HIGH LEVEL SPECIFICATION AND VALIDATION OF THE BUSINESS PROCESS EXECUTION LANGUAGE FOR WEB SERVICES

by

Mona Vajihollahi

B.Sc., Computer Engineering, Sharif University of Technology, 2001

THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

In the School
of
Computing Science

© Mona Vajihollahi 2004
Simon Fraser University
April 2004

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

Approval

Name: **Mona Vajihollahi**

Degree: **Master of Computing Science.**

Title of Thesis: **High Level Specification and Validation of the Business Process Execution Language for Web Services**

Examining Committee:

Chair: **Dr. Joseph G. Peters**
Professor of Computing Science

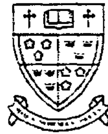
Dr. Uwe Glässer
Senior Supervisor
Associate Professor of Computing Science

Dr. Evgenia Ternovksa
Supervisor
Assistant Professor of Computing Science

Dr. David G. Mitchell
Examiner
Assistant Professor
School of Computing Science
Simon Fraser University

Date Approved: April 6. 2004

SIMON FRASER UNIVERSITY



Partial Copyright Licence

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

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

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

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

Bennett Library
Simon Fraser University
Burnaby, BC, Canada

Abstract

The Business Process Execution Language for Web Services (BPEL) is an XML based formal language for the design of networking protocols for automated business processes. Originally introduced by leading e-business vendors, including IBM and Microsoft, BPEL is now a forthcoming industrial standard as the work on the language continues at OASIS[1] within the technical committee on the Web Services Business Process Execution Language (WSBPEL TC).

We formally define an abstract executable semantics for the language in terms of a distributed abstract state machine (DASM). The DASM paradigm has proven to be a feasible, yet robust, approach for modeling architectural and programming languages and has been used as the basis for industrial standardization before.

The goal of this work is to support the design and standardization of BPEL by eliminating weak points in the language definition and validating key system attributes through experimental validation. The necessity of formalisation in the standardization process is well recognized by the OASIS WSBPEL TC and is formulated as one of the basic issues by the technical committee. *"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."*[2]

We take a hierarchical refinement approach to model the language. Starting from an abstract ground model of the core attributes of the language, we perform step-wise

[1] Organization for the Advancement of Structured Information Standards (OASIS), www.oasis-open.org

[2] Issue #42, WSBPEL Issue List, WSBPEL TC at OASIS

refinements obtaining a hierarchy of ground models at different levels of abstraction which leads to the final executable model. The executable model is then used together with a graphical visualization tool to experimentally validate the key attributes of the language through simulation of abstract machine runs.

Dedication

In loving memory of my grand fathers,

two outstanding individuals whose spirits always shine on my life.

Acknowledgements

I am deeply grateful to my senior supervisor Dr. Uwe Glässer whose knowledge and experience as well as inspiring support and encouragement truly guided me to explore new dimensions in computing science.

I would like to thank Roozbeh Farahbod, my colleague, dear friend, and loving husband both for his professional contributions in this research and for all his devotion and encouraging presence through our journey together. I am also very thankful to my parents for their continuous support and unlimited love.

Table of Contents

Approval ii
Abstract iii
Dedication v
Acknowledgements vi
Table of Contents vii
List of Figures x
List of Abbreviations and Acronyms xi
Chapter 1. Introduction 1
1.1. Motivation and Objective 2
1.2. Thesis Organization 3
Chapter 2. Business Process Execution Language for Web Services 4
2.1. Introduction 4
2.2. Overview of BPEL 5
2.3. Initial Example 6
2.4. Abstract Syntax Tree 7
2.5. Correlation 8
2.6. Activities 9
2.6.1. Basic Activities 10
2.6.2. Structured Activities 12
2.7. e-Book Store Re-visited 14
2.7.1. Partners 15
2.7.2. e-Book Store Business Process 16
2.8. Long-running Business Processes and Compensation Behaviour 19
Chapter 3. Abstract State Machines 20
3.1. Basic Abstract State Machines 20
3.1.1. Non-determinism 22
3.1.2. Parallelism 22
3.2. Distributed Abstract State Machines 23
3.2.1. Concurrency in Sample DASM models 25
3.2.2. Reactivity 28
3.2.3. Real-Time Behaviour 31
3.3. Our DASM 31

3.4. Notational Conventions 32

Chapter 4. Formalization of the BPEL Web Services Architecture 34

4.1. Overall Organization 34
4.2. BPEL Abstract Model: Overview 36
4.3. BPEL Abstract Model: Details 41
4.3.1. Basic Activities 41
4.3.2. Structured Activities 45

Chapter 5. Complete Formal Model 52

5.1. Inbox Manager 52
5.1.1. Assign Message 53
5.1.2. Pick Activity Clearance 55
5.2. Outbox Manager 56
5.3. Execute Activity 56
5.4. Receive Activity 60
5.5. Reply Activity 61
5.6. Invoke Activity 62
5.7. Terminate Activity 62
5.8. Wait Activity 62
5.9. Sequence Agent 64
5.10. Switch Agent 64
5.11. While Agent 66
5.12. Pick Agent 66
5.12.1. Pick Message Agent 68
5.12.2. Pick Alarm Agent 69
5.13. Flow Agent 70
5.13.1. Link Semantics 71

Chapter 6. Executable Model 73

6.1. Introduction to AsmL 73
6.2. The AsmL Model 74
6.2.1. Original Model 75
6.2.2. Internal Structure 76
6.2.3. Execution-Specific Additions to the ASM Model 77
6.2.4. GUI-Related extensions 78
6.2.5. Communication Model 80
6.3. Experimental Validation 81

Chapter 7. Critical Analysis of BPEL 82

7.1. Ambiguities 82
7.1.1. Correlations 82
7.1.2. Synchronous Receive/Reply 83
7.2. Loose Ends 84
7.2.1. Partners Communication 85
7.2.2. Re-Initiating a Correlation Set 85
7.3. Inconsistencies 88

Chapter 8. Conclusion and Future Work 90

Appendices 92

Appendix A. BPEL Abstract Syntax Tree 92
Appendix B. Abstract Model 97
B.1. Initial Definitions 97
B.2. Programs 100
Appendix C. Complete Formal Model 106
C.1. Initial Definitions 106
C.2. Programs 113
Appendix D. Executable Model 122
D.1. Original Model 122
D.2. Execution-Specific Additions to the ASM Model 129
D.3. GUI-Related Additions 134
D.4. Internal Structure 141

References 148

List of Figures

Figure 2-1 The structure of a BPEL process definition 5
Figure 2-2 The e-Book store business process 6
Figure 2-3 A flow activity with synchronization dependencies 14
Figure 2-4 Partners and port types of the e-Book Store business process 16
Figure 3-1 A partial ordered set of moves 24
Figure 3-2 All possible runs of the DASM of Example 3.1 26
Figure 3-3 Some segment of possible runs of Example 3.2 27
Figure 4-1 The composition of the BPEL service model and the network model 35
Figure 4-2 A three layer approach: From formal documentation to the executable model 36
Figure 4-3 High-level abstract structure of our BPEL model 36
Figure 4-4 The combination of all potential control structures of DASM activity agents at the top-level layer 39
Figure 5-1 The structure of an e-book Store business process instance in our model 60
Figure 6-1 Graphical user interface of a sample AsmL model 79

List of Abbreviations and Acronyms

ASM	Abstract State Machine
BPEL	Business Process Execution Language for Web Services
DASM	Distributed Abstract State Machine
GUI	Graphical User Interface
LRM	Language Reference Manual
OASIS	Organization for the Advancement of Structured Information Standards
SOAP	Simple Object Access Protocol
WSBPEL TC	Web Services Business Process Execution Language Technical Committee
WSDL	Web Services Description Language
XML	eXtensible Markup Language

Chapter 1. Introduction

In this thesis, we formally define an abstract operational semantics for the Business Process Execution Language for Web Services – *BPEL4WS* (or *BPEL*) [10] – in terms of a real-time *distributed abstract state machine* (DASM) model [23], [20]. Version 1.1 of the informal language description [10], henceforth called the language reference manual or LRM, is a forthcoming industrial standard proposed by the OASIS[3] Web Services Business Process Execution Language Technical Committee (WSBPEL TC) [34]. BPEL is an XML based formal language for modeling and design of the Web services orchestration and automated business processes. As such, it builds on other existing standards for the Internet and World Wide Web and, in particular, is defined on top of the service model of the Web Services Description Language (WSDL) [32]. A BPEL process and its partners are considered as abstract WSDL services that interact with each other by sending and receiving abstract messages as defined by the WSDL model for service interaction.

The *abstract state machine* (ASM) paradigm has been extensively used for formal specification of programming languages (e.g. Java [31], Prolog [4], [5]) and system modeling languages (e.g. SDL [12], [13], [18], VHDL [7], [8], SystemC [29]). The ASM formalism supports the integration of high-level modeling and analysis in the development cycle [9] which enables it to serve as a modeling basis in industrial standardization (e.g. ITU-T SDL-2000) [28].

---

[3] Organization for the Advancement of Structured Information Standards (OASIS)

1.1. Motivation and Objective

This work was mainly inspired by the successful experience from applying the asynchronous DASM model for semantic modeling to various industrial system design languages, including the ITU-T language SDL-2000 [12], [13], [18]. The resulting SDL formal semantics was officially approved by the International Telecommunication Union (ITU) as part of the SDL language definition [18].

The goal of our work is twofold. Formalization of BPEL semantics serves two main purposes, namely: (1) to eliminate deficiencies hidden in natural language descriptions, for instance, such as ambiguities, loose ends, and inconsistencies; (2) to establish a platform for experimental validation of key language attributes by making abstract operational specifications executable on real machines. For the development of BPEL, the responsible TC at OASIS has listed about one hundred basic issues. Among those, the necessity of formalization in the standardization process is well recognized as a powerful means for dealing with weak points of the LRM [15].

"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."—Issue #42, OASIS WSBPEL TC [34].

Formalization of language semantics based on informally specified requirements faces the non-trivial problem of 'turning English into mathematics'. Ideally, the formal and the informal language definition should complement each other in the endeavor to sharpen requirements into specifications. That is, the formal model provides the ultimate reference whenever the clarification of subtle language issues that are difficult to articulate in plain English requires mathematical precision. The gradual formalization of the key language attributes at different levels of abstraction and with a degree of detail and precision as needed would be certainly beneficial for practical purposes, such as industrial standardization [18].

Our definition of the abstract operational semantics presented here forms a service abstract machine and is organized into three basic layers reflecting different levels of abstraction. The top layer, called *abstract model,* provides an overview and defines the modeling framework comprehensively. The second layer, called *intermediate model*, which is the result of the first refinement step, specifies the relevant technical details and provides the complete formal model of the core constructs of the language. Finally, the third layer, called *executable model*, provides an abstract executable semantics of BPEL implemented in AsmL [1]. To this end, the service abstract machine model forms a hierarchy consisting of three DASM ground models [2], [9] obtained as the result of stepwise refinements of the abstract model. The executable model is complemented by a graphical user interface (GUI), facilitating experimental validation through simulation and animation of abstract machine runs.

1.2. Thesis Organization

The thesis provides brief introductions to BPEL and the ASM paradigm, presents the BPEL service abstract machine at different levels of abstraction, and discusses the results and possible future work. Chapter 2 introduces BPEL and describes the core aspects of the language. Chapter 3 provides an overview on abstract state machine paradigm and investigates the DASM model. Chapter 4 introduces the abstract model. In Chapter 5, the result of the first refinement step, i.e. the intermediate model, is introduced. In Chapter 6, the executable model is introduced and some results of the experimental validation are presented. Chapter 7 provides a critical analysis of BPEL based on the experience achieved through the formal modeling process. Chapter 8 concludes the thesis and discusses the possible future work.

Chapter 2. Business Process Execution Language for Web Services

2.1. Introduction

Several XML based Web standards have been introduced to define the Web services space and facilitate interoperability between a variety of Web applications, for instance, in e-business. Each of these standards targets a specific domain within the Web services space. For example, the widely used Simple Object Access Protocol (SOAP) [30] defines a standard message passing protocol, while WSDL provides a standard way of describing Web services [32].

These standards basically provide us with a structural view of Web services. They enable us to view Web services as communication endpoints which interact with each other by sending and receiving messages via a collection of *ports* associated with each of the communication endpoints. To this end, WSDL and SOAP support a stateless model of Web services.

The Business Process Execution Language for Web Services (BPEL) builds on top of WSDL (and indirectly also on SOAP) effectively introducing a stateful interaction model that allows to exchange sequences of messages between business partners (i.e. Web services). Like many other *domain specific languages (DSLs),* BPEL is designed to capture the problems in a particular application domain which is stateful interaction of Web services. Domain specific languages are usually less expressive than general-purpose languages; however they contain appropriate domain specific notations and high level domain specific abstractions that make them suitable to be used in their desired application domain [11].

In April 2003, members of OASIS, including IBM and Microsoft among other leading companies in the e-business market, formed the Web Services Business Process Execution Language Technical Committee (WSBPEL TC) [34] in order to continue work on BPEL version 1.1. As for other standardization attempts, e.g. ITU-T standards for telecommunication, standardization of BPEL is inspired by the facilitation it provides in trades and transferring technologies and is expected to increase interoperability, reliability and consumer comfort. The objective of the WSBPEL TC is to standardize the common concepts for a business process execution language that forms the technical foundation for designing and executing business processes [34].

2.2. *Overview of BPEL*

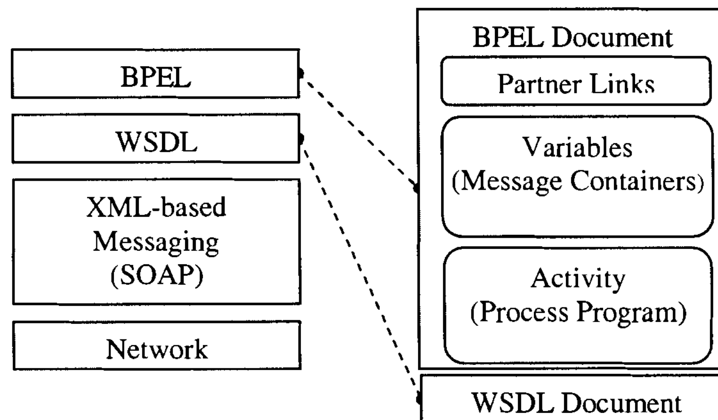


Figure 2-1 The structure of a BPEL process definition

The BPEL process model is built on top of WSDL. A BPEL process and its partners are defined as abstract WSDL services, and they use abstract messages defined by WSDL model for interaction. Figure 2-1 gives an overall view of the general structure of a BPEL business process document. A process is defined by specifying its *partners* (Web services that this process interacts with), a set of *variables* that keep the state of the process and an *activity* defining the logic behind the interactions between the process and its partners. This definition is just a template for creating business process instances. Process creation in BPEL is always implicit and is done by defining *start activities*. A *start activity* is

either a *receive* or a *pick* activity that is annotated with '*createInstance* = *yes*' causing a new process instance being created whenever a matching message is received. At least one such start activity must be defined in a template. Whenever a message arrives for a start activity, a new instance of the business process is created and starts its execution.

