.3-8) to obtain STARTPICK (R8.5.1-51) using the following refinement.

- In parallel with

**set**  $activityState$  **from**  $enabled$  **to**  $running$

apply

**forall**  $event \in pickEvents(pick)$  **where**  $eventType(event) = onAlarm$  **do**  
 INITALARMEVENT( $sl, event, now$ )

to initialize the *Time* conditions for each onAlarm event of a *Pick*.

STARTPICK (R8.5.1-51)

$(sl \in SubInstance, pick \in Pick) \equiv$

**if**  $allLinksSet(sl, pick) \wedge joinConditionSatisfied(sl, pick) = true$  **then**  
   **set** *activityState* **from** *enabled* **to** *running*  
   **forall**  $event \in pickEvents(pick)$  **where**  $eventType(event) = onAlarm$  **do**  
     INITALARMEVENT(*sl*, *event*)

refines STARTACTIVITY (R5.4.3-8),  
 referenced in EXECUTEPICK (RB.5.5-134),  
 references INITALARMEVENT (R8.5.1-50)

Please note that *getCurrentTime* in INITALARMEVENT (R8.5.1-50) always evaluates to the same element in *Time* for any *onAlarm* event of a *Pick*. Furthermore, the *Pick* has no running child activity yet.

## 8.5.2 Running Pick

Once a *Pick* is set to *running* it waits for the first event to occur. As soon as an event occurs, *Pick* executes the activity which is associated to that event and doesn't handle any further events. In other words: as long as *Pick* hasn't determined which child activity it has to execute (*pickChosenActivity*), it waits for the first event to occur. This behaviour formally reads as follows.

RUNPICK (R8.5.2-52)

$(sl \in SubInstance, pick \in Pick) \equiv$

**if**  $pickChosenActivity(sl, pick) = undef$  **then**  
   PICKCHOSEBRANCH(*sl*, *pick*)  
**else**  
   PICKRUNCHOSENBRANCH(*sl*, *pick*)

referenced in EXECUTEPICK (RB.5.5-134),  
 referencing PICKCHOSEBRANCH (R8.5.2-53), PICKRUNCHOSENBRANCH (R8.5.2-55)

### Pick – Select the first Event

Waiting for the first event to occur is rendered difficult, as events may occur concurrently. In a timely manner, a *Pick* may observe the occurrence of an event *after* its occurrence. This requires us to put events in a partial order and choose among those events which occurred at the earliest.

To determine which events have occurred up to now, evaluate the derived function  $pickActivatedEvents : SubInstance \times Pick \rightarrow \mathcal{P}(EventDescriptor)$  (D B.5.5-1). It returns the set of all events of the given *Pick* which occurred up to now.

To determine whether an event occurred earlier than another one, evaluate the derived function  $eventOccurrence : SubInstance \times EventDescriptor \rightarrow Time$  (D B.5.5-2). By pairwise comparing the moments of time of the occurrence of all events (use the partial order  $\leq_{Time}$ , c.f. 5.8), we get the set of events which occurred at the earliest. From those we may non-deterministically chose one event which we are going to handle. The formal definition of both functions is given in the Appendix B.5.5.

Having selected an event, we enable its activity to handle it. This involves further steps and is defined in PICKACTIVATEBRANCH (R8.5.2-54). The ASM rule to select an event and to enable its activity reads as follows.

PICKCHOSEBRANCH (R8.5.2-53)

$(sl \in SubInstance, pick \in Pick) \equiv$

**let** activatedEvents =  $pickActivatedEvents(sl, pick)$  **in**  
**select** ev  $\in$  activatedEvents  
**where**  $\forall ev' \in activatedEvents$   
 $eventOccurrence(sl, ev) \leq_{Time} eventOccurrence(sl, ev')$  **in**  
 PICKACTIVATEBRANCH(sl, pick, ev)

referenced in RUNPICK (R8.5.2-52),

referencing PICKACTIVATEBRANCH (R8.5.2-54)

The operational steps to handle the determined event are known to us from *Switch*. We refine SWITCHCHOSEBRANCH (R8.4.1-48), where we replace the chosen *SwitchBranch* by the determined *EventDescriptor*.

SWITCHCHOSEBRANCH (R8.4.1-48)

$(sl \in SubInstance, switch \in Switch) \equiv$

**let** branch =  $fulfilledSwitchBranch(sl, switchFirstBranch(switch))$ ,  
 child =  $switchBranchActivity(branch)$  **in**  
**add** (sl, child) **to**  $activityRunningChilds(sl, switch)$   
**signal**  $signalEnable$  **to** child **in** sl **via**  $signalChannel_{down}$   
**forall** skippedChild  $\in childActivities(switch) \setminus \{child\}$  **do**  
 INITDPE(sl, switch, skippedChild)

- In the initial **let** operator  
 remove branch = ..., and  
 replace  $switchBranchActivity(branch)$  by  $pickEventActivity(event)$ ,  
 since we already determined the event we want to handle.
- In parallel with **let**, apply

**if**  $eventType(ev) = onMsg$  **then**

AWAITANDRECEIVECORRELATINGMESSAGE $_{pick}(sl, pick, event_{Msg}PortDescriptor(event))$

to handle an *onMsg* event by receiving the triggering message and copying the contents of the message to an appropriate variable.

Hence the behaviour to activate the activity of the chosen event reads as follows.

**PICKACTIVATEBRANCH** (R8.5.2-54)

$(sl \in SubInstance, pick \in Pick, event \in EventDescriptor) \equiv$

```

let child = pickEventActivity(event) in
  add (sl, child) to activityRunningChilds(sl, pick)
  signal signalEnable to child in sl via signalChanneldown

  forall skippedChild  $\in$  childActivities(pick) \ {child} do
    INITDPE(sl, pick, skippedChild)

if eventType(event) = onMsg then
  AWAITANDRECEIVECORRELATINGMESSAGEpick(sl, pick, eventMsgPortDescriptor(event))

```

refines SWITCHCHOSEBRANCH (R8.4.1-48),

referenced in PICKCHOSEBRANCH (R8.5.2-53),

references AWAITANDRECEIVECORRELATINGMESSAGE<sub>pick</sub> (RB.5.5-135), INITDPE (R5.6.0-14)

A *Pick* which handles an *onMsg* event receives the message as abstractly specified in Section 6.5.3. We refine AWAITANDRECEIVECORRELATINGMESSAGE<sub>pattern</sub> (R6.5.3-19) as follows:

- Replace the abstract *port* by *localPort<sub>in</sub>*,  
to receive the message at an endpoints provided by the activity's BPEL process.

The full formal definition of AWAITANDRECEIVECORRELATINGMESSAGE<sub>pick</sub> (RB.5.5-135) can be found in Appendix B.5.5.

### Pick – Run the chosen Branch

Now that we've enabled the child activity of *Pick* we want to execute, the remaining operational steps for *Pick* are to wait for the chosen child activity to complete. This behaviour is already formalized in RUNACTIVITY<sub>structured</sub> (RB.5.1-129):

**PICKRUNCHOSEBRANCH** (R8.5.2-55)

$(sl \in SubInstance, pick \in Pick) \equiv$

```

  RUNACTIVITYstructured(sl, pick)

```

referenced in RUNPICK (R8.5.2-52)

### 8.5.3 Pick – Final Refinement

To obtain EXECUTE PICK (RB.5.5-134) from EXECUTEACTIVITY<sub>pattern,structured</sub> (R8.1.6-44) apply the following refinement:

- Replace STARTACTIVITY (R5.4.3-8) by STARTPICK (R8.5.1-51), which additionally activates the first satisfied branch of a *Switch* and initiates the dead-path-elimination on all other branches;
- Replace RUNACTIVITY by RUNPICK (R8.5.2-52), which defines the positive control flow of *Switch*: to wait for the chosen branch to complete its execution;

The final rule EXECUTE PICK (RB.5.5-134) can be found in the appendix B.5.5.

## 8.6 While

A *while* activity defines an iteratively repeated execution of its child activity as long as its condition holds.

```
<while condition="bool-expr" standard-attributes>
  standard-elements
  activity
</while>
```

**static functions** (see *BooleanExpression*, c.f. sec. 5.4.2)

**Universe:** *While*

$$whileCondition : While \rightarrow BooleanExpression$$

### 8.6.1 Starting While

To start a *While*, no further operational steps are required.

### 8.6.2 Running While

Running *While* involves two stages. In the first stage (*head*), the *whileCondition* is evaluated. If it evaluates to *false*, the execution of *While* completes. Otherwise (*true*), we execute the body of *While* (i.e. its child activity) and enter the second stage (*body*).

Being in the second stage, we await the completion of the current instance of the child activity. When this happens, we return to the first stage, to evaluate the *whileCondition* again. We model both stages in the usual way, using the following function.

**dynamic functions** (F8.6.2-1)

$$whileIterationState : SubInstance \times While \rightarrow \{head, body\}$$

requirement for the initial state (8.6.2-1)

$$\forall sl \in SubInstance \forall while \in While \text{ whileIterationState}(sl, while) = head$$

The two-staged behaviour as described above can be formalized as follows.

RUNWHILE (R8.6.2-56)

$(sl \in SubInstance, while \in While) \equiv$

**if**  $\text{whileIterationState}(sl, while) = head$  **then**

**if**  $\text{evaluateConditionExpression}(sl, \text{whileCondition}(while)) = true$  **then**

WHILESTARTNEWBODY( $sl, while$ )

**else**

**set**  $activityState$  **from** *running* **to** *completing*

**if**  $\text{whileIterationState}(sl, while) = body$  **then**

WHILEENDBODY( $sl, while$ )

referenced in EXECUTEWHILE (RB.5.6-136),

references WHILESTARTNEWBODY (R8.6.2-57), WHILEENDBODY (RB.5.6-137)

### Starting an Execution of the Body

To distinguish any two iterations of a *While* activity, each time its child activity *child* is executed a new instance of it has to be created. To do this it suffices to create a new *SubInstance* *sl* using **new** to discriminate the new instance from preceding ones. Then this new instance (*sl, child*) is enabled by *signalEnable*.

Having enable the child activity, *While* awaits its successful completion by switching to the second stage (*body*). In addition, the subinstance-tree needs to be extended properly.

WHILESTARTNEWBODY (R8.6.2-57)

$(sl \in SubInstance, while \in While) \equiv$

**let**  $sl_{body} = \text{new}(SubInstance)$ ,

$child = \text{childActivity}(while)$  **in**

**add** ( $sl_{body}, child$ ) **to**  $activityRunningChilds(sl, while)$

**signal**  $signalEnable$  **to**  $childActivity(while)$  **in**  $sl_{body}$  **via**  $signalChannel_{down}$

EXTENDSUBINSTANCETREE( $sl, sl_{body}$ )

**set**  $whileIterationState$  **from** *head* **to** *body*

referenced in RUNWHILE (R8.6.2-56)

**Note:** The structures of the subinstance tree currently do not define the *order* in which the subinstances of the body of a *While* are executed. The informal specification does not require such a notion. But the definitions can easily be extended to support this kind of order.

### Completing the Execution of the Body

Being in the second stage *body*, *While* behaves like any other structured activity and waits for all of its running child activities (which is exactly one) to complete. When this happens, *While* returns to the first stage *head*. This gives rise to the following refinement of  $\text{RUNACTIVITY}_{\text{pattern}}$  (R8.1.2-42).

- Instantiate  $\text{RUNACTIVITYNORUNNINGCHILDS}$  with

**set** *whileIterationState* **from** *head* **to** *body*

to check the *whileCondition* again.

The refinement results in  $\text{WHILEENDBODY}$  (RB.5.6-137), defined in the Appendix B.5.6. From the definition of  $\text{WHILESTARTNEWBODY}$  (R8.6.2-57) and  $\text{WHILEENDBODY}$  (RB.5.6-137) follows that

$$\forall \text{sl} \in \text{SubInstance} \quad \forall \text{while} \in \text{While} \quad |\text{activityRunningChilds}(\text{sl}, \text{while})| \leq 1$$

holds in every state.

### 8.6.3 While – Final Refinement

To obtain  $\text{EXECUTEWHILE}$  (RB.5.6-136) from  $\text{EXECUTEACTIVITY}_{\text{pattern}}$  (R5.9.0-18) apply the following refinement:

- Replace  $\text{RUNACTIVITY}$  by  $\text{RUNWHILE}$  (R8.6.2-56), which defines the positive control flow of *While*.

The final rule  $\text{EXECUTEWHILE}$  (RB.5.6-136) can be found in the appendix B.5.6.

## 9 The Scope, Fault Handling & Event Handling

Although a *Scope* implements the same interface for activity-to-activity communication like any other activity in BPEL, it's behaviour is quite different. This is due to the handlers which are attached to the *Scope*. Unlike the “normal” activities of the previous sections, the execution of a *Scope* involves more communication between it and its handlers.

In a first step, we will present the *Scope*'s general architecture and its behaviour in the normal case, the positive control-flow. Then, we will introduce event handling and fault handling. This knowledge is required to define the behaviour of a *Scope* in case of a fault. The compensation handling is formalized the next chapter 10.

### 9.1 Scope – General Architecture

A **scope** encapsulates a part of the process with respect to handle faults, to handle events and to undo performed behaviour as defined by the business logic.

A **scope** has exactly one *primary* child activity which is executed if the scope is executed.

A **scope** defines a *fault handler*<sup>1</sup> which catches any fault that occurs within the **scope**. Additionally, a **scope** defines a number of **eventHandlers** each being capable of handling a specific event. The **scope** also defines a **compensationHandler** which is capable of undoing work that has been done within the scope.

In the scenario where no fault occurs, a **scope** completes if its primary activity completes and all **eventHandlers** finished to handle their events. In case a fault occurs, the **scope** completes once its fault handler finished to handle the fault.

Furthermore, a **scope** may define local **variables** which are accessible from inside the **scope** only. In the same way, local **correlationSets** may be defined at a scope, to restrict communication executed within the **scope**.

```
<scope variableAccessSerializable="yes|no" standard-attributes>
  standard-elements
  <variables>?
    ...
</variables>
<correlationSets>?
  ...
</correlationSets>
<faultHandlers>?
  ...
</faultHandlers>
<compensationHandler>?
```

---

<sup>1</sup>Unlike the syntax suggests, the fault handling mechanism requires exactly one fault handler, c.f. sec. 9.3.

```

    ...
  </compensationHandler>
  <eventHandlers>?
    ...
  </eventHandlers>

  activity
</scope>

```

### static functions

**Universe:** *Scope*

**Universe:** *FaultHandler* (cf. sec. 9.3)

**Universe:** *EventHandler* (cf. sec. 9.5)

**Universe:** *CorrelationSet* (cf. sec. 6.4.2)

$$\begin{aligned}
 \text{scopePrimaryActivity} & : \text{Scope} \rightarrow \text{Activity} \quad (\text{activity}) \\
 \text{scopeFaultHandler} & : \text{Scope} \rightarrow \text{FaultHandler} \quad (\text{faultHandlers}) \\
 \text{scopeEventHandlers} & : \text{Scope} \rightarrow \mathcal{P}(\text{EventHandler}) \quad (\text{eventHandlers}) \\
 \text{scopeCompensationHandler} & : \text{Scope} \rightarrow \text{CompensationHandler} \quad (\text{compensationHandler}) \\
 \text{scopeVariables} & : \text{Scope} \rightarrow \mathcal{P}(\text{Variable}) \quad (\text{variables, cf. sec. 4.2.4}) \\
 \text{scopeCorrelationSets} & : \text{Scope} \rightarrow \mathcal{P}(\text{CorrelationSet}) \quad (\text{correlationSets, cf. sec. 6.4.2})
 \end{aligned}$$

Because of the Compensation Handling mechanism (Chapter 10) each execution of a *Scope* is performed on its own values of its *scopeVariables* and its *scopeCorrelationSets*. In the terms of the subinstance tree: for each instance *sl* of a *Scope*, the values of the local *Variables* and local *CorrelationSets* are also local to *sl* (cf. sections 6.3 and 6.4.2). Hence their initial values are not initialized in any instance:

requirement for the initial state (9.1.0-1)

$$\begin{aligned}
 & \forall \text{sl} \in \text{SubInstance} \quad \forall \text{scope} \in \text{Scope} \\
 & \quad \forall \text{var} \in \text{scopeVariable}(\text{scope}) \\
 & \quad \quad \text{variablePartValue}(\text{sl}, \text{var}, \text{variableType}(\text{var})) = \text{undef} \\
 & \quad \wedge \forall \text{cs} \in \text{scopeCorrelationSets}(\text{scope}) \quad \forall \text{prop} \in \text{correlationProperties}(\text{cs}) \\
 & \quad \quad \text{correlationPropertyValue}(\text{sl}, \text{scope}, \text{prop}) = \text{undef}
 \end{aligned}$$

Like stated in the introduction of this chapter, a *Scope* implements the interface of an activity as we presented it in Section 8.1: A scope supports dead-path-elimination and the propagation of *signalTerminate*. It reacts on *signalStop* and implements some finite state machine where the execution is triggered by *signalEnable*.

The main differences are that a *Scope* has three internal modes (*ScopeMode*) and that a *Scope* handles *signalStop* differently.

Firstly, the *scopeMode* keeps track of what kind of faults did occur. Being in the initial mode *positive*, no fault has occurred yet. A *Scope* switches to *negative* as soon as its fault handler as caught a fault. Both modes require a certain behaviour of the *Scope*. Yet it might

happen that during the execution of the *Scope*'s fault handler another fault occurs which cannot be handled. Then the *Scope* enters the mode *faulted* and must not do anything.

The behaviour of the three modes is clearly distinct. The *scopeMode* works like a switch in the behaviour at the very top level.

**Universe:**  $ScopeMode = \{positive, negative, faulted\}$

dynamic functions (F9.1.0-1)

$$scopeMode : SubInstance \times Scope \rightarrow ScopeMode$$

requirement for the initial state (9.1.0-2)

$$\forall sl \in SubInstance \forall scope \in Scope \\ scopeMode(sl, scope) = positive$$

Secondly, stopping a scope upon receiving *signalStop* involves more than just following the stop-concept as defined in Section 5.5.3. The behaviour deserves its own rule SCOPEHANDLESIGNALSTOP (R9.1.1-59) which will be defined in the following.

Taking both changes together, we may modify the abstract ASM rule EXECUTEACTIVITY<sub>pattern</sub> (R5.9.0-18) to define the correct rule for the *Scope*.

EXECUTESCOPE (R9.1.0-58)

( $sl \in SubInstance, scope \in Scope$ )  $\equiv$

PROPAGATEDPE( $sl, scope$ )

PROPAGATETERMINATE( $sl, scope$ )

**onSignal** *signalStop* **from** *parentActivity*( $scope$ ) **in** *sI* **via** *signalChannel<sub>down</sub>* **do**  
SCOPEHANDLESIGNALSTOP( $sl, scope$ )

**otherwise**

**if**  $scopeMode(sl, scope) = positive$  **then**  
EXECUTESCOPE<sub>positive</sub>( $sl, scope$ )

**if**  $scopeMode(sl, scope) = negative$  **then**  
EXECUTESCOPE<sub>negative</sub>( $sl, scope$ )

**if**  $scopeMode(sl, scope) = faulted$  **then**  
**skip**

references PROPAGATEDPE (R5.6.0-15), PROPAGATETERMINATE (R5.7.0-17),  
SCOPEHANDLESIGNALSTOP (R9.1.1-59), EXECUTESCOPE<sub>positive</sub> (RB.6.1-138),  
EXECUTESCOPE<sub>negative</sub> (R9.4.0-73)

### 9.1.1 Handling *signalStop*

We introduced *signalStop* for the purpose of preemptively terminating a running activity. This preemptive termination is always triggered by the *Scope* which encloses that activity. The intention of preemptive termination also applies to a *Scope*, yet the behaviour is different.

Whenever a *Scope* shall preemptively be terminated due to a fault in its enclosing *Scope*, the fault *forcedTermination* is thrown at the former *Scope*. This triggers the fault handling mechanism which finally results in a preemptive termination of the execution of the *Scope* but provides additional behaviour with respect to the business logic of the BPEL process.

From the informal specification [CGK<sup>+</sup>03], we deduce the following behaviour: Receiving *signalStop* from the parent activity and there is no fault handler running, *forcedTermination* is thrown at the current *Scope*. The fault *forcedTermination* must not be thrown if there is a running fault handler, since the *Scope* is already on its way to a preemptive termination of its execution. This behaviour may be applied only, if the *Scope* hasn't reached its mode *faulted* yet.

dynamic functions (F9.1.1-1)

$$\text{scopeFHRunning} : \text{SubInstance} \times \text{Scope} \rightarrow \{\text{true}, \text{false}\}$$

requirement for the initial state (9.1.1-1)

$$\forall \text{sl} \in \text{SubInstance} \forall \text{scope} \in \text{Scope} \text{scopeFHRunning}(\text{sl}, \text{scope}) = \text{false}$$

The remaining case, where the *scopeMode* is *faulted* and a *signalStop* arrives from the enclosing *Scope*, is possible. Since a *Scope* being in *scopeMode faulted* already terminated any behaviour, it may immediately confirm its termination with *signalStopped*, although there is no causal dependency.

This behaviour formally reads as follows.

SCOPEHANDLESIGNALSTOP (R9.1.1-59)

$(\text{sl} \in \text{SubInstance}, \text{scope} \in \text{Scope}) \equiv$

**if** *scopeMode*(*sl*, *scope*)  $\in$  {*positive*, *negative*} **then**

**if** *scopeFHRunning*(*sl*, *scope*) = *false* **then**

    THROWFORCEDTERMINATION(*sl*, *scope*)

**if** *scopeFHRunning*(*sl*, *scope*) = *true* **then**

**skip**

**if** *scopeMode*(*sl*, *scope*) = *faulted* **then**

**signal** *signalStopped* **to** *parentActivity*(*scope*) **in** *sl* **via** *signalChannel<sub>up</sub>*

referenced in EXECUTESCOPE (R9.1.0-58),

references THROWFORCEDTERMINATION (R9.1.2-61)

In any scenario, where *signalStop* arrives at a *Scope* while the scope hasn't *faulted* yet, the *Scope* will preemptively terminate its execution. Either due to the *signalStop*, or because it is already terminating its execution. Regardless of the cause of preemptive termination, *signalCompleted* or *signalStopped* will be sent to the *Scope*'s parent activity by the behaviour which is formalized in the following sections. Thus, the requirements of the stop-concept, that a *signalStop* must be confirmed by *signalStopped* or *signalCompleted*, is always fulfilled.

### 9.1.2 Organizing Flow of Faults

In Section 9.1.1, we already mentioned the propagation of faults at a *Scope*. Yet, the behaviour significantly differs from the propagation of faults at normal activities. First of all, any fault which is sent from an activity of the positive control flow (i.e. the *Scope*'s primary activity or one of its event handlers), now has reached the enclosing scope. Hence it is not propagated to the parent activity of the *Scope*, but it is forwarded to the *Scope*'s fault handler.

This requires a unique channel for faults going *downwards* the activity tree from a *Scope* to its fault handler.

dynamic functions (F9.1.2-1)

$$faultChannel_{down} : SubInstance \times Scope \times FaultHandler \rightarrow \mathcal{P}(Fault)$$

The behaviour of propagating faults is a refinement of PROPAGATEFAULTSACTIVITY (R5.5.2-11).

- Restrict the **forall** operator on *activityRunningChilds*(sl,scope) to instances (sl<sub>child</sub>, child) where child ≠ scopeFaultHandler(scope) holds, to propagate faults of the positive control flow only.
- Replace *faultChannel<sub>up</sub>*(sl, act, parentActivity(act)) by *faultChannel<sub>down</sub>*(sl, scope, scopeFaultHandler(scope)), to propagate any such fault to the fault handler of the *Scope*.

PROPAGATEFAULTSCOPE (R9.1.2-60)

(sl ∈ SubInstance, scope ∈ Scope) ≡

**forall** (sl<sub>child</sub>, child) ∈ *activityRunningChilds*(sl, act)  
**where** child ≠ *scopeFaultHandler*(scope) **do**  
**forall** fault ∈ *faultChannel<sub>up</sub>*(sl<sub>child</sub>, child, scope) **do**  
**add** fault **to** *faultChannel<sub>down</sub>*(sl, scope, *scopeFaultHandler*(scope))  
**remove** fault **from** *faultChannel<sub>up</sub>*(sl<sub>child</sub>, child, scope)

refines PROPAGATEFAULTSACTIVITY (R5.5.2-11),  
referenced in EXECUTESCOPE<sub>positive</sub> (RB.6.1-138)

Throwing *forcedTermination* as it is required to preemptively stop the execution of a *Scope* differs from THROW (R5.5.1-10) in the way that new fault with the name *forcedTermination* is passed to the *Scope*'s fault handler instead of its parent activity. Hence the behaviour for throwing this fault is formalized by the following rule.

THROWFORCEDTERMINATION (R9.1.2-61)

$(sl \in SubInstance, scope \in Scope) \equiv$

```
let fault = new(Fault) in
  faultName(fault) := forcedTermination
  add fault to faultChanneldown(sl, scope, scopeFaultHandler(scope))
```

referenced in SCOPEHANDLESIGNSTOP (R9.1.1-59)

Finally, there is a third scenario which we have to consider when talking about the propagation of faults at a *Scope*: a fault which is thrown by the *Scope*'s fault handler cannot be sent back to the fault handler since it already failed. Instead, the fault is *rethrown* to the *Scope*'s parent activity. The behaviour is another data refinement of PROPAGATEFAULTSACTIVITY (R5.5.2-11).

- Restrict the **forall** operator on *activityRunningChilds*(*sl*, *scope*) to instances (*sl<sub>child</sub>*, *child*) where *child* = *scopeFaultHandler*(*scope*) holds, to propagate faults from the fault handler only. Since this refinement puts the **forall** operator in a special case, we drop it and replace any occurrence of *child* by *scopeFaultHandler*(*scope*) and *sl<sub>child</sub>* by *sl* respectively.

RETHROWFAULTSCOPE (R9.1.2-62)

$(sl \in SubInstance, scope \in Scope) \equiv$

```
forall fault ∈ faultChannelup(sl, scopeFaultHandler(scope), scope) do
  add fault to faultChannelup(sl, scope, parentActivity(scope))
  remove fault from faultChannelup(sl, scopeFaultHandler(scope), scope)
```

refines PROPAGATEFAULTSACTIVITY (R5.5.2-11),

referenced in EXECUTESCOPE<sub>negative</sub> (R9.4.0-73)

Please note that PROPAGATEFAULTSCOPE and RETHROWFAULTSCOPE together cover all scenarios of fault propagation. The application of both rules is limited to the scenarios (i.e. the internal modes) in which they might contribute to a change of state.

## 9.2 Scope – Positive Control-Flow

In this section, we will define the behaviour of a *Scope* in the “all-well-case” where the *Scope* follows the intended behaviour of the BPEL process. This behaviour resembles the one of a structured activity:

- Receiving *signalEnable* from the parent activity triggers the execution which is expressed in terms of the internal finite state machine. We will properly define the behaviour in each of the states in the following.
- Any fault which occurs inside the *Scope* is propagated, although the destination of propagation is different this time. We already defines this behaviour in Section 9.1.2 above.
- The execution of the positive control flow of a *Scope* is interrupted whenever this is triggered by *signalStop*. Here, *signalStop* is not received from the parent activity, but from the *Scope*'s fault handler, since it is the task of the fault handler to trigger the stop-concept inside of a *Scope*. Anyway else, the *Scope*'s behaviour upon receiving *signalStop* from its own fault handler is equivalent to the behaviour of a structured activity.

Hence we may define the *Scope*'s behaviour in *positive* as a refinement of  $\text{EXECUTEACTIVITY}_{\text{pattern}}$ . As usual, we give the behaviour for each internal state and the final refinement in the following subsections.

Considering the requirements above, the formalized behaviour of a *Scope* in its *positive* stage reads as follows.

### 9.2.1 Initiate Stopping of a Scope in mode *positive*

To initiate the preemptive termination of a *Scope*, we may apply the according ASM rule  $\text{ACTIVITYSETTOSTOPPING}$  of a structured activity, since in this scenario, a *Scope* behaves just like a structured activity.

Additionally, receiving *signalStop* from the *Scope*'s fault handler implies that the fault handler is now running. We need to remember that in the dedicated dynamic function *scopeFHRunning* to properly react on *signalStop* from the *Scope*'s parent activity (cf. sec. 9.1.1).

This results in the following ASM rule.

$$\begin{array}{l} \text{SCOPESETTOSTOPPING} \\ (\text{sl} \in \text{SubInstance}, \text{scope} \in \text{Scope}) \equiv \\ \text{ACTIVITYSETTOSTOPPING}(\text{sl}, \text{scope}) \\ \text{scopeFHRunning}(\text{sl}, \text{scope}) := \text{true} \end{array} \quad (\text{R9.2.1-63})$$

referenced in  $\text{EXECUTESCOPE}_{\text{positive}}$  (RB.6.1-138),  
 references  $\text{ACTIVITYSETTOSTOPPING}$  (R5.5.3-12)

### 9.2.2 Starting a Scope in mode *positive*

A *Scope* is started by starting its primary activity, all of its event handlers and its fault handler. Having done this, the *Scope* enters its state *running*. This behaviour is similar to starting a *Flow* (section 8.2). Hence, refine  $\text{STARTFLOW}$  (R8.2.1-45) by

- Replace any occurrence of  $childActivities(flow)$  by  $childActivities(scope) \setminus \{scopeCompensationHandler(scope)\}$ , which enables the execution of the primary activity, the event handlers and the fault handler.  $activityRunningChilids$  is updated accordingly.

which results in  $STARTSCOPE_{positive}$  (RB.6.1-139). Its full definition is given in the Appendix B.6.1.

### 9.2.3 Running a Scope in mode *positive*

A *Scope* is executed by awaiting the completion of its primary activity. The *Scope*'s handlers may still be running. Therefore we refine  $RUNACTIVITY_{structured}$  (RB.5.1-129) by

- Replace

**if**  $activityRunningChilids(sl, act) = \emptyset$  **then**

by

**if**  $(sl, scopePrimaryActivity(scope)) \notin activityRunningChilids(sl, act)$  **then**

to just check the completion of the  $scopePrimaryActivity$ ,

which results in  $RUNSCOPE_{positive}$  (RB.6.1-140), defined in Appendix B.6.1.

### 9.2.4 Stopping a Scope in mode *positive*

Stopping a *Scope* requires a slight refinement of  $STOPACTIVITY_{structured}$  (R8.1.4-43), because the fault handler which is also a running child activity must not be preemptively terminated.