2.3. Initial Example

To better understand the basic structure and some fundamental concepts of BPEL, we will provide an example: a fictitious *e*-Book Store. The process of buying a book from this online store is simple. A customer first sends the order to the *e*-Book Store. The book store then sends the order to the publisher and also sends a shipping request to a shipping company. The book store then waits to receive a call-back from the shipping company containing the shipping schedule. Upon receiving that call-back, it replies back to the customer indicating the order is received and processed successfully.

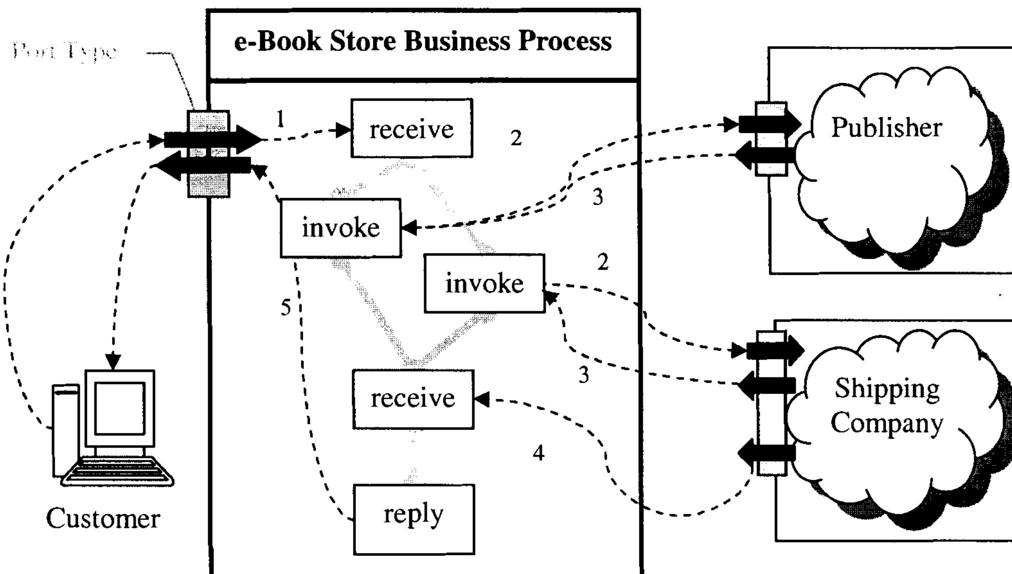


Figure 2-2 The e-Book store business process

Figure 2-2 illustrates the structure of the interaction between publisher, shipping company, and customer for the sample business process of our *e*-Book Store. A business

process interacts with other services through its ports, where each port is of a certain *port type* specifying some set of *operations*. Operations can be Input-Only, Output-Only, or Input-Output.

An abstract schema of the *e*-Book Store business process can also be found in Figure 2-2, where the numbers show the order in which the events occur. The BPEL process consists of 5 basic activities, two of which being executed concurrently (as indicated by identical order numbers annotating these two events). A process is instantiated when a message is received from the customer (interaction #1). This specific process instance is then responsible for serving the customer request. It will contact the publisher and the shipping company, at the same time, by invoking the corresponding Web services (interactions #2) and waiting for their confirmation (interactions #3). After receiving a call-back from the shipping company containing the shipping schedule and other required information (interaction #4), the e-Book store process instance will send a reply message to the costumer including all the required information (interaction #5). The customer's order request is then serviced and the transaction is completed.

2.4. Abstract Syntax Tree

A systematic approach to capture the complete structure of a BPEL process (focusing on the relevant aspects rather than syntactical details) is its representation in the form of an *abstract syntax tree* [18]. Many times during this project we had to refer to a precise and concise definition of the structure of a BPEL process. As the language definition in the LRM is currently lacking an abstract syntax, we have defined our own abstract syntax as outlined below. The complete definition of the abstract syntax tree is presented in Appendix A.

```
ProcessDef          ::=     ProcessName
                            ProcessAttributes
                            PartnerLinks?
                            Partners?
                            Variables?
                            CorrelationSets?
                            FaultHandlers?
                            CompensationHandler?
                            EventHandlers?
                            Activity

Activity            ::=     BasicActivity
                            |StructuredActivity
                            |ScopeRelatedActivity

BasicActivity       ::=     ReceiveAct  | ReplyAct   |InvokeAct
                            |AssignAct  | ThrowAct   |TerminateAct
                            |WaitAct    | EmptyAct

StructuredActivity ::=      SequenceAct | SwitchAct |WhileAct
                            |PickAct    | FlowAct

ScopeRelatedActivity ::=    ScopeAct | CompensateAct
```

2.5. *Correlation*

One of the main challenges in integrating Web services, and specifically business processes, is to deal with stateful interactions. Business processes normally act according to a history of external interactions. Therefore, it is necessary to keep track of the state of each business process instance. Since we have different instances of a business process, messages need to be delivered not only to the correct port, but also to the correct instance of the business process. To ensure global interoperability and avoid implementation dependencies, the mechanism required for dynamic binding of messages needs to be defined in a generic manner rather than leaving this to the individual implementations [10].

The need for such a mechanism can be seen in our *e*-Book Store example. Each order that is sent by the customer is handled by an *e*-Book Store business process instance. For each order that is sent from this process instance to the publisher, there is also one business

process instance at the publisher side. These pairs of process instances need to interact with each other and as a result they need to "know" each other. Therefore, there must be a mechanism to route messages to the correct process instances. One standard approach to this problem is to carry a business token (e.g. such as an order number) in all transactions between *e*-Book Store and the publisher. This business token acts as a key indicating the exact business process instances. When a message arrives at each Web service, it is routed to the correct process instance that is identified based on the value of the business token in the message. In this way, all the messages that arrive for a specific process instance should carry the desired business token value.

Such a mechanism is supported in BPEL by providing the ability to define a set of such *correlation tokens*; i.e. a set of tokens shared by all messages in a correlation group. This set is called a *correlation set*. Once a correlation set is initiated, the values of correlation tokens must be identical for all the messages in that correlation group. In this way, an application-level conversation between business process instances is identified.

2.6. Activities

Activities that can be performed by a business process instance are categorized into *basic activities* and *structured activities*. Basic activities perform simple operations like *receive, reply, invoke, assign, throw, terminate, wait,* and *empty*. Structured activities impose an execution order to a collection of activities. It is important to note that structured activities can be nested. Structured activities include *sequence*, *switch*, *flow*, *pick* and *while*. The following sections briefly describe the semantics of BPEL activities.

2.6.1. Basic Activities

Receive Activity

The role of a *receive* activity is twofold: it is used both for providing Web services operations to the partners and for creating new instances of the business process. A receive activity specifies the partner link from which a message is received and the port type and operation that is used in receiving the message. If receive activity is annotated with the createInstance attribute set to 'yes', a new instance of the business process must be created when the expected message arrives. In this sense, receive acts as the start activity of the process and has an important role in its life cycle. It is worth mentioning that such start activity must be an initial activity as well, that is every other basic activity that is performed prior to or concurrently with this receive activity must be annotated with the createInstance attribute set to 'yes' as well.

Reply Activity

A *reply* activity is always associated with a receive activity. It is meant to send a response to a request that is accepted by the associated receive activity. Such an interaction is viewed as a *synchronous interaction* in BPEL, whereas *asynchronous* responses are treated as Web services invocation and performed by an invoke activity.

Invoke Activity

An *invoke* activity enables a business process instance to use the services that are provided by its partners. These services are used by invoking certain operations provided by the Web services. The operations can be synchronous request/response or asynchronous one-way operations, as described in [32]. An invoke activity can perform both types of operations by defining corresponding input and output messages. In an

asynchronous interaction only the input variable is defined[4], whereas in synchronous interactions both input and output variables are mandatory.

Wait Activity

A *wait* activity defines a period of time for which the business process instance will have to wait. This period of time is specified either by a duration (*for*) or by a deadline (*until*).

Terminate Activity

A *terminate* activity stops all the activities currently running in a business process instance and terminates the behaviour of the business process instance.

Empty Activity

An *empty* activity performs the simplest job; it does nothing.

Assign Activity

An *assign* activity is used to (1) copy data from one variable to another one; (2) construct new data using expressions; and (3) copy endpoints references to and from partner links. In a valid assign activity, the elements need to be type compatible. Type compatibility constraints and further details can be found in [10].

---

[4] An input variable carries the message that is sent to the partner in order to invoke its operation. It is called "input" variable, because it carries the input for that operation.

Throw Activity

A *throw* activity is used to report an internal fault explicitly. It specifies the name of the generated fault and, optionally, it can also fill out a fault variable with further information about the fault and pass it to the respective fault handler.

Note: Since the behaviour of the assign activity and the throw activity is not captured in this project, the description given here is minimal. Capturing the behaviour of these two activities requires further refinements and considerations and is carried out as part of another project in our group. For more details about the behaviour of these activities, the reader is referred to [10].

2.6.2. Structured Activities

Sequence Activity

A *sequence* activity structures a collection of activities to take place one after another. A sequence activity is completed when the last activity in the sequence is completed.

Switch Activity

A *switch* activity provides the ability to choose among a collection of activities. A set of conditional branches, called *case elements*, are introduced in switch activity and are examined in the order they appear. The first branch with true condition is chosen and its corresponding activity is then executed. If none of the cases is true, the *otherwise* branch will be taken and its activity is executed. A default otherwise branch is assumed to exist with an empty activity.

Flow Activity

A *flow* activity enables the concurrent execution of a set of activities together with synchronization between these activities. A flow activity is completed when all its

activities have finished execution. The *synchronization dependencies* are expressed by defining *links* between concurrent activities.

Link Semantics: Each BPEL activity includes the standard *source* and *target* elements that are used to link two activities. An activity can be defined as the source or the target of a set of links. Links are defined to impose synchronization dependencies on concurrent activities. If activity *A* is the source of link *L* and activity *B* is the target of link *L*, then we say *B* has synchronization dependency on *A;* i.e., if *B* is ready to start execution, it has to wait until the *status* of link *L* (and any other incoming links) is determined. Once *A* is completed, the status of all outgoing links (including *L*) is determined based on their `transitionCondition`; if this condition is true the status of the link is *positive*, otherwise it is *negative*. Once the status of all incoming links of *B* (including *L*) is determined the standard `joinCondition` of *B* is evaluated. If the condition is true then *B* is executed; otherwise a fault is thrown[5]. Figure 2-3 shows how the synchronization dependencies are specified in a flow activity. *A, S,* and *D* are three activities that are executed concurrently in flow *F*. *S* is a sequence activity in which *B* and *C* are executed in the given order. Two links are defined in the flow activity: *LinkAtoB* and *LinkDtoB*. When the flow activity is executed, *A, S* and *D* start their execution concurrently. However, *S* will stop immediately because its first activity (*B)* has synchronization dependency both on *A* and *D*. Thus, it has to wait until *A* and *D* are completed and the status of *LinkAtoB* and *LinkDtoB* is determined. Once completed, *LinkAtoB* becomes positive and *LinkDtoB* becomes negative. The `joinCondition` of *B* is then evaluated and *B* is executed if the join condition is true.

---

[5] "If the explicit `joinCondition` is missing, the implicit condition requires the status of at least one incoming link to be positive [10, 12.5.1]."

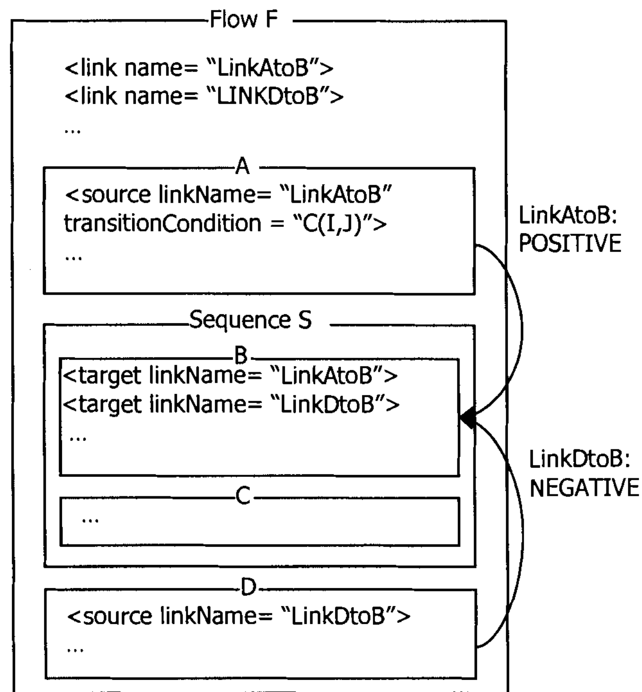


Figure 2-3 A flow activity with synchronization dependencies

Pick Activity

A *pick* activity waits on a set of events for one of them to occur and then executes its corresponding activity. If more than one event occurs then the pick activity will choose the one that has occurred first. As soon as an event is chosen, the pick activity no longer accepts any of the other events. Basically, there are two types of events: *onMessage* events and *onAlarm* events. The semantics of an onMessage event is very similar to a receive activity. An onMessage event occurs as soon as its corresponding message is received. onAlarm events are very similar to timers. They wait *for* a period of time or *until* a certain deadline is reached before they occur.

2.7. e-Book Store Re-visited

Based on the description of BPEL activities presented above, we now present the sample business process from our e-Book Store in a pseudo-code-like style. Although the

definition is written in a syntax similar to BPEL, there are certain details and requirements that have not been considered. Hence, the complete and correct BPEL definition requires dealing with these details and following the precise syntax.

2.7.1. Partners

The first step in defining a business process is to identify its partners. This includes identifying the shape of the conversation with partners by specifying messages and port types used in the interactions. The services with which a business process interacts are identified with *partner links.* From a partner link one can characterize the conversations between two services and the port types that are used in the communication. Thus, a partner link provides the *static shape* of the conversation. Nevertheless, it is worth mentioning that communicating with a partner via a partner link requires additional information about the actual partner service and communication bindings, which can be set as part of the business process deployment. This is outside the scope of BPEL [10, Section 7.2]. Figure 2-4 illustrates the static shape of e-Book Store relationships with its partners. Each circle specifies one partner link of the e-Book store business process. The business process interacts with three partners, through three partner links: purchasing, publishing, and shipping. These identifications are used in the definition of the business process, as it can be seen in the next section.

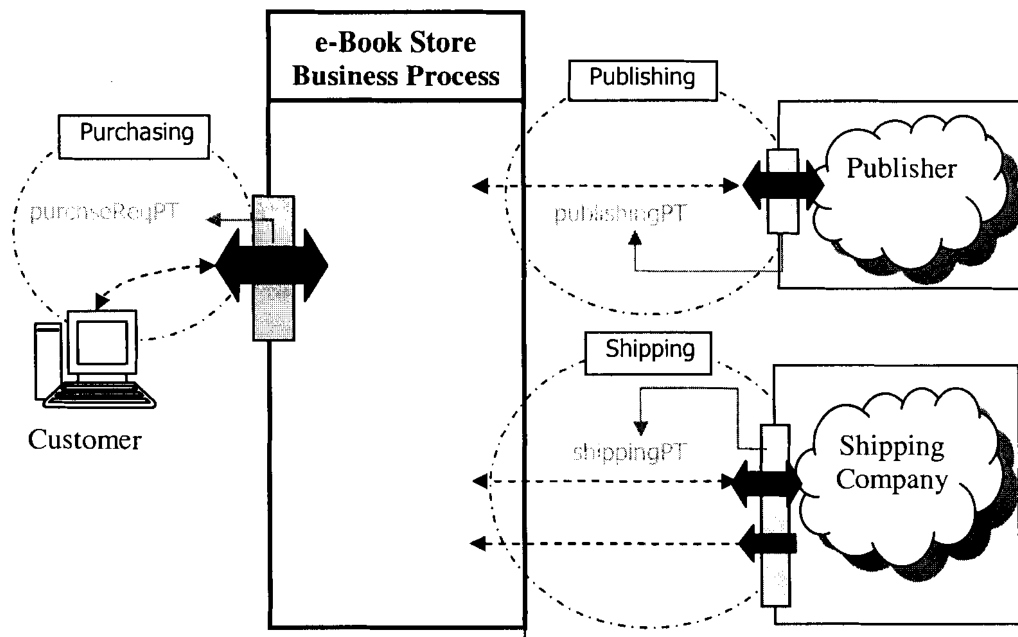


Figure 2-4 Partners and port types of the e-Book Store business process

2.7.2. e-Book Store Business Process

As illustrated below, the business process of e-Book store is named `eBookStoreProcess` and contains three partner links: purchasing, publishing and shipping. A set of variables is introduced in the process to maintain the state of the process. Each variable corresponds to a message that is communicated between the business process and its partners. In a way these variables work as wrappers for the messages; i.e. as soon as a message arrives, the business process wraps it in a variable and uses it afterwards. For example, `order` is a variable that embraces `orderMessage` as soon as it arrives at the business process. It is then used to conduct further communication with other services. `bookOrder` is a correlation set which identifies the business token that is required for specifying business process instances. Although the details are not presented here, `orderNo` is associated with a specific part of each of the messages belonging to the correlation group. In `eBookStoreProcess` all of the interactions are annotated with this correlation set, meaning that all interactions belong to

the same correlation group. Hence, this business token must be carried by all the incoming and outgoing messages.

```
Process
    ProcessName = "eBookStoreProcess"
//Partner Definitions
    PartnerLinks
         PartnerLink name="purchasing"
         PartnerLink name="publishing"
         PartnerLink name="shipping"
//Variable Definitions
    Variables
         Variable
              name="order"
              messageType="orderMessage"
         Variable
              name="shippingConfirmation"
              messageType="shippingConfirmationMessage"
         Variable
              name="publishingConfirmation"
              messageType="publishingConfirmationMessage"
         Variable
              name="shippingInfo"
              messageType="shippingInfoMessage"
//Correlation Sets
    CorrelationSets
         CorrelationSet
              name="bookOrder"
              properties= "orderNo"
//Main Activity: Sequence
    Sequence
//Interaction #1: Receive order from the customer
         Receive
              partnerLink="purchasing"
              portType="purchasePT"
              operation="sendPurchaseOrder"
              variable="order"
              createInstance="yes"
              correlation
                   set="bookOrder" initiate="yes"
//Interactions #2,#3: Concurrent Invokes
         Flow
              Invoke
                   partnerLink="publishing"
                   portType="publishPT"
                   operation="bookRequest"
                   inputVariable ="order"
                   outputVariable="publisherConfirmation"
```

```
                    correlation
                          set="bookOrder" initiate="No"

              Invoke
                    partnerLink=="shipping"
                    portType="shippingPT"
                    operation="shippingRequest"
                    inputVariable="order"
                    outputVariable="shippingConfirmation"
                    correlation
                          set="bookOrder" initiate="No"
//Interaction #4:Receive the schedule from the shipping company
        Receive
              partnerLink="shipping"
              portType="shippingPT"
              operation="sendInfo"
              variable="shippigInfo"
              correlation
                    set="bookOrder" initiate="No"
//Interaction #5: Reply to the customer
        Reply
              partnerLink="purchasing"
              portType="sendPurchaseOrder"
              operation="sendInfo"
              variable="shippingInfo"
              correlation
                    set ="bookStore" initiate="No"
```

The main activity of this process is a sequence activity, which causes a sequence of actions taking place one after another. The first activity in the sequence is a start activity; if a message is received which belongs to the conversation identified by the `purchasing` partner link, `purchasePT` port type, and `sendPurchaseOrder` operation, a new instance of the business process must be created to handle this request. Moreover, the `bookStore` correlation set is initiated which initializes and keeps the value of `orderNo` for further interactions. In the next step, the business process instance contacts the publisher and the shipping company at the same time and requests an order confirmation from both. This is accomplished by a flow activity which performs two invoke activities, one for the publisher and one for the shipping company. It is important to note that both invoke activities follow the `bookStore` correlation set, and hence the messages must carry the correlation token. When the confirmations are received from both partners, the

business process instance performs the next action, which is to receive a call back from the shipping company specifying the shipping schedule and other information. The last activity is a reply activity that passes this information to the customer. As mentioned before, both of these activities belong to the `bookStore` correlation group, therefore the message (`shippingInfo`) must carry the `orderNo` information.

2.8. Long-running Business Processes and Compensation Behaviour

Business processes are meant to define the interactions between several partners that are based on certain business logic. These processes usually have long durations and include asynchronous message passing between the partners. Consequently, error handling in such an environment is not easy. It is done by *compensation,* i.e. *"application specific activities that attempt to reverse the effects of a previous activity that was carried out as a part of a larger unit of work that is being abandoned."* [10, Section 13.2] This ability of compensating exceptions in an application-specific manner enables business processes to have so-called Long-Running (Business) Transactions (LRTs).

Compensation and fault handling in BPEL is done using the *scope* activity. Scope defines a logical unit of work for which a *compensation handler* or a set of *fault handlers* can be defined. A compensation handler defines the compensating behaviour of the logical unit in case of an error. A fault handler defines the reaction of the logical unit to an error. However, BPEL only deals with LRTs locally and within a single business process instance. The problem of achieving distributed agreement is addressed in [33]. As outlined in the LRM, the need to combine WS-Transaction with BPEL is well recognized. Clearly, the formal definition of BPEL and WS-Transaction will be an asset in this regard. Although they are not addressed in this project, the formal definition of compensation behaviour and fault handling is captured by another work in our group.

Chapter 3. Abstract State Machines

Our approach to modelling is based on the *abstract state machine* (ASM) paradigm. In this chapter, we first give a brief introduction to the basic ASM concepts, including *parallelism* and *non-determinism*. In the second section, we introduce *distributed abstract state machines* (DASMs) as a generalization of basic ASMs. The DASM computation model is widely used for modelling concurrent and distributed systems; hence we try to investigate the main DASM concepts, namely *concurrency*, *reactivity* and *real-time* behaviour, in more detail. The last section describes the operations and convention that have been introduced in our DASM model and used in this project.

The definitions recalled here should be sufficient for the purpose of this thesis. For a more comprehensive and rigorous definition, we refer the reader to the original literature on the theory of ASMs [23], [3] and their applications [9].

3.1. Basic Abstract State Machines

A *basic* ASM consists of a *program*, a set of *states* which can be viewed as first-order structures in mathematical logic, and a collection of *initial states*. A state S of vocabulary V consists of a *base set* X and the interpretations of function and relation names defined in V. An r-ary function name is interpreted as an r-ary function from X^r to X, called *basic function* of S. Similarly, an r-ary relation name is interpreted as an r-ary function from X^r to $\{true, false\}$, a *basic relation* of S. Every vocabulary contains static logic symbols *true, false, undef* and standard Boolean operations. The default value for basic functions is *undef* and is *false* for basic relations. Constants are represented as nullary function names and are interpreted as elements of X. Unary relation names can be interpreted as

special *universes* and allow a state to be viewed as a many-sorted structure where each universe represents some sort [23], [9].

A basic ASM program is just a rule. In basic ASMs this rule can be an *update rule,* a *conditional rule,* a *do-in-parallel rule* or an *import rule.*

An update rule has the form

$$f(t_1,\ldots,t_n) := t_0$$

where f is a *dynamic* function (or relation) and each t_i is a *term* (recursively defined as in first-order logic). A *location* of a state S is defined as a pair $(f,\bar{x})$ where f is an r-ary dynamic function name and $\bar{x}$ is an r-ary tuple of elements. The *content* of this location is defined as $c = f(\bar{x})$. An *update* (l, c') of state S replaces the old content of l with c' in the next state. An update rule of the above form fires an update (l, v_0) where $l = f(v_1,\ldots,v_n)$ and v_i is the value of each t_i.

A conditional rule has the form

if *e* **then** *R1* **else** *R2*

where e is a Boolean term and *R1, R2* are ASM rules. If e is evaluated to *true* then *R1* is executed, otherwise *R2* is executed.

A do-in-parallel rule has the form

do-in-parallel
R1
R2

where *R1*and *R2* are ASM rules. Such a rule executes *R1* and *R2* simultaneously.

In order to capture all sequential algorithms, the import rule is introduced in addition to these three basic rules. By using the import rule, we can model dynamic resource allocation for instance, such as adding a new elements to the model, e.g. add a new node to a graph. For further details the reader is referred to [23].

3.1.1. Non-determinism

Non-determinism is often required for describing algorithms at higher levels of abstraction. The basic ASM model is extended with the *choose rule* to capture explicit non-determinism.

A choose rule has the form

> **choose** $x \in S$
>
> $R(x)$

where $R(x)$ is a rule. To execute this rule, any element of S is chosen non-deterministically and $R(x)$ is executed.

A generalized version of choose is also introduced in [23], where a satisfying condition can be added to the rule:

> **choose** $x \in S$ **with** $g(x)$
>
> $R(x)$

where $g(x)$ is a Boolean term. The meaning of this rule is to choose an arbitrary x among those elements of S that satisfy g; i.e. $\{y | \, y \in S, \, g(y) = true\}$.

3.1.2. Parallelism

The notion of parallelism is introduced in the basic ASM model by means of the *forall rule.*

A forall rule has the form

> **forall** $x \in S$
>
> $R(x)$

where $R(x)$ is a rule. It executes all rules $R(x)$, where x is an element of S, simultaneously. Analogous to the definition of choose, the forall rule can be generalized by introducing a satisfying condition.

3.2. Distributed Abstract State Machines

A *distributed abstract state machine* M includes a set of *agents*. The behaviour of each agent is described by its *program*. A DASM M is defined over a given vocabulary V with a program Π_M and a non-empty set I_M of initial states. An initial state specifies a possible interpretation of V over some potentially infinite base set X. The behaviour of an agent a in a given state S of M is defined by $program_S(a)$. The dynamic universe *AGENT* represents the set of all agents in a DASM and the static universe *PROGRAM* represents the set of programs that these agents can execute. Agents can be dynamically added to or removed from AGENT.

In every state S reachable from an initial state of M, the set AGENT is well defined as follows.

$$AGENT_S \equiv \{ x \in X : program_S(x) \in PROGRAM\}$$

Each computation step of a single agent is called a *move*. Agents operate concurrently. As stated in [23], every run ρ of a DASM M is given by a triple (P, λ, σ) satisfying the following conditions:

1- P is a partially ordered set of moves where each move has only finitely many predecessors; i.e. $\{y | \ y \leq x\}$ is finite. Figure 3-1 presents one such partially ordered set of moves where each m_i represents a move.

2- The set of moves of a single agent are linearly ordered. λ is a function on P associating agents with moves, so $\{x | \ \lambda(x) = a\}$ is linearly ordered for every agent a. In Figure 3-1, m_1, m_2, m_4, and m_6 belong to agent a_1 while m_3 and m_5 belong to agent a_2.

3- σ(X) returns a state of M resulted by performing all moves in X; i.e. for each initial segment Y of P, $\sigma(P)$ specifies a state of M. $\sigma(\emptyset)$ is an initial state. An initial segment of P is a substructure Y of P such that if $y \in Y$ and $x < y$ in P then $x \in Y$. In Figure 3-1, the circles specify initial segments of P.

4- The coherence condition: If x is a maximal element in a finite initial segment X of P and $Y = X - \{x\}$ then $\sigma(X)$ is obtained from $\sigma(Y)$ by firing $\lambda(x)$ at $\sigma(Y)$ ($\lambda(x)$ is an agent in $\sigma(Y)$). In Figure 3-1, m_6 is the maximal element of X and $Y = X - \{m_6\}$.

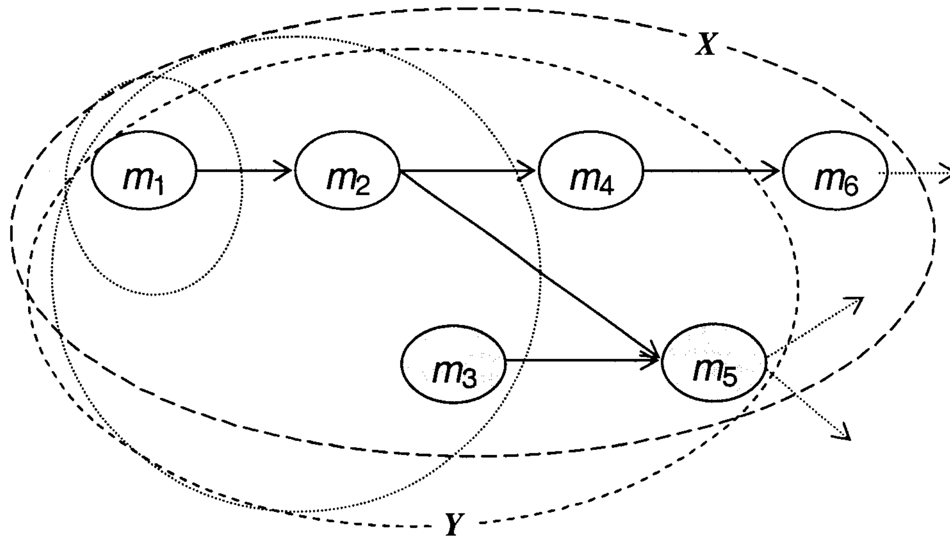


Figure 3-1 A partial ordered set of moves

While the above definition is concise, it needs further investigation to fully understand the implication of the coherence condition on the runs of a DASM. Each partially order run specifies a class of possible executions of a DASM. One immediate corollary of the coherence condition is expressed in terms of the linearizations of partially ordered runs; i.e. if ρ' is a finite initial segment of ρ, then all linearizations of ρ' yield to the same final state of M.