Refine  $STOPACTIVITY_{structured}$  (R8.1.4-43) by

- Restrict the **forall** operator on  $(sl_{child}, child) \in activityRunningChilids(sl, act)$  to instances where  $child \neq scopeFaultHandler(scope)$  holds, to stop activities of the positive control flow only.
- Replace

**if**  $activityRunningChilids(sl, act) = \emptyset$  **then**

by

**if**  $activityRunningChilids(sl, scope) \subseteq (sl, scopeFaultHandler(scope))$  **then**

to just check the termination of the activities of the positive control flow.

- Replace  $STOPACTIVITY$  by  $SCOPESWITCHTONEGATIVE$ , to enter  $scopeMode\ negative$ .

which yields  $STOPSCOPE_{positive}$  (RB.6.1-141). The definition is given in the Appendix B.6.1.

SCOPESWITCHTONEGATIVE (R9.2.4-64)

$(sl \in SubInstance, scope \in Scope) \equiv$

**set** *scopeMode* **from** *positive* **to** *negative*  
**set** *activityState* **from** *stopping* **to** *enabled*

referenced in  $STOPSCOPE_{positive}$  (RB.6.1-141)

### 9.2.5 Completing a Scope in mode *positive*

A scope may complete its execution if its primary activity completed its execution, but it cannot complete as long as an event is handled by the according event handler. This scenario is a general one, since event handlers are executed concurrently to the scope's activity.

To prevent the handling of further events, all event handlers are notified by the scope to finish their current execution and ignore any further events. As soon as each event handler acknowledged its completion<sup>2</sup>, the scope requires its fault handler to complete. Once this has happened, the scope may transition to *completed*. **Universe:**  $ScopeCompleteMode = \{sendingComplete, awaitingCompleted_{event}, awaitingCompleted_{fault}\}$

dynamic functions (F9.2.5-1)

$$scopeCompleteMode : SubInstance \times Scope \rightarrow ScopeCompleteMode$$

requirement for the initial state (9.2.5-1)

$$\forall sl \in SubInstance \forall scope \in Scope \\ scopeCompleteMode(sl, scope) = sendingComplete$$

COMPLETESCOPE<sub>positive</sub> (R9.2.5-65)

$(sl \in SubInstance, scope \in Scope) \equiv$

**if**  $scopeCompletingState(sl, scope) = sendingComplete$  **then**  
 SCOPEINITIATEEVENTHANDLERCOMPLETION(*sl*, *scope*)  
 $scopeCompletingState(sl, scope) := awaitingCompleted_{event}$

**if**  $scopeCompletingState(sl, scope) = awaitingCompleted_{event}$  **then**  
 SCOPEAWAITEVENTHANDLERCOMPLETION(*sl*, *scope*)

**if**  $scopeCompletingState(sl, scope) = awaitingCompleted_{fault}$  **then**  
**onSignal** *signalCompleted* **from**  $scopeFaultHandler(scope)$   
**in** *sl* **via**  $signalChannel_{up}$  **do**  
 COMPLETESCOPE(*sl*, *scope*)

referenced in  $EXECUTESCOPE_{positive}$  (RB.6.1-138),  
 references COMPLETESCOPE (R9.2.5-67)

<sup>2</sup>This mechanism is the only way to complete the execution of an onMessage event handler.

Initiating the completion of the *Scope*'s event handlers is trivially done by sending them a dedicated *signalComplete*.

**SCOPEINITIATEEVENTHANDLERCOMPLETION** (R9.2.5-66)

$(sl \in SubInstance, scope \in Scope) \equiv$

**forall**  $(sl_{eh}, eh) \in activityRunningChilds(sl, scope)$   
**where**  $eh \in scopeEventHandlers(scope)$  **do**  
**signal** *signalComplete* **to**  $eh$  **in**  $sl_{eh}$  **via** *signalChannel<sub>down</sub>*

referenced in COMPLETESCOPE<sub>positive</sub> (R9.2.5-65)

Awaiting the completion of the *Scope*'s event handlers is behaviorally equivalent to running a structured activity. Additionally, when all event handlers completed, the *Scope*'s fault handler is initiated to complete. Therefore we refine RUNACTIVITY<sub>pattern</sub> (R8.1.2-42) by

- Restrict the **forall** operator on *activityRunningChilds*( $sl, act$ ) to instances  $(sl_{eh}, eh)$  where  $eh \in scopeEventHandlers(scope)$  holds.
- Replace

**if** *activityRunningChilds*( $sl, act$ ) =  $\emptyset$  **then**

by

**if**  $\forall (sl_{child}, child) \in activityRunningChilds(sl, scope)$   
 $child \notin scopeEventHandlers(scope)$  **then**

to await the completion of the *Scope*'s event handlers.

- Instantiate RUNACTIVITYNORUNNINGCHILDS with

**signal** *complete* **to** *scopeFaultHandler*( $scope$ ) **in**  $sl$  **via** *signalChannel<sub>down</sub>*  
 $scopeCompletingState(sl, scope) := awaitingCompleted_{fault}$

to complete the fault handler.

which yields SCOPEAWAITEVENTHANDLERCOMPLETION (RB.6.1-142), defined in Appendix B.6.1.

Unlike a normal activity, a *Scope* provides further behaviour after its completion. So in addition to the normal completion of a *Scope*, it installs a *compensation handler* and prepares itself for compensation. Hence the behaviour of a *Scope* finally completing its execution is a conservative extension of the well-known behaviour.

COMPLETESCOPE (R9.2.5-67)

( $sl \in SubInstance, scope \in Scope$ )  $\equiv$

COMPLETEACTIVITY( $sl, scope$ )  
INSTALLCOMPENSATIONHANDLER( $sl, scope$ )

referenced in COMPLETESCOPE<sub>positive</sub> (R9.2.5-65),  
references COMPLETEACTIVITY (R5.4.4-9), INSTALLCOMPENSATIONHANDLER (R10.1.2-84)

For now it suffices to assume that INSTALLCOMPENSATIONHANDLER (R10.1.2-84) properly prepares the scope for its compensation. A detailed definition of the entire compensation handling mechanism is given in the next chapter 10.

### 9.2.6 Scope in mode *positive* – Final Refinement

We define EXECUTESCOPE<sub>positive</sub> (RB.6.1-138) by refining EXECUTEACTIVITY<sub>pattern,structured</sub> (R8.1.6-44) by

- Remove PROPAGATEDPE and PROPAGATETERMINATE, since these are already applied in EXECUTESCOPE (R9.1.0-58) <sup>3</sup>.

- Replace

**onSignal** *signalStop* **from** *parentActivity*(act)

by

**onSignal** *signalStop* **from** *scopeFaultHandler*(scope)

since a *Scope*'s preemptive termination in mode *positive* is initialized by its fault handler.

- Replace ACTIVITYSETTOSTOPPING by SCOPESETTOSTOPPING.
- Replace PROPAGATEFAULTSACTIVITY by PROPAGATEFAULTSCOPE (R9.1.2-60).
- Replace STARTACTIVITY by STARTSCOPE<sub>positive</sub>.
- Replace RUNACTIVITY<sub>structured</sub> by RUNSCOPE<sub>positive</sub>.
- Replace STOPACTIVITY<sub>structured</sub> by STOPSCOPE<sub>positive</sub>.
- Replace COMPLETEACTIVITY by COMPLETESCOPE<sub>positive</sub>.

The full definition of EXECUTESCOPE<sub>positive</sub> (RB.6.1-138) is given in the Appendix B.6.1.

---

<sup>3</sup>Their application here would create exactly the same update set as in EXECUTESCOPE, hence we may skip them here

## 9.3 Fault Handler

A fault handler is in charge of handling any faults occurring within the scope it is attached to.

A fault handler can distinguish faults by their names (`faultName`) and by the type of their contents (which must match the type of a specified `faultVariable`). For each kind of fault a fault handler provides its an activity to be executed in case a fault is caught. Furthermore the fault handler may specify an activity which is executed in the unmatched case (`catchAll`).

```
<faultHandlers>?
  <catch faultName="qname"? faultVariable="ncname"?>*
    activity
  </catch>
  <catchAll>?
    activity
  </catchAll>
</faultHandlers>
```

In the formal semantics we slightly alter the terminology in order to sharpen the semantics: Instead of letting a *Scope* have multiple fault handlers – one for each type of fault, as the informal specification [CGK<sup>+</sup>03] describes – we let a scope have exactly one fault handler with one or more *catch nodes*. A catch node is *not* an activity, but a sub-structure of the fault handler. We do so because a fault handler handles no more than one fault and – by the informal specification – at most one fault may be handled the scope’s fault handlers.

### static functions

**Universe:** *FaultHandler*

**Universe:** *CatchNode*

A *CatchNode* has a *FaultName*, references a *Variable* and encloses an activity. Each fault handler has a most general catch node. Its activity is in charge of handling unexpected faults.

$$\begin{aligned} \text{faultHandlerCatchNodes} & : \text{FaultHandler} \rightarrow \mathcal{P}(\text{CatchNode}) \\ \text{catchNodeFaultName} & : \text{CatchNode} \rightarrow \text{FaultName} \cup \{\text{catchAll}\} \\ \text{catchNodeActivity} & : \text{CatchNode} \rightarrow \text{Activity} \\ \text{catchNodeVariable} & : \text{CatchNode} \rightarrow \text{Variable} \end{aligned}$$

The fault handler assumes that the activity of a catch node is appropriate to handle a fault if the fault names match and if the data-types of the fault and of the referenced variable match.

### 9.3.1 Starting a Fault Handler

A *FaultHandler* cannot be the target activity of a *Link*. Hence it transitions to *running* as soon as it is *enabled*. This behaviour is directly implied by STARTACTIVITY (R5.4.3-8) and *joinConditionSatisfied* (D 5.4.2-1).

### 9.3.2 Running a Fault Handler

A fault handler is run in three stages: Firstly, it waits for a fault to be caught. Secondly, if a fault is caught, it stops the execution of the process inside the fault handler's scope. Thirdly, it executes an activity to handle the fault. A fault handler catches at most one fault.

**Universe:**  $FaultHandlerStage = \{awaitFault, stopScope, executeCatch\}$

dynamic functions (F9.3.2-1)

$$faultHandlerStage : SubInstance \times FaultHandler \rightarrow FaultHandlerStage$$

A fault handler which hasn't caught a fault yet (stage *awaitFault*) can be required to successfully complete its execution by its scope<sup>4</sup>. In case the fault handler is not executed (because there was no *Fault*), we need to initialize the dead path elimination for all child activities of the fault handler. The formal definition of this behaviour reads as follows.

**RUNFAULTHANDLER** (R9.3.2-68)

```
(sl ∈ SubInstance, fh ∈ FaultHandler) ≡
if faultHandlerStage(sl, fh) = awaitFault then
  onSignal signalComplete from parentActivity(fh)
    in sl via signalChanneldown do
      set activityState from running to completing
      forall child ∈ childActivities(fh) do
        INITDPE(sl, fh, child)
      otherwise
        WAITFORFAULT(sl, fh)
if faultHandlerStage(sl, fh) = stopScope then
  FAULTHANDLERSTOPSCOPE(sl, fh)
if faultHandlerStage(sl, fh) = executeCatch then
  EXECUTEATCH(sl, fh)
```

referenced in EXECUTEFAULTHANDLER (RB.6.2-144),  
 references INITDPE (R5.6.0-14), WAITFORFAULT (R9.3.2-69), FAULTHANDLERSTOPSCOPE  
 (R9.3.2-70), EXECUTEATCH (R9.3.2-72)

#### Catching a Fault

From section 9.1.2 (Organizing Flow of Faults) we know that the scope forwards any *Fault* which reached the scope to its fault handler via the *faultChannel*<sub>down</sub>.

<sup>4</sup>cf. completing a scope in mode *positive*, sec. 9.2.5

Hence a fault handler being in stage *awaitFault* catches a *Fault* by receiving it via this channel. In our semantics we decided to choose the most appropriate activity to handle the fault as soon as the fault has been caught. The fault handler has to remember its choice.

dynamic functions (F9.3.2-2)

$$\begin{aligned} \mathit{faultHandlerCaughtFault} &: \mathit{SubInstance} \times \mathit{FaultHandler} \rightarrow \mathit{Fault} \\ \mathit{faultHandlerChosenActivity} &: \mathit{SubInstance} \times \mathit{FaultHandler} \rightarrow \mathit{Activity} \end{aligned}$$

Before the fault handler may execute its chosen activity, it needs to stop the positive control flow of its scope. Therefore upon catching a *Fault* the fault handler transitions to *stopScope*, remembers the fault it has caught and determines the best fault handling activity by applying **CHOOSECATCHNODE** (RB.6.2-145).

WAITFORFAULT (R9.3.2-69)

( $\mathit{sl} \in \mathit{SubInstance}, \mathit{fh} \in \mathit{FaultHandler}$ )  $\equiv$

**select**  $\mathit{fault} \in \mathit{faultChannel}_{\mathit{down}}(\mathit{sl}, \mathit{parentActivity}(\mathit{fh}), \mathit{fh})$  **in**  
 $\mathit{faultHandlerStage}(\mathit{sl}, \mathit{fh}) := \mathit{stopScope}$   
 $\mathit{faultHandlerCaughtFault}(\mathit{sl}, \mathit{fh}) := \mathit{fault}$   
**CHOOSECATCHNODE**( $\mathit{sl}, \mathit{fh}, \mathit{fault}$ )

referenced in **RUNFAULTHANDLER** (R9.3.2-68),  
 references **CHOOSECATCHNODE** (RB.6.2-145)

Choosing the best activity to handle the caught fault is formalized in **CHOOSECATCHNODE** (RB.6.2-145), given in the Appendix B.6.2. The choice is stored in *faultHandlerChosenActivity* and will be executed once the positive control flow of the scope has preemptively been terminated.

In case no fault handling activity is defined for the caught fault, the fault handler applies its default mechanism. We formally represent this case by defining *faultHandlerChosenActivity* to *undef* at the corresponding location.

### Stopping Positive Control Flow of the FaultHandler's Scope

To preemptively terminate the execution of the scope's positive control-flow, we employ a modification of the stop concept: The fault handler sends *signalStop upwards* the activity tree to the scope and awaits *signalStopped* in return<sup>5</sup> which implies that the fault handler may start the execution of the fault handling activity. Hence upon receiving *signalStopped* the fault handler enters stage *executeCatch*.

This behaviour is *not* identical with the termination of the control flow of the fault handler. This requires a new function.

dynamic functions (F9.3.2-3)

<sup>5</sup>see 9.2.4 (Stopping a Scope in mode *positive*)

$$fHStopMode : SubInstance \times FaultHandler \rightarrow StopMode$$

requirement for the initial state (9.3.2-1)

$$\forall sl \in SubInstance \forall fh \in FaultHandler \ fHStopMode(sl, fh) = sendingStop$$

FAULTHANDLERSTOPSCOPE (R9.3.2-70)

$$(sl \in SubInstance, fh \in FaultHandler) \equiv$$

**if**  $fHStopMode(sl, fh) = sendingStop$  **then**  
**signal**  $signalStop$  **to**  $parentActivity(fh)$  **in**  $sl$  **via**  $signalChannel_{up}$   
 $fHStopMode(sl, fh) := awaitingStopped$   
**if**  $fHStopMode(sl, fh) = awaitingStopped$  **then**  
**onSignal**  $signalStopped$  **from**  $parentActivity(fh)$   
**in**  $sl$  **via**  $signalChannel_{down}$  **do**  
 $faultHandlerStage(sl, fh) := executeCatch$

referenced in RUNFAULTHANDLER (R9.3.2-68)

Sending  $signalStop$  triggers the application of SCOPESETTOSTOPPING (R9.2.1-63) by the fault handler's scope. Thus, STOPSCOPE<sub>positive</sub> (RB.6.1-141) will be applied by that scope in consequence.

### Handling the Caught Fault

Having reached  $executeCatch$ , the fault handler executes the chosen fault handler activity. If  $faultHandlerChosenActivity$  (F9.3.2-2) is defined, the execution is performed in the well-known way. If there is no matching fault handling activity, the fault handler has to apply the default behaviour and rethrows the caught fault.

1. The execution of a defined  $faultHandlerChosenActivity$  requires two steps.

**Universe:**  $FHExecuteMode = \{sendingEnable, awaitingCompleted\}$

dynamic functions (F9.3.2-4)

$$FHExecuteMode : SubInstance \times FaultHandler \rightarrow FHExecuteMode$$

requirement for the initial state (9.3.2-2)

$$\forall sl \in SubInstance \forall fh \in FaultHandler \ FHExecuteMode(sl, fh) = sendingEnable$$

2. Rethrowing the caught fault essentially means to send  $faultHandlerCaughtFault$  back to the scope and entering the internal state  $faulted$ .

RETHROWFAULT (R9.3.2-71)

```
(sl ∈ SubInstance, fh ∈ FaultHandler) ≡
let fault = faultHandlerCaughtFault(sl, fh) in
  add fault to faultChannelup(sl, fh, parentActivity(fh))
  set activityState from running to faulted
```

referenced in EXECUTE CATCH (R9.3.2-72)

Since the fault handler executes at most one of its child activities the fault handler needs to run the dead path elimination on all child activities which are not executed.

When the *faultHandlerChosenActivity* completed its execution, the fault handler may complete its execution as well; receiving *signalCompleted* defines the transition to the internal state *completing*.

EXECUTE CATCH (R9.3.2-72)

```
(sl ∈ SubInstance, fh ∈ FaultHandler) ≡
if fHExecuteMode(sl, fh) = sendingEnable then
  let act = faultHandlerChosenActivity(sl, fh) in
    if act ≠ undef then
      signal signalEnable to act in sl via signalChanneldown
      add (sl, act) to activityRunningChilds(sl, fh)
      fHExecuteMode(sl, fh) := awaitingCompleted
    else
      RETHROWFAULT(sl, fh)
  forall child in childActivities(fh) \ {act} do
    INITDPE(sl, fh, child)

  if fHExecuteMode(sl, fh) = awaitingCompleted then
    RUNACTIVITYstructured(sl, fh)
```

referenced in RUNFAULTHANDLER (R9.3.2-68),

references INITDPE (R5.6.0-14), RETHROWFAULT (R9.3.2-71), RUNACTIVITY<sub>structured</sub> (RB.5.1-129)

From the transitions of *faultHandlerStage* one can easily deduce that a fault handler catches at most one *Fault*.

### 9.3.3 Fault Handler – Final Refinement

The final refinement from EXECUTEACTIVITY<sub>pattern,structured</sub> (R8.1.6-44) is straight forward:

- Replace RUNACTIVITY by RUNFAULTHANDLER.

The full definition of EXECUTEFAULTHANDLER (RB.6.2-144) is defined in the appendix B.6.2. Please note that since a fault handler cannot be the source of a *Link*, EVALUATEOUTGOINGLINKS (R5.4.2-6) which is applied in COMPLETEACTIVITY (R5.4.4-9) doesn't change the state.

## 9.4 Scope – Negative Control-Flow

Knowing how the fault handler behaves, and what the fault handler requires from its scope in case of a caught fault, we may now define the behaviour of the *Scope* in its mode *negative*. Remember that a *Scope* enters *negative* when *Scope*'s positive control flow preemptively terminated due to *signalStop* from its fault handler:

SCOPESWITCHTONEGATIVE (R9.2.4-64)

$$(\text{sl} \in \text{SubInstance}, \text{scope} \in \text{Scope}) \equiv$$

**set** *scopeMode* **from** *positive* **to** *negative*  
**set** *activityState* **from** *stopping* **to** *enabled*

Being in mode *negative* and knowing that the fault handler is now executing its *faultHandlerChosenActivity*, the scope awaits the fault handler to successfully complete its execution.

Therefore *Scope* neither awaits a *signalEnable*, since it is already running, nor it may preemptively terminate its execution due to a *signalStop*, since this already happened. Furthermore, any *Fault* which might reach the *Scope* now is sent from its fault handler.

derived functions ( $\mathcal{D}$  9.4.0-1)

$$\mathcal{D} \text{ fhHasThrownFault} : \text{SubInstances} \times \text{Scope} \rightarrow \{\text{true}, \text{false}\}$$

$$(\text{sl}, \text{scope}) \mapsto \begin{cases} \text{true}, & \text{faultChannel}_{\text{up}}(\text{sl}, \text{scopeFaultHandler}(\text{scope}), \text{scope}) \neq \emptyset \\ \text{false}, & \text{else} \end{cases}$$

Simple propagation semantics as in PROPAGATEFAULTSACTIVITY are not applicable: any *Fault* which is thrown by the *Scope*'s fault handler implies that the execution of the fault handler failed. The informal specification [CGK<sup>+</sup>03] jointly requires to immediately rethrow any such *Fault* and to terminate the execution of the fault handler.

Hence the formally defined behaviour of a *Scope* in mode *negative* is not a refinement of EXECUTEACTIVITY<sub>pattern</sub> but a conservative extension of a refined ACTIVITYSTATEMACHINE.

EXECUTESCOPE<sub>negative</sub> (R9.4.0-73)

$$(\text{sl} \in \text{SubInstance}, \text{scope} \in \text{Scope}) \equiv$$

**if** *fhHasThrownFault*(*sl*, *scope*) = *true* **then**  
    RETHROWFAULTSCOPE(*sl*, *scope*)

```

SCOPESETTOSTOPPING(sl, scope)
else
  if activityState(sl, act) = enabled then
    STARTSCOPEnegative(sl, act)
  if activityState(sl, act) = running then
    RUNACTIVITYstructured(sl, act)
  if activityState(sl, act) = completing then
    COMPLETESCOPEnegative(sl, act)
  if activityState(sl, act) = stopping then
    STOPSCOPEnegative(sl, act)

```

referenced in EXECUTESCOPE (R9.1.0-58),

references SCOPESETTOSTOPPING (R9.2.1-63), RETHROWFAULTSCOPE  
(R9.1.2-62), STARTSCOPE<sub>negative</sub> (R9.4.1-74), RUNACTIVITY<sub>structured</sub> (RB.5.1-129),  
COMPLETESCOPE<sub>negative</sub> (RB.6.1-143), STOPSCOPE<sub>negative</sub> (R9.4.4-75)

#### 9.4.1 Starting a Scope in mode *negative*

There is no condition which an enabled *Scope* in mode *negative* has to meet. Hence it directly transitions to *running*. We must not evaluate the conditions of the links again.

```

STARTSCOPEnegative (R9.4.1-74)
  (sl ∈ SubInstance, scope ∈ Scope) ≡
  set activityState from enabled to running

```

referenced in EXECUTESCOPE<sub>negative</sub> (R9.4.0-73)

#### 9.4.2 Running a Scope in mode *negative*

The intended behaviour of a running *Scope* in mode *negative* is to await the completion of its fault handler. Since this is the only running child activity, we may apply RUNACTIVITY<sub>structured</sub> (RB.5.1-129) without change.

#### 9.4.3 Completing a Scope in mode *negative*

After the fault handler completes, the *Scope* may finish its execution as well. Yet, the execution was not successful. Hence the *Scope* enters the final state *stopped*. In any other aspect, the behaviour is identical to COMPLETEACTIVITY (R5.4.4-9). We refine:

- For the update of *activityState*, replace *completed* by *stopped*, to transition to the unsuccessful, terminal state.

The definition of COMPLETESCOPE<sub>negative</sub> (RB.6.1-143) is given in the Appendix B.6.1.

#### 9.4.4 Stopping a Scope in mode *negative*

A *Scope* which has a running fault handler (hence the *scopeMode* is *negative*) enters *stopping* only if its fault handler has thrown or rethrown a fault. Either because the fault handler couldn't handle the fault, or because another fault occurred during the execution of the fault handler. In either case, the fault has already been rethrown to the *Scope*'s parent activity as defined in EXECUTESCOPE<sub>negative</sub> (R9.4.0-73).

The informal specification states that in this scenario the fault handler's execution needs to be terminated and the *Scope* shall terminate its execution immediately. Therefore the *Scope* doesn't await *signalStopped* from the fault handler and transitions to *stopped* immediately.

As stated in the informal specification, if a fault was rethrown then the outgoing links of a *Scope* have to be evaluated to *false*.

$\frac{\text{STOPSCOPE}_{negative}}{(sl \in \text{SubInstance}, \text{scope} \in \text{Scope})} \equiv$ <p><b>signal</b> <i>signalStop</i> <b>to</b> <i>scopeFaultHandler</i>(<i>scope</i>) <b>in</b> <i>sl</i> <b>via</b> <i>signalChannel</i><sub>down</sub> <b>set</b> <i>scopeMode</i> <b>from</b> <i>negative</i> <b>to</b> <i>faulted</i> <b>set</b> <i>activityState</i> <b>from</b> <i>stopping</i> <b>to</b> <i>stopped</i> DISABLEOUTGOINGLINKS(<i>sl</i>, <i>scope</i>)</p>	(R9.4.4-75)
--	-------------

referenced in EXECUTESCOPE<sub>negative</sub> (R9.4.0-73),  
references DISABLEOUTGOINGLINKS (R5.4.2-7)

## 9.5 Event Handler

This section defines the common behaviour of all event handlers in BPEL. The specific behaviour for specific types of events is defined in the subsequent sections 9.6, 9.7.

An event handler is associated to a scope. It is enabled to react on events as long as the scope's primary activity is active. BPEL defines two types of *events*.

**onAlarm** events are timed events. They wait **for** a certain period of time or **until** time a reached a given value. An **onAlarm** event may occur at most once.

An **onMessage** event occurs each time a message arrives from a remote web service at the designated interface of the process. An **onMessage** event may occur several times.

Each time an event occurs, the according event handler executes its child activity. For the **onMessage** event handler this allows a concurrent execution of several instances of its child activity.

```
<eventHandlers>
  <onMessage partnerLink="ncname" portType="qname"
    operation="ncname" variable="ncname"?>*
```

```

    <correlations>
      <correlation set="ncname" initiate="yes|no">+
    </correlations>

    activity
  </onMessage>

  <onAlarm for="duration-expr"? until="deadline-expr"?>*
    activity
  </onAlarm>
</eventHandlers>

```

Each `onMessage` and each `onAlarm` element defines a new event handler in the meaning of a new *Activity*. Each event handler handles exactly one event.

**Universe:** *EventHandler*

see also *EventDescriptor*, cf. sec. 8.5.

static functions (9.5.0-1)

$$eventHandlerEvent : EventHandler \rightarrow EventDescriptor$$

### 9.5.1 Dead Path Elimination and Event Handlers

An event handler needn't to support the DPE because no link with its source activity being inside of an event handler is allowed to have its target activity outside of that event handler.

By this structural property, we may deduce that for any activity for which the DPE may be effective (i.e. where the incoming link needs to be evaluated to *false*), and where the DPE is initiated outside of an event handler, the existence of any event handler is irrelevant. Thus, the DPE's *signalNegateLinks* needn't to be propagated through an event handler.

### 9.5.2 Executing an Event Handler

An event handler supports the interface of the activity-to-activity communication just like a structured activity, except for the dead path elimination.

But the behaviour of an event handler handling an *onAlarm* event significantly differs from the behaviour of one which handles an *onMsg* event. Therefore, we need to define distinct ASM rules of an event handler's behaviour for each type of event: EXECUTEEVENTHANDLER<sub>onAlarm</sub> (RB.6.4-150) defines the behaviour of an event handler which handles an *onAlarm* event, EXECUTEEVENTHANDLER<sub>onMessage</sub> (RB.6.5-151) does likewise for an *onMsg* event.

Hence the ASM rule defining the behaviour of an event handler is a refinement of EXECUTEACTIVITY<sub>pattern</sub> (R5.9.0-18) together with a distinction of the behaviour by its event. Their definitions are given in the subsequent sections.

EXECUTEEVENTHANDLER (R9.5.2-76)

( $sl \in SubInstance, eh \in EventHandler$ )  $\equiv$

**if**  $eventType(eventHandlerEvent(eh)) = onAlarm$  **then**

EXECUTEEVENTHANDLER<sub>onAlarm</sub>( $sl, eh$ )

**if**  $eventType(eventHandlerEvent(eh)) = onMsg$  **then**

EXECUTEEVENTHANDLER<sub>onMessage</sub>( $sl, eh$ )

referenced in EXECUTEACTIVITY<sub>structured</sub> (RB.2.2-106)

referencing

EXECUTEEVENTHANDLER<sub>onAlarm</sub>

(RB.6.4-150),

EXECUTEEVENTHANDLER<sub>onMessage</sub> (RB.6.5-151)

## 9.6 OnAlarm Event Handler

As stated in the previous section, we conceive an *onAlarm EventHandler* to be a structured *Activity*. We give its formal semantics as a refinement of EXECUTEACTIVITY<sub>pattern,structured</sub> (R8.1.6-44). Its *eventHandlerEvent* is of type *onAlarm*.

Upon an *onAlarm EventHandler* starts handling its event, it ignores any further occurrence of this event. It completes when its child activity completes.

### 9.6.1 Starting an OnAlarm Event Handler

An *EventHandler* cannot be the target of a *Link*. Together with its transition to *running*, it has to evaluate its expressions to obtain the moment of time at which the event occurs. This behaviour is already defined for the *EventDescriptor* in INITIALARMEVENT (R8.5.1-50) at the *Pick* activity in section 8.5.1.

STARTEVENTHANDLER<sub>onAlarm</sub> (R9.6.1-77)

( $sl \in SubInstance, eh \in EventHandler$ )  $\equiv$

**set** *activityState* **from** *enabled* **to** *running*

INITALARMEVENT( $sl, eventHandlerEvent(eh)$ )

referencing INITIALARMEVENT (R8.5.1-50)

### 9.6.2 Running an OnAlarm Event Handler

Since the *onAlarm EventHandler* must not handle a second occurrence of its event, we model its behaviour in *running* in two stages: the *EventHandler* waits in the first stage *waitEvent* until its event occurs. If this happens, the event is handled by executing its child activity ( HANDLEEVENT<sub>onAlarm</sub> (R9.6.2-79)). Then the *EventHandler* transitions to *handleEvent* where it waits for its child activity to complete ( FINISHEVENT<sub>onAlarm</sub> (R9.6.2-80)).

dynamic functions (F9.6.2-1)

$$eventHandlerStage : SubInstance \times EventHandler \rightarrow \{awaitEvent, handleEvent\}$$

requirement for the initial state (9.6.2-1)

$$\forall sl \in SubInstance \forall eh \in EventHandler \\ eventType(eventHandlerEvent(eh)) = onAlarm$$

Any time during the execution of an *EventHandler* the positive control flow of its *Scope* may complete successfully, which leads to a *signalComplete* arriving from its *Scope* (COMPLETESCOPE<sub>positive</sub> (R9.2.5-65)).

The *EventHandler* immediately completes its execution if its event hasn't occurred yet; formally, while being in stage *awaitEvent*. But once the event has occurred, it may complete to handle the event. The following ASM rule formally defines this behaviour.

$$\frac{\text{RUNEVENTHANDLER}_{onAlarm}}{(sl \in SubInstance, eh \in EventHandler) \equiv} \quad (R9.6.2-78)$$

**if** *eventHandlerStage*(*sl*, *eh*) = *awaitEvent* **then**  
**onSignal** *signalComplete* **from** *parentActivity*(*sl*, *eh*)  
**in** *sl* **via** *signalChannel*<sub>down</sub> **do**  
**set** *activityState* **from** *running* **to** *completing*  
**otherwise**  
**if** *eventOccurred*(*sl*, *eventHandlerEvent*(*eh*)) = *true* **then**  
HANDLEEVENT<sub>onAlarm</sub>(*sl*, *eh*)  
**if** *eventHandlerStage*(*sl*, *eh*) = *handleEvent* **then**  
FINISHEVENT<sub>onAlarm</sub>(*sl*, *eh*)

referenced in EXECUTEEVENTHANDLER<sub>onAlarm</sub> (RB.6.4-150),  
references HANDLEEVENT<sub>onAlarm</sub> (R9.6.2-79), FINISHEVENT<sub>onAlarm</sub> (R9.6.2-80)

### Handling *onAlarm* events

An *onAlarm* event is handled by executing the *EventHandler*'s child activity. This also requires the transition to *eventHandlerStage* *handleEvent* to handle at most one occurrence of the event.

$$\frac{\text{HANDLEEVENT}_{onAlarm}}{(sl \in SubInstance, eh \in EventHandler) \equiv} \quad (R9.6.2-79)$$

**signal** *signalEnable* **to** *childActivity*(*eh*) **in** *sl* **via** *signalChannel*<sub>down</sub>  
**set** *eventHandlerStage* **from** *awaitEvent* **to** *handleEvent*  
**add** (*sl*, *childActivity*(*eh*)) **to** *activityRunningChilds*(*sl*, *eh*)

referenced in RUNEVENTHANDLER<sub>onAlarm</sub> (R9.6.2-78)

### Finishing the Handling of *onAlarm* events

To finish the handling of an *onAlarm* event, the *EventHandler* waits for its child activity to complete successfully.

$$\frac{\text{FINISHEVENT}_{onAlarm}}{(sl \in SubInstance, eh \in EventHandler)} \equiv \quad (R9.6.2-80)$$

$\text{RUNACTIVITY}_{structured}(sl, eh)$

referenced in  $\text{RUNEVENTHANDLER}_{onAlarm}$  (R9.6.2-78), references  $\text{RUNACTIVITY}_{structured}$  (RB.5.1-129)

### 9.6.3 OnAlarm Event Handler – Final Refinement

To obtain  $\text{EXECUTEEVENTHANDLER}_{onAlarm}$  (RB.6.4-150) from  $\text{EXECUTEACTIVITY}_{pattern,structured}$  (R8.1.6-44) apply the following refinement:

- Remove  $\text{PROPAGATEDPE}$ , because an event handler is not required to support DPE (see sec. 9.5.1);
- Replace  $\text{STARTACTIVITY}$  (R5.4.3-8) by  $\text{STARTEVENTHANDLER}_{onAlarm}$  (R9.6.1-77), which additionally initializes the event handler to watch for its *onAlarm* event;
- Replace  $\text{RUNACTIVITY}$  by  $\text{RUNEVENTHANDLER}_{onAlarm}$  (R9.6.2-78), which defines the positive control flow of an *onAlarm EventHandler*: to wait for the event to occur and then to execute its child activity.

The final rule  $\text{EXECUTEEVENTHANDLER}_{onAlarm}$  (RB.6.4-150) can be found in the appendix B.6.4.

## 9.7 OnMessage Event Handler

Like the *onAlarm EventHandler*, the *onMsg EventHandler* is a structured activity – its formal behaviour will be a refinement of  $\text{EXECUTEACTIVITY}_{pattern,structured}$  (R8.1.6-44). The *onMsg EventHandler* handles each occurrence of its event concurrently to previous events: for each occurrence, the *EventHandler* creates a new instance of its child activity which is in charge of processing the contents of the received message. This is done until the *EventHandler*'s *Scope* requires it to complete its execution.

### 9.7.1 Starting an OnMessage Event Handler

The *onMsg EventHandler* requires no special preparations for starting.

### 9.7.2 Running an OnMessage Event Handler

Similar to the *onAlarm EventHandler*, the *onMsg EventHandler* may be required to complete its execution by a request from its parent activity. Unless this request (*signalComplete*) is received, the *EventHandler* handles occurring events and finishes the handling of previous events concurrently.

$\frac{\text{RUNEVENTHANDLER}_{onMessage}}{(sl \in SubInstance, eh \in EventHandler) \equiv} \quad (R9.7.2-81)$

```

onSignal signalComplete from parentActivity(sl, eh)
  in sl via signalChanneldown do
    set activityState from running to completing
otherwise
  if eventOccurred(sl, event) = true then
    HANDLEEVENTonMsg(sl, eh, eventHandlerEvent(eh))
    FINISHEVENTonMsg(sl, eh)

```

referenced in EXECUTEEVENTHANDLER<sub>onMessage</sub> (RB.6.5-151),  
 references HANDLEEVENT<sub>onMsg</sub> (R9.7.2-82), FINISHEVENT<sub>onMsg</sub> (RB.6.5-153)

#### Handling *onMsg* events

The occurrence of the event handler's event is checked by *eventOccurrence* (D B.5.5-2), which is already defined for *Pick*. To handle the event, the *BPELMsg* which caused the event must be received and a new instance of the *EventHandler*'s child activity must be executed. This requires the creation of a new *SubInstance*.

Receiving a message includes copying its contents to the specified variable (see section 9.5). Although the informal specification [CGK<sup>+</sup>03] doesn't state this, from the concurrent handling of *onMsg* events follows that the variable must provide a clearly distinct value for each occurrence of the event. Furthermore, a variable is declared at a *Scope* only.

Hence it follows that the definition of an *onMsg EventHandler* implicitly defines its child activity to be a *Scope* at which the specified variable is declared. We assume this structural property to hold in the initial state. Then by receiving a message, its contents is assigned to the value of the variable in the newly created *SubInstance*.

An *onMsg* event is handled by waiting for and by receiving a message at the specified interface.

$\frac{\text{HANDLEEVENT}_{onMsg}}{(sl \in SubInstance, eh \in EventHandler, event \in EventDescriptor) \equiv} \quad (R9.7.2-82)$

```

let portdescr = eventPortDescriptor(event) in
  AWAITANDRECEIVECORRELATINGMESSAGEevent(sl, eh, portdescr)

```

referenced in `RUNEVENTHANDLERonMessage` (R9.7.2-81),  
 references `AWAITANDRECEIVECORRELATINGMESSAGEevent` (RB.6.5-152)

To define how the message is received, we refine `AWAITANDRECEIVECORRELATINGMESSAGEpattern` (R6.5.3-19) by:

- replace `port` by `localPortin`,  
 to receive the message at the local interface of the process,
- replace

`RECEIVEMESSAGEpattern(sl, portdescr, msg)`

by

`let slchild = new(SubInstance) in`  
`RECEIVEMESSAGEpattern(slchild, portdescr, msg)`  
`RUNEVENTHANDLERCHILDACTIVITY(sl, slchild, eh, event)`  
`EXTENDSUBINSTANCETREE(sl, slchild)`

to create a new instance of the child activity, to store the contents of the received message in that new subinstance and to extend the subinstance tree properly.

The full definition of `AWAITANDRECEIVECORRELATINGMESSAGEevent` (RB.6.5-152) is given in the Appendix B.6.5. Please note that by applying `RECEIVEMESSAGEpattern` (R6.5.3-20) with the parameter `slchild`, the contents of the message is assigned to the variable in the new `SubInstance`.

The operational steps to run the new instance of the child activity are well-known.

`RUNEVENTHANDLERCHILDACTIVITY` (R9.7.2-83)  
`(sl ∈ SubInstance, slchild ∈ SubInstance,`  
`eh ∈ EventHandler, event ∈ EventDescriptor) ≡`

**signal** `signalEnable` **to** `eventActivity(event)` **in** `slchild` **via** `signalChanneldown`  
**add** `(slchild, eventActivity(event))` **to** `ehRunningInstances(sl, eh)`

referenced in `AWAITANDRECEIVECORRELATINGMESSAGEevent` (RB.6.5-152)

### Finishing the Handling of `onAlarm` events

To finish the handling of an `onMsg` event, the execution of the corresponding child activity must complete. This behaviour is known from `RUNACTIVITYpattern` (R8.1.2-42). Yet, even if there are no running child activities, the `EventHandler` may not complete its execution. We refine `RUNACTIVITYpattern` (R8.1.2-42) by

- Instantiate `RUNACTIVITYNORUNNINGCHILDS` with **skip** ,  
 which leads to removing the **if** operator.

to obtain `FINISHEVENTonMsg` (RB.6.5-153). The definition is given in the Appendix B.6.5.

### 9.7.3 Completing an OnMessage Event Handler

An *EventHandler* which has reached *completing* behaves like any structured activity and waits for the completion of all instances of its child activity. Hence another refinement of  $\text{RUNACTIVITY}_{pattern}$  (R8.1.2-42) applies:

- Instantiate  $\text{RUNACTIVITYNORUNNINGCHILDS}$  with

```
set activityState from completing to completed
signal signalCompleted to parentActivity(eh) in sl via signalChannelup
```

to finish the execution of the *EventHandler* and inform the *Scope* about the result.

The full definition of  $\text{COMPLETEEVENTHANDLER}_{onMessage}$  (RB.6.5-154) is given in the Appendix B.6.5. **Note:** An *onMsg EventHandler* cannot be the source of a *Link*. Therefore  $\text{EVALUATEOUTGOINGLINKS}$  is not applied.

### 9.7.4 OnMessage Event Handler – Final Refinement

To obtain  $\text{EXECUTEEVENTHANDLER}_{onMessage}$  (RB.6.5-151) from  $\text{EXECUTEACTIVITY}_{pattern,structured}$  (R8.1.6-44) apply the following refinement:

- Remove  $\text{PROPAGATEDPE}$ ,  
because an event handler is not required to support DPE (see sec. 9.5.1);
- Replace  $\text{RUNACTIVITY}$  by  $\text{RUNEVENTHANDLER}_{onMessage}$  (R9.7.2-81),  
which defines the positive control flow of an *onAlarm Event Handler*: to wait for an incoming message to arrive and execute a new instance of its child activity, and to wait for any instance of its child activity to complete its execution;
- Replace  $\text{COMPLETEACTIVITY}$  by  $\text{COMPLETEEVENTHANDLER}_{onMessage}$  (RB.6.5-154),  
which defines that an *onMessage Event Handler* may complete only, if all instances of its child activity completed.

The final rule  $\text{EXECUTEEVENTHANDLER}_{onMessage}$  (RB.6.5-151) can be found in the appendix B.6.5.

## 10 Compensation Handling

The Compensation Handling mechanism in BPEL extends the semantics which we presented in the previous chapters. The entire mechanism involves some new behaviour of the *Scope*, and two new activities – the *compensation handler* and the *compensate* activity – together with some structures to coordinate the joint behaviour. We will also formalize the process’ property to compensate an instance whenever it completes faultlessly.

We present their definition in two steps. Initially, in Section 10.1, we define abstract semantics by using abstract functions and postponing a detailed definition of some ASM rules. This way we capture the level of uncertainty about the concrete structures and operational steps of the compensation handling mechanism as it is given in the informal specification.

In a second step we propose a most abstract solution to the unspecified aspects of compensation handling such that the solution seamlessly integrates with the semantics that have been presented in the previous chapters. This solution, which is given in Sections 10.2, 10.3 and 10.4, reduces the amount of informally specified functions and rules to a single abstract relation. This relation defines the correct order of compensation of completed scopes. We must assume this relation to be abstract because no proper formal or informal specification has been given yet.

We will close this chapter by providing the semantics for the compensation of a complete process instance wrt. the `enableInstanceCompensation` property of the process.

**Important Note:** Regarding the process state which is seen by a compensation handler, we explicitly refer to section 13.3.2. “Process State Usage in Compensation Handlers” of the informal BPEL specification working draft published by OASIS at 2004-09-08. [ABC<sup>+</sup>04].

### 10.1 Abstract Semantics for Compensation Handling

The formal definition of the behaviour of a scope’s compensation handling mechanism follows the conception that a scope never proactively seeks to execute its compensation. Instead we conceive the *compensate* activity and the implicit compensation mechanism to be the only source of invoking behaviour for compensation. We refer to them as *compensating entities*. The operation to invoke compensation is named *compensation call*.

From this conception we can derive two requirements:

1. a scope which becomes available for compensation must publish this information to the compensating entities, and
2. a scope which is available for compensation must provide an interface which is called by the compensating entities.

The first requirement gives rise to some shared structures, which are accessible from the scope and the compensating entities. The second requirement directly turns into basic requirements on the behaviour of a scope and of the compensating entities.

### 10.1.1 Required Shared Structures

The default order compensation (compensate all scopes in the reverse order of completion) is available only, if there was no named compensation call yet.

$$\text{defaultOrderCompensationAvailable} : \text{SubInstances} \times \text{Scope} \rightarrow \{\text{true}, \text{false}\}$$

requirement for the initial state (10.1.1-1)

$$\forall \text{sl} \in \text{SubInstance} \forall \text{scope} \in \text{Scope} \\ \text{defaultOrderCompensationAvailable}(\text{sl}, \text{scope}) = \text{true}$$

The compensating entities need to know about any instance of scopes which are directly enclosed and which have installed a compensation handler. A set will store all completed instances of scopes. From now on, we call such a pair of a *compensation module*. Since all compensating entities are associated to the scope which encloses the concerned compensation modules, we declare the following function.

dynamic functions (F10.1.1-1)

$$\text{scopeCompensationModules} : \text{SubInstances} \times \text{Scope} \rightarrow \mathcal{P}(\text{SubInstances} \times \text{Scope})$$

This set is *not* to be considered as a property of the scope, but as a shared property of the scope and all of its enclosed activities *act* which may reach the scope structurally by computing  $\text{enclosingScope}(\text{act})$  (cf. sec. B.1.1).

### 10.1.2 Compensation Handling in Scopes

We will now address the second requirement above.

#### Preparing a Scope for Compensation

Before a successfully completed scope can be compensated it needs to prepare itself for compensation. To do so, the completed instance of the scope must register its compensation module such that it is available for compensation. Then the scope needs to change its internal behaviour to be able to react on a compensation call.

Upon reaching internal state *completed* in the scope mode *positive*, ASM rule `INSTALLCOMPENSATIONHANDLER` is applied.

INSTALLCOMPENSATIONHANDLER (R10.1.2-84)

( $sl \in SubInstance, scope \in Scope$ )  $\equiv$

REGISTERCOMPENSATIONMODULE( $sl, scope$ )  
PREPARESCOPEFORCOMPENSATION( $sl, scope$ )

referenced in COMPLETESCOPE (R9.2.5-67)

references REGISTERCOMPENSATIONMODULE (R10.3.1-93), PREPARESCOPEFORCOMPENSATION (R10.3.1-95)

### Behaviour in the Case of Compensation

A scope which has prepared for compensation and which follows the second requirement waits for its compensation mechanism to be called. If the mechanism is called, it handles this call by executing its compensation handler.

The compensating entity which sent the call waits for the scope to complete its compensation. The completion of the compensation handler leads to the completion of compensation at the scope. The successful compensation is *confirmed* at the compensating entity.

Furthermore, a fault occurring within the compensation handler cannot be handled at the scope anymore since the scope already completed. The faults must be propagated to some entity that can handle it.

EXECUTESCOPE<sub>compensate, pattern</sub> (R10.1.2-85)

( $sl \in SubInstance, scope \in Scope$ )  $\equiv$

**if** *receivedCompensationCalls*( $sl, scope$ )  $\neq \emptyset$  **then**

HANDLECOMPENSATIONCALL<sub>pattern</sub>( $sl, scope$ )

**onSignal** *signalCompleted* **from** *scopeCompensationHandler*( $scope$ )

**in**  $sl$  **via** *signalChannel<sub>up</sub>* **do**

CONFIRMCOMPENSATION( $sl, scope$ )

**if** *faultChannel<sub>up</sub>*( $sl, scopeCompensationHandler$ ( $scope$ ),  $scope$ )  $\neq \emptyset$  **then**

PROPAGATEFAULTSFROMCH( $sl, scope$ )

references HANDLECOMPENSATIONCALL<sub>pattern</sub> (R10.1.2-86), CONFIRMCOMPENSATION (R10.1.2-87), PROPAGATEFAULTSFROMCH (R10.1.2-88)

Firstly we must assume that there is more than compensating entity which sent a call. Each call must be handled. Secondly, calling the same compensation module twice is not defined. The fault *repeatedCompensation*  $\in$  *FaultName* must be thrown to the compensating entity which sent a second or later call.

---

HANDLECOMPENSATIONCALL<sub>pattern</sub> (R10.1.2-86)

$(sl \in SubInstance, scope \in Scope) \equiv$

**if**  $scopeCHcalled(sl, scope) = false$  **then**

**choose**  $compensationCall \in receivedCompensationCalls(sl, scope)$  **in**

**remove**  $compensationCall$  **from**  $receivedCompensationCalls(sl, scope)$

$scopeCompensationCall(sl, scope) := compensationCall$

**signal**  $signalEnable$  **to**  $scopeCompensationHandler(scope)$  **in**  $sl$  **via**  $signalChannel_{down}$

$scopeCHcalled(sl, scope) := true$

**else**

**let**  $fault = new(Fault)$  **in**

$faultName(fault) := repeatedCompensation$

    NOTIFYFAULTEDCOMPENSATION( $sl, scope, scopeCompensationCall(sl, scope), fault$ )

references NOTIFYFAULTEDCOMPENSATION (R10.3.2-97)

Hence at most one compensation call is handled by executing the compensation handler. Compensation is confirmed at the entity which sent the call once the compensation handler completes.

CONFIRMCOMPENSATION (R10.1.2-87)

$(sl \in SubInstance, scope \in Scope) \equiv$

  NOTIFYCOMPLETEDCOMPENSATION( $sl, scope, scopeCompensationCall(sl, scope)$ )

referenced in EXECUTESCOPE<sub>compensate</sub> (RB.7.1-155),  
 references NOTIFYCOMPLETEDCOMPENSATION (R10.3.2-96)

Likewise, occurring faults are propagated to the sender of the compensation call. Since a compensation handler may send a fault only if it is executed due to a compensation call, the defined behaviour of propagating faults is sound.

PROPAGATEFAULTSFROMCH (R10.1.2-88)

$(sl \in SubInstance, scope \in Scope) \equiv$

**let**  $ch = scopeCompensationHandler(scope),$

$cc = scopeCompensationCall(sl, scope)$  **in**

**forall**  $fault$  **in**  $faultChannel_{up}(sl, ch, scope)$  **do**

**remove**  $fault$  **from**  $faultChannel_{up}(sl, ch, scope)$

    NOTIFYFAULTEDCOMPENSATION( $sl, scope, cc, fault$ )

referenced in EXECUTESCOPE<sub>compensate</sub> (RB.7.1-155),  
 references NOTIFYFAULTEDCOMPENSATION (R10.3.2-97)

### 10.1.3 Compensation Handler

A compensation handler defines an activity which is executed if its scope is called to be compensated.

```
<scope ... >
  <compensationHandler>?
    activity
  </compensationHandler>
  ...
</scope>
```

There is nothing special about the compensation handler. It simply wraps the execution of its child activity.<sup>1</sup>

**Universe:** *CompensationHandler*

<p><u>STARTCOMPENSATIONHANDLER</u> (<math>sl \in SubInstance, ch \in CompensationHandler</math>) <math>\equiv</math> <b>set</b> <i>activityState</i> <b>from</b> <i>enabled</i> <b>to</b> <i>running</i> <b>add</b> (<math>sl, childActivity(ch)</math>) <b>to</b> <i>activityRunningChilds(sl, ch)</i> <b>signal</b> <i>signalEnable</i> <b>to</b> <i>childActivity(ch)</i> <b>in</b> <math>sl</math> <b>via</b> <i>signalChannel<sub>down</sub></i></p>	<p>(R10.1.3-89)</p>
---	---------------------

referenced in EXECUTECOMPENSATIONHANDLER (RB.7.2-157)

The behaviour of a *CompensationHandler* in any other of the internal states has already been specified. The refinement of EXECUTEACTIVITY<sub>pattern,structured</sub> (R8.1.6-44) is straight forward. The full definition is given in the Appendix B.7.2.

### 10.1.4 Compensate

A compensate activity calls the compensation handler of a scope. It may be defined within a fault handler or a compensation handler. A compensate activity can call the compensation handler of a scope which is enclosed by their directly enclosing scope only. A compensate activity may either call the compensation handler of a specific scope (**scope=".."**) or the compensation handler of all scopes.

The compensation handlers are called in their “reverse order of completion”. A compensate activity is a compensating entity.

```
<compensate scope="ncname"? standard-attributes>
  standard-elements
</compensate>
```

---

<sup>1</sup>According to the semantics defined in this document so far, there is no need for a compensation handler activity with own behaviour at all.

**Universe:** *Compensate*

$$\text{compensateScope} : \text{Compensate} \rightarrow \text{Scope} \quad (\text{name} = "..")$$

The compensation handling of BPEL requires that the compensation handlers of the registered compensation modules are called in some kind of order. This order hasn't explicitly been defined yet. The default compensation order suggests some notion of stack which is worked off by a *Compensate* activity, we call this implicitly given structure the *compensation stack*.

Therefore we assume that the top of the compensation stack can be computed in the current state for a given instance of a *Compensate* activity: We introduce an abstract function to access the compensation modules which have to be compensated by a given instance of *Compensate* activity in the next step. There might be more than just one compensation module on the top of this stack in case there is no causal dependency for their compensation.

$$\text{A } \text{compensateNextTOS} : \text{SubInstance} \times \text{Compensate} \rightarrow \mathcal{P}(\text{SubInstance} \times \text{Scope})$$

*nextCompensationModules* returns the subset of compensation modules which are stored within *scopeCompensationModule* of the enclosing scope and which may be called by the given *Compensate* activity at the time this function is evaluated; *nextCompensationModules* returns the empty set if there is no more scope to be compensated. The top of stack denotes the set of all compensation modules which may be called *concurrently*. The definition of *compensateScope* for the given compensate activity restricts the returned set accordingly.

Since a *Compensate* activity not only invokes the compensation handler, but also waits until compensation is confirmed for each of the called modules, the top of the compensation stack must be memorized by the compensate activity – comparable to the set of running child activities of an activity.

$$\text{compensateTOS} : \text{SubInstance} \times \text{Compensate} \rightarrow \mathcal{P}(\text{SubInstance} \times \text{Scope})$$

### Starting Compensate

An enabled *Compensate* must satisfy its join condition at first. If the condition is satisfied it gets the top of the stack of the compensation modules it has to call and switches to *running*. The top

In case the *Compensate* has no *compensateScope* specified, it is meant to call all compensation modules. But this may happen only, if it is the first compensating entity accessing the compensation stack in this kind, i.e. if *defaultOrderCompensationAvailable* evaluates to *true*. Otherwise, the activity completes directly.

$$\frac{\text{STARTCOMPENSATE}_{\text{pattern}}}{(\text{sl} \in \text{SubInstance}, \text{comp} \in \text{Compensate})} \equiv \quad (\text{R10.1.4-90})$$

```

if  $allLinksSet(sl, act) \wedge joinConditionSatisfied(sl, act) = true$  then
  let  $scope = enclosingScope(comp)$ ,
     $sl_{scope} = currentSubInstanceFromInner(sl, scope)$  in
    if  $compensateScope(scope) = undef$ 
       $\wedge defaultOrderCompensationAvailable(sl_{scope}, scope) = false$  then
        set  $activityState$  from  $enabled$  to  $completing$ 
      else
        set  $activityState$  from  $enabled$  to  $running$ 
         $POP\_COMPENSATION\_STACK_{pattern}(sl, comp)$ 
        if  $compensateScope(comp) \neq undef$  then
           $defaultOrderCompensationAvailable(sl_{scope}, scope) := false$ 

```

references  $POP\_COMPENSATION\_STACK_{pattern}$  (R10.1.4-91)

In the abstract setting, popping the compensation stack is straight forward by the help of the abstract function  $compensateNextTOS$ .

$$\frac{POP\_COMPENSATION\_STACK_{pattern}}{(sl \in SubInstance, comp \in Compensate)} \equiv$$

$$compensateTOS(sl, comp) := compensateNextTOS(sl, comp) \quad (R10.1.4-91)$$

referenced in  $START\_COMPENSATE_{pattern}$  (R10.1.4-90),  $RUN\_COMPENSATE_{pattern}$  (R10.1.4-92)

### Running Compensate

Compensation by a compensate activity works in two stages: Firstly, call each compensation module in the current top of the stack of the compensate activity ( $callCompensation$ ). Then wait for each called module to confirm the completion of the compensation ( $awaitingCompensated$ ). When this happens, the next compensation modules are popped from the compensation stack.

It might happen that the compensation of a called module fails due to a fault. In such a case, the  $Compensate$  activity is notified likewise and it behaves as if the fault occurred directly at the  $Compensate$  activity.

This behaviour is iterated while the stack is not empty.

**Universe:**  $CompensationMode =_{def} \{callCompensation, awaitingCompensated\}$

$$compensationMode : SubInstance \times Compensate \rightarrow CompensationMode$$

$$\frac{RUN\_COMPENSATE_{pattern}}{(sl \in SubInstance, comp \in Compensate)} \equiv$$

$$\mathbf{let} \ TOS = compensateTOS(sl, comp) \ \mathbf{in}$$

```

if TOS =  $\emptyset$  then
  set activityState from running to completing
else
  if compensationMode(sl, comp) = callCompensation then
    forall (sl', scope')  $\in$  TOS do
      CALLCOMPENSATION(sl, comp, sl', scope')
      compensationMode(sl, comp) := awaitingCompensated
  if compensationMode(sl, comp) = awaitingCompensated then
    if  $\exists$ (sl', scope')  $\in$  TOS
      compensationFailed(sl, comp, sl', scope') = true then
        COMPENSATERETHROWFAULT(sl, comp)
        set activityState from running to faulted
    else if  $\forall$  (sl', scope')  $\in$  TOS
      compensationCompleted(sl, comp, sl', scope') = true then
        POPCOMPENSATIONSTACKpattern(sl, comp)
        compensationMode(sl, comp) := callCompensation

```

references POPCOMPENSATIONSTACK<sub>pattern</sub> (R10.1.4-91)

Consider CALLCOMPENSATION and COMPENSATERETHROWFAULT to be abstract rules, and consider *compensationCompleted* and *compensationFailed* to be abstract functions. Either shall be defined such that together with the abstract rules of a scope, calling a compensation module and confirming compensation becomes sound. A detailed definition for a refined notion of *Compensate* will be given in a later section.

### Completing Compensate

Completing *Compensate* is identical to completing a normal BPEL activity.

### Stopping Compensate

Stopping *Compensate* is not defined.

## 10.2 Structures for Compensation Handling

In this section we present a refined notion of the abstract function *compensateNextTOS* and the unspecified ASM rules which were given in the previous section. The refinement aims on integrating the compensating handling mechanism into the formal semantics of the preceding chapters. Furthermore, the loose ends in the definition of the scope and the compensate activity will be defined operationally.

### 10.2.1 Activity-to-Activity Communication for Compensation Handling

We cannot use the activity-to-activity communication used in the previous chapters for calling a compensation module. The signal channels which are defined along the tree of

instances of activities are not applicable: compensation is called directly and does not involve the instances of activities between the compensating entity and the called module. We evolve the concept of the signal channel for the purpose of compensation by introducing two new types channels and an “open” port.

A scope receives calls to invoke compensation handling from any activity that is allowed to do so. For the time being, calling is restricted to the *Compensate* activity and the implicit compensation handling<sup>2</sup>. A priori, the initiator of such a call is not known to the called scope. This has two implications:

Firstly, the call which invokes compensation handling must include the reference to the caller (to the instances of the calling activity).

**Universe:** *CompensationCall*

**Universe:** *CompensationCaller* =<sub>def</sub> *Compensate*

$$\begin{aligned} \textit{compensationCaller} & : \textit{CompensationCall} \rightarrow \textit{CompensationCaller} \\ \textit{compensationCallerSI} & : \textit{CompensationCall} \rightarrow \textit{SubInstance} \end{aligned}$$

Secondly, the instance of a scope must be able to receive compensation calls from any activity which knows that instance. Avoiding to introduce a signal channel for compensation calls to *each* activity which might call the scope (as this is an arbitrary but fixed number), we define an “open” port (consider it as a mailbox) located at the scope. An activity which knows the instance of the scope then may place a compensation call there. It is the strongest assumption on the structures we may impose so far.

dynamic functions (F10.2.1-1)

$$\textit{compensationPort}_{in,scope} : \textit{SubInstances} \times \textit{Scope} \rightarrow \mathcal{P}(\textit{CompensationCall})$$

The initiator of a compensation call needs to know about the success or failure of this call. Having received a compensation call, a scope knows which instance of activity is waiting for a confirmation of the call. Likewise, the caller obviously knows which instance of which scope was called. Thus we may employ the concept of the signal channel again. Each channel, directed from the called scope to its caller, is defined by a pair of instances of activities.<sup>3</sup>

dynamic functions (F10.2.1-2)

$$\begin{aligned} \textit{compensationChannel}_{confirm} & : \\ & \textit{SubInstances} \times \textit{Scope} \times \textit{SubInstances} \times \textit{CompensationCaller} \rightarrow \mathcal{P}(\textit{UpSignals}) \\ \textit{compensationChannel}_{fault} & : \\ & \textit{SubInstances} \times \textit{Scope} \times \textit{SubInstances} \times \textit{CompensationCaller} \rightarrow \mathcal{P}(\textit{Fault}) \end{aligned}$$

---

<sup>2</sup>We assume that the implicit compensation handling can be translated into a definition of the process where a compensate activity is executed.