To further illustrate the meaning of the coherence condition in the above definition and the relationship between partially ordered runs and their linearizations, the following section considers two simple but meaningful examples.

3.2.1. Concurrency in Sample DASM models

We show some implications of the coherence condition on the semantics of partially ordered runs through the following examples.

Example 3.1.[6] Suppose that we have three propositional variables (dynamic nullary relation symbols) *door, window* and *light.* Intuitively *door* = *true* means that "the door is open", *window* = *true* means that "the window is open" and *light* = *true* means that "the light is on". Now, consider a DASM consisting of three agents: a *door manager* (agent d), a *window manager* (agent w) and a *light manager* (agent l). The door manager opens the door only when the window is closed (move x), the window manager opens the window only when the door is closed (move y), and the light manager turns on the light when either the door or the window is closed (move z).

```
WindowManagerProgram ≡ if ¬door then window := true

DoorManagerProgram ≡ if ¬window then door := true

LightManagerProgram ≡ if ¬door or ¬window then light := true
```

Figure 3-2 shows all of the possible DASM runs assuming that in the initial state S_0 the door and the window are closed and the light is turned off. There are six possible runs (M_1 -M_6) yielding to two different final states (S_4, S_5).

$M_1 = (\{x, z\}, \langle\rangle)$, $M_2 = (\{x, z\}, \langle x < z\rangle)$, $M_3 = (\{x, z\}, \langle z < x\rangle)$,

$M_4 = (\{y, z\}, \langle\rangle)$, $M_5 = (\{y, z\}, \langle y < z\rangle)$, $M_6 = (\{y, z\}, \langle z < y\rangle)$.

We cannot have $x < y$ because w is disabled in state S_1 obtained from S_0 by performing x. Similarly we cannot have $y < x$ because d is disabled in state S_3 obtained from S_0 by performing y. Finally, we also cannot have a run where x and y are incomparable, that is neither $x < y$ nor $y < x$. This follows from the fact that all the linearizations of such a run

---

[6] This example is derived from [19].

must result in the same state (thus it is impossible to go from state S_0 to S_6 or S_7, or from state S_2 to S_7).

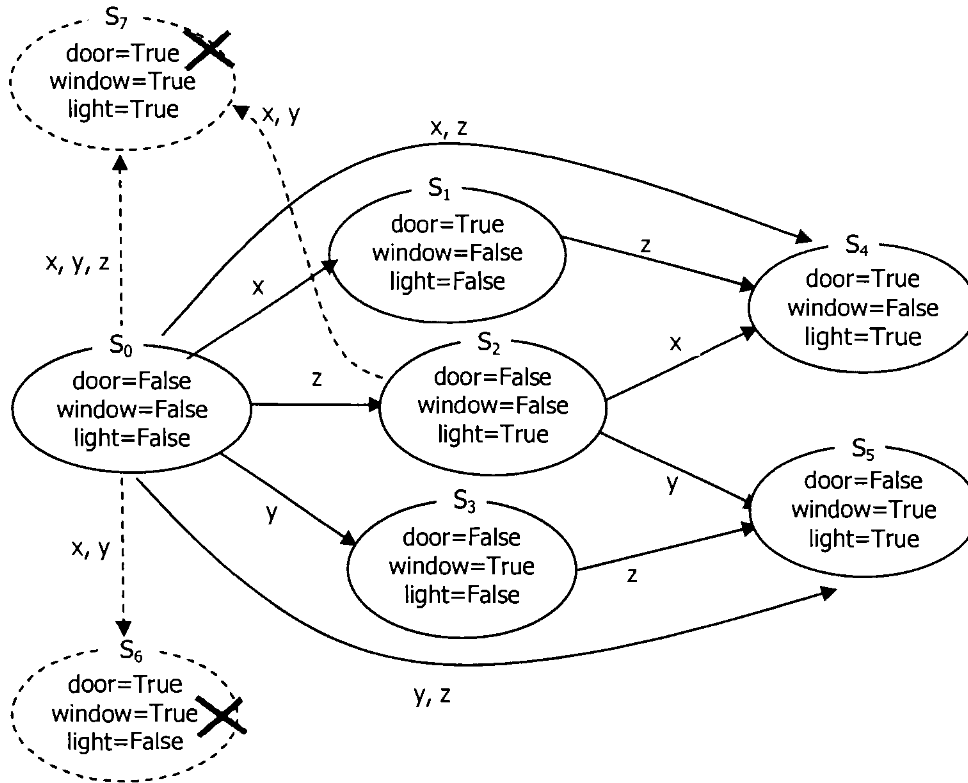


Figure 3-2 All possible runs of the DASM of Example 3.1

Example 3.2. Suppose a single *producer* agent is placing items, one by one, into a queue. Two *consumer* agents concurrently attempt to remove these items by popping the head of the queue. This example shows the effect of the coherence condition in the presence of a race condition (between the two consumers simultaneously trying to remove the same item of the queue).

We abstract from the details of adding items to the queue and removing items from it. In each step of the producer agent, it adds a single new item to the queue (move p). In each step of a consumer, it removes the head item if the queue is nonempty (moves c_1, c_2). The programs of the producer agent and the consumer agents can be written as follows.

ProducerProgram ≡ ADD_ITEM(queue, newItem)

ConsumerProgram ≡ **if** queue ≠ empty **then** item := headItem(queue)

In the initial state the queue is empty. The most important property of this DASM is that there is no run where c_1 and c_2 are incomparable. Note that if both consumers would make an attempt to remove the same head item at the same time (incomparable c_1 and c_2), this would not cause conflicting update operations on the queue; rather it would produce a logical conflict (notably, a duplication of this item). The coherence condition prohibits this behaviour as any linear execution of such a run, for instance $c_1 < c_2$, can not produce the same result. Figure 3-3 shows some segment of possible runs of this DASM and helps clarifying this argument. Clearly, it is not possible to go from states S_2 or S_3 to S_4; hence c_1 and c_2 are not incomparable.

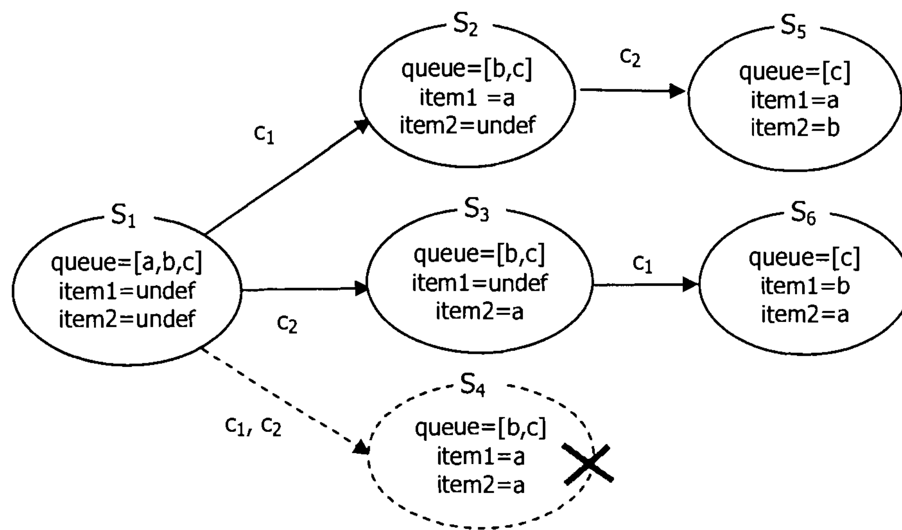


Figure 3-3 Some segment of possible runs of Example 3.2

3.2.2. Reactivity

In support of the principle of separation of concerns, the communication between an agent and its environment (or similarly among different agents in a DASM) is not supported by any specific mechanism. Instead, functions act as means of communication in ASMs. Thus, it is natural to categorize functions based on their role in a specific ASM M [9].

Basic functions of M are categorized in two main groups: *static* and *dynamic*. Static functions are those with constant values during all runs of M. On the other hand, the value of a dynamic function may change in different states of M. Dynamic functions are then categorized into different groups based on their role in M.

A *monitored* function of M is a dynamic function that is only updated by the environment. The machine reads the monitored function, but does not update it. In case of a DASM, a monitored function (defined for a specific agent) can also be updated by other agents. Monitored functions are the means of transferring information from the environment (or other agents) to a specific machine (agent) [9]. A typical example of such a function is the nullary function *now*, which is defined in real-time distributed ASMs (see Section 3.2.3). This function returns the global system time and is updated by the environment. As such, it truly resembles the behaviour of a clock, or watch, in the real world. We do not keep track of the time ourselves; we look at a watch to know the time.

On the other hand, *out* functions are the functions that are only updated, but never read, by M. Conversely, they are read but not updated by the environment (or other agents). Consequently, through out functions agents can forward information to the environment or to each other. To describe the interaction between an agent and the environment, one can also define *shared* dynamic functions that are updated and read both by the machine and the environment (or other agents) [9].

A clear distinction of various types of functions enables ASMs to support reactive behaviour as well as separation of concerns, information hiding, data abstraction and stepwise refinement [9]. The reactive behaviour of a system is captured in ASMs by introducing well-defined functions as interfaces of interactions between a machine and its environment. Specifying interactions through monitored, shared and out functions as interfaces, enable us to define the reactive behaviour of a system while abstracting from communication concerns. The following example shows how reactive behaviour is captured by the ASM paradigm.

Example 3.3. Assume an asynchronous interaction model between three autonomously operating entities that are involved in Automated Teller Machine (ATM) transactions, namely: an *ATM manager*, an *authentication manager*, and an *account manager*. For simplicity, here we restrict our attention to the withdrawal transaction of the ATM.

Performing a withdrawal transaction requires the following logical steps:

1- Input the bank card, PIN code and withdrawal amount.

2- Authenticate the bank card and PIN code.

3- Check the account balance against the credit line.

4- On approval update the account balance.

5- Output cash or notification about denial.

Assuming an unreliable communication medium, timeout mechanisms may cause the cancellation of a transaction at any time.

The Abstract Model. In this initial model we do not formally define the behaviour of the account manager and the authentication manager. Instead, we consider these two entities as parts of the environment and focus on the behaviour of the ATM manager.

The ATM agent communicates with its environment through various monitored functions. An activation event occurs whenever a user requests the service. The user then

enters the card number, PIN code and the desired withdrawal amount. Beyond reading this data from the environment, the machine can also perform more complex interactions with the environment to get other, non-trivial information like authentication approval/rejection and transaction approval/denial. The abstraction mechanisms allow us not only to define (and decide about) the environment, but to freely choose the level of detail and precision.

The behaviour of the ATM control is described as follows:

```
if Idle and activationEvent then
   data := getCardData
   code := getPinCode
   amount := getWithdrawAmount
   mode := processing
if Processing and isAuthenticated(data, code) and ¬cancellationEvent then
   if isValidTransaction(data, amount) then
      RELEASE_CASH(amount)
      UPDATE_ACCOUNT_BALANCE(data, amount)
   else
      OUTPUT_CANCELLATION_NOTIFICATION
   mode := idle
if Processing and (¬isAuthenticated(data,code) or cancellationEvent) then
   OUTPUT_CANCELLATION_NOTIFICATION
   mode := idle
where
   Idle ≡ mode = idle,
   Processing ≡ mode = processing
```

The machine is idle in the initial state. activationEvent is a monitored predicate that causes the DASM to become active. Other monitored functions getCardData, getPinCode and getWithdrawAmount serve to obtain the user's data and withdrawal amount. In this way, a series of interactions between the DASM and the environment takes place and in each step some required information, ranging from the requested withdrawal amount to user authentication, is obtained from the environment and is used to perform the operation. isAuthenticated and isValidTransaction are two important monitored functions that respectively provide the authentication and account management services to the ATM abstract machine. At this level of abstraction, the ATM manager does not issue any

information (data, code or amount) to the authentication manager or the account manager. Alternatively, we assume that these communications take place in the background. cancellationEvent is another important monitored predicate that indicates cancellation of the operation caused by the timeout mechanism.

This example also makes use of another convenient feature. RELEASE_CASH, UPDATE_ACCOUNT_BALANCE and OUTPUT_CANCELLATION_NOTIFICATION are parts of the model that are meant to perform the final operations. However, we do not want to deal with the details of such operations at this level of abstraction. Thus, we left the definition of these rules abstract assuming that more detailed definition of these rules will be provided as part of the next refinement step.

3.2.3. Real-Time Behaviour

In order to capture real time behaviour, additional constraints are imposed on DASM runs ensuring that the agents react instantaneously and environmental changes take place instantaneously [25]. We introduce an abstract notion of local system time for modelling timeout events. In a given state S of M, the global time (as measured by some global system clock) is given by a nullary monitored function *now* taking values in some linearly ordered domain *TIME*. Time values are represented as positive real numbers. Additionally, '∞' represents a distinguished time value such that $t < \infty$ for all $t \in TIME - \{\infty\}$. We assume the values of *now* to increase monotonically over runs of M. Our semantic model of time resembles those defined in [8], [18], and [22].

3.3. Our DASM

Our formal definition of an abstract operational semantics of BPEL is based on the real-time distributed abstract state machine model. However, we introduce two additional operations that facilitate the creation and termination of DASM agents. These operations

are different from normal creation and termination in the sense that they also update the (sub-) domain of the agent.

new a : ⟨domain⟩

new creates a new agent a of type ⟨domain⟩ and sets *program(a).* Additionally, it also adds agent a to the associated domain of agents.

stop a

stop discards agent a from the associated domain of agents and resets *program(a)* to *undef*.

To cope with partial updates of sets, we follow the solution proposed in [26] and use the following operations for adding/removing an element to/from a set.

add a **to** A

add inserts element a into set A of elements.

remove a **from** A

remove deletes element a from set A of elements.

3.4. Notational Conventions

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

- Program names are entirely written in capital letters with no separator between individual words (e.g. PROCESSPROGRAM)

- Function names start with a lowercase first letter. The individual words start with capital letters and the rest of the letters are written in lowercase (e.g. functionName).
- Abstract rule names are entirely written in capital letters and the individual words are divided by underscore '_' (e.g. INITIATE_CORRELATION).
- Abstract predicate names are entirely written in lowercase letters. The individual words are divided by underscore '_' (e.g. message_is_received).
- Rule names start with a capital letter. The individual words also start with capital letters and are separated by underscore '_'. The rest of the letters are written in lower case (e.g. Pick_Activity_Clearance).
- ASM keywords are written in lowercase using bold font (e.g. **else**).
- Domains are written in all capital letters (e.g. MESSAGE).

Chapter 4. Formalization of the BPEL Web Services Architecture

We formalize here the key functional attributes of the BPEL Web services architecture based on the asynchronous computation model of distributed abstract state machines [23]. The primary focus is on the concurrent and reactive behaviour of Web services and their interaction through TCP/IP communication networks. This includes concurrent control structures, communication primitives, and dynamic creation and termination of services. For dealing with real time aspects, we define an abstract notion of global system time and impose additional constraints on the runs defining the behaviour of our BPEL abstract machine.