<sup>3</sup>Since the caller and the scope are not neighbors of the activity tree, each subinstance must be given explicitly.

### 10.2.2 The Compensation Stack

We already introduced *scopeCompensationModules* in section 10.1.1 and the abstract function *compensateNextTOS* which together behave like a stack, although there is no explicit stack structure. We will not change the way the contents of the stack is specified as this is not covered by the informal specification. But we will refine the notion of “top of stack” from the perspective of a *Compensate* activity which *is* covered by the required behaviour.

Firstly, we introduce an abstract relation  $\leq_{comp}$  defined over compensation modules:

$$\mathcal{A} \quad \leq_{comp} \subseteq (\text{SubInstance} \times \text{Scope}) \times (\text{SubInstance} \times \text{Scope})$$

where  $(sI, scope) \leq_{comp} (sI', scope')$  iff  $(sI, scope)$  must not be compensated before  $(sI', scope')$ .

Then, for a given set of compensation modules, we can easily compute the set of compensation modules which may be compensated next.

derived functions ( $\mathcal{D}$  10.2.2-1)

$$\mathcal{D} \text{ nextCompensationModules} : \mathcal{P}(\text{SubInstance} \times \text{Scope}) \rightarrow \mathcal{P}(\text{SubInstance} \times \text{Scope})$$

$$\text{CompModules} \mapsto \{(sl, scope) \in \text{CompModules} \mid \forall (sl', scope') \in \text{CompModules} \\ (sl', scope') \leq_{comp} (sl, scope)\}$$

We may conceive the result of this function as the “top of the stack” which is induced by  $\leq_{comp}$  on a given set of compensation modules.

## 10.3 Compensation Handling in Scopes

The formal definition of the behaviour of a scope which is compensated needs to be joined with the scope’s ASM rules of chapter 9. The behaviour of a scope that is compensated is completely disjoint from the behaviour of a scope we have presented so far. This gives rise to a new scope mode:

**Universe:**  $\text{ScopeMode} = \{\text{positive}, \text{negative}, \text{faulted}, \text{compensate}\}$

### 10.3.1 Preparing a Scope for Compensation

As abstractly specified in section 10.1.2, a scope prepares for compensation by registering its compensation module and switching its internal behaviour (`INSTALLCOMPENSATIONHANDLER` (R10.1.2-84)).

Having defined *scopeCompensationModules* (F10.1.1-1), registering a compensation module for compensation is fairly easy: just push it to the “stack” by adding it to *scopeCompensationModules*. The abstract relation  $\leq_{comp}$  will do the ordering.

REGISTERCOMPENSATIONMODULE (R10.3.1-93)

$(sl \in SubInstance, scope \in Scope) \equiv$

PUSHTOCOMPENSATIONSTACK( $sl, scope$ )

referenced in INSTALLCOMPENSATIONHANDLER (R10.1.2-84)

references PUSHTOCOMPENSATIONSTACK (R10.3.1-94)

PUSHTOCOMPENSATIONSTACK (R10.3.1-94)

$(sl \in SubInstance, scope \in Scope) \equiv$

**let**  $scope' = enclosingScope(scope)$ ,  
 $sl' = currentSubInstance_{fromInner}(sl, scope)$  **in**  
**add**  $(sl, scope)$  **to**  $scopeCompensationModules(sl', scope')$

referenced in REGISTERCOMPENSATIONMODULE (R10.3.1-93)

Switching the behaviour is a simple transition of  $scopeMode$ .

PREPARESCOPEFORCOMPENSATION (R10.3.1-95)

$(sl \in SubInstance, scope \in Scope) \equiv$

**set**  $scopeMode$  **from**  $positive$  **to**  $compensate$

referenced in INSTALLCOMPENSATIONHANDLER (R10.1.2-84)

### 10.3.2 Behaviour in the Case of Compensation

Based on the structures of section 10.2 we may define the concrete operational steps of a scope being compensated by refining the rules of section 10.1.2.

Refine HANDLECOMPENSATIONCALL $_{pattern}$  (R10.1.2-86) by

- Replace  $receivedCompensationCalls$  by  $compensationPort_{in,scope}$ , which makes the the scope handle compensation calls which are sent to the dedicated port.

which results in HANDLECOMPENSATIONCALL (RB.7.1-156).

Refine EXECUTESCOPE $_{compensate,pattern}$  (R10.1.2-85) by

- Replace  $receivedCompensationCalls$  in by  $compensationPort_{in,scope}$ , which makes the scope listen for compensation calls on the dedicated port.

- Replace  $\text{HANDLECOMPENSATIONCALL}_{pattern}$  by  $\text{HANDLECOMPENSATIONCALL}$ , to apply the refined ASM rule for handling compensation calls.

which results in  $\text{EXECUTESCOPE}_{compensate}$  (RB.7.1-155). The full definitions are given in the Appendix B.7.1.

Having introduced the compensation channels (F10.2.1-2), the formal definition of confirming a compensation call and notifying a faulted compensation is fairly simple. To confirm add  $signalCompleted$  to the channel which ends at the caller of the compensation call.

NOTIFYCOMPLETEDCOMPENSATION (R10.3.2-96)  
 $(sl \in SubInstance, scope \in Scope, cc \in CompensationCall) \equiv$

**let**  $act = compensationCaller(cc)$ ,  
 $sl_{act} = compensationCallerSI(cc)$  **in**  
**add**  $signalCompleted$  **to**  $compensationChannel_{confirm}(sl, scope, sl_{act}, act)$

referenced in  $\text{CONFIRMCOMPENSATION}$  (R10.1.2-87)

To notify about a faulted compensation, add the fault to the channel which ends at the caller of the compensation call.

NOTIFYFAULTEDCOMPENSATION (R10.3.2-97)  
 $(sl \in SubInstance, scope \in Scope, cc \in CompensationCall, fault) \equiv$

**let**  $act = compensationCaller(cc)$ ,  
 $sl_{act} = compensationCallerSI(cc)$  **in**  
**add**  $fault$  **to**  $compensationChannel_{fault}(sl, scope, sl_{act}, act)$

referenced in  $\text{HANDLECOMPENSATIONCALL}_{pattern}$  (R10.1.2-86),  $\text{PROPAGATEFAULTSFROMCH}$  (R10.1.2-88)

### 10.3.3 Integrating Compensation Handling into the Scope

To add the behaviour of a scope in case of compensation we need to extend  $\text{EXECUTESCOPE}$  (R9.1.0-58) as we defined it in the last chapter 9. Compensating a scope is disjoint from its behaviour in either of the scope modes *positive*, *negative* and *faulted*. A scope enters its mode *compensate* if it is ready to participate in compensation handling. The behaviour for this case was already defined in the previous subsection 10.3.2.

Furthermore, being in that scope mode, the scope doesn't need to handle a  $signalStop$  coming from its parent activity: The scope reaches this mode only if it completed its execution successfully. Hence, it has sent  $signalCompleted$  to its parent. From the stop-concept (cf. sec. 5.5.3) we may deduce that the scope's parent activity does not expect the  $signalStop$  to be handled by a scope that is in the scope mode *compensate*.

Therefore, we may conservatively extend  $\text{EXECUTESCOPE}$  (R9.1.0-58) to define  $\text{EXECUTESCOPE}_{compensationHandling}$  (R10.3.3-98).

$$\frac{\text{EXECUTESCOPE}_{\text{compensationHandling}}}{(\text{sl} \in \text{SubInstance}, \text{scope} \in \text{Scope})} \equiv \text{EXECUTESCOPE}(\text{sl}, \text{scope})$$

**if**  $\text{scopeMode}(\text{sl}, \text{scope}) = \text{compensate}$  **then**  
 $\text{EXECUTESCOPE}_{\text{compensate}}(\text{sl}, \text{scope})$

(R10.3.3-98)

referenced in EXECUTEACTIVITY<sub>structured</sub> (RB.2.2-106)references EXECUTESCOPE (R9.1.0-58), EXECUTESCOPE<sub>compensate</sub> (RB.7.1-155)

## 10.4 Compensate

Finally, we may refine the behaviour of the *Compensate* activity, now using the more detailed structures which were introduced in section 10.2.

We want to get rid of the abstract function *compensateNextTOS* which evaluates to the set of compensation modules which are to be called next by a given *Compensate* activity. Instead, we would like to represent the state of the compensation stack explicitly by the help of the abstract relation  $\leq_{\text{comp}}$ .

But we do not want to modify *scopeCompensationModules* (F10.1.1-1) to update the state of the compensation stack: The behaviour of compensation is defined locally for the execution of a *Compensate* activity in the informal specification [CGK<sup>+</sup>03]. Especially no statements about race conditions of concurrent *Compensate* activities are made. Hence the notion of a global state of the compensation stack is not defined.

This fact suggests to define a “private” state of the compensation stack for each *Compensate* activity. It is a property of the *Compensate*’s instance. Modifications of the stack – like popping – are not visible to the rest of the process.

### 10.4.1 The Private Compensation Stack

The set of all compensation modules a compensate activity has to call can be calculated by a derived function. It is a restriction of *scopeCompensationModules* (F10.1.1-1) by the activity’s *compensateScope* property.

derived functions ( $\mathcal{D}$  10.4.1-1)

$$\mathcal{D} \text{ compensateCompensationModules} : \\ \text{SubInstances} \times \text{Compensate} \rightarrow \mathcal{P}(\text{SubInstance} \times \text{Scope})$$

$$(\text{sl}, \text{comp}) \mapsto \{(\text{sl}', \text{scope}') \mid \begin{aligned} &\text{scope} = \text{enclosingScope}(\text{comp}) \\ &\wedge \text{sl}_{\text{scope}} = \text{currentSubInstance}_{\text{fromInner}}(\text{sl}, \text{scope}) \\ &\wedge (\text{sl}', \text{scope}') \in \text{scopeCompensationModules}(\text{sl}_{\text{scope}}, \text{scope}) \\ &\wedge (\text{compensateScope}(\text{comp}) \neq \text{undef} \\ &\quad \Rightarrow \text{compensateScope}(\text{comp}) = \text{scope}') \} \end{aligned}$$

Compensation modules which already have been called by a compensate activity do not have to be called again by this activity. We declare the following dynamic function to map to the set of all compensation modules which confirmed a successful compensation.

dynamic functions (F10.4.1-1)

$$\begin{aligned} & \textit{compensateFinishedModules} : \\ & \textit{SubInstances} \times \textit{Compensate} \rightarrow \mathcal{P}(\textit{SubInstance} \times \textit{Scope}) \end{aligned}$$

requirement for the initial state (10.4.1-1)

$$\begin{aligned} & \forall \textit{sl} \in \textit{SubInstance} \forall \textit{comp} \in \textit{Compensate} \\ & \textit{compensateFinishedModules}(\textit{sl}, \textit{comp}) = \emptyset \end{aligned}$$

By these functions, the set of all compensation modules which are to be called next by a compensate activity can easily be computed from the difference: Let **stack** be the set of all compensation modules for this activity and let **finished** be the set of successfully compensated modules. Then

$$\textit{nextCompensationModules}(\textit{stack} \setminus \textit{finished})$$

is the “private” top of stack of the compensate activity (see function definition ( $\mathcal{D}$  10.2.2-1)). Working down the compensation stack is fairly simple,

1. get the top of the private stack,
2. call each of the modules on top of stack and await the confirmation of their successful compensation,
3. pop the next compensation modules from the stack by adding the compensated modules to the finished ones and starting all over.

Let  $\textit{compensateTOS}(\textit{sl}, \textit{comp})$  be a set of compensation modules which have confirmed a successful compensation to the compensate activity  $(\textit{sl}, \textit{comp})$ . Then the application of the following ASM rule adds these modules to the set of the  $\textit{compensateFinishedModules}$  (F10.4.1-1) and pops the private compensation stack. The new top of stack is assigned to  $\textit{compensateTOS}(\textit{sl}, \textit{comp})$  for calling the compensation.

POPCOMPENSATIONSTACK (R10.4.1-99)

$$(\textit{sl} \in \textit{SubInstance}, \textit{comp} \in \textit{Compensate}) \equiv$$

```

let stack = compensateCompensationModules(sl, comp),
    finished = compensateFinishedModules(sl, comp),
    currentTOS = compensateTOS(sl, comp) in
let nextModules = nextCompensationModules(stack \ (finished  $\cup$  currentTOS)) in
    compensateTOS(sl, comp) := nextModules
    compensateFinishedModules(sl, comp) := finished  $\cup$  currentTOS

```

refines POPCOMPENSATIONSTACK<sub>pattern</sub> (R10.1.4-91)

referenced in STARTCOMPENSATE (RB.7.3-159), RUNCOMPENSATE (RB.7.3-160)

### 10.4.2 Starting Compensate

The only abstract aspect of  $\text{STARTCOMPENSATE}_{pattern}$  (R10.1.4-90) was the behaviour of popping the compensation stack. Hence we refine  $\text{STARTCOMPENSATE}_{pattern}$ :

- Replace  $\text{POPCOMPENSATIONSTACK}_{pattern}$  by  $\text{POPCOMPENSATIONSTACK}$ ,

which results in  $\text{STARTCOMPENSATE}$  (RB.7.3-159). The definition is given in Appendix B.7.3

### 10.4.3 Running Compensate

The behaviour of a running compensate we defined in section 10.1.4 abstracts from the handling of the compensation stack and how the compensate activity communicates with a compensation module.

To capture the new level of abstraction, we refine  $\text{RUNCOMPENSATE}_{pattern}$  (R10.1.4-92) and we provide definitions for the functions *compensationCompleted* and *compensationFailed* and for the referenced, but yet undefined ASM rules  $\text{CALLCOMPENSATION}$  and  $\text{COMPENSATERETHROWFAULT}$ .

The mentioned refinement is straight forward:

- Replace  $\text{POPCOMPENSATIONSTACK}_{pattern}$  by  $\text{POPCOMPENSATIONSTACK}$ .

which results in  $\text{RUNCOMPENSATE}$  (RB.7.3-160). The definition is given in Appendix B.7.3.

To finally call a compensation module from the top of the stack, its *compensationPort<sub>in</sub>* must receive a *CompensationSignal* which points back to the compensate activity. The latter requirement ensures that the called scope informs the calling activity about failure or success.

$\text{CALLCOMPENSATION}$	(R10.4.3-100)
$(\text{sl} \in \text{SubInstance}, \text{comp} \in \text{CompensationCaller}, \text{sl}' \in \text{SubInstance}, \text{scope}' \in \text{Scope}) \equiv$	
$\text{let } \text{cs} = \text{new}(\text{CompensationSignal}) \text{ in}$	
$\text{compensationCallerSI}(\text{cs}) = \text{sl}$	
$\text{compensationCaller}(\text{cs}) = \text{comp}$	
$\text{add cs to compensationPort}_{in,scope}(\text{sl}', \text{scope}')$	

referenced in  $\text{RUNCOMPENSATE}$  (RB.7.3-160),  $\text{CALLINSTANCECOMPENSATION}$  (R10.5.0-101)

Using the communication structures introduced in section 10.2, the compensation of a scope completed if *signalCompleted* comes from the called compensation module via *compensationChannel<sub>confirm</sub>* (F10.2.1-2). A module failed its compensation if a *Fault* is in the respective *compensationChannel<sub>fault</sub>*.

derived functions ( $\mathcal{D}$  10.4.3-1)

$\mathcal{D}$  *compensationCompleted* :

$SubInstance \times CompensationCaller \times SubInstance \times Scope \rightarrow \{true, false\}$

$$(sl, comp, sl', scope') \mapsto \begin{cases} true, & signalCompleted \in \\ & compensationChannel_{confirm}(sl', scope', sl, comp) \\ false, & else \end{cases}$$

$\mathcal{D}$  *compensationFailed* :

$SubInstance \times CompensationCaller \times SubInstance \times Scope \rightarrow \{true, false\}$

$$(sl, comp, sl', scope') \mapsto \begin{cases} true, & compensationChannel_{fault}(sl', scope', sl, comp) \neq \emptyset \\ false, & else \end{cases}$$

In case a fault was passed from the compensation module to the compensate activity the fault is rethrown as if it occurred at the compensate activity. We may refine PROPAGATEFAULTSACTIVITY (R5.5.2-11):

- Replace  $(sl_{child}, child) \in activityRunningChilds(sl, act)$   
by  $(sl', scope') \in compensateTOS(sl, comp)$ ,  
to look for faults which were sent from the called compensation modules.
- Replace  $faultChannel_{up}(sl_{child}, child, act)$   
by  $compensationChannel_{fault}(sl', scope', sl, comp)$ ,  
to rethrow faults which were sent along this channel from the called compensation modules.
- In parallel apply

**set** *activityState* **from** *running* **to** *faulted*

since a fault which is rethrown by a compensate activity is conceived to have occurred at the compensate activity.

The refinement results in COMPENSATERETHROWFAULT (RB.7.3-161) defined in the Appendix B.7.3.

#### 10.4.4 Compensate – Final Refinement

Refine EXECUTEACTIVITY<sub>pattern,basic</sub> (R7.1.3-28) by

- Replace STARTACTIVITY by STARTCOMPENSATE.
- Replace RUNACTIVITY by RUNCOMPENSATE.
- Remove

```

onSignal signalStop from parentActivity(act)
  in sI via signalChanneldown do
    STOPACTIVITY(sl, act)
otherwise

```

because preemptive termination does not apply to *Compensate*.

which results in EXECUTECOMPENSATE (RB.7.3-158) which is given in the Appendix B.7.3.

## 10.5 Compensation Handling at the Process Level by enableInstanceCompensation="yes"

A process may be defined in a way, such that each process instance that completes successfully must run the compensation handler of its *processScope*. This behaviour is determined by a property of the process-element.

```
<process ... enableInstanceCompensation="yes|no"? />
```

static functions (10.5.0-1)

$$processEnableInstanceCompensation : Process \rightarrow \{true, false\}$$

A process instance may be compensated only if the *processScope* has installed a compensation handler, which happens only if it completes successfully. The execution of this compensation handler cannot be triggered by any enclosing activity as there is none. But the *processInstanceManager* which we introduced in section 6.6.3 is hierarchically suitable to execute this behaviour.

Therefore, we refine RUNINSTANCMANAGER. First, let us recall the rule's definition.

RUNINSTANCMANAGER (R6.6.3-26)

(self)  $\equiv$

```

let instanceManager = myManager(self),
  process = managerProcess(self),
  scope = processScope(scope) in
forall pl  $\in$  processRunningInstances(process) do
  if signalComplete  $\in$  signalChannelup(pl, scope, scope) then
    remove signalComplete from signalChannelup(pl, scope, scope)
    remove pl from processRunningInstances(process)
  if signalTerminate  $\in$  signalChannelup(pl, scope, scope) then
    remove signalTerminate from signalChannelup(pl, scope, scope)
    remove pl from processRunningInstances(process)

```

Instead of just removing the process instance from the set of running instances, we first have to check whether the process definition requires the compensation of the instance. If this is the case, we send a compensation call to the *processScope* of the completed instance and then await the completion of its compensation in turn. This requires some extensions to the structures we have defined so far. The extensions are canonical by the extension of

**Universe:**  $CompensationCaller =_{def} Compensate \cup ProcessManager$

The instance-manager does not need to follow any stack of compensation modules, but it has to keep track of the compensation modules it has called, which are unambiguously identified by the identifier of the process instance.

dynamic functions (F10.5.0-1)

$$\begin{aligned} &instanceManagerCompensationModules : \\ &ProcessManager \rightarrow \mathcal{P}(ProcessInstance) \end{aligned}$$

requirement for the initial state (10.5.0-1)

$$\begin{aligned} &\forall instanceManager \in ProcessManager \\ &instanceManagerCompensationModules(instanceManager) = \emptyset \end{aligned}$$

Then initiating the compensation of a process instance by the instance-manager reads as follows.

<p><u>CALLINSTANCECOMPENSATION</u>  <math>(instanceManager \in ProcessManager, pl \in ProcessInstance) \equiv</math>  <b>let</b> process = <i>managerProcess</i>(instanceManager) <b>in</b>  CALLCOMPENSATION(pl, instanceManager, pl, <i>processScope</i>(process))  <b>add</b> pl <b>to</b> <i>instanceManagerCompensationModules</i>(instanceManager)</p>	<p>(R10.5.0-101)</p>
--	----------------------

referenced in  $RUNINSTANCEMANAGER_{instanceCompensation}$  (R10.5.0-103),  
references CALLCOMPENSATION (R10.4.3-100)

Once a process instance has finished its compensation, it will notify its caller. We use structures that we defined in section 10.2.1. A process instance which completed its compensation can finally be removed from the set of running instances. The semantics in case of a failure during the compensation of the entire instance are not specified.

<p><u>AWAITINSTANCECOMPENSATION</u>  <math>(instanceManager \in ProcessManager) \equiv</math>  <b>let</b> process = <i>managerProcess</i>(instanceManager),</p>	<p>(R10.5.0-102)</p>
---	----------------------

```

scope = processScope(process) in
forall pl ∈ instanceManagerCompensationModules(instanceManager) do
  if signalCompleted ∈ compensationChannelconfirm(pl, scope, pl, instanceManager) then
    remove signalCompleted from compensationChannelconfirm(pl, scope, pl, instanceManager)
    remove pl from instanceManagerCompensationModules(instanceManager)
    remove pl from processRunningInstances(process)

```

referenced in  $\text{RUNINSTANCEMANAGER}_{\text{instanceCompensation}}$  (R10.5.0-103)

Now, we may refine  $\text{RUNINSTANCEMANAGER}$  by

- In the **if**  $\text{signalEnable} \in \dots$  - operator, replace

```
remove pl from processRunningInstances(process)
```

by

```

if processEnableInstanceCompensation(process) = false then
  remove pl from processRunningInstances(process)
else
  CALLINSTANCECOMPENSATION(instanceManager, pl)

```

to compensate a completed process instance where applicable.

- In parallel with both **if** -operators, apply

```
AWAITINSTANCECOMPENSATION(instanceManager)
```

to update the set of running instances in case of instance compensation.

Then the refined ASM rule reads as follows.

<pre> RUNINSTANCEMANAGER<sub>instanceCompensation</sub> (self) ≡ let instanceManager = myManager(self),     process = managerProcess(self),     scope = processScope(scope) in forall pl ∈ processRunningInstances(process) do   if signalComplete ∈ signalChannel<sub>up</sub>(pl, scope, scope) then     remove signalComplete from signalChannel<sub>up</sub>(pl, scope, scope)   if processEnableInstanceCompensation(process) = false then     remove pl from processRunningInstances(process)   else     CALLINSTANCECOMPENSATION(instanceManager, pl) </pre>	(R10.5.0-103)
---	---------------

```
AWAITINSTANCECOMPENSATION(instanceManager)
if signalTerminate ∈ signalChannelup(pl, scope, scope) then
  remove signalTerminate from signalChannelup(pl, scope, scope)
  remove pl from processRunningInstances(process)
```

refines RUNINSTANCMANAGER (R6.6.3-26),  
references CALLINSTANCECOMPENSATION (R10.5.0-101), AWAITINSTANCECOMPENSATION  
(R10.5.0-102)

Please note that the informal specification gives no requirement on how the compensation of a complete process instance is triggered. Whether our design choice is appropriate, is subject to evaluation.

## 11 Summary & Conclusion

In this report, we explained, analyzed, and completely formalized the semantics of BPEL. We provided a detailed definition of the structures and the behavior of *each* activity that is defined in the language's informal specification [CGK<sup>+</sup>03]. We formalized how processes and activities are related to process instances and their executions. Furthermore, we provided an abstract definition of three process managers that render process instances transparent to the process' environment.

Our approach to create an ASM ground model for BPEL was lead by the language's architecture. We identified structures and behaviour that are shared by all activities, and which lead us to an abstract definition of an activity in BPEL. In effect, this abstract definition proposes an abstract framework for executing distributed, reactive systems and is not limited to BPEL. It can easily be reused for specifying other systems of this kind. From thereon we defined subsequent refinement steps for a full formal definition of the behaviour and the structures of each the language's activities.

The resulting model formalizes structures by function symbols and behaviour by ASM rules at the level of abstraction given in [CGK<sup>+</sup>03]. They are defined in a way such that they can be applied to any well-formed BPEL process. A concrete BPEL process has to be translated into the initial state of a distributed multi-agent ASM having those function symbols and ASM rules. The semantic domain of the translation are the runs of this ASM.

Because the ASM model encodes both – static *and* dynamic structures of the language – in a state of the ASM, its transitions are denoted in a general way such that their syntactical representation (the ASM rules) apply to any well-formed BPEL process. Together with the identical level of abstraction, the formal model can be verified against the informal specification by comparing informal description and formal definition directly.

By this property, the proposed ASM ground model complements the existing informal specification of BPEL. We supplement the complicated description of the language's complex semantics with a rigorous formal definition which allows for mathematical reasoning about properties of BPEL processes and the entire language. The model may also serve as a starting point for an implementation.

This is because the formal model addresses several problems of the informal specification. It integrates general requirements that constrain executions of BPEL processes into operational steps of activities, and it provides a clear structured view on the language's architecture. We think that our model is defined in such a way that changes to the language, or extensions like new activities can properly be reflected at the respective part of the model, making it a robust formal representation of BPEL.

Furthermore, using the mathematical rigourousness of first-order structures and the concept of abstract functions, the model properly identifies those parts of the language where insufficient semantics are provided. Namely there are:

- 
1. creation of process instances and matching of messages to process instances,
  2. order of compensation of completed scopes, and
  3. the entire compensation mechanism.

We were not able to provide abstract semantics of serializable scopes as depicted in [CGK<sup>+</sup>03, Section 16.3]. The description given therein leaves the required operational steps unknown.

The sharpness of the mathematical definitions forced us to make design decisions in order to formalize imprecise descriptions of the language's semantics. The decisions we have made are

1. the definition of inbox-manager, outbox-manager and instance-manager,
2. the definition of signal channels to coordinate the execution of instances of activities,
3. the definition of a hierarchical protocol to terminate an entire process instance,
4. the last refinement-steps towards an executional model of compensation handling,
5. the semantics of compensation of a complete process instance,
6. and that we assume a normal form of a BPEL process definition wrt. synchronous invokes, the default compensation handler, the `suppressJoinFailure` attribute and an unambiguous definition of variable names and activity names.

Either of these design decisions needs to be critically evaluated using criteria like falsifiability, testability, and non-functional requirements that arise from the field of application of the language.

Inconsistencies, which we have found in the course of defining the semantics mostly involved the preemptive termination of scopes due to a fault or the compensation handling mechanism, and link semantics. In large parts, these seem to be solved by the recent developments in defining BPEL v2.0 by OASIS [WT04]. A critical examination of our formal model regarding inconsistencies is subject to future work.

The semantics formally defined in this report are effectively an extension and a variation of the formal model of Farahbod et al. [FGV05] We added definitions for correlation handling, event handling and dead-path elimination. Furthermore, we provided different definitions of fault handling and compensation handling. We adapted the model's framework accordingly. Thus the semantics defined herein are the first complete operational semantics of BPEL with respect to its activities.

Some design decisions we made differ from those made by Farahbod et al. This gives a great opportunity to evaluate the effect of either decision on the language's semantics. Together, we are going to create a combined model which puts a highlight on these aspects and provides further information about the uncertainties of BPEL's informal specification.

We agree with Farahbod et al that the aim of providing a reliable and maintainable standard for BPEL is likely to fail without a "proper formalization of the fundamental

semantic issues". But the domain of e-Business highly demands for reliable standards to build on. [FGV05]. Evaluating our design decisions to analyze the insufficiently specified parts, as well as verifying the formal model of the language and turning it into an executable specification will help to make BPEL's definition, or those of a similar language, more coherent and consistent.

---

## **Acknowledgements**

We'd like to thank Wolfgang Reisig and Christian Stahl for their valuable comments and discussions in the course of creating this report, as well as Roozbeh Farahbod, Mona Vajihollahi and Uwe Glässer for making me write a proper introduction to the model and for the interesting discussions we had on this topic.

## A Abstract-State Machines

### A.1 Specific Rules for the BPEL semantics

#### A.1.1 Sets

**add element to Set**  $\equiv$

$\text{Set} := \text{Set} \cup \{\text{element}\}$

**remove element from Set**  $\equiv$

$\text{Set} := \text{Set} \setminus \{\text{element}\}$

On application of **add** and **remove** we assume the semantics of partial updates. [GT01]

#### A.1.2 State Transitions (of finite state machines)

The following rule describes all allowed state transitions from states in  $\{\text{oldStates}\}$  to  $\text{newState}$ .

**set state from oldStates to newState**  $\equiv$

**if**  $\text{state}(\text{myCurrentInstance}(\text{self}), \text{myStatic}(\text{self})) \in \{\text{oldStates}\}$  **then**  
     $\text{state}(\text{myCurrentInstance}(\text{self}), \text{myStatic}(\text{self})) := \text{newState}$

#### A.1.3 Asynchronous message passing via determined channel

To send a signal between activities, we use **signal** .

**signal sig to targetActivity in subInstance via channel**  $\equiv$

**add sig to channel**(subInstance,  $\text{myStatic}(\text{self})$ , targetActivity)

To receive one or more signals, we use **onSignal** .

**onSignal sigs from sourceActivity in subInstance via channel do R otherwise R'**  $\equiv$

**if**  $\{\text{sigs}\} \cap \text{channel}(\text{subInstance}, \text{sourceActivity}, \text{myStatic}(\text{self})) \neq \emptyset$  **then**  
        **forall**  $s \in \{\text{sigs}\} \cap \text{channel}(\text{subInstance}, \text{sourceActivity}, \text{myStatic}(\text{self}))$  **do**  
            **remove s from channel**(subInstance, sourceActivity,  $\text{myStatic}(\text{self})$ )  
            R

**else**  
        R'

With **signal ... toAll** and **onSignal ... fromAll** we may send and receive signals between sets of activities.