4.1. Overall Organization

Logically, the BPEL Web services architecture splits into two basically different components, namely: (1) the TCP/IP communication network, and (2) the BPEL services residing at the communication endpoints. We separate the behaviour of the network from the behaviour of services by decomposing our architecture model of the BPEL abstract machine into two sub-models, each of which in turn is a distributed ASM, or DASM.

In this project, we concentrated on the *service abstract machine* model, whereas a *network abstract machine* model is defined in [20]. The composition of these two machines is well defined by the underlying semantics of the DASM computation model. Any interaction between these models is restricted to actions and events occurring at well identified interfaces. Each Web service in the service model interacts with the external world (i.e. the communication network and remote Web services) through two well

defined interfaces, one for incoming messages and the other for outgoing messages. The network model delivers the messages to the input mailbox (inbox space) of a service and carries the messages from the output mailbox (outbox space) of a service to remote services. The service abstract machine model and the network abstract machine model both are based on asynchronous models; hence they can easily be composed into one coherent and consistent DASM. It is worth mentioning that since BPEL is defined on top of WSDL, it is sometimes necessary to take into account the service or message bindings described by the WSDL definitions. Thus, a transformation phase is required to make the messages conform to the correct format and carry the required information. Figure 4-1 gives an overview of the composition of the two models.

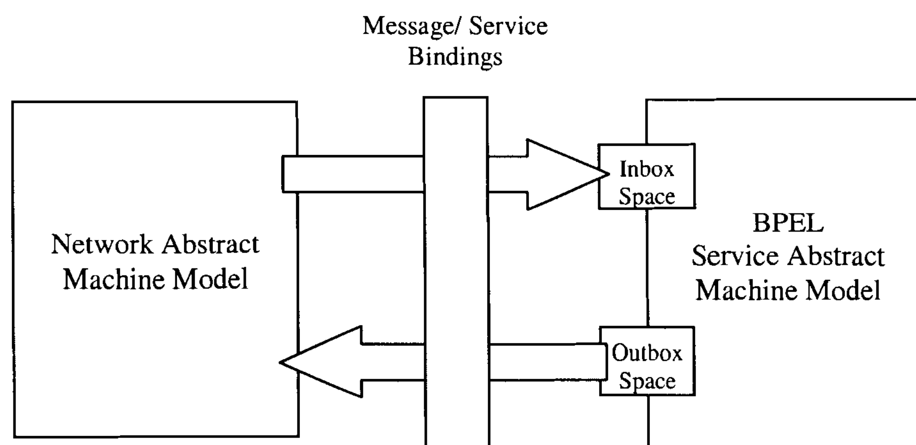


Figure 4-1 The composition of the BPEL service model and the network model

The overall organization of the BPEL abstract machine splits into three different layers as illustrated in Figure 4-2. The abstract model is introduced in this chapter. The intermediate model and the executable model are presented in Chapter 5 and Chapter 6.

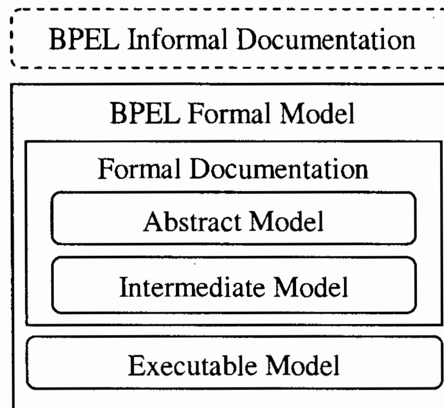


Figure 4-2 A three layer approach: From formal documentation to the executable model

4.2. BPEL Abstract Model: Overview

The top layer of the BPEL abstract machine, called the *abstract model*, provides an overview of the abstract machine architecture and defines the underlying modelling framework. A BPEL document abstractly defines a Web service consisting of a collection of business process instances. A *process instance* maintains a continuous interaction with the external world through two interface components, called *inbox manager* and *outbox manager*, as shown in Figure 4-3. As for the composition of the service model and the network model mentioned in the previous section (Section 4.2), the inbox manager and the outbox manager are attached to the network model and operate as the interfaces between the network model and the service model.

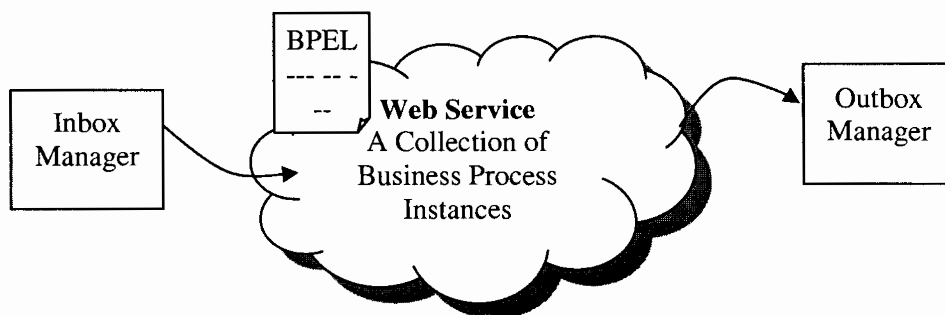


Figure 4-3 High-level abstract structure of our BPEL model

The inbox manager operates on the *inbox space,* a possibly empty set of inbound messages, and takes care of all the messages that arrive at the Web service. For each such message, the inbox manager is responsible to find a process instance that is waiting for that message, and assigns the message to this instance. The outbox manager, on the other hand, delivers outbound messages from process instances to the network. Inbox managers, outbox managers, and process instances are modelled by three different types of DASM agents. Additionally, we introduce another agent type, *activity agent.* Each process agent executes a single process instance and it uses dynamically created activity agents for executing complex (structured) activities.

```
AGENT ≡ INBOX_MANAGER ∪ OUTBOX_MANAGER ∪ PROCESS ∪ ACTIVITY_AGENT
```

In the initial DASM state, there are only three DASM agents: the inbox manager, the outbox manager and a dummy process. The role of the dummy process instance merely is to simplify the creation of new process instances. There is always one and only one such process instance waiting on its start activity. By receiving the first matching message, the dummy process instance becomes a normal running process instance and a new dummy process instance will be created automatically by the inbox manager. The DASM program given below specifies the behaviour of the inbox manager agent.

```
domain MESSAGE
inboxSpace: INBOX_MANAGER → MESSAGE-set
//initial value: ∅
//Keeps the messages that have arrived for a business process and have
//not yet been actively processed.

match: (PROCESS, MESSAGE) → BOOLEAN
//Tells whether a messages matches a process instance or not.

waiting: PROCESS → BOOLEAN
//Tells whether a process instance is waiting for a message or not.
```

```
INBOXMANAGERPROGRAM ≡
   if inboxSpace(self) ≠ ∅ then
      choose p ∈ PROCESS, m ∈ inboxSpace(self)
            with match(p, m) and waiting(p)
         ASSIGN_MESSAGE(p, m)
         //Effectively assigns message m to process instance p.
         if p = dummyProcess then
            new newDummy : PROCESS
               dummyProcess := newDummy
```

In each step, the inbox manager chooses a message among the messages waiting in the inbox space and tries to find a matching process instance to assign the message to this process instance. The predicate match(p: PROCESS, m: MESSAGE) checks whether message *m* can be delivered to process instance *p* or not, trying to match the message type and the correlation information between the waiting process instance and the incoming message. If the matching is successful, the message is assigned to the process instance by calling ASSIGN_MESSAGE(p, m) which is left abstract at this level but will be defined as part of the next refinement step.

The outbox manager operates on the *outbox space*, a possibly empty set of output descriptors, one for each outgoing message that is to be generated. The outbox manager then performs the actual output operation based on the information specified by the respective output descriptor. The following DASM program defines the behaviour of the outbox manager.

```
domain OUTPUT_DESCRIPTOR
outboxSpace: OUTBOX_MANAGER → OUTPUT_DESCRIPTOR-set
//initial value: ∅
//This set keeps the information on all the outbound messages in a given state.

OUTBOXMANAGERPROGRAM ≡
   if outboxSpace(self) ≠ ∅ then
      choose od ∈ outboxSpace(self)
         SEND(od)          //Effective send operation
```

To send a message to a specific remote destination, the process instance which needs to send the message creates an outbox descriptor in the outbox space. This descriptor encapsulates sufficient information on the message destination and the message itself. In

each step, the outbox manager chooses a single output descriptor and generates the corresponding message. The output operation itself is not further defined leaving the details of the operation SEND abstract.

In general, a BPEL program combines two different types of activities: basic activities and structured activities. Structured activities impose an execution order on a collection of activities. These activities can be both basic and structured activities. The execution of each structured activity inside a process instance is modelled by a single DASM agent of type activity agent. Figure 4-4 combines all the potential control structures of DASM activity agents at the top-level layer. A process instance uses five types of activity agents (*sequence agent*, *while agent*, *pick agent*, *switch agent* and *flow agent*) to execute different structured activities. A pick agent uses a *pick alarm agent* and a *pick message agent* to handle onAlarm events and onMessage events respectively. A flow agent creates a number of *flow thread agents* to concurrently execute its activities.

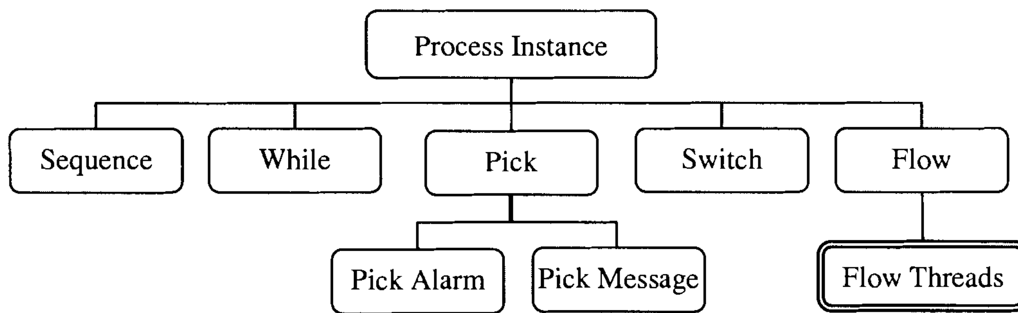


Figure 4-4 The combination of all potential control structures of DASM activity agents at the top-level layer

Below is the DASM program that abstractly specifies the behaviour of process agents.

```
RUNNING_AGENT ≡ PROCESS U ACTIVITY_AGENT
//RUNNING_AGENT is the set of agents that are executing (running) an activity.

startedExecution: PROCESS → BOOLEAN
//initial value: false
//Tells whether a process has started executing its activity or not.
```

```
busy: RUNNING_AGENT → BOOLEAN
//initial value: false
//An agent is busy while one of its activities is being executed.

activity: RUNNING_AGENT → ACTIVITY
//Returns the activity that must be executed in a running agent.
//derived from the BPEL document and defined in the initial state

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

The program of a process agent describes its behaviour in one DASM step. The routine is to execute the activity defined inside the process, and if the execution is completed the process agent is terminated. startedExecution is a predicate that specifies whether the execution of the activity is started or not. When the execution is started (startedExecution(self) = true), the process agent becomes busy (by setting the predicate busy to true) and remains busy during the execution. Once the execution is completed, the agent is released; i.e. busy is reset to false (by EXECUTE_ACTIVIY). Thus, the process agent knows the execution is completed and is terminated. Despite its important role, EXECUTE_ACTIVTY is not further defined at this level and is left abstract[7].

Modelling the behaviour of a BPEL process requires certain information that is specific for the given business process to be derived from the underlying BPEL document. For instance, the process agent program is defined to execute the main activity of the BPEL process which can only be extracted from the underlying BPEL process definition. The construction of the initial state is not further detailed here; rather we assume that the

[7] Note that we introduced a constraint on EXECUTE_ACTIVITY which requires it to release the agent when the execution of the activity is completed.

relevant information is generated automatically in a pre-processing step through static analysis of the underlying BPEL document using standard compiler techniques. This information is formalized by a set of statically defined functions (like activity) as part of the definition of the initial state of the DASM[8].

4.3. BPEL Abstract Model: Details

The BPEL abstract model captures the behaviour of a business process, the underlying framework and core BPEL activities in a high level of abstraction. In this sense, the abstract model provides guidelines for the specification of each activity. This section provides these high level specifications along with basic descriptions. The specifications are refined and discussed in more detail in the next chapter.

4.3.1. Basic Activities

The behaviour of each basic activity is defined by a single ASM rule in our model. As mentioned in Section 4.2, the process program fetches an activity and executes that activity. For basic activities the execution is handled by the corresponding ASM rule. For instance, Execute_Receive is responsible for executing a receive activity. These rules are formally defined and described in the following subsections.

4.3.1.1. Receive Activity

Executing a receive activity is done in two phases:

1- The running agent informs the inbox manager that a receive activity is executed and a message is anticipated.

---

[8] These functions are identified in the BPEL DASM with a comment (*"derived from the BPEL document and defined in the initial state"*).

2- The running agent waits until the message arrives and is assigned to the agent by the inbox manager

As a result, Execute_Receive works in two modes which are distinguished by receiveMode, a unary predicate that specifies whether an agent is waiting to receive a message or not. If receiveMode is false, it means that the inbox manager has not yet been informed about the receive activity and the anticipated message. The inbox manager can assign a message to an agent only when it is informed which agent is waiting for which message. The information about the expected message and the waiting agent is collected in an *input descriptor* and is passed to the inbox manager through a set, called *waiting set.* If receiveMode is true, it means that the input descriptor is already added to the waiting set and the agent has to wait until the message is received. When the message is received, the receiveMode is toggled (back to false) and the agent is released by setting busy to false.

```
receiveMode: RUNNING_AGENT → BOOLEAN
//initial value : false
//Tells whether a runnig agent is waiting to receive a message or not.

Execute_Receive (activity : RECEIVE) ≡
  if ¬receiveMode(self) then
    receiveMode(self) := true   //The running agent waits to receive a message
    ADD_INPUT_DESCRIPTOR_TO_WAITING_SET(activity)
  else
    if message_is_received(activity) then
      receiveMode(self) := false
      busy(self) := false
```

4.3.1.2. Reply Activity

Executing a reply activity requires sending a message out. As described in 4.2, in our model the outbox manager is responsible for the outgoing messages. Thus, in order to send a message out the running agent has to inform the outbox manager about the outgoing message. This is done by adding an *output descriptor*, which contains the required information for an outgoing message, to the outbox space of the outbox manager.

```
Execute_Reply ( activity : REPLY) ≡
   ADD_OUTPUT_DESCRIPTOR_TO_OUTBOX_SPACE(activity)
   busy(self) := false
```

4.3.1.3. Invoke Activity

As described in 2.6.1 the behaviour of an invoke activity is very similar to a combination of reply and receive. Initially, an invoke activity invokes a service of a partner by sending an appropriate message to that partner. Similar to what is done in the reply activity, this behaviour is captured by informing the outbox manager. However, if the invoke activity is synchronous (synchronous(activity) = true) the agent has to wait for a response from the partner. Basically, this behaviour is captured by the same approach that is applied to the receive activity. Thus, if the invoke activity is synchronous the receiveMode is set to true and the agent will wait to receive the message from the partner, as follows.

```
synchronous: INOVKE → BOOLEAN
//returns true if the invoke activity contains synchronous interactions;
//i.e. request/response
//derived from the BPEL document and defined in the initial state

Execute_Invoke (activity : INVOKE) ≡
   if ¬receiveMode(self) then
      ADD_OUTPUT_DESCRIPTOR_TO_OUTBOX_SPACE(activity)
      if ¬synchronous(activity) then
         busy(self) := false

      if synchronous(activity) then
         receiveMode(self) := true
         ADD_INPUT_DESCRIPTOR_TO_WAITING_SET(activity)

   if receiveMode(self) and message_is_received(activity) then
      receiveMode(self) := false
      busy(self) := false
```

4.3.1.4. Terminate Activity

Executing a terminate activity requires stopping the current business process instance and all its subordinate agents.

```
Execute_Terminate ≡
   STOP_ALL_SUBORDINATE_AGENTS
   stop rootProcess(self)
```

4.3.1.5. Wait Activity

When a wait activity is executed, the agent has to wait *for* a specified period of time, or *until* a specified time. While the latter can be accomplished by checking the current time against the deadline, modelling the former requires keeping track of the time that the waiting period starts. Thus, when the execution of a wait activity starts, the current time is recorded as the starting point of the waiting period, and then the agent remains busy until the waiting period is completed.

```
Execute_Wait (activity : WAIT) ≡
   if wait_just_started(activity) then
      RECORD_WAIT_START_TIME(activity)
   else
      if wait_completed(activity) then
         busy(self) := false
```

4.3.1.6. Empty Activity

Empty activity makes the agent do nothing for one DASM step. Thus, the agent becomes busy when this activity is fetched and is released when the empty activity is executed.

```
Execute_Empty (activity : EMPTY) ≡
   busy(self) := false
```

4.3.2. Structured Activities

As mentioned in Section 4.2, the behaviour of each structured activity is captured by a single DASM agent created to handle that activity. These DASM agents are called activity agents and include *sequence agents, switch agents, while agents, pick agents,* and *flow agents.* In addition, there are three other types of DASM agents that are categorized as activity agents and help in modelling the behaviour of structured activities: *flow thread agents, pick message agents,* and *pick alarm agents.*

domain SEQUENCE_AGENT
domain SWITCH_AGENT
domain WHILE_AGENT
domain PICK_AGENT
domain FLOW_AGENT
domain PICK_ALARM_AGENT
domain PICK_MESSAGE_AGENT
domain FLOW_THREAD_AGENT

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

The high level behaviour of each of these agents is abstractly presented in the following sections. It is important to notice that upon completion an activity agent has to take an additional action, which is releasing its parent. In Section 4.3.1 we showed that each execution rule is finished by setting busy to false, in order to inform the current agent that the execution of the activity is completed. In case of structured activities, the *parent agent* has to be informed about the completion; therefore RELEASE_PARENT is called whenever an activity agent completes its execution. RELEASE_PARENT has to set busy to false for the parent agent.

4.3.2.1. Sequence Activity

A sequence activity is handled by a sequence agent which executes a sequence of activities one by one. currentActivity is a function that indicates the activity that is being executed. A sequence agent program starts by setting this function to the first activity

specified in the sequence activity. When the activity is fetched, the agent becomes busy and starts executing it. When the execution is completed, the sequence agent is released; i.e. busy is set to false again. Thus, the next activity can be fetched and this continues until the last activity of the sequence is executed.

```
currentActivity: SEQUENCE_AGENT → ACTIVITY
//Keeps track of the current activity which is being executed

SEQUENCEPROGRAM ≡
  if ¬busy(self) then
    SET_CURRENT_ACTIVITY_TO_NEXT_ACTIVITY
    busy(self) := true

  if busy(self) then
    if sequence_is_not_completed then  //There are still some activities to execute
      EXECUTE_ACTIVITY(currentActivity(self))
    else            //No more activities
      stop self
      RELEASE_PARENT
```

4.3.2.2. Switch Activity

Switch activity performs two main tasks:

1- Finds the first branch with a true condition (or selects the *otherwise* branch)

2- Executes the activity associated with that branch.

In our model, a switch activity is handled by a switch agent which performs the two tasks mentioned above. First, it selects the appropriate branch and becomes busy. This selection is always successful because the LRM introduces a default *otherwise* branch for every switch activity. Second, the agent remains busy until the execution of the activity associated with the selected branch is completed. foundBranch is a function that indicates this activity and is updated when the branch is selected. When the execution is completed, the agent is required to release its parent agent and terminate its execution.

```
foundBranch: SWITCH_AGENT → ACTIVITY
//The activity associated with the branch that is chosen by switch to be executed

SWITCHPROGRAM ≡
  if ¬busy(self) and ¬branch_found then //No branch is selected yet
    FIND_BRANCH          //foundbranch(self) is set to the selected branch.
                         //Always successful (because of the default otherwise)
    busy(self) := true

  if busy(self) then
    EXECUTE_ACTIVITY(foundBranch(self))

  if ¬busy(self) and branch_found then
    stop self
    RELEASE_PARENT
```

4.3.2.3. While Activity

A while activity is handled by a while agent. If the while condition is true, the while agent becomes busy immediately and starts executing the activity defined inside the while loop. The while agent remains busy until the activity is executed once, then busy becomes false again. Whenever the while agent is not busy (either in the beginning or when the execution of the activity is completed), the while condition is checked. If the while condition is still true the agent goes through the same steps again, otherwise the while agent releases its parent agent and terminates itself.

```
WHILEPROGRAM ≡
  if ¬busy(self) and true_while_condition then
    busy(self) := true

  if busy(self) then
    EXECUTE_ACTIVITY(activity(self)))
    //Executing the activity inside while
    //after it is executed once, busy becomes false

  if ¬busy(self) and false_while_condition then
    stop self
    RELEASE_PARENT
```

4.3.2.4. Pick Activity

A pick agent is responsible for handling a pick activity. A pick agent waits for an onMessage event or an onAlarm event to occur. To handle each of these two categories of events, the pick agent is assisted by two other DASM agents: pick message agent and pick alarm agent. A pick message agent is responsible for the onMessage events while a pick alarm agent is responsible for the onAlarm events. The pick agent starts by creating a pick alarm agent and a pick message agent, and then becomes busy, meaning that the agent is waiting for an event to happen. As soon as one or more events happen, the pick agent chooses the event that has happened first and sets chosenActivity to its associated activity. If two or more events happen at the same time, one of them is chosen non-deterministically[9]. Once the activity is chosen, the pick agent has to wait until the execution of the chosenActivity is completed. Upon completion, the pick agent releases its parent agent and terminates its execution.

```
chosenActivity: PICK_AGENT → ACTIVITY
//The activity that is chosen by a pick agent to be executed

PICKPROGRAM ≡
  if ¬busy(self) then
    if activity_is_not_chosen then
      CREATE_PICK_ALARM_AGENT          // To manage onAlaram events
      CREATE_PICK_MESSAGE_AGENT        // To manage onMessage events
      busy(self) := true          //The agent is waiting for an event to happen
    else
      RELEASE_PARENT
      stop self
  if busy(self)  then
    if activity_is_not_chosen then
      CHOOSE_EARLIEST_HAPPENED_EVENT
      //Among the onMessage and onAlarm events choose the one that occurred
      //first; chosenActivity is set to the corresponding activity of that event
    else
      EXECUTE_ACTIVITY(chosenActivity(self))
```

[9] According to the LRM, *"If the events occur almost simultaneously, there is a race and the choice of activity to be performed is dependent on both timing and implementation."* [10, Section 12.4]

Pick Message Agent

A pick message agent performs the following steps:

1- It notifies the inbox manager about all of the related onMessage events; i.e. it informs the inbox manager that any of the messages requested by the onMessage events can be assigned to this agent. This is done by adding one input descriptor for each onMessage event to the waiting set.

2- Once the inbox manager is informed, the pick message agent waits until one such event is completed; i.e. the corresponding message has been received.

3- It informs the pick agent that such an event has occurred and terminates itself.

Meanwhile, whenever an onAlarm event occurs (time-out) the pick message agent must terminate. However, before termination the pick message agent has to remove all the input descriptors (if any) from the waiting set informing the inbox manager that it is not waiting for such messages anymore.

```
PICKMESSAGEPROGRAM ≡
  if onAlarm_event_occured then
    REMOVE_ALL_ONMESSAGE_INPUT_DESCRIPTORS_FROM_WAITING_SET
    stop self
  else
    if ¬busy(self) then
      ADD_ALL_ONMESSAGE_INPUT_DESCRIPTORS_TO_WAITING_SET
      busy(self) := true
    else
      CHOOSE_A_COMPLETED_ONMESSAGE_EVENT_AND_INFORM_PICK_AGENT
        stop self
```

Pick Alarm Agent

A pick alarm agent is responsible for all of the onAlarm events defined inside a pick activity. Similar to the while activity, onAlarm events are defined in terms of a period of time ('for'), or a deadline ('until'). Therefore, the pick alarm agent has to record the starting time at the very beginning. As soon as the waiting period is completed or the deadline is passed, the pick alarm agent informs the pick agent and terminates thereafter. Note that it is possible that more than one alarm occur between two steps of a DASM. In

such a case the pick agent is informed about all the triggered alarms and it will pick the earliest one.

Moreover, if an onMessage event occurs while the pick alarm agent is waiting for an alarm to trigger, the pick alarm agent must terminate.

```
PICKALARMPROGRAM ≡
   if onMessage_event_occured then
      stop self
   else
      if ¬busy(self) then
         RECORD_ALARM_START_TIME
         busy(self) := true
      else
         FORALL_PASSED_ONALARM_EVENTS_INFORM_PICK_AGENT
            stop self
```

4.3.2.5. Flow Activity

Each flow activity is handled by a flow agent which concurrently executes the set of activities defined inside the flow activity. To allow concurrent execution, the flow agent is assisted by another type of DASM agents, called flow thread agent. Each flow thread agent is responsible for executing one of the concurrent activities.

A flow agent performs the following steps:

1- It creates one flow thread agent for each activity defined inside the flow activity and becomes busy. The flow agent keeps track of these thread agents by a set called *flow agent set*

2- The flow agent is completed when all of the flow thread agents are completed. After completion, the flow agent releases its parent agent and terminates itself.

```
flowActivitySet: FLOW → ACTIVITY-set
//Set of the activities defined inside a FLOW
//derived from the BPEL document and defined in the initial state

FLOWPROGRAM ≡
  if ¬busy(self) then
    //Creates threads to concurrently execute activities grouped inside the flow.
    forall activity in flowActivitySet(self)
      CREATE_A_FLOWTHREAD_AGENT_AND_ ADD_TO_FLOWAGENTSET(activity)
    busy(self) := true

  if busy(self) and empty_flowagentset then
    //All threads are done, flow activity is completed.
    RELEASE_PARENT
    stop self
```

The flow agent set has a main role in defining the behaviour of the flow activity. The thread agents are created and added to this set. When the threads are completed, they remove themselves from this set. When this set becomes empty, the flow agent will know that all of the threads are completed and the flow activity is completed as well.

A flow thread agent executes a single activity. Thus, its behaviour is very similar to a process agent, except that when completed, it has to remove itself from the flow agent set.

```
startedExecution: PROCESS ∪ FLOW_THREAD_AGENT → BOOLEAN
//initial value: false
//Tells whether a process or a flow thread agent has started executing
//its activity or not.

FLOWTHREADPROGRAM ≡
  if ¬busy(self) and ¬startedExecution(self) then
    startedExecution(self) := true
    busy(self) := true

  if busy(self) then
    EXECUTE_ACTIVITY(activity(self))

  if ¬busy(self) and startedExecution(self) then
    REMOVE_SELF_FROM_FLOWAGENTSET
    stop self
```

Chapter 5. Complete Formal Model

By refining the abstract model of Chapter 4, we obtain the intermediate model which provides the complete BPEL service abstract machine model of the core constructs of BPEL. The intermediate model forms the basis for deriving the executable model in Chapter 6. In the following sections, different parts of the intermediate model are described while the full model is available in Appendix C. In this chapter the goal is to clarify notable and principle parts of the intermediate model whereas details that are more related to making the formal model executable on real machines are considered in Chapter 6. Note that the required specifics on a given business process definition as extracted from the underlying BPEL document are encoded in terms of statically defined functions as part of the initial state of the DASM (see also Section 4.2 for details).

5.1. Inbox Manager

The refined inbox manager program presented in the intermediate model describes the behaviour of the inbox manager more concretely and in more detail. At this level of abstraction, the inbox manager uses the information carried by the input descriptors to identify the input activity (or event) that is waiting for the message. A message is matched to an input operation (activity or event) of a business process instance, if it satisfies the requirements specified by the waiting operation (including correlations). If the matching is successful, the message is assigned to that specific operation waiting in a process instance (or one of its subordinate agents). The refined inbox manager program is presented below.

```
INBOXMANAGERPROGRAM ≡
  if inboxSpace(self) ≠ Ø then
    choose p ∈ PROCESS, m ∈ inboxSpace(self),
           (agent, op) ∈ waitingForMessage(p) with match(p, op, m)
      Assign_Message(p, agent, op, m)
      Pick_Activity_Clearance(p, agent, op)

      if p = dummyProcess then
        new newDummy : PROCESS
          dummyProcess := newDummy
```

waitingForMessage is a function that identifies the set of waiting operations of each process. Moreover, match and ASSIGN_MESSAGE are refined as follows to incorporate additional parameters that identify the waiting operation and the agent to which it belongs, so that the message can be assigned to that specific activity/event.

```
IN_OPERATION ≡ RECEIVE ∪ INVOKE ∪ ONMESSAGE

waitingForMessage: PROCESS→ (RUNNING_AGENT X IN_OPERATION)-set

match: PROCESS X IN_OPERATION X MESSAGE → BOOLEAN
```

5.1.1. Assign Message

To assign a message to the correct process instance, the inbox manager has to deal with correlations in two basically different ways:

1- If an input operation belongs to a correlation group then the message must contain the appropriate correlation token values.

2- If the input operation is responsible for initiating a new correlation set then the inbox manager has to deal with initiating this correlation set.

The match predicate takes care of the first condition by considering the constraints imposed through correlations in matching a message with a process instance. Correlation initiation is managed by an INITIATE_CORRELATION[10] operation.