**signal sig toAll targetActivities in subInstance via channel**  $\equiv$

**forall**  $act \in targetActivities$  **do**  
    **signal sig to act in subInstance via channel**

**onSignal sigs fromAll sourceActivities in subInstance via channel do R otherwise R'**  $\equiv$

**if**  $\forall act \in sourceActivities$   
     $\{sigs\} \cap channel(subInstance, act, myStatic(self)) \neq \emptyset$  **then**  
    **forall**  $act \in sourceActivities$  **do**  
      **forall**  $s \in \{sigs\} \cap channel(subInstance, act, myStatic(self))$  **do**  
        **remove s from channel(subInstance, act, myStatic(self))**  
    R  
  **else**  
    R'

## B ASM-Rules for BPEL

### B.1 Rules and Functions for the Static Structure of a BPEL-Process

**Universe:**  $Activity_{basic} = Empty \cup Wait \cup Throw \cup Terminate \cup Receive \cup Reply \cup Invoke \cup Assign \cup Compensate$

**Universe:**  $Activity_{structured} = Flow \cup Sequence \cup Switch \cup Pick \cup While \cup Scope \cup EventHandler \cup FaultHandler \cup CompensationHandler$

**Universe:**  $Activity = Activity_{basic} \cup Activity_{structured}$

#### B.1.1 Activity Tree

$\mathcal{D} \text{ childActivity} : Activity \rightarrow Activity$

$$\text{act} \mapsto \begin{cases} \text{child}, & \{\text{child}\} = \text{childActivities}(\text{act}) \\ \text{undef}, & \text{else} \end{cases}$$

$\mathcal{D} \text{ enclosingScope} : Activity \rightarrow Scope$

$$\text{act} \mapsto \begin{cases} \text{parent}, & \text{parent} = \text{parentActivity}(\text{act}) \\ & \wedge \text{parent} \neq \text{undef} \wedge \text{parent} \in \text{Scope} \\ \text{sc}, & \text{parent} = \text{parentActivity}(\text{act}) \\ & \wedge \text{parent} \neq \text{undef} \wedge \text{parent} \notin \text{Scope} \\ & \wedge \text{sc} = \text{enclosingScope}(\text{parent}) \\ \text{undef}, & \text{else} \end{cases}$$

## B.2 Rules and Functions for the Dynamic Structure of a BPEL-Process

### B.2.1 Process Managers' ASM Rules

RUNINBOXMANAGER (self)  $\equiv$  (R6.6.1-24)

```

let msgManager = myManager(self),
    process = managerProcess(self) in
select msg  $\in$  processInbox(process) in
    select pl  $\in$  processRunningInstances(process)
        where msgMatchesInstance(msg, process, pl) = true in
            messageReceiveTime(msg) := getCurrentTime(pl)
            ASSIGNMESSAGE TO INSTANCE(msg, process, pl)
    ifnone
        let pl = processWaitingInstance(process) in
            messageReceiveTime(msg) := getCurrentTime(pl)
            ASSIGNMESSAGE TO INSTANCE(msg, process, pl)
        add pl to processRunningInstances(process)

        let plnew = new(ProcessInstance) in
            processWaitingInstance(process) := plnew
            let scope = processScope(process) in
                add signalEnable to signalChanneldown(plnew, scope, scope)

```

RUNOUTBOXMANAGER (self)  $\equiv$  (R6.6.2-25)

```

let msgManager = myManager(self),
    process = managerProcess(self) in
forall p  $\in$  availablePorts(process) do
    forall msg  $\in$  localPortout(p) do
        remove msg from localPortout(p)
        add msg to processOutbox(process)

    forall msg  $\in$  localPortfault(p) do
        remove msg from localPortfault(p)
        add msg to processOutbox(process)

    forall msg  $\in$  remotePortin(p) do
        remove msg from remotePortin(p)
        add msg to processOutbox(process)

```

RUNINSTANCMANAGER<sub>instanceCompensation</sub> (self)  $\equiv$  (R10.5.0-103)

```

let instanceManager = myManager(self),

```

```

    process = managerProcess(self),
    scope = processScope(scope) in
forall pl ∈ processRunningInstances(process) do
    if signalComplete ∈ signalChannelup(pl, scope, scope) then
        remove signalComplete from signalChannelup(pl, scope, scope)
    if processEnableInstanceCompensation(process) = false then
        remove pl from processRunningInstances(process)
    else
        CALLINSTANCECOMPENSATION(instanceManager, pl)
AWAITINSTANCECOMPENSATION(instanceManager)
if signalTerminate ∈ signalChannelup(pl, scope, scope) then
    remove signalTerminate from signalChannelup(pl, scope, scope)
    remove pl from processRunningInstances(process)

```

### B.2.2 Activities' ASM Rule

RUNBPELACTIVITY (RB.2.2-104)

(self) ≡

```

let sl = mySubInstance(self),
    act = myActivity(self) in
if act ∈ Activitybasic then
    EXECUTEACTIVITYbasic(sl, act)
if act ∈ Activitystructured then
    EXECUTEACTIVITYstructured(sl, act)

```

refines RUNBPELACTIVITY<sub>pattern</sub> (R5.2.0-4)

references EXECUTEACTIVITY<sub>basic</sub> (RB.2.2-105), EXECUTEACTIVITY<sub>structured</sub> (RB.2.2-106)

EXECUTEACTIVITY<sub>basic</sub> (RB.2.2-105)

(sl ∈ SubInstance, act ∈ Activity) ≡

```

if act ∈ Empty then
    EXECUTEEMPTY(sl, act)
if act ∈ Wait then
    EXECUTEWAIT(sl, act)
if act ∈ Throw then
    EXECUTETHROW(sl, act)
if act ∈ Terminate then
    EXECUTETERMINATE(sl, act)
if act ∈ Receive then
    EXECUTERECEIVE(sl, act)

```

**if**  $act \in Reply$  **then**  
     EXECUTEREPLY( $sl, act$ )  
**if**  $act \in Invoke$  **then**  
     EXECUTEINVOKE( $sl, act$ )  
**if**  $act \in Assign$  **then**  
     EXECUTEASSIGN( $sl, act$ )  
**if**  $act \in Compensate$  **then**  
     EXECUTECOMPENSATE( $sl, act$ )

referenced in RUNBPELACTIVITY (RB.2.2-104)

references EXECUTEEMPTY (RB.4.1-112), EXECUTEWAIT (RB.4.2-113), EXECUTETHROW  
 (RB.4.3-114), EXECUTETERMINATE (RB.4.4-115), EXECUTERECEIVE (RB.4.5-116),  
 EXECUTEREPLY (RB.4.6-118), EXECUTEINVOKE (RB.4.7-123), EXECUTEASSIGN (RB.4.8-127),  
 EXECUTECOMPENSATE (RB.7.3-158)

EXECUTEACTIVITY<sub>structured</sub> (RB.2.2-106)  
 ( $sl \in SubInstance, act \in Activity$ )  $\equiv$

**if**  $act \in Flow$  **then**  
     EXECUTEFLOW( $sl, act$ )  
**if**  $act \in Sequence$  **then**  
     EXECUTESEQUENCE( $sl, act$ )  
**if**  $act \in Switch$  **then**  
     EXECUTESWITCH( $sl, act$ )  
**if**  $act \in Pick$  **then**  
     EXECUTEPICK( $sl, act$ )  
**if**  $act \in While$  **then**  
     EXECUTEWHILE( $sl, act$ )  
**if**  $act \in EventHandler$  **then**  
     EXECUTEEVENTHANDLER( $sl, act$ )  
**if**  $act \in FaultHandler$  **then**  
     EXECUTEFAULTHANDLER( $sl, act$ )  
**if**  $act \in CompensationHandler$  **then**  
     EXECUTECOMPENSATIONHANDLER( $sl, act$ )  
**if**  $act \in Scope$  **then**  
     EXECUTESCOPE<sub>compensationHandling</sub>( $sl, act$ )

referenced in RUNBPELACTIVITY (RB.2.2-104)

references EXECUTEFLOW (RB.5.2-130), EXECUTESEQUENCE (RB.5.3-131),  
 EXECUTESWITCH (RB.5.4-133), EXECUTEPICK (RB.5.5-134), EXECUTEWHILE (RB.5.6-136),  
 EXECUTEEVENTHANDLER (R9.5.2-76), EXECUTEFAULTHANDLER (RB.6.2-144),  
 EXECUTECOMPENSATIONHANDLER (RB.7.2-157), EXECUTESCOPE<sub>compensationHandling</sub>  
 (R10.3.3-98)

### B.2.3 The Subinstance Tree

derived functions ( $\mathcal{D}$  B.2.3-1)

$\mathcal{D}$  *currentSubInstanceFromInner* : *SubInstance* × *Activity* × *Activity* → *SubInstance*

$$(\text{sl}_{act}, act, act') \mapsto \begin{cases} \text{sl}_{act}, & act = act' \\ \text{sl}', & \text{parent} = \text{parentActivity}(act) \\ & \wedge (\text{parent} \in (While \cup EventHandler)) \\ & \quad \Rightarrow \text{sl}_{parent} = \text{parentSubInstance}(\text{sl}_{act}) \\ & \wedge (\text{parent} \notin (While \cup EventHandler)) \\ & \quad \Rightarrow \text{sl}_{parent} = \text{sl}_{act} \\ & \wedge \text{sl}' = \text{currentSubInstanceFromInner}(\text{sl}_{parent}, \text{parent}, act') \\ \text{undef}, & \text{else} \end{cases}$$

### B.2.4 Links

$\mathcal{D}$  *processLinks* : *Process* →  $\mathcal{P}(\text{Link})$

$$\text{process} \mapsto \{link \mid \exists flow \in Flow \\ \quad flow \in \text{activity*TreeNode}(\text{processScope}(\text{process})) \\ \quad \wedge link \in \text{flowLinks}(flow)\}$$

$\mathcal{D}$  *activityTargetLinks* : *Activity* →  $\mathcal{P}(\text{Link})$

$$act \mapsto \{l \in \text{processLinks}(\text{activityProcess}(act)) \mid \text{linkTarget}(l) = act\}$$

$\mathcal{D}$  *activitySourceLinks* : *Activity* →  $\mathcal{P}(\text{Link})$

$$act \mapsto \{l \in \text{processLinks}(\text{activityProcess}(act)) \mid \text{linkSource}(l) = act\}$$

## B.3 Rules and Functions for Communication

### B.3.1 Correlation Handling

The correlation sets of a process are defined within scopes. Thus, the correlation values of a correlation set are stored relatively to the scope and the *SubInstance* in which it is executed. A process may have several correlation values for the same correlation set, at most one for each *SubInstance* the possessing scope is executed in.

$\mathcal{D}$  *correlationSatisfied* :  $SubInstance \times Scope \times PropertyAlias \times Value \rightarrow \{true, false\}$

$$(sl, scope, alias, msgPartValue) \mapsto \begin{cases} true, & \begin{aligned} & correlationPropertyValue(sl, alias) = undef \\ & \vee (subvalue \\ & \quad = messagePartSubValue(msgPartValue, \\ & \quad \quad aliasMessageTypePart(alias), \\ & \quad \quad aliasQueryPath(alias)) \\ & \wedge correlationPropertyValue(sl, scope, alias) \\ & \quad = subValue) \end{aligned} \\ false, & else \end{cases}$$

Checking the correctness of the satisfaction of a correlation set by a message or a variable depends on the correlation values of the possessing scope.

The possessing scope  $S$  always encloses the activity  $A$  checking the correctness. Otherwise the correlation set wouldn't be known to  $A$ . Hence we may determine the *SubInstance*  $sI_{scope}$  in which  $S$  is currently executed:  $sI_{scope}$  is either equal to the *SubInstance* of  $A$ , or it is a (transitive) parent of  $A$ 's *SubInstance*. The function *currentSubInstanceFromInner* delivers the required node of the sub instance tree.

$\mathcal{D}$  *correlationSatisfied<sub>msg</sub>* :  $SubInstance \times Activity \times BPELMsg \times \mathcal{P}(CorrelationSet) \rightarrow \{true, false\}$

$$(sl, act, msg, CS) \mapsto \begin{cases} true, & \begin{aligned} & \forall cs \in CS \\ & \quad scope = correlationSetScope(cs) \\ & \wedge sl_{scope} = currentSubInstanceFromInner(sl, act, scope) \\ & \wedge \forall property \in cs \\ & \quad msgMessageType(msg) = aliasMsgType(property) \\ & \quad \wedge msgTypePart = aliasMessageTypePart(property) \\ & \quad \wedge msgTypePart \\ & \quad \quad \in messageTypeParts(msgMessageType(msg)) \\ & \quad \wedge correlationSatisfied(sl_{scope}, scope, property, \\ & \quad \quad msgPartValue(msg, msgTypePart)) \end{aligned} \\ false, & else \end{cases}$$

$\mathcal{D}$  *correlationSatisfied<sub>var</sub>* :  $SubInstance \times Activity \times Variable \times \mathcal{P}(CorrelationSet) \rightarrow \{true, false\}$

$$(\text{sl}, \text{act}, \text{msg}, \text{CS}) \mapsto \left\{ \begin{array}{l} \text{true, } \forall \text{cs} \in \text{CS} \\ \quad \text{scope} = \text{correlationSetScope}(\text{cs}) \\ \quad \wedge \text{sl}_{\text{scope}} = \text{currentSubInstance}_{\text{fromInner}}(\text{sl}, \text{act}, \text{scope}) \\ \quad \wedge \forall \text{property} \in \text{cs} \\ \quad \quad \text{variableMessageType}(\text{var}) = \text{aliasMsgType}(\text{property}) \\ \quad \quad \wedge \text{msgTypePart} = \text{aliasMessageTypePart}(\text{property}) \\ \quad \quad \wedge \text{msgTypePart} \\ \quad \quad \quad \in \text{messageTypeParts}(\text{variableMessageType}(\text{var})) \\ \quad \quad \wedge \text{correlationSatisfied}(\text{sl}_{\text{scope}}, \text{scope}, \text{property}, \\ \quad \quad \quad \text{variablePartValue}(\text{sl}, \text{var}, \text{msgTypePart})) \\ \text{false, else} \end{array} \right.$$

The same argument holds for setting the values of a correlation set.

SETCORRELATIONPROPERTY (RB.3.1-107)

$(\text{sl} \in \text{ProcessInstance}, \text{scope} \in \text{Scope}, \text{property} \in \text{PropertyAlias},$   
 $\text{msgPartValue} \in \text{Value}) \equiv$

**if**  $\text{correlationPropertyValue}(\text{sl}, \text{scope}, \text{property}) = \text{undef}$  **then**  
 $\text{correlationPropertyValue}(\text{sl}, \text{scope}, \text{property}) :=$   
 $\text{messagePartSubValue}(\text{msgPartValue},$   
 $\text{aliasMessageTypePart}(\text{property}),$   
 $\text{aliasQueryPath}(\text{property}))$

referenced in CORRELATEINSTANCETOMESSAGE (RB.3.1-108),  
CORRELATEINSTANCETO VARIABLE (RB.3.1-109)

CORRELATEINSTANCETOMESSAGE (RB.3.1-108)

$(\text{sl} \in \text{SubInstance}, \text{act} \in \text{Activity}, \text{msg} \in \text{BPELMsg}, \text{CS} \in \mathcal{P}(\text{CorrelationSet}),$   
 $\text{CS}_{\text{init}} \in \mathcal{P}(\text{CorrelationSet})) \equiv$

**if**  $\text{correlationSatisfied}_{\text{msg}}(\text{pl}, \text{msg}, \text{CS})$  **then**  
**forall**  $\text{cs} \in \text{CS}$  **where**  $\text{cs} \in \text{CS}_{\text{init}}$  **do**  
**let**  $\text{scope} = \text{correlationSetScope}(\text{cs}),$   
 $\text{sl}_{\text{scope}} = \text{currentSubInstance}_{\text{fromInner}}(\text{sl}, \text{act}, \text{scope})$  **in**  
**forall**  $\text{property} \in \text{cs}$  **do**  
**let**  $\text{value} = \text{msgPartValue}(\text{msg}, \text{aliasMessageTypePart}(\text{property}))$  **in**  
SETCORRELATIONPROPERTY( $\text{sl}_{\text{scope}}, \text{scope}, \text{property}, \text{value}$ )

references SETCORRELATIONPROPERTY (RB.3.1-107)  
referenced in RECEIVEMESSAGE<sub>pattern</sub> (R6.5.3-20)

CORRELATEINSTANCETOVARIABLE (RB.3.1-109)

( $sl \in SubInstance, act \in Activity, msg \in BPELMsg, CS \in \mathcal{P}(CorrelationSet),$   
 $CS_{init} \in \mathcal{P}(CorrelationSet)$ )  $\equiv$

**if**  $correlationSatisfied_{var}(sl, msg, CS)$  **then**  
**forall**  $cs \in CS$  **where**  $cs \in CS_{init}$  **do**  
**let**  $scope = correlationSetScope(cs),$   
 $sl_{scope} = currentSubInstance_{fromInner}(sl, act, scope)$  **in**  
**forall**  $property \in cs$  **do**  
**let**  $value = variablePartValue(sl, var, aliasMessageTypePart(property))$  **in**  
 $SETCORRELATIONPROPERTY(sl_{scope}, scope, property, value)$

references  $SETCORRELATIONPROPERTY$  (RB.3.1-107)

referenced in  $GENERATEANDSENDMESSAGE_{pattern}$  (R6.5.3-22)

### B.3.2 Receiving and Sending Messages

COPYMSGTOVARIABLE (RB.3.2-110)

( $sl \in ProcessInstance, msg \in BPELMsg, var \in Variable$ )  $\equiv$

**if**  $var \neq undef \wedge messageType(msg) = variableMessageType(var)$  **then**  
**forall**  $part \in messageTypeParts(messageType(var))$  **do**  
 $variablePartValue(pl, var, part) := messagePartValue(msg, part)$

referenced in  $RECEIVEMESSAGE_{pattern}$  (R6.5.3-20)

COPYVARIABLETOMSG (RB.3.2-111)

( $sl \in ProcessInstance, var \in Variable, msg \in BPELMsg$ )  $\equiv$

**if**  $var \neq undef$  **then**  
**forall**  $part \in messageTypeParts(variableMessageType(var))$  **do**  
 $msgPartValue(msg, part) := variablePartValue(pl, var, part)$   
 $msgMessageType(msg) := variableMessageType(var)$

referenced in  $GENERATEANDSENDMESSAGE_{pattern}$  (R6.5.3-22),  $RUNTHROW$  (R7.4.1-32)

## B.4 Rules and Functions for Basic Activities

### B.4.1 Empty

EXECUTEEMPTY (RB.4.1-112)

$(sl \in SubInstance, empty \in Empty) \equiv$

PROPAGATEDPE(sl, empty)

PROPAGATETERMINATE<sub>basic</sub>(sl, empty)

**onSignal** *signalStop* **from** *parentActivity*(empty)

**in** *sI* **via** *signalChannel<sub>down</sub>* **do**

STOPACTIVITY(sl, empty)

**otherwise**

**onSignal** *signalEnable* **from** *parentActivity*(empty)

**in** *sI* **via** *signalChannel<sub>down</sub>* **do**

**set** *activityState* **from** *disabled* **to** *enabled*

**if** *activityState*(sl, empty) = *enabled* **then**

STARTACTIVITY(sl, empty)

**if** *activityState*(sl, empty) = *running* **then**

RUNEMPTY(sl, empty)

**if** *activityState*(sl, empty) = *completing* **then**

COMPLETEACTIVITY(sl, empty)

refines EXECUTEACTIVITY<sub>pattern,basic</sub> (R7.1.3-28),

referenced in EXECUTEACTIVITY<sub>basic</sub> (RB.2.2-105), referencing RUNEMPTY (R7.2.0-29)

### B.4.2 Wait

EXECUTEWAIT (RB.4.2-113)

$(sl \in SubInstance, wait \in Empty) \equiv$

PROPAGATEDPE(sl, wait)

PROPAGATETERMINATE<sub>basic</sub>(sl, wait)

**onSignal** *signalStop* **from** *parentActivity*(wait)

**in** *sI* **via** *signalChannel<sub>down</sub>* **do**

STOPACTIVITY(sl, wait)

**otherwise**

**onSignal** *signalEnable* **from** *parentActivity*(wait)

**in** *sI* **via** *signalChannel<sub>down</sub>* **do**

**set** *activityState* **from** *disabled* **to** *enabled*

**if** *activityState*(sl, wait) = *enabled* **then**

```

STARTWAIT(sl, wait)
if activityState(sl, wait) = running then
  RUNWAIT(sl, wait)
if activityState(sl, wait) = completing then
  COMPLETEACTIVITY(sl, wait)

```

refines EXECUTEACTIVITY<sub>pattern,basic</sub> (R7.1.3-28),  
 referenced in EXECUTEACTIVITY<sub>basic</sub> (RB.2.2-105), referencing STARTWAIT (R7.3.1-30),  
 RUNWAIT (R7.3.2-31)

### B.4.3 Throw

EXECUTETHROW (RB.4.3-114)

```

(sl ∈ SubInstance, throw ∈ Empty) ≡
PROPAGATEDPE(sl, throw)
PROPAGATERMINATEbasic(sl, throw)
onSignal signalStop from parentActivity(throw)
  in sI via signalChanneldown do
  STOPACTIVITY(sl, throw)
otherwise
  onSignal signalEnable from parentActivity(throw)
    in sI via signalChanneldown do
    set activityState from disabled to enabled
  if activityState(sl, throw) = enabled then
    STARTACTIVITY(sl, throw)
  if activityState(sl, throw) = running then
    RUNTHROW(sl, throw)

```

refines EXECUTEACTIVITY<sub>pattern,basic</sub> (R7.1.3-28),  
 referenced in EXECUTEACTIVITY<sub>basic</sub> (RB.2.2-105), referencing RUNTHROW (R7.4.1-32)

### B.4.4 Terminate

EXECUTETERMINATE (RB.4.4-115)

```

(sl ∈ SubInstance, terminate ∈ Terminate) ≡
PROPAGATEDPE(sl, terminate)
PROPAGATERMINATEbasic(sl, terminate)
onSignal signalStop from parentActivity(terminate)
  in sI via signalChanneldown do

```

```

    STOPACTIVITY(sl, terminate)
  otherwise
    onSignal signalEnable from parentActivity(terminate)
      in sI via signalChanneldown do
        set activityState from disabled to enabled
    if activityState(sl, terminate) = enabled then
      STARTACTIVITY(sl, terminate)
    if activityState(sl, terminate) = running then
      RUNTERMINATE(sl, terminate)
    if activityState(sl, terminate) = completing then
      COMPLETEACTIVITY(sl, terminate)

```

refines EXECUTEACTIVITY<sub>pattern,basic</sub> (R7.1.3-28),  
 referenced in EXECUTEACTIVITY<sub>basic</sub> (RB.2.2-105), referencing RUNTERMINATE (R7.5.1-33)

#### B.4.5 Receive

EXECUTERECEIVE (RB.4.5-116)  
 (sl ∈ *SubInstance*, receive ∈ *Receive*) ≡

```

  PROPAGATEDPE(sl, receive)
  PROPAGATETERMINATEbasic(sl, receive)
  onSignal signalStop from parentActivity(receive)
    in sI via signalChanneldown do
      STOPACTIVITY(sl, receive)
  otherwise
    onSignal signalEnable from parentActivity(receive)
      in sI via signalChanneldown do
        set activityState from disabled to enabled
    if activityState(sl, receive) = enabled then
      STARTACTIVITY(sl, receive)
    if activityState(sl, receive) = running then
      RUNRECEIVE(sl, receive)
    if activityState(sl, receive) = completing then
      COMPLETEACTIVITY(sl, receive)

```

refines EXECUTEACTIVITY<sub>pattern,basic</sub> (R7.1.3-28),  
 referenced in EXECUTEACTIVITY<sub>basic</sub> (RB.2.2-105), referencing RUNRECEIVE (R7.6.1-34)

RUNRECEIVE (R7.6.1-34)  
 (sl ∈ *SubInstance*, receive ∈ *Receive*) ≡  
 let *portdescr* = *receivePortDescriptor*(receive) in

AWAITANDRECEIVECORRELATINGMESSAGE<sub>receive</sub>(sl, receive, portdescr)

AWAITANDRECEIVECORRELATINGMESSAGE<sub>receive</sub> (RB.4.5-117)  
 (sl ∈ SubInstance, receive ∈ Receive, portdescr ∈ PortDescriptor) ≡

```

let partnerL = portPartnerLink(portdescr),
    pT = portPortType(portdescr),
    op = portOperation(portdescr),
    pl = processInstance(sl) in
select msg ∈ localPortin(pl, partnerL, pT, op) in
  let CS = portCorrelationSet(portdescr) in
    if correlationSatisfied(sl, receive, msg, CS) then
      remove msg from localPortin(pl, partnerL, pT, op)
      RECEIVEMESSAGEpattern(sl, receive, portdescr, msg)
      set activityState from running to completing
    else
      THROW(sl, receive, correlationViolation)

```

referenced in RUNRECEIVE (R7.6.1-34),

refines AWAITANDRECEIVECORRELATINGMESSAGE<sub>pattern</sub> (R6.5.3-19),

references RECEIVEMESSAGE<sub>pattern</sub> (R6.5.3-20), THROW (R5.5.1-10)

#### B.4.6 Reply

EXECUTEREPLY (RB.4.6-118)  
 (sl ∈ SubInstance, reply ∈ Reply) ≡

```

PROPAGATEDPE(sl, reply)
PROPAGATETERMINATEbasic(sl, reply)
onSignal signalStop from parentActivity(reply)
  in sI via signalChanneldown do
  STOPACTIVITY(sl, reply)
otherwise
  onSignal signalEnable from parentActivity(reply)
    in sI via signalChanneldown do
    set activityState from disabled to enabled
  if activityState(sl, reply) = enabled then
    STARTACTIVITY(sl, reply)
  if activityState(sl, reply) = running then
    RUNREPLY(sl, reply)
  if activityState(sl, reply) = completing then
    COMPLETEACTIVITY(sl, reply)

```

refines EXECUTEACTIVITY<sub>pattern,basic</sub> (R7.1.3-28),  
 referenced in EXECUTEACTIVITY<sub>basic</sub> (RB.2.2-105), referencing RUNREPLY (R7.7.1-36)

RUNREPLY (R7.7.1-36)  
 $(sl \in SubInstance, reply \in Reply) \equiv$

**let** portdescr = *replyPortDescriptor*(reply) **in**  
 GENERATEANDSENDCORRELATINGMESSAGE<sub>reply</sub>(sl, reply, portdescr)

GENERATEANDSENDCORRELATINGMESSAGE<sub>reply</sub> (R7.7.1-35)

$(sl \in SubInstance, act \in Activity, portdescr \in PortDescriptor) \equiv$   
**let** var = *portVariable*(portdescr),  
 CS = *portCorrelationSets*(portdescr) **in**  
**if** *correlationSatisfied*<sub>var</sub>(sl, reply, var, CS) = true **then**  
**if** *replyFaultName*(reply) = undef **then**  
 GENERATEANDSENDMESSAGE<sub>reply</sub>(sl, reply, portdescr)  
**else**  
 GENERATEANDSENDMESSAGE<sub>fault</sub>(sl, reply, portdescr)  
**set** *activityState* **from** *running* **to** *completing*  
**else**  
 THROW(sl, act, *correlationViolation*)

GENERATEANDSENDMESSAGE<sub>reply</sub> (RB.4.6-119)  
 $(sl \in SubInstance, reply \in Reply, portdescr \in PortDescriptor) \equiv$

**let** message = **new**(*BPELMsg*) **in**  
**let** var = *portVariable*(portdescr) **in**  
 COPYVARIABLETOMSG(sl, var, message)  
**let** CS = *portCorrelationSets*(portdescr),  
 CS<sub>init</sub> = {cs ∈ CS | *portInitCorrelation.Set*(portdescr, cs) = true} **in**  
 CORRELATEINSTANCETOVARIABLE(sl, reply, var, CS, CS<sub>init</sub>)  
 SENDMESSAGE<sub>reply</sub>(sl, portdescr, message)

refines GENERATEANDSENDMESSAGE<sub>pattern</sub> (R6.5.3-22),  
 referenced in GENERATEANDSENDCORRELATINGMESSAGE<sub>reply</sub> (R7.7.1-35),  
 references COPYVARIABLETOMSG (RB.3.2-111), CORRELATEINSTANCETOVARIABLE  
 (RB.3.1-109), SENDMESSAGE<sub>reply</sub> (RB.4.6-121)

GENERATEANDSENDMESSAGE<sub>fault</sub> (RB.4.6-120)  
 $(sl \in SubInstance, reply \in Reply, portdescr \in PortDescriptor) \equiv$

**let** fault = **new**(*Fault*) **in**

```

let var = portVariable(portdescr) in
  COPYVARIABLETOMSG(sl, var, fault)
let CS = portCorrelationSets(portdescr),
  CSinit = {cs ∈ CS | portInitCorrelationSet(portdescr, cs) = true} in
  CORRELATEINSTANCETOVARIABLE(sl, reply, var, CS, CSinit)
  SENDMESSAGEfault(sl, portdescr, fault)
  faultFName(fault) := replyFaultName(reply)

```

refines GENERATEANDSENDMESSAGE<sub>pattern</sub> (R6.5.3-22),  
 referenced in GENERATEANDSENDCORRELATINGMESSAGE<sub>reply</sub> (R7.7.1-35),  
 references COPYVARIABLETOMSG (RB.3.2-111), CORRELATEINSTANCETOVARIABLE  
 (RB.3.1-109), SENDMESSAGE<sub>fault</sub> (RB.4.6-122)

SENDMESSAGE<sub>reply</sub> (RB.4.6-121)

(sl ∈ *SubInstance*, portdescr ∈ *PortDescriptor*, msg ∈ *BPELMsg*) ≡

```

let partnerL = portPartnerLink(portdescr),
  pT = portPortType(portdescr),
  op = portOperation(portdescr),
  pl = processInstance(sl) in
  add msg to localPortout(pl, partnerL, pT, op)

```

refines SENDMESSAGE<sub>pattern</sub> (R6.5.3-23),  
 referenced in GENERATEANDSENDMESSAGE<sub>reply</sub> (RB.4.6-119)

SENDMESSAGE<sub>fault</sub> (RB.4.6-122)

(sl ∈ *SubInstance*, portdescr ∈ *PortDescriptor*, fault ∈ *Fault*) ≡

```

let partnerL = portPartnerLink(portdescr),
  pT = portPortType(portdescr),
  op = portOperation(portdescr),
  pl = processInstance(sl) in
  add fault to localPortfault(pl, partnerL, pT, op)

```

refines SENDMESSAGE<sub>pattern</sub> (R6.5.3-23),  
 referenced in GENERATEANDSENDMESSAGE<sub>fault</sub> (RB.4.6-120)

## B.4.7 Invoke

### derived functions

We define the following derived functions to filter from the *invokeCorrelationSets* these sets which correspond to a specific direction of communication (outgoing and incoming).

$$\mathcal{D} \text{ invokeCorrelationSet}_{out} : \text{Invoke} \rightarrow \mathcal{P}(\text{CorrelationSet})$$

$$\begin{aligned}
 \text{invoke} \mapsto & \{cs \in \text{invokeCorrelationSets}(\text{invoke}) \mid \\
 & \text{out} \in \text{invokeCorrelationSetDirection}(\text{invoke}, cs)\} \\
 \mathcal{D} \text{ invokeCorrelationSet}_{in} : & \text{Invoke} \rightarrow \mathcal{P}(\text{CorrelationSet}) \\
 \text{invoke} \mapsto & \{cs \in \text{invokeCorrelationSets}(\text{invoke}) \mid \\
 & \text{in} \in \text{invokeCorrelationSetDirection}(\text{invoke}, cs)\}
 \end{aligned}$$

requirement for the initial state (B.4.7-1)

$$\begin{aligned}
 \forall \text{ invoke} \in \text{Invoke} \exists \text{ pd}_s, \text{pd}_r \in \text{PortDescriptor} \\
 ( & \text{invokePortDescriptor}_{send}(\text{invoke}) = \text{pd}_s \\
 \wedge & \text{portPartnerLink}(\text{pd}_s) = \text{invokePartnerLink}(\text{invoke}) \\
 \wedge & \text{portPortType}(\text{pd}_s) = \text{invokePortType}(\text{invoke}) \\
 \wedge & \text{portOperation}(\text{pd}_s) = \text{invokeOperation}(\text{invoke}) \\
 \wedge & \text{portVariable}(\text{pd}_s) = \text{invokeInputVariable}(\text{invoke}) \\
 \wedge & \text{portCorrelationSets}(\text{pd}_s) = \text{invokeCorrelationSet}_{out}(\text{invoke}) \\
 \wedge & \forall cs \in \text{portCorrelationSets}(\text{pd}_s) \\
 & \text{portInitCorrelationSets}(\text{pd}, cs) = \text{invokeInitCorrelationSet}(\text{invoke}, cs) ) \\
 \wedge & (\text{invokeOutputVariable}(\text{invoke}) \neq \text{undef} \Rightarrow \\
 & \text{invokePortDescriptor}_{receive}(\text{invoke}) = \text{pd}_r \\
 \wedge & \text{portPartnerLink}(\text{pd}_r) = \text{invokePartnerLink}(\text{invoke}) \\
 \wedge & \text{portPortType}(\text{pd}_r) = \text{invokePortType}(\text{invoke}) \\
 \wedge & \text{portOperation}(\text{pd}_r) = \text{invokeOperation}(\text{invoke}) \\
 \wedge & \text{portVariable}(\text{pd}_r) = \text{invokeOutputVariable}(\text{invoke}) \\
 \wedge & \text{portCorrelationSets}(\text{pd}_r) = \text{invokeCorrelationSet}_{out}(\text{invoke}) \\
 \wedge & \forall cs \in \text{portCorrelationSets}(\text{pd}_r) \\
 & \text{portInitCorrelationSets}(\text{pd}, cs) = \text{invokeInitCorrelationSet}(\text{invoke}, cs) ) \\
 \wedge & (\text{invokeOutputVariable}(\text{invoke}) = \text{undef} \Rightarrow \\
 & \text{invokePortDescriptor}_{receive}(\text{invoke}) = \text{undef})
 \end{aligned}$$

EXECUTEINVOKE

(RB.4.7-123)

( $sl \in \text{SubInstance}, \text{invoke} \in \text{Invoke}$ )  $\equiv$

PROPAGATEDPE( $sl, \text{invoke}$ )

PROPAGATETERMINATE<sub>basic</sub>( $sl, \text{invoke}$ )

**onSignal**  $signalStop$  **from**  $parentActivity(\text{invoke})$

**in**  $sI$  **via**  $signalChannel_{down}$  **do**

STOPACTIVITY( $sl, \text{invoke}$ )

**otherwise**

**onSignal**  $signalEnable$  **from**  $parentActivity(\text{invoke})$

**in**  $sI$  **via**  $signalChannel_{down}$  **do**

**set**  $activityState$  **from**  $disabled$  **to**  $enabled$

```

if activityState(sl, invoke) = enabled then
  STARTACTIVITY(sl, invoke)
if activityState(sl, invoke) = running then
  RUNINVOKE(sl, invoke)
if activityState(sl, invoke) = completing then
  COMPLETEACTIVITY(sl, invoke)

```

refines EXECUTEACTIVITY<sub>pattern,basic</sub> (R7.1.3-28),  
 referenced in EXECUTEACTIVITY<sub>basic</sub> (RB.2.2-105), referencing RUNINVOKE (R7.8.1-37)

RUNINVOKE (R7.8.1-37)  
 (sl ∈ *SubInstance*, invoke ∈ *Invoke*) ≡

```

if invokeState(sl, invoke) = sending then
  RUNINVOKEsending(sl, invoke)
if invokeState(sl, invoke) = receiving then
  RUNINVOKEreceiving(sl, invoke)

```

RUNINVOKE<sub>sending</sub> (R7.8.1-38)  
 (sl ∈ *SubInstance*, invoke ∈ *Invoke*) ≡

```

let portdescrsend = invokePortDescriptorsend(invoke) in
  GENERATEANDSENDCORRELATINGMESSAGEinvoke(sl, invoke, portdescrsend)

```

GENERATEANDSENDCORRELATINGMESSAGE<sub>invoke</sub> (R7.8.1-39)  
 (sl ∈ *SubInstance*, invoke ∈ *Invoke*, portdescr ∈ *PortDescriptor*) ≡

```

let var = portVariable(portdescr),
  CS = portCorrelationSets(portdescr) in
if correlationSatisfiedvar(sl, var, CS) = true then
  GENERATEANDSENDMESSAGEinvoke(sl, invoke, portdescr)
if invokePortDescriptorreceive(invoke) ≠ undef then
  invokeState(sl, invoke) := receiving
else
  set activityState from running to completing
else
  THROW(sl, act, correlationViolation)

```

GENERATEANDSENDMESSAGE<sub>invoke</sub> (RB.4.7-124)  
 (sl ∈ *SubInstance*, invoke ∈ *Invoke*, portdescr ∈ *PortDescriptor*) ≡

```

let message = new(BPELMsg) in
  let var = portVariable(portdescr) in
  COPYVARIABLETOMSG(sl, var, message)
  let CS = portCorrelationSets(portdescr),

```

$CS_{init} = \{cs \in CS \mid portInitCorrelationSet(portdescr, cs) = true\}$  **in**  
CORRELATEINSTANCETOVARIABLE(*sl*, *invoke*, *var*, *CS*,  $CS_{init}$ )  
SENDMESSAGE<sub>*invoke*</sub>(*sl*, *portdescr*, *message*)

refines GENERATEANDSENDCORRELATINGMESSAGE<sub>*pattern*</sub> (R6.5.3-21),  
referenced in GENERATEANDSENDCORRELATINGMESSAGE<sub>*invoke*</sub> (R7.8.1-39),  
references COPYVARIABLETOMSG (RB.3.2-111), CORRELATEINSTANCETOVARIABLE  
(RB.3.1-109), SENDMESSAGE<sub>*invoke*</sub> (RB.4.7-125)

SENDMESSAGE<sub>*invoke*</sub> (RB.4.7-125)  
(*sl*  $\in$  *SubInstance*, *portdescr*  $\in$  *PortDescriptor*, *msg*  $\in$  *BPELMsg*)  $\equiv$

**let** *partnerL* = *portPartnerLink*(*portdescr*),  
*pT* = *portPortType*(*portdescr*),  
*op* = *portOperation*(*portdescr*),  
*pl* = *processInstance*(*sl*) **in**  
**add msg to** *remotePort<sub>in</sub>*(*pl*, *partnerL*, *pT*, *op*)

refines SENDMESSAGE<sub>*pattern*</sub> (R6.5.3-23),  
referenced in GENERATEANDSENDMESSAGE<sub>*invoke*</sub> (RB.4.7-124)

RUNINVOKE<sub>*receiving*</sub> (R7.8.1-40)  
(*sl*  $\in$  *SubInstance*, *invoke*  $\in$  *Invoke*)  $\equiv$

**let** *portdescr<sub>receive</sub>* = *invokePortDescriptor<sub>receive</sub>*(*invoke*) **in**  
**let** *partnerL* = *portPartnerLink*(*portdescr*),  
*pT* = *portPortType*(*portdescr*),  
*op* = *portOperation*(*portdescr*),  
*pl* = *processInstance*(*sl*) **in**  
**select** *fault*  $\in$  *remotePort<sub>fault</sub>*(*pl*, *partnerL*, *pT*, *op*) **in**  
**signal** *fault* **to** *parentActivity*(*invoke*) **in** *sl* **via** *signalChannel<sub>fault</sub>*  
**remove fault from** *remotePort<sub>fault</sub>*(*pl*, *partnerL*, *pT*, *op*)  
**ifnone**  
AWAITANDRECEIVECORRELATINGMESSAGE<sub>*invoke*</sub>(*sl*, *invoke*, *portdescr<sub>receive</sub>*)

AWAITANDRECEIVECORRELATINGMESSAGE<sub>*invoke*</sub> (RB.4.7-126)  
(*sl*  $\in$  *SubInstance*, *invoke*  $\in$  *Invoke*, *portdescr*  $\in$  *PortDescriptor*)  $\equiv$

**let** *partnerL* = *portPartnerLink*(*portdescr*),  
*pT* = *portPortType*(*portdescr*),  
*op* = *portOperation*(*portdescr*),  
*pl* = *processInstance*(*sl*) **in**

```

select msg ∈ remotePortout(pl, partnerL, pT, op) in
  let CS = portCorrelationSet(portdescr) in
    if correlationSatisfied(sl, invoke, msg, CS) then
      remove msg from remotePortout(pl, partnerL, pT, op)
      RECEIVEMESSAGEpattern(sl, invoke, portdescr, msg)
      set activityState from running to completing
    else
      THROW(sl, invoke, correlationViolation)

```

referenced in RUNINVOKE<sub>receiving</sub> (R7.8.1-40),

refines AWAITANDRECEIVECORRELATINGMESSAGE<sub>pattern</sub> (R6.5.3-19),

references RECEIVEMESSAGE<sub>pattern</sub> (R6.5.3-20), THROW (R5.5.1-10)

### B.4.8 Assign

```

<assign standard-attributes>
  standard-elements
  <copy>+
    from-spec
    to-spec
  </copy>
</assign>

```

where from-spec is one of the following elements

```

<from variable="ncname" part="ncname"?/>
<from partnerLink="ncname" endpointReference="myRole|partnerRole"/>
<from variable="ncname" property="qname"/>
<from expression="general-expr"/>
<from> ... literal value ... </from>

```

and to-spec is one of the following elements

```

<to variable="ncname" part="ncname"?/>
<to partnerLink="ncname"/>
<to variable="ncname" property="qname"/>

```

**Universe:** *LiteralExpression* (for ...literal value...)

**Universe:** *AssignSpecType* =<sub>def</sub> {*specTypeVariablePart*, *specTypeVariableProperty*, *specTypePartnerLink*, *specTypeExpression*, *specTypeLiteral*}

```

specType : AssignSpec → AssignSpecType
specVariable : AssignSpec → Variable
specMsgTypePart : AssignSpec → MessageTypePart
specMsgProperty : AssignSpec → MessageProperty
specPartnerLink : AssignSpec → PartnerLink
specRoleType : AssignSpec → RoleType
specLiteral : AssignSpec → LiteralExpression
specExpression : AssignSpec → Expression

```

The following functions allow us to consider a concrete *EndpointReference* as an abstract *Value* and vice versa. This technicality is required to match BPEL's semantics of assign and the abstraction from values used in ASM.

$A \text{ endpointReferenceValue} : \text{EndpointReference} \rightarrow \text{Value}$   
 $A \text{ valueEndpointReference} : \text{Value} \rightarrow \text{EndpointReference}$

EXECUTEASSIGN (RB.4.8-127)

$(sl \in \text{SubInstance}, \text{assign} \in \text{Assign}) \equiv$

PROPAGATEDPE(*sl*, *assign*)

PROPAGATETERMINATE<sub>basic</sub>(*sl*, *assign*)

**onSignal** *signalStop* **from** *parentActivity*(*assign*)

**in** *sI* **via** *signalChannel*<sub>down</sub> **do**

STOPACTIVITY(*sl*, *assign*)

**otherwise**

**onSignal** *signalEnable* **from** *parentActivity*(*assign*)

**in** *sI* **via** *signalChannel*<sub>down</sub> **do**

**set** *activityState* **from** *disabled* **to** *enabled*

**if** *activityState*(*sl*, *assign*) = *enabled* **then**

STARTACTIVITY(*sl*, *assign*)

**if** *activityState*(*sl*, *assign*) = *running* **then**

RUNASSIGN(*sl*, *assign*)

**if** *activityState*(*sl*, *assign*) = *completing* **then**

COMPLETEACTIVITY(*sl*, *assign*)

refines EXECUTEACTIVITY<sub>pattern,basic</sub> (R7.1.3-28),

referenced in EXECUTEACTIVITY<sub>basic</sub> (RB.2.2-105), referencing RUNASSIGN (R7.9.1-41)

RUNASSIGN (R7.9.1-41)

$(sl \in \text{SubInstance}, \text{assign} \in \text{Assign}) \equiv$

**forall** *copySpec*  $\in$  *assignSpecs*(*assign*) **do**

**let** *value* = *assignSpecValue*(*sl*, *assignFromSpec*(*copySpec*)) **in**

SETVALUEBYSPEC(*sl*, *value*, *assignToSpec*(*copySpec*))

SETVALUEBYSPEC (RB.4.8-128)

$(sl \in \text{SubInstance}, \text{value} \in \text{Value}, \text{spec} \in \text{AssignSpec}) \equiv$

**if** *specType*(*spec*) = *specTypeVariablePart* **then**

**let** *var* = *specVariable*(*spec*),

*part* = *specMsgTypePart*(*spec*),

```

    type = variableType(var) in
if type ∈ XMLschemaType ∨ (type ∈ MessageType ∧ part = undef) then
    variablePartValue(sl, var, type) := value
else
    variablePartValue(sl, var, part) := value
if specType(spec) = specTypeVariableProperty then
let var = specVariable(spec),
    prop = specMsgProperty(spec),
    part = aliasMsgTypePart(messagePropertyAlias(prop)) in
    variablePartValue(sl, var, part) := value
if specType(spec) = specTypePartnerLink then
let partnerL = specPartnerLink(spec),
    role = partnerLinkRole(partnerL, partnerRole),
    endpoint = valueEndpointReference(value) in
    partnerLinkRoleEndpoint(processInstance(sl), partnerL, role) := endpoint

```

referenced in RUNASSIGN (R7.9.1-41)

$$\mathcal{D} \text{ assignSpecValue} : \text{SubInstance} \times \text{AssingSpec} \rightarrow \text{Value}$$

$(sl, spec) \mapsto$	$\left\{ \begin{array}{l} \\ \\ \\ \\ \\ \\ \end{array} \right.$	litValue,	$specType(spec) = specTypeLiteral$ $\wedge litExpr = specLiteral(spec)$ $\wedge litValue = evaluateExpression(sl, litExpr)$
		exprValue,	$specType(spec) = specTypeExpression$ $\wedge expr = specExpression(spec)$ $\wedge exprValue = evaluateExpression(sl, expr)$
		value,	$specType(spec) = specTypeVariablePart$ $\wedge var = specVariable(spec)$ $\wedge type = variableMessageType(var)$ $\wedge (type \in XMLschemaType$ $\quad \Rightarrow value = variablePartValue(sl, var, type))$ $\wedge (type \in MessageType \wedge part = specMsgTypePart(spec)$ $\quad \wedge (part \neq undef$ $\quad \quad \Rightarrow value = variablePartValue(sl, var, part))$ $\quad \wedge (part = undef$ $\quad \quad \quad \Rightarrow value = variablePartValue(sl, var, type)))$
		propValue,	$specType(spec) = specTypeVariableProperty$ $\wedge var = specVariable(spec)$ $\wedge msgProp = specMsgProperty(spec)$ $\wedge alias = messagePropertyAlias(msgProp)$ $\wedge part = aliasMsgTypePart(alias)$ $\wedge propValue = variablePartValue(sl, var, part)$
		partnerLValue,	$specType(spec) = specTypePartnerLink$ $\wedge pl = processInstance(sl)$ $\wedge partnerL = specPartnerLink(spec)$ $\wedge role = partnerLinkRole(partnerL, specRoleType(spec))$ $\wedge reference$ $\quad = partnerLinkRoleEndpoint(pl, partnerL, role)$ $\wedge partnerLValue$ $\quad = endpointReferenceValue(reference)$

## B.5 Rules and Functions for Structured Activities

### B.5.1 Structured Activities

RUNACTIVITY<sub>structured</sub> (RB.5.1-129)  
 $(sl \in SubInstance, act \in Activity_{structured}) \equiv$

**if** *activityRunningChilds*(sl, act) =  $\emptyset$  **then**  
     **set** *activityState* **from** *running* **to** *completing*  
**else**  
     **forall** (sl<sub>child</sub>, child)  $\in$  *activityRunningChilds*(sl, act) **do**  
         **onSignal** *signalCompleted* **from** child **in** sl<sub>child</sub> **via** *signalChannel<sub>up</sub>* **do**  
             **remove** (sl<sub>child</sub>, child) **from** *activityRunningChilds*(sl, act)

refines RUNACTIVITY<sub>pattern</sub> (R8.1.2-42),  
 referenced in EXECUTEACTIVITY<sub>pattern,structured</sub> (R8.1.6-44)

STOPACTIVITY<sub>structured</sub> (R8.1.4-43)  
 $(sl \in SubInstance, act \in Activity_{structured}) \equiv$

**if** *activityRunningChilds*(sl, act) =  $\emptyset$  **then**  
     STOPACTIVITY(sl, act)  
**else**  
     **if** *activityStopMode*(sl, act) = *sendingStop* **then**  
         **forall** (sl<sub>child</sub>, child)  $\in$  *activityRunningChilds*(sl, act) **do**  
             **signal** *signalStop* **to** child **in** sl<sub>child</sub> **via** *signalChannel<sub>down</sub>*  
             **set** *activityStopMode* **from** *sendingStop* **to** *awaitingStopped*  
     **if** *activityStopMode*(sl, act) = *awaitingStopped* **then**  
         **forall** (sl<sub>child</sub>, child)  $\in$  *activityRunningChilds*(sl, act) **do**  
             **onSignal** *signalCompleted*, *signalStopped* **from** child  
                 **in** sl<sub>child</sub> **via** *signalChannel<sub>up</sub>* **do**  
                     **remove** (sl<sub>child</sub>, child) **from** *activityRunningChilds*(sl, act)

### B.5.2 Flow

EXECUTEFLOW (RB.5.2-130)  
 $(sl \in SubInstance, flow \in Flow) \equiv$

PROPAGATEDPE(sl, flow)  
 PROPAGATETERMINATE(sl, flow)  
**onSignal** *signalStop* **from** *parentActivity*(flow)  
     **in** sl **via** *signalChannel<sub>down</sub>* **do**  
     ACTIVITYSETTOSTOPPING(sl, flow)

**otherwise**  
**onSignal** *signalEnable* **from** *parentActivity*(*flow*)  
     **in** *sl* **via** *signalChannel<sub>down</sub>* **do**  
         **set** *activityState* **from** *disabled* **to** *enabled*  
         PROPAGATEFAULTSACTIVITY(*sl*, *flow*)  
         **if** *activityState*(*sl*, *flow*) = *enabled* **then**  
             STARTFLOW(*sl*, *flow*)  
         **if** *activityState*(*sl*, *flow*) = *running* **then**  
             RUNACTIVITY<sub>structured</sub>(*sl*, *flow*)  
         **if** *activityState*(*sl*, *flow*) = *completing* **then**  
             COMPLETEACTIVITY(*sl*, *flow*)  
         **if** *activityState*(*sl*, *flow*) = *stopping* **then**  
             STOPACTIVITY<sub>structured</sub>(*sl*, *flow*)

refines EXECUTEACTIVITY<sub>pattern,structured</sub> (R8.1.6-44), referenced in  
 EXECUTEACTIVITY<sub>structured</sub> (RB.2.2-106), referencing STARTFLOW (R8.2.1-45)

STARTFLOW (R8.2.1-45)

(*sl* ∈ *SubInstance*, *flow* ∈ *Flow*) ≡

**if** *allLinksSet*(*sl*, *act*) ∧ *joinConditionSatisfied*(*sl*, *act*) = *true* **then**  
     **forall** *child* ∈ *childActivities*(*flow*) **do**  
         **signal** *signalEnable* **to** *child* **in** *sl* **via** *signalChannel<sub>down</sub>*  
         **add** (*sl*, *child*) **to** *activityRunningChilds*(*sl*, *flow*)  
         **set** *activityState* **from** *enabled* **to** *running*

### B.5.3 Sequence

EXECUTESEQUENCE (RB.5.3-131)

(*sl* ∈ *SubInstance*, *sequence* ∈ *Sequence*) ≡

PROPAGATEDPE(*sl*, *sequence*)  
 PROPAGATETERMINATE(*sl*, *sequence*)  
**onSignal** *signalStop* **from** *parentActivity*(*sequence*)  
     **in** *sl* **via** *signalChannel<sub>down</sub>* **do**  
         ACTIVITYSETTOSTOPPING(*sl*, *sequence*)  
**otherwise**  
     **onSignal** *signalEnable* **from** *parentActivity*(*sequence*)  
         **in** *sl* **via** *signalChannel<sub>down</sub>* **do**  
             **set** *activityState* **from** *disabled* **to** *enabled*  
         PROPAGATEFAULTSACTIVITY(*sl*, *sequence*)  
         **if** *activityState*(*sl*, *sequence*) = *enabled* **then**

```

STARTSEQUENCE(sl, sequence)
if activityState(sl, sequence) = running then
  RUNSEQUENCE(sl, sequence)
if activityState(sl, sequence) = completing then
  COMPLETEACTIVITY(sl, sequence)
if activityState(sl, sequence) = stopping then
  STOPACTIVITYstructured(sl, sequence)

```

refines EXECUTEACTIVITY<sub>pattern,structured</sub> (R8.1.6-44), referenced in  
 EXECUTEACTIVITY<sub>structured</sub> (RB.2.2-106), referencing STARTSEQUENCE (R8.3.1-47),  
 RUNSEQUENCE (RB.5.3-132)

STARTSEQUENCE (R8.3.1-47)

(sl ∈ SubInstance, seq ∈ Sequence) ≡

```

if allLinksSet(sl, act) ∧ joinConditionSatisfied(sl, act) = true then
  ENABLESEQUENCECHILD(sl, seq, sequenceFirstActivity(seq))
set activityState from enabled to running

```

ENABLESEQUENCECHILD (R8.3.1-46)

(sl ∈ SubInstance, seq ∈ Sequence, next ∈ Activity) ≡

```

signal signalEnable to next in sl via signalChanneldown
add (sl, next) to activityRunningChlds(sl, seq)

```

RUNSEQUENCE (RB.5.3-132)

(sl ∈ SubInstance, seq ∈ Sequence) ≡

```

if activityRunningChlds(sl, seq) = ∅ then
  set activityState from running to completing
else
  forall (slchild, child) ∈ activityRunningChlds(sl, seq) do
    onSignal signalCompleted from child in slchild via signalChannelup do
      remove (slchild, child) from activityRunningChlds(sl, seq)
      let next = sequenceActivityNext(child) in
        if next ≠ undef then
          ENABLESEQUENCECHILD(sl, seq, next)

```

refines RUNACTIVITY<sub>structured</sub> (RB.5.1-129),  
 referenced in EXECUTESEQUENCE (RB.5.3-131)

## B.5.4 Switch

EXECUTESWITCH (RB.5.4-133)

( $sl \in SubInstance, switch \in Switch$ )  $\equiv$

PROPAGATEDPE( $sl, switch$ )

PROPAGATETERMINATE( $sl, switch$ )

**onSignal** *signalStop* **from** *parentActivity*( $switch$ )

**in** *sI* **via** *signalChannel<sub>down</sub>* **do**

ACTIVITYSETTOSTOPPING( $sl, switch$ )

**otherwise**

**onSignal** *signalEnable* **from** *parentActivity*( $switch$ )

**in**  $sl$  **via** *signalChannel<sub>down</sub>* **do**

**set** *activityState* **from** *disabled* **to** *enabled*

PROPAGATEFAULTSACTIVITY( $sl, switch$ )

**if** *activityState*( $sl, switch$ ) = *enabled* **then**

STARTSWITCH( $sl, switch$ )

**if** *activityState*( $sl, switch$ ) = *running* **then**

RUNACTIVITY<sub>structured</sub>( $sl, switch$ )

**if** *activityState*( $sl, switch$ ) = *completing* **then**

COMPLETEACTIVITY( $sl, switch$ )

**if** *activityState*( $sl, switch$ ) = *stopping* **then**

STOPACTIVITY<sub>structured</sub>( $sl, switch$ )

refines EXECUTEACTIVITY<sub>pattern,structured</sub> (R8.1.6-44), referenced in  
EXECUTEACTIVITY<sub>structured</sub> (RB.2.2-106), referencing STARTSWITCH (R8.4.1-49)

STARTSWITCH (R8.4.1-49)

( $sl \in SubInstance, seq \in Sequence$ )  $\equiv$

**if** *allLinksSet*( $sl, act$ )  $\wedge$  *joinConditionSatisfied*( $sl, act$ ) = *true* **then**

SWITCHCHOSEBRANCH( $sl, switch$ )

**set** *activityState* **from** *enabled* **to** *running*

SWITCHCHOSEBRANCH (R8.4.1-48)

( $sl \in SubInstance, switch \in Switch$ )  $\equiv$

**let** *branch* = *fulfilledSwitchBranch*( $sl, switchFirstBranch$ ( $switch$ )),

*child* = *switchBranchActivity*(*branch*) **in**

**add** ( $sl, child$ ) **to** *activityRunningChilds*( $sl, switch$ )

**signal** *signalEnable* **to** *child* **in**  $sl$  **via** *signalChannel<sub>down</sub>*

**forall** *skippedChild*  $\in$  *childActivities*( $switch$ )  $\setminus$  {*child*} **do**

INITDPE( $sl, switch, skippedChild$ )

derived functions ( $\mathcal{D}$  B.5.4-1)

$$\mathcal{D} \text{ fulfilledSwitchBranch} : \text{SubInstance} \times \text{SwitchBranch} \rightarrow \text{SwitchBranch}$$

$$(\text{sl}, \text{branch}) \mapsto \begin{cases} \text{branch,} & \text{condition} = \text{switchBranchCondition}(\text{branch}) \\ & \wedge (\text{condition} = \text{otherwise} \\ & \quad \vee \text{evaluateConditionExpression}(\text{sl}, \text{condition}) = \text{true}) \\ \text{next,} & \text{next} = \text{fulfilledSwitchBranch}(\text{sl}, \text{switchBranchNext}(\text{branch})) \end{cases}$$

### B.5.5 Pick

derived functions ( $\mathcal{D}$  B.5.5-1)

$$\mathcal{D} \text{ pickActivatedEvents} : \text{SubInstance} \times \text{Pick} \rightarrow \mathcal{P}(\text{EventDescriptor})$$

$$(\text{sl}, \text{pick}) \mapsto \{\text{ev} \in \text{pickEvents}(\text{pick}) \mid \text{eventOccured}(\text{sl}, \text{ev})\}$$

$$\mathcal{D} \text{ eventOccured} : \text{SubInstance} \times \text{EventDescriptor} \rightarrow \{\text{true}, \text{false}\}$$

$$(\text{sl}, \text{ev}) \mapsto \begin{cases} \text{true,} & (\text{eventType}(\text{ev}) = \text{onMsg} \\ & \quad \wedge \text{msgEventReceivedMsgs}(\text{sl}, \text{ev}) \neq \emptyset) \\ & \vee (\text{eventType}(\text{ev}) = \text{onAlarm} \\ & \quad \wedge \text{now} = \text{getCurrentTime}(\text{processInstance}(\text{sl})) \\ & \quad \wedge \text{eventAlarmEndTime}(\text{sl}, \text{ev}) \leq_{\text{Time}} \text{now}) \\ \text{false,} & \text{else} \end{cases}$$

derived functions ( $\mathcal{D}$  B.5.5-2)

$$\mathcal{D} \text{ eventOccurrence} : \text{SubInstance} \times \text{EventDescriptor} \rightarrow \text{Time}$$

$$(\text{sl}, \text{ev}) \mapsto \begin{cases} \text{endTime,} & \text{eventType}(\text{ev}) = \text{onAlarm} \\ & \quad \wedge \text{endTime} = \text{eventAlarmEndTime}(\text{sl}, \text{ev}) \\ \text{receiveTime,} & \text{eventType}(\text{ev}) = \text{onMsg} \\ & \quad \wedge \text{msg} \in \text{msgEventReceivedMsgs}(\text{sl}, \text{ev}) \\ & \quad \wedge \forall \text{msg}' \in \text{msgEventReceivedMsgs}(\text{sl}, \text{ev}) \\ & \quad \quad \text{messageReceiveTime}(\text{msg}) \\ & \quad \quad \leq_{\text{Time}} \text{messageReceiveTime}(\text{msg}') \\ \text{undef,} & \text{else} \end{cases}$$

$$\mathcal{D} \text{ msgEventReceivedMsgs} : \text{SubInstance} \times \text{EventDescriptor} \rightarrow \mathcal{P}(\text{BPELMsg})$$

$$(\text{sl}, \text{ev}) \mapsto \{\text{msg} \mid \text{pl} = \text{processInstance}(\text{sl}) \\ \wedge \text{portdescr} = \text{eventMsgPortDescriptor}(\text{ev}) \\ \wedge \text{partnerL} = \text{portPartnerLink}(\text{portdescr}) \\ \wedge \text{pT} = \text{portPortType}(\text{portdescr}) \\ \wedge \text{op} = \text{portOperation}(\text{portdescr}) \\ \wedge \text{msg} \in \text{localPort}_{in}(\text{pl}, \text{partnerL}, \text{pT}, \text{op})\}$$

EXECUTE PICK (RB.5.5-134)

( $sl \in SubInstance, pick \in Pick$ )  $\equiv$

PROPAGATEDPE( $sl, pick$ )