```
completedInOperations: PROCESS → (RUNNING_AGENT X IN_OPERATION
                                   X TIME)-set
//initial value: ∅

Assign_Message (p : PROCESS, agent : RUNNING_AGENT, op : IN_OPERATION,
                m : MESSAGE) ≡
   if initiateCorrelation(op) then
      INITIATE_CORRELATION(p, agent, op, m)

   remove m from inboxSpace(self)
   remove (agent,op) from waitingForMessage(p)
   add (agent, op, now) to completedInOperations(p)
```

When assigning a message to a process instance, the related input descriptor is removed from the waiting set (since the associated request is served and is not waiting anymore). In addition, a new descriptor containing the information on the input operation and its serving time is added to the completedInOperations set. For each process this set indicates the input activities (or onMessage events) that have received a message together with the receiving time. The role of the receiving time becomes clear when the behavior of the pick activity is discussed in Section 5.12. Assign_Message also updates the inbox space by removing the assigned message from it[11].

[10] In the absence of scopes and variables, the behaviour of INITIATE_CORRELATION is reduced to assigning values to a set of properties. Thus, this rule is left abstract at this level and is refined in the executable model.

[11] Please note that since this work does not capture the behaviour of variables, the value of a message is not explicitly assigned to a variable inside a process instance. According to the BPEL LRM, this is a valid behaviour for abstract business processes. However, the future work on the variable extension captures this issue.

5.1.2. Pick Activity Clearance

Pick_Activity_Clearance introduces a new responsibility for the inbox manager. In the abstract formal definition of the pick activity behaviour (Section 4.3.2.4), we mentioned that for each onMessage event, the pick message agent adds one input descriptor to the waiting set. According to the LRM, once one of these messages is received, the business process must not accept any of the other messages. Thus, it is required to remove the remaining input descriptors from the waiting set. Here we had two design choices:

1- Assigning this responsibility to the pick message agent; i.e. the pick message agent is responsible to remove the remaining input descriptors once a message is received.

2- Assigning this responsibility to the inbox manager; i.e. whenever the inbox assigns a message to a pick activity, it is responsible to remove the remaining unwanted input descriptors.

The second choice was preferred over the first one because there might be a delay between the time a message is assigned to an onMessage event and the time the pick message agent is informed. This delay may cause the inbox manager to accept another message for one of the other onMessage events which would violate the semantics of the business process as it is defined by the LRM. Pick_Activity_Clearance is responsible for such an action.

```
Pick_Activity_Clearance (p : PROCESS, a : RUNNING_AGENT,
                          op : IN_OPERATION) ≡
  if a ∈ PICK_MESSAGE_AGENT then
     forall (a, op') ∈ waitingForMessage(p) with op' ≠ op
        remove (a,op') from waitingForMessage(p)
```

5.2. *Outbox Manager*

In the outbox manager program, presented in Section 4.2, elements of the domain OUTPUT_DESCRIPTOR were used to access the required information on the outgoing message and its destination. In the intermediate model, we refine this domain by identifying the information carried by output descriptors more concretely. Similar to an input descriptor, we suggest that an output descriptor must identify a waiting activity and the waiting agent to which it belongs. The required information for generating an outbound message can then be extracted from the output descriptors. Thus, OUTBOX_DESCRIPTOR is refined as follows.

OUT_OPERATION ≡ REPLY ∪ INVOKE OUTBOX_DESCRIPTOR ≡ RUNNING_AGENT X OUT_OPERATION

As mentioned in Section 4.1, sending a message out also requires the BPEL abstract machine model to interact with the network abstract machine model through a well-defined interface. As the details of the composition of two models and the communication were not further detailed in this project, SEND is left abstract and must be considered in another refinement step where the network model and the BPEL service model are combined into a single DASM model.

5.3. *Execute Activity*

Section 4.2 described how a process agent executes its main activity, without actually defining Execute_Activity. According to the LRM, an activity can be any of the structured or basic activities. It is worth mentioning that for the purpose of this project we restrict the domain ACTIVITY to only include the core activities allowed in BPEL[12]. By focusing

[12] These activities include receive, reply, invoke, wait, terminate, empty, sequence, switch, while, pick and flow. Not included are the following activities: assign, throw, scope, and compensate.

on the core BPEL activities, we show how our approach to formally modelling the core activities solves the problem in principle. Basically, the same approach can be used in formal modelling of the remaining activities and concepts of BPEL defined by the LRM. An extension of the core model presented in this work, including all remaining concepts of BPEL, is being developed as part of another project in our group.

```
domain RECEIVE
domain SEQUENCE
... //and all other BPEL core activities
ACTIVITY ≡ REPLY ∪ RECEIVE ∪ INVOKE ∪ WAIT ∪ TERMINATE ∪ EMPTY
          ∪ SEQUENCE ∪ SWITCH ∪ WHILE ∪ PICK ∪ FLOW
```

The partial definition of Execute_Activity is presented below while the complete definition is available in Appendix C. To execute a basic activity the corresponding rule is invoked. For executing a structured activity, a new activity agent is created to handle that specific activity.

```
Execute_Activity(activitiy: ACTIVITY) ≡
  if linkStatusDefined then
  //checks whether the predecessors of the activity are completed or not
    if activityJoinCondition(activity) then
    //evaluates the join conditon defined by the activity
      if activity ∈ REPLY then
        Execute_Reply(activity)
      if activity ∈ RECEIVE then
        Execute_Receive(activity)

      ... //and all other basic activities

      if activity ∈ SEQUENCE then
        if assignedAgent(activity) = undef then
          new s : SEQUENCE_AGENT
            assignedAgent(activity) := s
            Initialize(s, activity)
```

```
        if activity ∈ FLOW then
           if assignedAgent(activity) = undef then
              new f : FLOW_AGENT
                 assignedAgent(activity) := f
                 Initialize(f, activity)
        ... //and all other structured activities

     else
        THROW_JOIN_FAILURE
       //JoinCondition is false. A fault (joinFailure) is thrown.
  //else
  //There are some activities linked to this activity that are not yet completed.
  //Therefore, the activity can not be executed yet.
  where
    linkStatusDefined ≡
              ∀x (x ∈ targetLinkSet(activity) → linkStatus(x) ≠ NOTDEFINED)
```

As defined in the above rule, before starting the execution of an activity, two conditions must be satisfied which are specified by two predicates: linkStatusDefined and activityJoinCondition. The first condition (linkStatusDefined) checks whether the activity is ready to be executed. The synchronization dependencies between concurrent activities introduced in the LRM suggest that each activity must be executed only after its predecessors, as defined by the related *links* (see Section 2.6.2), are completed. The first condition captures this behaviour and is further explored in Section 5.13.1. Moreover, as described in Section 2.6.2, each BPEL activity is defined along with a *join condition* and it can be executed only if the join condition is true. The second condition (activityJoinCondition) checks this requirement and according to the LRM, if this condition is not satisfied a fault must be thrown. This behaviour is captured in the intermediate model by THROW_JOIN_FAILURE, but is not further explored since fault handling is part of another project in our group[13].

In connection with structured activities, we define a function parentAgent for linking the parent agent and the subordinate activity agent. A process instance has one subordinate

[13] THROW_JOIN_FAILURE can be refined in the refinement step where faults are captured.

agent for each structured activity that is being executed inside it. For each activity agent, a derived dynamic function rootProcess is defined that returns the process instance to which the agent belongs. Furthermore, the root process has to keep track of all its subordinate agents. subordinateAgentSet is another derived dynamic function which provides the set of subordinate agents of a process instance. In order to identify the activity agents that are responsible for executing structured activities, we further define a function assignedAgent. For each structured activity that is being executed, this function indicates the agent that handles the activity. These functions are defined as follows.

```
parentAgent: RUNNING_AGENT → RUNNING_AGENT

rootProcess: RUNNING_AGENT → PROCESS

rootProcess(a: RUNNING_AGENT) ≡ { a                              : a ∈ PROCESS,
                                { rootProcess(parentAgent(a))    : otherwise.

subordinateAgentSet: PROCESS → ACTIVITY_AGENT-set

subordinateAgentSet(p: PROCESS) ≡ {a | a ∈ ACTIVITY_AGENT, rootProcess(a) = p}

assignedAgent: ACTIVITY → ACTIVITY_AGENT
```

The parentAgent relation is maintained by calling the Initialize rule. Whenever a new activity agent is created (either in an Execute_Activity rule or inside activity agents like the flow agent) Initialize is called. This rule also updates baseActivity which is the activity that must be executed by this activity agent.

```
baseActivity: ACTIVITY_AGENT→ ACTIVITY

Initialize (agent: ACTIVITY_AGENT, activity: ACTIVITY) ≡
   parentAgent(agent) := self
   baseActivity(agent) := activity
```

Figure 5-1 clarifies these relations, by illustrating the structure of an e-book Store business process instance (presented in Section 2.3) in our model and the relationships among its subordinate agents. An e-Book Store business process instance executes a sequence of activities which is handled by a sequence agent in our model. The sequence agent executes the activities one after another in the order of their appearance. The basic

activities defined inside the sequence, including receiving a request from a customer and sending a response to the customer are handled by basic ASM rules inside the sequence agent. However, to execute the flow activity which defined two invoke activities to be executed concurrently, the sequence agent creates a flow agent. The flow agent then assigns each of these concurrent activities to one flow thread agent and waits until both threads are completed. All the created agents form the subordinate agent set of the process instance and the process instance is the root process of all of them.

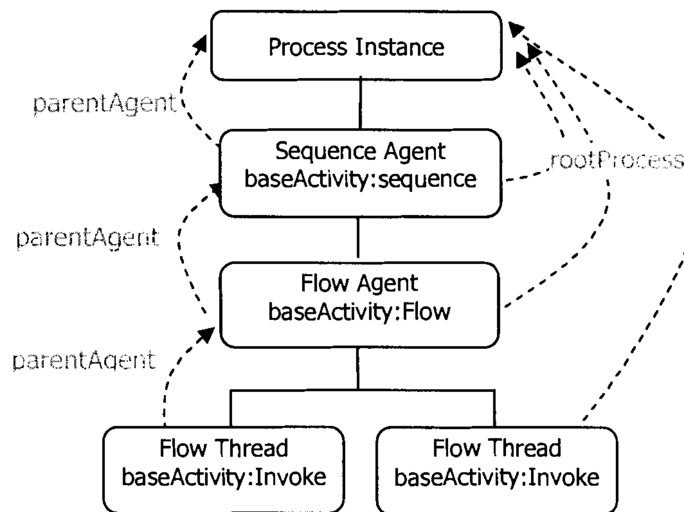


Figure 5-1 The structure of an e-book Store business process instance in our model

5.4. Receive Activity

In the abstract formal definition of Execute_Receive, presented in Section 4.3.1.1, some parts were left abstract. Here we take the next step to refine two abstract parts: ADD_INPUT_DESCRIPTOR_TO_WAITING_SET and message_is_received.

As mentioned before, to execute a receive activity of a given process instance, the inbox manager has to be informed that the process instance (or one of its subordinate agents) is waiting for a message. To accomplish this, we define a set called waitingForMessage for each process instance. This set indicates the input descriptors that provide the required information on the expected messages and the waiting agents. An input descriptor is

defined as (self, activity); therefore it identifies both the waiting agent and the waiting activity which contains the required information on the partner of the interaction (including port type and operation). Informing the inbox manager is performed by adding an inputDescriptor to the waitingForMessage set of the root process.

```
ADD_INPUT_DESCRIPTOR_TO_WAITING_SET(activity: ACTIVITY) ≡
   add inputDescriptor to waitingSet
   where inputDescriptor = (self, activity),
         waitingSet = waitingForMessage(rootProcess(self))
```

The inbox manager inspects the waiting set to identify the expected messages. Once one such message arrives, the inbox manager assigns it to the matching process instance. As described in Section 5.1, the inbox manager also removes the input descriptor from this set whenever the assignment is performed. Thus, the agent will be informed that the assignment is performed and the message is received, whenever the input descriptor is removed from the waiting set. The agent can then proceed with processing the message.

```
message_is_received(activity : ACTIVITY) ≡ inputDescriptor ∉ waitingSet
                                           where inputDescriptor = (self, activity)
```

The complete formal definition of Execute_Receive is presented in Appendix C.

5.5. *Reply Activity*

According to the abstract formal definition of Execute_Reply (presented in Section 4.3.1.2), to model a reply activity an output descriptor is placed in the outbox space of the outbox manager. An output descriptor is an element of the domain OUTPUT_DESCRIPTOR. Hence, based on the refinement introduced in Section 5.2, an output descriptor is defined as (self, activity). The reply activity contains the required information about the destination partner (including port type and operation).

Thus, ADD_OUTPUT_DESCRIPTOR_TO_OUTBOX_SPACE is refined as follows.

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

The complete formal definition of Execute_Reply is presented in Appendix C.

5.6. *Invoke Activity*

As mentioned in Section 4.3.1.3, invoke activity can be viewed as a combination of receive and reply. Therefore, the same refinements are applied to the invoke activity as well. For the detailed formal definition of Execute_Invoke the reader is referred to Appendix C.

5.7. *Terminate Activity*

As mentioned in Section 4.3.1.4, a terminate activity has to stop all the subordinate agents of a process instance as well as the instance itself. In the intermediate model, the derived function subordinateAgentSet identifies the subordinate agents of a process agent, so it is fairly simple to stop all of them.

```
STOP_ALL_SUBORDINATE_AGENTS ≡
   forall agent in subordinateAgentSet(rootProcess(self))
      stop agent
```

5.8. *Wait Activity*

The abstract formal definition of Execute_Wait (presented in Section 4.3.1.5), includes two abstract predicates (wait_just_started, wait_completed) and one abstract rule (RECORD_WAIT_START_TIME) that must be refined.

In the refinement step, we define two functions startTime and completionTime. startTime is used to record the starting time of a wait activity. Its initial value is *undef* and it is updated by the agent as soon as the execution of the wait activity is started. Therefore, if startTime is *undef* it means that the execution is just started.

```
startTime: WAIT → TIME
//initial value: undef

wait_just_started(activity: WHILE) ≡        startTime(activity) = undef
```

completionTime is a function that indicates the time when a wait activity is completed. If the wait activity uses 'until' to specify a deadline, then this function simply returns this deadline. On the other hand, if the wait activity is defined with 'for' and specifies a period of time, then this function uses the starting time to calculate the deadline.

```
completionTime: WAIT → TIME
```

In modelling the behaviour of a wait activity we are benefited from the abstract notion of time in real-time DASM as described in Section 3.2.3. The nullary monitored function *now* specifies the global system time and is well used in this refinement step. As soon as a wait activity is executed, startTime is updated and records the current time denoted by the monitored function *now*.

```
RECORD_WAIT_START_TIME(activity: WHILE) ≡  startTime(activity) := now
```

The completion of a wait activity is determined by checking its completionTime against the current time. If the completion time reaches (or is passed[14]) then the wait activity is completed.

```
wait_completed(activity: WHILE) ≡        completionTime(activity) ≤  now
```

The complete formal definition of Execute_Wait is presented in Appendix C.

---

[14] Note that the completion time may reach in between of two DASM steps; hence it can be less than the current time.

5.9. Sequence Agent

One important task of a sequence agent is to fetch the activities defined in the corresponding sequence activity one by one. In the intermediate model, fetching the activities is performed by a function called sequenceCounter. Each time it is called, the sequence counter indicates the next activity in the sequence and this activity is recorded as the current activity.