PROPAGATE TERMINATE( $sl, pick$ )

**onSignal** *signalStop* **from** *parentActivity*( $pick$ )

**in** *sI* **via** *signalChannel<sub>down</sub>* **do**

ACTIVITYSETTOSTOPPING( $sl, pick$ )

**otherwise**

**onSignal** *signalEnable* **from** *parentActivity*( $pick$ )

**in**  $sl$  **via** *signalChannel<sub>down</sub>* **do**

**set** *activityState* **from** *disabled* **to** *enabled*

PROPAGATEFAULTSACTIVITY( $sl, pick$ )

**if** *activityState*( $sl, pick$ ) = *enabled* **then**

STARTPICK( $sl, pick$ )

**if** *activityState*( $sl, pick$ ) = *running* **then**

RUNPICK( $sl, pick$ )

**if** *activityState*( $sl, pick$ ) = *completing* **then**

COMPLETEACTIVITY( $sl, pick$ )

**if** *activityState*( $sl, pick$ ) = *stopping* **then**

STOPACTIVITY<sub>structured</sub>( $sl, pick$ )

refines EXECUTEACTIVITY<sub>pattern,structured</sub> (R8.1.6-44), referenced in  
EXECUTEACTIVITY<sub>structured</sub> (RB.2.2-106), referencing STARTPICK (R8.5.1-51), RUNPICK  
(R8.5.2-52)

STARTPICK (R8.5.1-51)

( $sl \in SubInstance, pick \in Pick$ )  $\equiv$

**if** *allLinksSet*( $sl, pick$ )  $\wedge$  *joinConditionSatisfied*( $sl, pick$ ) = *true* **then**

**set** *activityState* **from** *enabled* **to** *running*

**forall**  $event \in pickEvents(pick)$  **where** *eventType*( $event$ ) = *onAlarm* **do**

INITALARMEVENT( $sl, event$ )

INITALARMEVENT (R8.5.1-50)

( $sl \in SubInstance, event \in EventDescriptor$ )  $\equiv$

**let**  $then = evaluateTimeExpression(sl, eventAlarmExpression(event))$  **in**

**if** *eventAlarmType*( $event$ ) = *for* **then**

*eventAlarmEndTime*( $sl, event$ ) := *getCurrentTime*( $sl$ ) + *Time* **then**

**if** *eventAlarmType*( $event$ ) = *until* **then**

*eventAlarmEndTime*( $sl, event$ ) :=  $then$

RUNPICK (R8.5.2-52)

```
(sl ∈ SubInstance, pick ∈ Pick) ≡
if pickChosenActivity(sl, pick) = undef then
  PICKCHOSEBRANCH(sl, pick)
else
  PICKRUNCHOSENBRANCH(sl, pick)
```

PICKCHOSEBRANCH (R8.5.2-53)

```
(sl ∈ SubInstance, pick ∈ Pick) ≡
let activatedEvents = pickActivatedEvents(sl, pick) in
  select ev ∈ activatedEvents
    where ∃ ev' ∈ activatedEvents
      eventOccurrence(sl, ev) ≤Time eventOccurrence(sl, ev') in
    PICKACTIVATEBRANCH(sl, pick, ev)
```

PICKACTIVATEBRANCH (R8.5.2-54)

```
(sl ∈ SubInstance, pick ∈ Pick, event ∈ EventDescriptor) ≡
let child = pickEventActivity(event) in
  add (sl, child) to activityRunningChilds(sl, pick)
  signal signalEnable to child in sl via signalChanneldown
  forall skippedChild ∈ childActivities(pick) \ {child} do
    INITDPE(sl, pick, skippedChild)
if eventType(event) = onMsg then
  AWAITANDRECEIVECORRELATINGMESSAGEpick(sl, pick, eventMsgPortDescriptor(event))
```

AWAITANDRECEIVECORRELATINGMESSAGE<sub>pick</sub> (RB.5.5-135)

```
(sl ∈ SubInstance, pick ∈ Pick, portdescr ∈ PortDescriptor) ≡
let partnerL = portPartnerLink(portdescr),
    pT = portPortType(portdescr),
    op = portOperation(portdescr),
    pl = processInstance(sl) in
  select msg ∈ localPortin(pl, partnerL, pT, op) in
    let CS = portCorrelationSet(portdescr) in
      if correlationSatisfied(sl, pick, msg, CS) then
        remove msg from localPortin(sl, partnerL, pT, op)
        RECEIVEMESSAGEpattern(sl, pick, portdescr, msg)
      else
        THROW(sl, pick, correlationViolation)
```

referenced in PICKACTIVATEBRANCH (R8.5.2-54),  
 refines AWAITANDRECEIVECORRELATINGMESSAGE<sub>pattern</sub> (R6.5.3-19),  
 references RECEIVEMESSAGE<sub>pattern</sub> (R6.5.3-20), THROW (R5.5.1-10)

PICKRUNCHOSENBRANCH (R8.5.2-55)  
 $(sl \in SubInstance, pick \in Pick) \equiv$   
 RUNACTIVITY<sub>structured</sub>(sl, pick)

### B.5.6 While

EXECUTEWHILE (RB.5.6-136)  
 $(sl \in SubInstance, while \in While) \equiv$   
 PROPAGATEDPE(sl, while)  
 PROPAGATETERMINATE(sl, while)  
**onSignal** *signalStop* **from** *parentActivity*(while)  
     **in** *sI* **via** *signalChannel<sub>down</sub>* **do**  
         ACTIVITYSETTOSTOPPING(sl, while)  
**otherwise**  
     **onSignal** *signalEnable* **from** *parentActivity*(while)  
         **in** *sl* **via** *signalChannel<sub>down</sub>* **do**  
             **set** *activityState* **from** *disabled* **to** *enabled*  
         PROPAGATEFAULTSACTIVITY(sl, while)  
         **if** *activityState*(sl, while) = *enabled* **then**  
             STARTACTIVITY(sl, while)  
         **if** *activityState*(sl, while) = *running* **then**  
             RUNWHILE(sl, while)  
         **if** *activityState*(sl, while) = *completing* **then**  
             COMPLETEACTIVITY(sl, while)  
         **if** *activityState*(sl, while) = *stopping* **then**  
             STOPACTIVITY<sub>structured</sub>(sl, while)  
 refines EXECUTEACTIVITY<sub>pattern,structured</sub> (R8.1.6-44), referenced in  
 EXECUTEACTIVITY<sub>structured</sub> (RB.2.2-106), referencing RUNWHILE (R8.6.2-56)

RUNWHILE (R8.6.2-56)  
 $(sl \in SubInstance, while \in While) \equiv$   
**if** *whileIterationState*(sl, while) = *head* **then**  
     **if** *evaluateConditionExpression*(sl, *whileCondition*(while)) = *true* **then**  
         WHILESTARTNEWBODY(sl, while)  
     **else**  
         **set** *activityState* **from** *running* **to** *completing*  
**if** *whileIterationState*(sl, while) = *body* **then**  
     WHILEENDBODY(sl, while)

WHILESTARTNEWBODY (R8.6.2-57)

( $sl \in SubInstance, while \in While$ )  $\equiv$   
**let**  $sl_{body} = \mathbf{new}(SubInstance)$ ,  
      $child = childActivity(while)$  **in**  
     **add** ( $sl_{body}, child$ ) **to**  $activityRunningChilds(sl, while)$   
     **signal**  $signalEnable$  **to**  $childActivity(while)$  **in**  $sl_{body}$  **via**  $signalChannel_{down}$   
     EXTENDSUBINSTANCETREE( $sl, sl_{body}$ )  
**set**  $whileIterationState$  **from**  $head$  **to**  $body$

WHILEENDBODY (RB.5.6-137)

( $sl \in SubInstance, while \in While$ )  $\equiv$   
**if**  $activityRunningChilds(sl, act) = \emptyset$  **then**  
     **set**  $activityState$  **from**  $running$  **to**  $completing$   
**else**  
     **forall** ( $sl_{child}, child$ )  $\in activityRunningChilds(sl, act)$  **do**  
         **onSignal**  $signalCompleted$  **from**  $child$  **in**  $sl_{child}$  **via**  $signalChannel_{up}$  **do**  
             **remove** ( $sl_{child}, child$ ) **from**  $activityRunningChilds(sl, act)$

refines RUNACTIVITY<sub>pattern</sub> (R8.1.2-42) referenced in RUNWHILE (R8.6.2-56)

## B.6 Rules and Functions for the Scope and the Handlers

### B.6.1 Scope

EXECUTESCOPE (R9.1.0-58)  
 $(sl \in SubInstance, scope \in Scope) \equiv$

PROPAGATEDPE( $sl, scope$ )

PROPAGATETERMINATE( $sl, scope$ )

**onSignal**  $signalStop$  **from**  $parentActivity(scope)$  **in**  $sI$  **via**  $signalChannel_{down}$  **do**

SCOPEHANDLESIGNALSTOP( $sl, scope$ )

**otherwise**

**if**  $scopeMode(sl, scope) = positive$  **then**

EXECUTESCOPE<sub>positive</sub>( $sl, scope$ )

**if**  $scopeMode(sl, scope) = negative$  **then**

EXECUTESCOPE<sub>negative</sub>( $sl, scope$ )

**if**  $scopeMode(sl, scope) = faulted$  **then**

**skip**

SCOPEHANDLESIGNALSTOP (R9.1.1-59)  
 $(sl \in SubInstance, scope \in Scope) \equiv$

**if**  $scopeMode(sl, scope) \in \{positive, negative\}$  **then**

**if**  $scopeFHRunning(sl, scope) = false$  **then**

THROWFORCEDTERMINATION( $sl, scope$ )

**if**  $scopeFHRunning(sl, scope) = true$  **then**

**skip**

**if**  $scopeMode(sl, scope) = faulted$  **then**

**signal**  $signalStopped$  **to**  $parentActivity(scope)$  **in**  $sl$  **via**  $signalChannel_{up}$

THROWFORCEDTERMINATION (R9.1.2-61)  
 $(sl \in SubInstance, scope \in Scope) \equiv$

**let**  $fault = new(Fault)$  **in**

$faultName(fault) := forcedTermination$

**add**  $fault$  **to**  $faultChannel_{down}(sl, scope, scopeFaultHandler(scope))$

### Positive Control Flow

EXECUTESCOPE<sub>positive</sub> (RB.6.1-138)  
 $(sl \in SubInstance, scope \in Scope) \equiv$

**onSignal**  $signalStop$  **from**  $scopeFaultHandler(scope)$

**in**  $sl$  **via**  $signalChannel_{up}$  **do**

SCOPESETTOSTOPPING( $sl, scope$ )

```

otherwise
  onSignal signalEnable from parentActivity(scope)
    in sl via signalChanneldown do
      set activityState from disabled to enabled

  PROPAGATEFAULTSCOPE(sl, scope)

  if activityState(sl, act) = enabled then
    STARTSCOPEpositive(sl, act)
  if activityState(sl, act) = running then
    RUNSCOPEpositive(sl, act)
  if activityState(sl, act) = completing then
    COMPLETESCOPEpositive(sl, act)
  if activityState(sl, act) = stopping then
    STOPSCOPEpositive(sl, act)
    
```

refines EXECUTEACTIVITY<sub>pattern</sub> (R5.9.0-18),  
 referenced in EXECUTESCOPE (R9.1.0-58),  
 references SCOPESETTOSTOPPING (R9.2.1-63), PROPAGATEFAULTSCOPE (R9.1.2-60),  
 STARTSCOPE<sub>positive</sub> (RB.6.1-139), RUNSCOPE<sub>positive</sub> (RB.6.1-140), COMPLETESCOPE<sub>positive</sub>,  
 STOPSCOPE<sub>positive</sub>

SCOPESETTOSTOPPING (R9.2.1-63)

(*sl* ∈ *SubInstance*, scope ∈ *Scope*) ≡

ACTIVITYSETTOSTOPPING(*sl*, scope)  
*scopeFHRunning*(*sl*, scope) := true

PROPAGATEFAULTSCOPE (R9.1.2-60)

(*sl* ∈ *SubInstance*, scope ∈ *Scope*) ≡

**forall** (*sl<sub>child</sub>*, *child*) ∈ *activityRunningChildds*(*sl*, act)  
**where** *child* ≠ *scopeFaultHandler*(scope) **do**  
**forall** *fault* ∈ *faultChannel<sub>up</sub>*(*sl<sub>child</sub>*, *child*, scope) **do**  
**add** *fault* **to** *faultChannel<sub>down</sub>*(*sl*, scope, *scopeFaultHandler*(scope))  
**remove** *fault* **from** *faultChannel<sub>up</sub>*(*sl<sub>child</sub>*, *child*, scope)

STARTSCOPE<sub>positive</sub> (RB.6.1-139)

(*sl* ∈ *SubInstance*, scope ∈ *Scope*) ≡

**if** *allLinksSet*(*sl*, scope) = true ∧ *joinConditionSatisfied*(*sl*, scope) = true **then**  
**forall** *child* ∈ *childActivities*(scope) \ {*scopeCompensationHandler*(scope)} **do**  
**signal** *signalEnable* **to** *child* **in** *sl* **via** *signalChannel<sub>down</sub>*  
**add** (*sl*, *child*) **to** *activityRunningChildds*(*sl*, scope)  
**set** *activityState* **from** *enabled* **to** *running*

refines STARTFLOW (R8.2.1-45),  
 referenced in EXECUTESCOPE<sub>positive</sub> (RB.6.1-138)

RUNSCOPE<sub>positive</sub> (RB.6.1-140)

( $sl \in SubInstance, scope \in Scope$ )  $\equiv$

**if** ( $sl, scopePrimaryActivity(scope)$ )  $\notin$   $activityRunningChlds(sl, scope)$  **then**  
   **set**  $activityState$  **from** *running* **to** *completing*  
**else**  
   **forall** ( $sl_{child}, child$ )  $\in$   $activityRunningChlds(sl, scope)$  **do**  
     **onSignal**  $signalCompleted$  **from**  $child$  **in**  $sl_{child}$  **via**  $signalChannel_{up}$  **do**  
       **remove** ( $sl_{child}, child$ ) **from**  $activityRunningChlds(sl, scope)$

refines RUNACTIVITY<sub>structured</sub> (RB.5.1-129),  
 referenced in EXECUTESCOPE<sub>positive</sub> (RB.6.1-138)

STOPSCOPE<sub>positive</sub> (RB.6.1-141)

( $sl \in SubInstance, scope \in Scope$ )  $\equiv$

**if**  $activityRunningChlds(sl, scope) \subseteq (sl, scopeFaultHandler(scope))$  **then**  
   SCOPESWITCHTONEGATIVE( $sl, scope$ )  
**else**  
   **if**  $activityStopMode(sl, scope) = sendingStop$  **then**  
     **forall** ( $sl_{child}, child$ )  $\in$   $activityRunningChlds(sl, scope)$   
       **where**  $child \neq scopeFaultHandler(scope)$  **do**  
         **signal**  $signalStop$  **to**  $child$  **in**  $sl_{child}$  **via**  $signalChannel_{down}$   
         **set**  $activityStopMode$  **from** *sendingStop* **to** *awaitingStopped*  
   **if**  $activityStopMode(sl, scope) = awaitingStopped$  **then**  
     **forall** ( $sl_{child}, child$ )  $\in$   $activityRunningChlds(sl, scope)$   
       **where**  $child \neq scopeFaultHandler(scope)$  **do**  
         **onSignal**  $signalCompleted, signalStopped$  **from**  $child$   
           **in**  $sl_{child}$  **via**  $signalChannel_{up}$  **do**  
           **remove** ( $sl_{child}, child$ ) **from**  $activityRunningChlds(sl, scope)$

refines STOPACTIVITY<sub>structured</sub> (R8.1.4-43),  
 referenced in EXECUTESCOPE<sub>positive</sub> (RB.6.1-138),  
 references SCOPESWITCHTONEGATIVE (R9.2.4-64)

SCOPESWITCHTONEGATIVE (R9.2.4-64)

( $sl \in SubInstance, scope \in Scope$ )  $\equiv$

**set**  $scopeMode$  **from** *positive* **to** *negative*  
**set**  $activityState$  **from** *stopping* **to** *enabled*

COMPLETESCOPE<sub>positive</sub> (R9.2.5-65)

( $sl \in SubInstance, scope \in Scope$ )  $\equiv$

**if**  $scopeCompletingState(sl, scope) = sendingComplete$  **then**  
 SCOPEINITIALETEVENTHANDLERCOMPLETION( $sl, scope$ )  
 $scopeCompletingState(sl, scope) := awaitingCompleted_{event}$

**if**  $scopeCompletingState(sl, scope) = awaitingCompleted_{event}$  **then**  
 SCOPEAWAITEVENTHANDLERCOMPLETION( $sl, scope$ )

**if**  $scopeCompletingState(sl, scope) = awaitingCompleted_{fault}$  **then**  
**onSignal**  $signalCompleted$  **from**  $scopeFaultHandler(scope)$   
**in**  $sl$  **via**  $signalChannel_{up}$  **do**  
 COMPLETESCOPE( $sl, scope$ )

SCOPEINITIALETEVENTHANDLERCOMPLETION (R9.2.5-66)

( $sl \in SubInstance, scope \in Scope$ )  $\equiv$

**forall** ( $sl_{eh}, eh$ )  $\in$   $activityRunningChilds(sl, scope)$   
**where**  $eh \in scopeEventHandlers(scope)$  **do**  
**signal**  $signalComplete$  **to**  $eh$  **in**  $sl_{eh}$  **via**  $signalChannel_{down}$

SCOPEAWAITEVENTHANDLERCOMPLETION (RB.6.1-142)

( $sl \in SubInstance, scope \in Scope$ )  $\equiv$

**if**  $\forall (sl_{child}, child) \in activityRunningChilds(sl, scope)$   
 $child \notin scopeEventHandlers(scope)$  **then**  
**signal**  $complete$  **to**  $scopeFaultHandler(scope)$  **in**  $sl$  **via**  $signalChannel_{down}$   
 $scopeCompletingState(sl, scope) := awaitingCompleted_{fault}$

**else**  
**forall** ( $sl_{eh}, eh$ )  $\in$   $activityRunningChilds(sl, scope)$   
**where**  $eh \in scopeEventHandlers(scope)$  **do**  
**onSignal**  $signalCompleted$  **from**  $eh$  **in**  $sl_{eh}$  **via**  $signalChannel_{up}$  **do**  
**remove** ( $sl_{eh}, eh$ ) **from**  $activityRunningChilds(sl, scope)$

refines RUNACTIVITY<sub>pattern</sub> (R8.1.2-42),  
 referenced in COMPLETESCOPE<sub>positive</sub> (R9.2.5-65)

COMPLETESCOPE (R9.2.5-67)

( $sl \in SubInstance, scope \in Scope$ )  $\equiv$

COMPLETEACTIVITY( $sl, scope$ )  
 INSTALLCOMPENSATIONHANDLER( $sl, scope$ )

## Negative Control Flow

EXECUTESCOPE<sub>negative</sub> (R9.4.0-73)

$(sl \in SubInstance, scope \in Scope) \equiv$   
**if**  $fhHasThrownFault(sl, scope) = true$  **then**  
    RETHROWFAULTSCOPE( $sl, scope$ )  
    SCOPESETTOSTOPPING( $sl, scope$ )  
**else**  
    **if**  $activityState(sl, act) = enabled$  **then**  
        STARTSCOPE<sub>negative</sub>( $sl, act$ )  
    **if**  $activityState(sl, act) = running$  **then**  
        RUNACTIVITY<sub>structured</sub>( $sl, act$ )  
    **if**  $activityState(sl, act) = completing$  **then**  
        COMPLETESCOPE<sub>negative</sub>( $sl, act$ )  
    **if**  $activityState(sl, act) = stopping$  **then**  
        STOPSCOPE<sub>negative</sub>( $sl, act$ )

RETHROWFAULTSCOPE (R9.1.2-62)

$(sl \in SubInstance, scope \in Scope) \equiv$   
**forall**  $fault \in faultChannel_{up}(sl, scopeFaultHandler(scope), scope)$  **do**  
    **add**  $fault$  **to**  $faultChannel_{up}(sl, scope, parentActivity(scope))$   
    **remove**  $fault$  **from**  $faultChannel_{up}(sl, scopeFaultHandler(scope), scope)$

COMPLETESCOPE<sub>negative</sub> (RB.6.1-143)

$(sl \in SubInstance, scope \in Scope) \equiv$

**set**  $activityState$  **from**  $completing$  **to**  $stopped$   
**signal**  $signalCompleted$  **to**  $parentActivity(scope)$  **in**  $sl$  **via**  $signalChannel_{up}$   
EVALUATEOUTGOINGLINKS( $sl, scope$ )

refines COMPLETEACTIVITY (R5.4.4-9),  
referenced in EXECUTESCOPE<sub>negative</sub> (R9.4.0-73),  
references EVALUATEOUTGOINGLINKS (R5.4.2-6)

STOPSCOPE<sub>negative</sub> (R9.4.4-75)

$(sl \in SubInstance, scope \in Scope) \equiv$

**signal**  $signalStop$  **to**  $scopeFaultHandler(scope)$  **in**  $sl$  **via**  $signalChannel_{down}$   
**set**  $scopeMode$  **from**  $negative$  **to**  $faulted$   
**set**  $activityState$  **from**  $stopping$  **to**  $stopped$   
DISABLEOUTGOINGLINKS( $sl, scope$ )

## B.6.2 Fault Handler

EXECUTEFAULTHANDLER (RB.6.2-144)

( $sl \in SubInstance, fh \in FaultHandler$ )  $\equiv$

PROPAGATEDPE( $sl, fh$ )

PROPAGATETERMINATE<sub>structured</sub>( $sl, fh$ )

**onSignal** *signalStop* **from** *parentActivity*( $fh$ ) **in** *sI* **via** *signalChannel<sub>down</sub>* **do**  
ACTIVITYSETTOSTOPPING( $sl, fh$ )

**otherwise**

**onSignal** *signalEnable* **from** *parentActivity*( $fh$ )  
**in** *sl* **via** *signalChannel<sub>down</sub>* **do**  
**set** *activityState* **from** *disabled* **to** *enabled*

PROPAGATEFAULTSACTIVITY( $sl, fh$ )

**if** *activityState*( $sl, fh$ ) = *enabled* **then**

STARTACTIVITY( $sl, fh$ )

**if** *activityState*( $sl, fh$ ) = *running* **then**

RUNFAULTHANDLER( $sl, fh$ )

**if** *activityState*( $sl, fh$ ) = *completing* **then**

COMPLETEACTIVITY( $sl, fh$ )

**if** *activityState*( $sl, fh$ ) = *stopping* **then**

STOPACTIVITY<sub>structured</sub>( $sl, fh$ )

refines EXECUTEACTIVITY<sub>pattern,structured</sub> (R8.1.6-44), referenced in  
EXECUTEACTIVITY<sub>structured</sub> (RB.2.2-106), RUNFAULTHANDLER (R9.3.2-68)

RUNFAULTHANDLER (R9.3.2-68)

( $sl \in SubInstance, fh \in FaultHandler$ )  $\equiv$

**if** *faultHandlerStage*( $sl, fh$ ) = *awaitFault* **then**

**onSignal** *signalComplete* **from** *parentActivity*( $fh$ )

**in** *sl* **via** *signalChannel<sub>down</sub>* **do**

**set** *activityState* **from** *running* **to** *completing*

**forall** *child*  $\in$  *childActivities*( $fh$ ) **do**

INITDPE( $sl, fh, child$ )

**otherwise**

WAITFORFAULT( $sl, fh$ )

**if** *faultHandlerStage*( $sl, fh$ ) = *stopScope* **then**

FAULTHANDLERSTOPSCOPE( $sl, fh$ )

**if** *faultHandlerStage*( $sl, fh$ ) = *executeCatch* **then**

EXECUTECATCH( $sl, fh$ )

WAITFORFAULT (R9.3.2-69)

( $sl \in SubInstance, fh \in FaultHandler$ )  $\equiv$

**select** *fault*  $\in$  *faultChannel<sub>down</sub>*( $sl, parentActivity$ ( $fh$ ),  $fh$ ) **in**

$faultHandlerStage(sl, fh) := stopScope$   
 $faultHandlerCaughtFault(sl, fh) := fault$   
 $CHOOSECATCHNODE(sl, fh, fault)$

derived functions ( $\mathcal{D}$  B.6.2-1)

$\mathcal{D} faultHasData : Fault \rightarrow \{true, false\}$

$fault \mapsto \begin{cases} true, & msgMsgType(fault) \neq undef \\ false, & else \end{cases}$

$\mathcal{D} messageTypeEquals_{catch, fault} : CatchNode \times Fault \rightarrow \{true, false\}$

$(catch, fault) \mapsto \begin{cases} true, & var = catchNodeVariable(catch) \\ & \wedge msgMsgType(fault) = variableMsgType(var) \\ false, & else \end{cases}$

CHOOSECATCHNODE (RB.6.2-145)

$(sl \in SubInstance, fh \in FaultHandler, fault \in Fault) \equiv$

**if**  $faultHasData(fault) = false$  **then**  
 $CHOOSECATCHNODE_{noVar}(sl, fh, fault)$   
**else**  
 $CHOOSECATCHNODE_{Var}(sl, fh, fault)$

referenced in WAITFORFAULT (R9.3.2-69),

references  $CHOOSECATCHNODE_{noVar}$  (RB.6.2-146),  $CHOOSECATCHNODE_{Var}$  (RB.6.2-147)

CHOOSECATCHNODE<sub>noVar</sub> (RB.6.2-146)

$(sl \in SubInstance, fh \in FaultHandler, fault \in Fault) \equiv$

**select**  $catch \in faultHandlerCatchNodes(fh)$   
**where**  $catchNodeFaultName(catch) = faultName(fault)$  **in**  
 $REMEMBERCATCHNODE(sl, fh, catch, fault)$   
**ifnone**  
 $CHOOSECATCHNODE_{default}(sl, fh, fault)$

referenced in CHOOSECATCHNODE (RB.6.2-145),

references  $REMEMBERCATCHNODE$  (RB.6.2-149),  $CHOOSECATCHNODE_{default}$  (RB.6.2-148)

CHOOSECATCHNODE<sub>Var</sub> (RB.6.2-147)

$(sl \in SubInstance, fh \in FaultHandler, fault \in Fault) \equiv$

```

select catch  $\in$  faultHandlerCatchNodes(fh)
  where catchNodeFaultName(catch) = faultName(fault)
     $\wedge$  messageTypeEqualscatch,fault(catch, fault)
  REMEMBERCATCHNODE(sl, fh, catch, fault)
ifnone
  select catch  $\in$  faultHandlerCatchNodes(fh)
    where messageTypeEqualscatch,fault(catch, fault)
    REMEMBERCATCHNODE(sl, fh, catch, fault)
  ifnone
    CHOOSECATCHNODEdefault(sl, fh, fault)

```

referenced in CHOOSECATCHNODE (RB.6.2-145),  
 references REMEMBERCATCHNODE (RB.6.2-149), CHOOSECATCHNODE<sub>default</sub> (RB.6.2-148)

CHOOSECATCHNODE<sub>default</sub> (RB.6.2-148)

$(sl \in SubInstance, fh \in FaultHandler, fault \in Fault) \equiv$

```

select catch  $\in$  faultHandlerCatchNodes(fh)
  where catchNodeFaultName(catch) = catchAll in
  REMEMBERCATCHNODE(sl, fh, catch, fault)
ifnone
  faultHandlerChosenActivity(sl, fh) := undef

```

referenced in CHOOSECATCHNODE<sub>Var</sub> (RB.6.2-147), CHOOSECATCHNODE<sub>noVar</sub> (RB.6.2-146),  
 references REMEMBERCATCHNODE (RB.6.2-149)

REMEMBERCATCHNODE (RB.6.2-149)

$(sl \in SubInstance, fh \in FaultHandler, catch \in CatchNode, fault \in Fault) \equiv$

```

faultHandlerChosenActivity(sl, fh) := catchNodeActivity(catch)
if faultHasData(fault) = true
   $\wedge$  catchNodeVariable(catch)  $\neq$  undef then
  COPYMSGTOVARIABLE(sl, fault, catchNodeVariable(catch))

```

referenced in CHOOSECATCHNODE<sub>Var</sub> (RB.6.2-147), CHOOSECATCHNODE<sub>noVar</sub> (RB.6.2-146),  
 CHOOSECATCHNODE<sub>default</sub> (RB.6.2-148),  
 references COPYMSGTOVARIABLE (RB.3.2-110)

FAULTHANDLERSTOPSCOPE (R9.3.2-70)

( $sl \in SubInstance, fh \in FaultHandler$ )  $\equiv$   
**if**  $fHStopMode(sl, fh) = sendingStop$  **then**  
  **signal**  $signalStop$  **to**  $parentActivity(fh)$  **in**  $sl$  **via**  $signalChannel_{up}$   
   $fHStopMode(sl, fh) := awaitingStopped$   
**if**  $fHStopMode(sl, fh) = awaitingStopped$  **then**  
  **onSignal**  $signalStopped$  **from**  $parentActivity(fh)$   
  **in**  $sl$  **via**  $signalChannel_{down}$  **do**  
   $faultHandlerStage(sl, fh) := executeCatch$

EXECUTE CATCH (R9.3.2-72)

( $sl \in SubInstance, fh \in FaultHandler$ )  $\equiv$   
**if**  $fHExecuteMode(sl, fh) = sendingEnable$  **then**  
  **let**  $act = faultHandlerChosenActivity(sl, fh)$  **in**  
  **if**  $act \neq undef$  **then**  
    **signal**  $signalEnable$  **to**  $act$  **in**  $sl$  **via**  $signalChannel_{down}$   
    **add**  $(sl, act)$  **to**  $activityRunningChilds(sl, fh)$   
     $fHExecuteMode(sl, fh) := awaitingCompleted$   
  **else**  
     $RETHROWFAULT(sl, fh)$   
  **forall**  $child$  **in**  $childActivities(fh) \setminus \{act\}$  **do**  
     $INITDPE(sl, fh, child)$   
  **if**  $fHExecuteMode(sl, fh) = awaitingCompleted$  **then**  
     $RUNACTIVITY_{structured}(sl, fh)$

RETHROWFAULT (R9.3.2-71)

( $sl \in SubInstance, fh \in FaultHandler$ )  $\equiv$   
  **let**  $fault = faultHandlerCaughtFault(sl, fh)$  **in**  
  **add**  $fault$  **to**  $faultChannel_{up}(sl, fh, parentActivity(fh))$   
  **set**  $activityState$  **from**  $running$  **to**  $faulted$

### B.6.3 Event Handler

EXECUTE EVENT HANDLER (R9.5.2-76)

( $sl \in SubInstance, eh \in EventHandler$ )  $\equiv$   
  **if**  $eventType(eventHandlerEvent(eh)) = onAlarm$  **then**  
     $EXECUTEEVENTHANDLER_{onAlarm}(sl, eh)$   
  **if**  $eventType(eventHandlerEvent(eh)) = onMsg$  **then**  
     $EXECUTEEVENTHANDLER_{onMessage}(sl, eh)$

### B.6.4 OnAlarm Event Handler

EXECUTEEVENTHANDLER<sub>onAlarm</sub> (RB.6.4-150)  
 $(sl \in SubInstance, act \in Activity) \equiv$

PROPAGATETERMINATE(sl, act)  
**onSignal** *signalStop* **from** *parentActivity*(act)  
  **in** *sI* **via** *signalChannel<sub>down</sub>* **do**  
  ACTIVITYSETTOSTOPPING(sl, act)  
**otherwise**  
  **onSignal** *signalEnable* **from** *parentActivity*(act)  
  **in** *sl* **via** *signalChannel<sub>down</sub>* **do**  
  **set** *activityState* **from** *disabled* **to** *enabled*  
  PROPAGATEFAULTSACTIVITY(sl, act)  
  **if** *activityState*(sl, act) = *enabled* **then**  
   STARTEVENTHANDLER<sub>onAlarm</sub>(sl, act)  
  **if** *activityState*(sl, act) = *running* **then**  
   RUNEVENTHANDLER<sub>onAlarm</sub>(sl, act)  
  **if** *activityState*(sl, act) = *completing* **then**  
   COMPLETEACTIVITY(sl, act)  
  **if** *activityState*(sl, act) = *stopping* **then**  
   STOPACTIVITY<sub>structured</sub>(sl, act)

refines EXECUTEACTIVITY<sub>pattern,structured</sub> (R8.1.6-44),    STARTEVENTHANDLER<sub>onAlarm</sub>  
(R9.6.1-77), RUNEVENTHANDLER<sub>onAlarm</sub> (R9.6.2-78)

STARTEVENTHANDLER<sub>onAlarm</sub> (R9.6.1-77)  
 $(sl \in SubInstance, eh \in EventHandler) \equiv$

**set** *activityState* **from** *enabled* **to** *running*  
INITALARMEVENT(sl, *eventHandlerEvent*(eh))

RUNEVENTHANDLER<sub>onAlarm</sub> (R9.6.2-78)  
 $(sl \in SubInstance, eh \in EventHandler) \equiv$

**if** *eventHandlerStage*(sl, eh) = *awaitEvent* **then**  
  **onSignal** *signalComplete* **from** *parentActivity*(sl, eh)  
  **in** *sl* **via** *signalChannel<sub>down</sub>* **do**  
  **set** *activityState* **from** *running* **to** *completing*  
**otherwise**  
  **if** *eventOccurred*(sl, *eventHandlerEvent*(eh)) = *true* **then**  
   HANDLEEVENT<sub>onAlarm</sub>(sl, eh)  
**if** *eventHandlerStage*(sl, eh) = *handleEvent* **then**  
  FINISHEVENT<sub>onAlarm</sub>(sl, eh)

HANDLEEVENT<sub>onAlarm</sub> (R9.6.2-79)

$(sl \in SubInstance, eh \in EventHandler) \equiv$   
**signal** *signalEnable* **to** *childActivity(eh)* **in** *sl* **via** *signalChannel<sub>down</sub>*  
**set** *eventHandlerStage* **from** *awaitEvent* **to** *handleEvent*  
**add** (*sl, childActivity(eh)*) **to** *activityRunningChilds(sl, eh)*

$\frac{FINISHEVENT_{onAlarm}}{(sl \in SubInstance, eh \in EventHandler) \equiv}$  (R9.6.2-80)  
 RUNACTIVITY<sub>structured</sub>(*sl, eh*)

### B.6.5 OnMessage Event Handler

$\frac{EXECUTEEVENTHANDLER_{onMessage}}{(sl \in SubInstance, eh \in EventHandler) \equiv}$  (RB.6.5-151)

PROPAGATETERMINATE(*sl, eh*)  
**onSignal** *signalStop* **from** *parentActivity(eh)*  
     **in** *sI* **via** *signalChannel<sub>down</sub>* **do**  
         ACTIVITYSETTOSTOPPING(*sl, eh*)  
**otherwise**  
     **onSignal** *signalEnable* **from** *parentActivity(eh)*  
         **in** *sl* **via** *signalChannel<sub>down</sub>* **do**  
             **set** *activityState* **from** *disabled* **to** *enabled*  
         PROPAGATEFAULTSACTIVITY(*sl, eh*)  
         **if** *activityState(sl, eh) = enabled* **then**  
             STARTACTIVITY(*sl, eh*)  
         **if** *activityState(sl, eh) = running* **then**  
             RUNEVENTHANDLER<sub>onMessage</sub>(*sl, eh*)  
         **if** *activityState(sl, eh) = completing* **then**  
             COMPLETEEVENTHANDLER<sub>onMessage</sub>(*sl, eh*)  
         **if** *activityState(sl, eh) = stopping* **then**  
             STOPACTIVITY<sub>structured</sub>(*sl, eh*)

refines EXECUTEACTIVITY<sub>pattern,structured</sub> (R8.1.6-44), referencing  
 RUNEVENTHANDLER<sub>onMessage</sub> (R9.7.2-81), COMPLETEEVENTHANDLER<sub>onMessage</sub>  
 (RB.6.5-154),

$\frac{RUNEVENTHANDLER_{onMessage}}{(sl \in SubInstance, eh \in EventHandler) \equiv}$  (R9.7.2-81)

**onSignal** *signalComplete* **from** *parentActivity(sl, eh)*  
     **in** *sl* **via** *signalChannel<sub>down</sub>* **do**  
         **set** *activityState* **from** *running* **to** *completing*  
**otherwise**

```

if eventOccurred(sl, event) = true then
    HANDLEEVENTonMsg(sl, eh, eventHandlerEvent(eh))
    FINISHEVENTonMsg(sl, eh)
    
```

HANDLEEVENT<sub>onMsg</sub> (R9.7.2-82)  
 (sl ∈ *SubInstance*, eh ∈ *EventHandler*, event ∈ *EventDescriptor*) ≡

```

let portdescr = eventPortDescriptor(event) in
    AWAITANDRECEIVECORRELATINGMESSAGEevent(sl, eh, portdescr)
    
```

AWAITANDRECEIVECORRELATINGMESSAGE<sub>event</sub> (RB.6.5-152)  
 (sl ∈ *SubInstance*, eh ∈ *Activity*, portdescr ∈ *PortDescriptor*) ≡

```

let partnerL = portPartnerLink(portdescr),
    pT = portPortType(portdescr),
    op = portOperation(portdescr),
    pl = processInstance(sl) in
    select msg ∈ localPortin(pl, partnerL, pT, op) in
        let CS = portCorrelationSet(portdescr) in
            if correlationSatisfied(sl, eh, msg, CS) then
                remove msg from localPortin(pl, partnerL, pT, op)
                let slchild = new(SubInstance) in
                    RECEIVEMESSAGEpattern(slchild, eh, portdescr, msg)
                    RUNEVENTHANDLERCHILDACTIVITY(sl, slchild, eh, event)
                    EXTENDSUBINSTANCETREE(sl, slchild)
            else
                THROW(sl, eh, correlationViolation)
    
```

refines AWAITANDRECEIVECORRELATINGMESSAGE<sub>pattern</sub> (R6.5.3-19),  
 referenced in HANDLEEVENT<sub>onMsg</sub> (R9.7.2-82),  
 references RECEIVEMESSAGE<sub>pattern</sub> (R6.5.3-20), RUNEVENTHANDLERCHILDACTIVITY  
 (R9.7.2-83), THROW (R5.5.1-10)

RUNEVENTHANDLERCHILDACTIVITY (R9.7.2-83)  
 (sl ∈ *SubInstance*, sl<sub>child</sub> ∈ *SubInstance*,  
 eh ∈ *EventHandler*, event ∈ *EventDescriptor*) ≡

```

signal signalEnable to eventActivity(event) in slchild via signalChanneldown
add (slchild, eventActivity(event)) to ehRunningInstances(sl, eh)
    
```

FINISHEVENT<sub>onMsg</sub> (RB.6.5-153)  
 (sl ∈ *SubInstance*, eh ∈ *EventHandler*) ≡

**forall** ( $sl_{child}, child$ )  $\in$   $activityRunningChlds(sl, act)$  **do**  
     **onSignal**  $signalCompleted$  **from**  $child$  **in**  $sl_{child}$  **via**  $signalChannel_{up}$  **do**  
         **remove** ( $sl_{child}, child$ ) **from**  $activityRunningChlds(sl, act)$

refines  $RUNACTIVITY_{pattern}$  (R8.1.2-42)

$COMPLETEEVENTHANDLER_{onMessage}$  (RB.6.5-154)

$(sl \in SubInstance, eh \in EventHandler) \equiv$

**if**  $activityRunningChlds(sl, act) = \emptyset$  **then**  
     **set**  $activityState$  **from**  $completing$  **to**  $completed$   
     **signal**  $signalCompleted$  **to**  $parentActivity(eh)$  **in**  $sl$  **via**  $signalChannel_{up}$   
**else**  
     **forall** ( $sl_{child}, child$ )  $\in$   $activityRunningChlds(sl, act)$  **do**  
         **onSignal**  $signalCompleted$  **from**  $child$  **in**  $sl_{child}$  **via**  $signalChannel_{up}$  **do**  
             **remove** ( $sl_{child}, child$ ) **from**  $activityRunningChlds(sl, act)$

refines  $RUNACTIVITY_{pattern}$  (R8.1.2-42)

## B.7 Rules and Functions for the Compensation Handling Mechanism

### B.7.1 Scope supporting Compensation Handling

#### Preparing Scope for Compensation

INSTALLCOMPENSATIONHANDLER (R10.1.2-84)  
 $(sl \in SubInstance, scope \in Scope) \equiv$

REGISTERCOMPENSATIONMODULE( $sl, scope$ )  
 PREPARESCOPEFORCOMPENSATION( $sl, scope$ )

REGISTERCOMPENSATIONMODULE (R10.3.1-93)  
 $(sl \in SubInstance, scope \in Scope) \equiv$

PUSHTOCOMPENSATIONSTACK( $sl, scope$ )

PUSHTOCOMPENSATIONSTACK (R10.3.1-94)  
 $(sl \in SubInstance, scope \in Scope) \equiv$

let  $scope' = enclosingScope(scope)$ ,  
 $sl' = currentSubInstance_{fromInner}(sl, scope)$  in  
 add ( $sl, scope$ ) to  $scopeCompensationModules(sl', scope')$

PREPARESCOPEFORCOMPENSATION (R10.3.1-95)  
 $(sl \in SubInstance, scope \in Scope) \equiv$

set  $scopeMode$  from *positive* to *compensate*

#### Behaviour in the Case of Compensation

EXECUTESCOPE<sub>compensationHandling</sub> (R10.3.3-98)  
 $(sl \in SubInstance, scope \in Scope) \equiv$

EXECUTESCOPE( $sl, scope$ )  
 if  $scopeMode(sl, scope) = compensate$  then  
 EXECUTESCOPE<sub>compensate</sub>( $sl, scope$ )

EXECUTESCOPE<sub>compensate</sub> (RB.7.1-155)  
 $(sl \in SubInstance, scope \in Scope) \equiv$

if  $compensationPort_{in,scope}(sl, scope) \neq \emptyset$  then  
 HANDLECOMPENSATIONCALL( $sl, scope$ )  
 onSignal  $signalCompleted$  from  $scopeCompensationHandler(scope)$   
 in  $sl$  via  $signalChannel_{up}$  do  
 CONFIRMCOMPENSATION( $sl, scope$ )  
 if  $faultChannel_{up}(sl, scopeCompensationHandler(scope), scope) \neq \emptyset$  then  
 PROPAGATEFAULTSFROMCH( $sl, scope$ )

refines EXECUTESCOPE<sub>compensate,pattern</sub> (R10.1.2-85),  
 referenced in EXECUTESCOPE<sub>compensationHandling</sub> (R10.3.3-98), references  
 HANDLECOMPENSATIONCALL (RB.7.1-156)

HANDLECOMPENSATIONCALL (RB.7.1-156)

( $sl \in SubInstance, scope \in Scope$ )  $\equiv$

**if**  $scopeCHcalled(sl, scope) = false$  **then**  
   **choose**  $compensationCall \in compensationPort_{in,scope}(sl, scope)$  **in**  
     **remove**  $compensationCall$  **from**  $compensationPort_{in,scope}(sl, scope)$   
      $scopeCompensationCall(sl, scope) := compensationCall$   
     **signal**  $signalEnable$  **to**  $scopeCompensationHandler(scope)$  **in**  $sl$  **via**  $signalChannel_{down}$   
      $scopeCHcalled(sl, scope) := true$   
**else**  
   **let**  $fault = new(Fault)$  **in**  
      $faultName(fault) := repeatedCompensation$   
     NOTIFYFAULTEDCOMPENSATION( $sl, scope, scopeCompensationCall(sl, scope), fault$ )

refines HANDLECOMPENSATIONCALL<sub>pattern</sub> (R10.1.2-86),  
 referenced in EXECUTESCOPE<sub>compensate</sub> (RB.7.1-155)

NOTIFYFAULTEDCOMPENSATION (R10.3.2-97)

( $sl \in SubInstance, scope \in Scope, cc \in CompensationCall, fault$ )  $\equiv$

**let**  $act = compensationCaller(cc),$   
    $sl_{act} = compensationCallerSI(cc)$  **in**  
   **add**  $fault$  **to**  $compensationChannel_{fault}(sl, scope, sl_{act}, act)$

CONFIRMCOMPENSATION (R10.1.2-87)

( $sl \in SubInstance, scope \in Scope$ )  $\equiv$

NOTIFYCOMPLETEDCOMPENSATION( $sl, scope, scopeCompensationCall(sl, scope)$ )

NOTIFYCOMPLETEDCOMPENSATION (R10.3.2-96)

( $sl \in SubInstance, scope \in Scope, cc \in CompensationCall$ )  $\equiv$

**let**  $act = compensationCaller(cc),$   
    $sl_{act} = compensationCallerSI(cc)$  **in**  
   **add**  $signalCompleted$  **to**  $compensationChannel_{confirm}(sl, scope, sl_{act}, act)$

PROPAGATEFAULTSFROMCH (R10.1.2-88)

( $sl \in SubInstance, scope \in Scope$ )  $\equiv$

**let**  $ch = scopeCompensationHandler(scope),$   
    $cc = scopeCompensationCall(sl, scope)$  **in**  
   **forall**  $fault$  **in**  $faultChannel_{up}(sl, ch, scope)$  **do**  
     **remove**  $fault$  **from**  $faultChannel_{up}(sl, ch, scope)$   
     NOTIFYFAULTEDCOMPENSATION( $sl, scope, cc, fault$ )

## B.7.2 Compensation Handler

EXECUTECOMPENSATIONHANDLER (RB.7.2-157)

( $sl \in SubInstance, ch \in CompensationHandler$ )  $\equiv$

PROPAGATEDPE( $sl, ch$ )

PROPAGATETERMINATE( $sl, ch$ )

**onSignal** *signalStop* **from** *parentActivity*( $ch$ )

**in**  $sl$  **via** *signalChannel<sub>down</sub>* **do**

ACTIVITYSETTOSTOPPING( $sl, ch$ )

**otherwise**

**onSignal** *signalEnable* **from** *parentActivity*( $ch$ )

**in**  $sl$  **via** *signalChannel<sub>down</sub>* **do**

**set** *activityState* **from** *disabled* **to** *enabled*

PROPAGATEFAULTSACTIVITY( $sl, ch$ )

**if** *activityState*( $sl, ch$ ) = *enabled* **then**

STARTCOMPENSATIONHANDLER( $sl, ch$ )

**if** *activityState*( $sl, ch$ ) = *running* **then**

RUNACTIVITY<sub>structured</sub>( $sl, ch$ )

**if** *activityState*( $sl, ch$ ) = *completing* **then**

COMPLETEACTIVITY( $sl, ch$ )

**if** *activityState*( $sl, ch$ ) = *stopping* **then**

STOPACTIVITY<sub>structured</sub>( $sl, ch$ )

refines EXECUTEACTIVITY<sub>pattern,structured</sub> (R8.1.6-44)

referenced in EXECUTEACTIVITY<sub>structured</sub> (RB.2.2-106), references

STARTCOMPENSATIONHANDLER (R10.1.3-89)

STARTCOMPENSATIONHANDLER (R10.1.3-89)

( $sl \in SubInstance, ch \in CompensationHandler$ )  $\equiv$

**set** *activityState* **from** *enabled* **to** *running*

**add** ( $sl, childActivity(ch)$ ) **to** *activityRunningChilds*( $sl, ch$ )

**signal** *signalEnable* **to** *childActivity*( $ch$ ) **in**  $sl$  **via** *signalChannel<sub>down</sub>*

## B.7.3 Compensate

EXECUTECOMPENSATE (RB.7.3-158)

( $sl \in SubInstance, comp \in Compensate$ )  $\equiv$

PROPAGATEDPE( $sl, comp$ )

PROPAGATETERMINATE<sub>basic</sub>( $sl, comp$ )

```

onSignal signalEnable from parentActivity(comp)
  in sI via signalChanneldown do
    set activityState from disabled to enabled

  if activityState(sl, comp) = enabled then
    STARTCOMPENSATE(sl, comp)
  if activityState(sl, comp) = running then
    RUNCOMPENSATE(sl, comp)
  if activityState(sl, comp) = completing then
    COMPLETEACTIVITY(sl, comp)

```

refines EXECUTEACTIVITY<sub>pattern,basic</sub> (R7.1.3-28),  
 referenced in EXECUTEACTIVITY<sub>basic</sub> (RB.2.2-105), references STARTCOMPENSATE  
 (RB.7.3-159), RUNCOMPENSATE (RB.7.3-160)

STARTCOMPENSATE (RB.7.3-159)

(*sl* ∈ *SubInstance*, *comp* ∈ *Compensate*) ≡

```

let scope = enclosingScope(comp),
  slscope = currentSubInstancefromInner(sl, scope) in
  if compensateScope(scope) = undef,
    ∧ defaultOrderCompensationAvailable(slscope, scope) = false then
    set activityState from enabled to completing
  else
    set activityState from enabled to running
    POPCOMPENSATIONSTACK(sl, comp)
    if compensateScope(comp) ≠ undef then
      defaultOrderCompensationAvailable(slscope, scope) := false

```

refines STARTCOMPENSATE (RB.7.3-159)<sub>pattern</sub>,  
 referenced in EXECUTECOMPENSATE (RB.7.3-158),  
 references POPCOMPENSATIONSTACK (R10.4.1-99)

POPCOMPENSATIONSTACK (R10.4.1-99)

(*sl* ∈ *SubInstance*, *comp* ∈ *Compensate*) ≡

```

let stack = compensateCompensationModules(sl, comp),
  finished = compensateFinishedModules(sl, comp),
  currentTOS = compensateTOS(sl, comp) in
  let nextModules = nextCompensationModules(stack \ (finished ∪ currentTOS)) in
    compensateTOS(sl, comp) := nextModules
    compensateFinishedModules(sl, comp) := finished ∪ currentTOS

```

RUNCOMPENSATE (RB.7.3-160)

$(sl \in SubInstance, comp \in Compensate) \equiv$

```

let TOS = compensateTOS(sl, comp) in
  if TOS =  $\emptyset$  then
    set activityState from running to completing
  else
    if compensationMode(sl, comp) = callCompensation then
      forall (sl', scope')  $\in$  TOS do
        CALLCOMPENSATION(sl, comp, sl', scope')
        compensationMode(sl, comp) := awaitingCompensated
    if compensationMode(sl, comp) = awaitingCompensated then
      if  $\exists$ (sl', scope')  $\in$  TOS
        compensationFailed(sl, comp, sl', scope') = true then
          COMPENSATERETHROWFAULT(sl, comp)
          set activityState from running to faulted
        else if  $\forall$  (sl', scope')  $\in$  TOS
          compensationCompleted(sl, comp, sl', scope') = true then
            POPCOMPENSATIONSTACK(sl, comp)
            compensationMode(sl, comp) := callCompensation
    
```

refines RUNCOMPENSATE<sub>pattern</sub> (R10.1.4-92),  
 referenced in EXECUTECOMPENSATE (RB.7.3-158),  
 references CALLCOMPENSATION (R10.4.3-100), COMPENSATERETHROWFAULT (RB.7.3-161),  
 POPCOMPENSATIONSTACK (R10.4.1-99)

CALLCOMPENSATION (R10.4.3-100)

$(sl \in SubInstance, comp \in CompensationCaller, sl' \in SubInstance, scope' \in Scope) \equiv$

```

let cs = new(CompensationSignal) in
  compensationCallerSI(cs) = sl
  compensationCaller(cs) = comp
  add cs to compensationPortin,scope(sl', scope')
    
```

COMPENSATERETHROWFAULT (RB.7.3-161)

$(sl \in SubInstance, comp \in Compensate) \equiv$

```

forall (sl', scope')  $\in$  compensateTOS(sl, comp) do
  forall fault  $\in$  compensationChannelfault(sl', scope', sl, comp) do
    remove fault from compensationChannelfault(sl', scope', sl, comp)
    add fault to faultChannelup(sl, act, parentActivity(act))
  set activityState from running to faulted
    
```

refines PROPAGATEFAULTSACTIVITY (R5.5.2-11),  
 referenced in RUNCOMPENSATE (RB.7.3-160)



## Bibliography

- [ABC<sup>+</sup>04] ASKARY, A. ; BLOCH, B. ; CURBERA, F. ; GOLAND, Y. ; KARTHA, N. ; LIU, C.K. ; THATTE, S. ; YENDLURI, P. ; YIU, A.: Business Process Execution Language for Web Services Version 1.1, Working Draft 01 / OASIS. 08 Sep 2004. – Specification. <http://www.oasis-open.org/committees/download.php/9094/wsbpel-specification-draft-Sept-08-2004.html>
- [ASMa] Die Webseite zu AsmGofer: <http://www.tydo.de/AsmGofer/>
- [Asmb] Die Webseite zu AsmL: <http://www.research.microsoft.com/foundations/asml/>
- [BFN05] BUTLER, M. ; FERREIRA, C. ; NG, M.Y.: Precise Modelling of Compensating Business Transactions and its Application to BPEL. In: *Journal of Universal Computer Science* 11 (2005), Mai, Nr. 5, S. 712–743. – [http://www.jucs.org/jucs\\_11\\_5/precise\\_modelling\\_of\\_compensating](http://www.jucs.org/jucs_11_5/precise_modelling_of_compensating)
- [BGM95] BÖRGER, E. ; GLÄSSER, U. ; MÜLLER, W.: Formal Definition of an Abstract VHDL'93 Simulator by EA-Machines. In: DELGADO KLOOS, C. (Hrsg.) ; BREUER, P. T. (Hrsg.): *Formal Semantics for VHDL*. Kluwer Academic Publishers, 1995, S. 107–139
- [BK05] VAN BREUGEL, Franck ; KOSHINKA, Mariya: Dead-Path-Elimination in BPEL4WS. In: *Proceedings of the 5th International Conference on Application of Concurrency to System Design*. St Malo, IEEE, June 2005
- [Bo95] BÖRGER, E.: Why use Evolving Algebras for Hardware and Software Engineering? In: BARTOSEK, M. (Hrsg.) ; STANDEK, J. (Hrsg.) ; WIEDERMANN, J. (Hrsg.): *SOFSEM'95, 22nd Seminar on Current Trends in Theory & Practice of Informatics* Bd. LNCS 1012, Springer-Verlag, 1995, S. 235–271
- [BS03] BÖRGER, E. ; STÄRK, R.: *Abstract State Machines – A Method for High-Level System Design and Analysis*. Springer-Verlag, 2003. – ISBN 3–540–00702–4
- [CB<sup>+</sup>03] CURBERA, A. ; BOX, D. [u. a.]: Web Services Addressing (WS-Addressing) / BEA, IBM, Microsoft. 2003. – Specification. <http://msdn.microsoft.com/ws/2003/03/ws-addressing/>
- [CGK<sup>+</sup>03] CURBERA, F. ; GOLAND, Y. ; KLEIN, J. ; LEYMAN, F. ; ROLLER, D. ; WEERAWARANA, S.: Business Process Execution Language for Web Services Version 1.1 / BEA Systems, IBM, Microsoft, SAP, Siebel. 05 May 2003. – Specification. <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbiz2k2/html/bpel1-1.asp>

- [EAA<sup>+</sup>04] ENDREI, M. ; ANG, J. ; ARSANJANI, A. ; CHUA, S. ; COMTE, P. ; KROGDAHL, P. ; LUO, M. ; NEWLING, T.: *Patterns: Service-oriented Architecture and Web Services*. IBM Redbooks. IBM, April 2004. – ISBN 073845317X
- [EGG<sup>+</sup>01] ESCHBACH, R. ; GLÄSSER, U. ; GOTZHEIN, R. ; VON LÖVIS, M. ; PRINZ, A.: Formal Definition of SDL-2000 - Compiling and Running SDL Specifications as ASM Models. In: *Journal of Universal Computer Science* 7 (2001), November, Nr. 11, S. 1024–1049
- [Fah04] FAHLAND, Dirk: Ein Ansatz einer formalen Semantik der Business Process Execution Language for Web Services mit Abstract State Machines / Humboldt-Universität zu Berlin. 2004. – Studienarbeit
- [Far04] FARAHBOD, Roozbeh: Extending and Refining an Abstract Operational Semantics of the Web Services Architecture for the Business Process Execution Language / Simon Fraser University, Burnaby B.C. Canada. 2004. – Master Thesis
- [FBS04] FU, Xiang ; BULTAN, Tevfik ; SU, Jianwen: Analysis of interacting BPEL web services. In: *WWW '04: Proceedings of the 13th international conference on World Wide Web*. New York, NY, USA : ACM Press, 2004. – ISBN 1–58113–844–X, S. 621–630
- [Fer04a] FERRARA, Andrea: Web services: a process algebra approach. In: *ICSOC '04: Proceedings of the 2nd international conference on Service oriented computing*. New York, NY, USA : ACM Press, 2004. – ISBN 1–58113–871–7, S. 242–251
- [Fer04b] FERRARA, Andrea: Web services: a process algebra approach / DIS - Università di Roma “La Sapienza”. 2004. – Technical Report. 17-04
- [FGV04a] FARAHBOD, Roozbeh ; GLÄSSER, Uwe ; VAJIHOLLAHI, Mona: Abstract Operational Semantics of the Business Process Execution Language for Web Services / Simon Fraser University, Burnaby B.C. Canada. 2004. – Technical Report. SFU-CMPT-TR-2004-03
- [FGV04b] FARAHBOD, Roozbeh ; GLÄSSER, Uwe ; VAJIHOLLAHI, Mona: Specification and Validation of the Business Process Execution Language for Web Services. In: *Abstract State Machines*, 2004, S. 78–94
- [FGV05] FARAHBOD, Roozbeh ; GLÄSSER, Uwe ; VAJIHOLLAHI, Mona: Abstract Operational Semantics of the Business Process Execution Language for Web Services / Simon Fraser University, Burnaby B.C. Canada. 2005. – Technical Report. SFU-CMPT-TR-2005-04
- [GT01] GUREVICH, Y. ; TILLMANN, N.: Partial Updates: Exploration. In: *Journal of Universal Computer Science* 7 (2001), November, Nr. 11, S. 917–951. – [http://www.jucs.org/jucs\\_7\\_11/partial\\_updates\\_exploration](http://www.jucs.org/jucs_7_11/partial_updates_exploration)

- 
- [Gur95] GUREVICH, Y.: Evolving Algebras 1993: Lipari Guide. In: BÖRGER, E. (Hrsg.): *Specification and Validation Methods*. Oxford University Press, 1995, S. 9–36
- [Gur97] GUREVICH, Y.: May 1997 Draft of the ASM Guide / University of Michigan EECS Department. 1997. – Forschungsbericht. CSE-TR-336-97
- [HW02] HUGGINS, J. ; WALLACE, C. *An Abstract State Machine primer*. 2002
- [KB03] KOSHINKA, Mariya ; VAN BREUGEL, Franck: Verification of Business Processes for Web Services / York University. 2003. – Technical Report. CS-2003-11
- [Kre01] KREGER, H.: Web Services Conceptual Architecture (WSCA 1.0) / IBM Software Group. 2001. – Whitepaper
- [Ley01] LEYMANN, F.: Web Service Flow Language (WSFL 1.0) / IBM Software Group. May 2001. – Specification
- [OAB<sup>+</sup>05] OUYANG, C. ; VAN DER AALST, W.M.P. ; BREUTEL, S. ; DUMAS, M. ; TER HOFSTEDÉ, A.H.M. ; VERBEEK, H.M.W.: Formal Semantics and Analysis of Control Flow in WS-BPEL. / BPMcenter.org. 2005. – Technical Report. BPM Report BPM-05-13
- [SS04] SCHMIDT, Karsten ; STAHL, Christian: A Petri net Semantic for BPEL4WS - Validation and Application. In: KINDLER, Ekkart (Hrsg.): *Proceedings of the 11th Workshop on Algorithms and Tools for Petri Nets (AWPN'04)*, Universität Paderborn, october 2004, S. 1–6
- [SSB01] STÄRK, R. ; SCHMID, J. ; BÖRGER, E.: *Java and the Java Virtual Machine: Definition, Verification, Validation*. Springer-Verlag, 2001
- [Sta05] STAHL, Christian: A Petri Net Semantics for BPEL / Humboldt-Universität zu Berlin. 2005 ( 188). – Informatik-Berichte
- [Tha01] THATTE, S.: XLANG – Web Service for Business Process Design / Microsoft Corporation. 2001. – Specification
- [VBS04] VIDAL, Jose M. ; BUHLER, Paul ; STAHL, Christian: Multiagent Systems with Workflows. In: *IEEE Internet Computing* 8 (2004), feb, Nr. 1, S. 76–82
- [WT04] WSBPEL-TC: WSBPEL issues list. 2004. – Public Documents. Organization for the Advancement of Structured Information Standards (OASIS)