```
sequenceCounter: SEQUENCE→ ACTIVITY
//returns undef when reaches the end of the list of activities
//derived from the BPEL document and defined in the initial state

SET_CURRENT_ACTIVITY_TO_NEXT_ACTIVITY ≡
                        currentActivity(self) := sequenceCounter(baseActivity(self))
```

Another refinement that is needed in the sequence agent program is for determining the completion of the sequence activity. The sequence is completed when the last activity defined in the sequence is completed. We assume that sequenceCounter returns *undef* when it reaches the end of the list of activities. Thus, the abstract predicate sequence_is_not_completed is refined as follows.

```
sequence_is_not_completed ≡      currentActivity(self) ≠ undef
```

It is also important to refine the abstract rule RELEASE_PARENT. As mentioned in Section 4.3.2, all the activity agents have to release their parent agents after completion. Thus, RELEASE_PARENT is refined as follows.

```
RELEASE_PARENT ≡          busy(parentAgent(self)) := false
```

The complete program of the sequence agent is presented in Appendix C.

5.10. Switch Agent

In the abstract switch agent program (presented in Section 4.3.2.2), one main issue which is finding the correct branch (FIND_BRANCH) was left abstract. Refining the abstract rule

FIND_BRANCH requires searching through all switch case elements and finding the first one with a true condition. In this refinement step we introduce the following functions and predicate.

```
swCaseSet: SWITCH → SWCASE-set
//derived from the BPEL document and defined in the initial state

swPrority: SWCASE → PRIORITY
//derived from the BPEL document and defined in the initial state

swCaseCondition: SWCASE → BOOLEAN
```

swCaseSet indicates the set of case elements defined inside a switch activity plus *otherwise* (or default *otherwise*). swCaseCondition evaluates the condition of a specific case element. In case of an *otherwise* it always returns true. To keep the order of the case elements and find the first one with a true condition, a priority is assigned to each case element. swPriority indicates the priority of a specific case element. The element with the highest priority is the first case element.

FIND_BRANCH is refined using the above mentioned functions and predicate, as follows.

```
foundBranch: SWITCH_AGENT → ACTIVITY

FIND_BRANCH ≡
  let caseSet = swCaseSet(baseActivity(self)) in
    choose c ∈ caseSet with (swCaseCondition(c) ∧
          ∀x ((x ∈ caseSet ∧ swCaseCondition(x))→ swPriority(c) ≥ swPriority(x)))
      foundBranch(self) := swCaseActivity(c)
```

To find the right branch the agent investigates the set of all case elements (caseSet), chooses the element with a true condition and ensures that the chosen branch is the one with the highest priority. The foundBranch function was defined in Section 4.3.2.2 and the signature is presented here again. Once the correct branch is selected, foundBranch is updated with the associated activity of the selected branch. As mentioned before, it is supposed that the *otherwise* branch is also an element of the case set with an always-true condition and the lowest priority. Therefore, if none of the conditional branches is selected, *otherwise* will be automatically selected.

Another abstract part of the definition of Section 4.3.2.2 is the branch_found predicate which indicates whether any branch is selected or not. The initial value of foundBranch is *undef* and it remains *undef* until a branch is selected. Therefore, the abstract predicate branch_found can be refined as follows.

```
foundBranch: SWITCH_AGENT → ACTIVITY
//initial value: undef

branch_found ≡      foundBranch(self) = undef
```

The complete formal definition of the switch agent program is presented in Appendix C.

5.11. While Agent

To refine the abstract while agent program presented in Section 4.3.2.3, we have to define a way to evaluate the condition of a while activity. This is performed by the waCondition predicate. Thus, the abstract predicates true_while_condition and false_while_condition are refined as follows. The complete formal definition of the while agent program can be found in Appendix C.

```
waCondition: WHILE → BOOLEAN

true_while_condition≡ waCondition(baseActivity(self))

false_while_condition ≡ ¬waCondition(baseActivity(self))
```

5.12. Pick Agent

As defined in the abstract pick agent program (Section 4.3.2.4), a pick agent starts with creating a pick message agent and a pick alarm agent abstractly performed by CREATE_PICK_ALARM_AGENT and CREATE_PICK_MESSAGE_AGENT. In the intermediate model these rules are refined as follows.

```
CREATE_PICK_ALARM_AGENT ≡
  new a : PICK_ALARM_AGENT
    Initialize(a, baseActivity(self))

CREATE_PICK_MESSAGE_AGENT ≡
  new b: PICK_MESSAGE_AGENT
    Initialize(b, baseActivity(self))
```

The next step is to refine the abstract predicate activity_is_not_chosen. The initial value of chosenActivity is *undef*, thus activity_is_not_chosen is refined as follows.

```
chosenActivity: PICK_AGENT → ACTIVITY
//initial value: undef

activity_is_not_chosen ≡   chosenActivity(self) = undef
```

As described in the abstract model, when any of the onAlarm or onMessage events occurs, the corresponding agent will inform the pick agent. In the refined model, this information exchange takes place through a set, called triggeredEvents. When an event occurs, the responsible agent records the occurrence of the event together with its occurrence time[15] in this set.

```
triggeredEvents: PICK_AGENT → (EVENT X TIME)-set
//initial value: ∅
```

The pick agent then checks this set to choose the event that happened first and executes its corresponding activity. The abstract rule CHOOSE_EARLIEST_HAPPENED_EVENT is refined as follows where onEventActivity is a function that specifies the activity associated with an event.

```
onEventActivity: EVENT → ACTIVITY
//derived from the BPEL document and defined in the initial state
```

[15] The occurrence time for an onMessage event is the time when the message is received, and in case of an onAlarm event it is the time when the alarm is triggered.

```
CHOOSE_EARLIEST_HAPPENED_EVENT ≡
   choose (event, time) ∈ triggeredEvents(self) with
                                        ∀e∀t ((e,t) ∈ triggeredEvents(self) → time ≤ t)
      chosenActivity(self) := onEventActivity(event)
```

5.12.1.Pick Message Agent

Refining the pick message agent program is fairly simple, since its behaviour is very similar to a receive activity. In the first step, input descriptors must be created for all onMessage events and added to the waitingForMessage set, as follows. onMessageSet is a function that specifies all the onMessage events listed inside a pick activity.

```
onMessageSet: PICK → ONMESSAGE-set
//derived from the BPEL document and defined in the initial state
ADD_ALL_ONMESSAGE_INPUT_DESCRIPTORS_TO_WAITING_SET ≡
   forall event ∈ onMessageSet(baseActivity(self))
      let inputDescriptor = (self, event) in
         add inputDescriptor to waitingForMessage(rootProcess(self))
```

The pick message agent will then wait until one of the messages is received to record the event occurrence together with the occurrence time. The occurrence time is extracted from the completedInOperations set, where the inbox manager records the completed input operations together with their completion time. Note that once one of these events happens, the inbox manager prevents the rest of them from happening by removing them from the waiting set (see Section 5.1 for details).

```
CHOOSE_A_COMPLETED_ONMESSAGE_EVENT_AND_INFORM_PICK_AGENT ≡
   choose event ∈ onMessageSet(baseActivity(self)) with
            (self, event, time) ∈ completedInOperations(rootProcess(self))
      add (event, time) to triggeredEvents(parentAgent(self))
```

Meanwhile, if an onAlarm event occurs while the pick message agent is waiting for an onMessage event, the pick message agent must terminate. Every event registers itself in the triggeredEvents set whenever it occurs, so if the triggeredEvents set is not empty it means that an onAlarm event has already happened.

```
onAlarm_event_occured ≡  triggeredEvents(parentAgent(self)) ≠ ∅
```

In addition to termination, the pick message agent has to remove all the input descriptors from the waiting set so that the inbox manager is informed that the agent is not waiting for those messages anymore.

```
REMOVE_ALL_ONMESSAGE_INPUT_DESCRIPTORS_FROM_WAITING_SET ≡
  forall event ∈ onMessageSet(baseActivity(self))
    let inputDescriptor = (self, event) in
      remove inputDescriptor from waitingForMessage(rootProcess(self))
```

5.12.2.Pick Alarm Agent

As mentioned in the abstract model, a pick alarm agent has to record its starting time to calculate the time when each alarm is triggered and inform the pick agent as soon as an onAlarm event occurs. Keeping track of the alarms is very similar to what is done for a wait activity (Section 5.7). Hence, the definition of the startTime function is extended to capture pick alarm agents as well as wait activities.

```
startTime: WAIT ∪ PICK_ALARM_AGENT → TIME

RECORD_ALARM_START_TIME ≡  startTime(self) := now
```

The pick alarm agent is then responsible to check all the alarms and see which one is due. The completion time of an onAlarm event is determined by a function, called triggerTime, with a similar functionality as completionTime defined for a wait activity (Section 5.7). triggerTime indicates the time when an alarm event is triggered. If the alarm uses 'until' to specify a deadline, this function simply returns the deadline. If the alarm is defined with 'for' and specifies a period of time, the function has to know when the alarm was started; i.e. the deadline is calculated using the starting time (second parameter).

```
triggerTime: ONALARM  X  TIME → TIME
```

The pick alarm agent waits for any alarm to occur and informs the pick agent about the triggered alarms as soon as an alarm is triggered, as follows.

```
FORALL_PASSED_ONALARM_EVENTS_INFORM_PICK_AGENT ≡
   forall event ∈ onAlarmSet(baseActivity(self)) with
                                          triggerTime(event, startTime(self)) ≤ now
      add (event, triggerTime(event, startTime(self))) to
                                                 triggeredEvents(parentAgent(self))
```

Moreover, the pick alarm agent must terminate if an onMessage event occurs while the alarm agent is waiting for an alarm to trigger. Analogous to the pick message agent, the pick alarm agent checks the triggeredEvents set to be informed if an onMessage event occurs.

```
onMessage_event_occured ≡      triggeredEvents(parentAgent(self)) ≠ ∅
```

For a complete formal definition of the pick agent program, the pick message agent program, and the pick alarm agent program we refer the reader to Appendix C.

5.13. Flow Agent

In Section 4.3.2.5, where the abstract flow agent program was presented, we mentioned the key role of the flow agent set. In the intermediate model, this set is concretely defined as a set of flow thread agents belonging to a flow agent that makes it easy to determine whether it is empty or not.

```
flowAgentSet: FLOW_AGENT→ FLOW_THREAD_AGENT-set
// initial value: Ø

empty_flowagentset ≡      flowAgentSet(self) = Ø
```

As discussed before, the flow agent program starts with creating one flow thread agent for each activity defined inside the flow and adding these thread agents to the flow agent set. This behaviour is refined in the intermediate model as follows.

```
CREATE_A_FLOWTHREAD_AGENT_AND_ ADD_TO_FLOWAGENTSET(activity:
ACTIVITY) ≡
   new fThread : FLOW_THREAD_AGENT
      Initialize(fThread, activity)
      add fThread to flowAgentSet(self)
```

The flow agent program is completed when all of these flow thread agents complete the execution of their associated activities. Once each thread is completed, it has to remove itself from the flow agent set (abstractly defined by REMOVE_SELF_FROM_FLOWAGENTSET rule in Section 4.3.2.5). Thus, the flow agent program is completed when the flow agent set becomes empty.

```
REMOVE_SELF_FROM_FLOWAGENTSET ≡
   remove self from flowAgentSet(parentAgent(self))
```

The complete formal definitions of the flow agent and the flow thread program are presented in Appendix C.

5.13.1.Link Semantics

As described in Section 2.6.2, the LRM allows synchronization dependencies to be defined between concurrent activities by introducing link semantics. An activity can be the source of a set of links or it can be the target of a set of links. sourceLinkSet and tragetLinkSet are two functions that indicate these sets of links respectively.

```
sourceLinkSet: ACTIVITY → LINK-set
//derived from the BPEL document and defined in the initial state

targetLinkSet: ACTIVITY → LINK-set
//derived from the BPEL document and defined in the initial state
```

Initially the status of all outgoing links is NOTDEFINED. linkStatus is a function that indicates the status of a single link. When activity *A* completes, certain steps are performed to determine the effect of the synchronization links on further execution. First, the status of all outgoing links of *A* must be determined. The status will become either POSITIVE or NEGATIVE.

```
linkStatus: LINK → {POSITIVE, NEGATIVE, NOTDEFINED}
//initial value: NOTDEFINED

linkTransitionCondition: LINK → BOOLEAN
```

To determine the status of each link its transition condition is evaluated. linkTransitionCondition is a predicate that evaluates this condition. If the condition is evaluated to true the status will be POSITIVE, otherwise it will be NEGATIVE. This behaviour is captured in our model by the Synchronization rule.

```
Synchronization(activity : ACTIVITY) ≡
  forall link ∈ sourceLinkSet(activity)
    if linkTransitionCondition(link)  then
      linkStatus(link) := POSITIVE
    else
      linkStatus(link) := NEGATIVE
```

Synchronization must be performed whenever the execution of an activity is completed, either in basic activities or structured activities. In the complete formal definition of the intermediate model presented in Appendix C, this rule is called at the end of all basic activity execution rules and whenever an activity agent is terminated.

Second, if an activity *B* which has a synchronization dependency on *A* is ready for execution, and the status of all incoming links of *B* is determined, the join condition of *B* is evaluated. If the join condition is true then *B* starts its execution, otherwise a standard fault is thrown. These pre-requisites are captured in Execute_Activity, and were introduced in Section 5.1.

```
linkStatusDefined ≡ ∀x (x ∈ targetLinkSet(activity) → linkStatus(x) ≠ NOTDEFINED)

activityJoinCondition: ACTIVITY → BOOLEAN
```

linkStatusDefined is a predicate that ensures the status of all the incoming links of an activity have been determined before executing that activity. If so, activityJoinCondition is evaluated and the activity can be executed only if the condition is true.

Chapter 6. Executable Model

This section introduces an abstract executable semantics of BPEL obtained from the intermediate model as the result of another refinement step. Experimental validation of abstract requirement specifications provides us with an effective instrument to further eliminate undesirable behaviour and hidden side effects in early design stages [18]. In combination with analytical techniques, simulation and testing can provide valuable feedback for establishing key system attributes and exploring alternative design choices. In this project, we use AsmL [1] for this purpose.

6.1. Introduction to AsmL

AsmL, developed by the Foundation of Software Engineering Group at Microsoft Research [16], is a high level executable specification language based on the concept of ASM. AsmL specifications are executable, hence they can be used to test and validate the specifications itself [19]. Experimental validation is widely accepted for checking the conformance of the specifications to the requirements [17], [18]. In most cases, only execution can reveal many loose ends and ambiguities both in the formal specification and the informal requirements [17]. Moreover, a machine compiler can effectively detect possible syntactic errors in the formal specification. AsmL provides us with the means to investigate specifications both syntactically and semantically. Besides, AsmL is integrated with Microsoft software development and run-time environments which facilitate creating useful user interfaces to simulate and test AsmL executable specifications.

AsmL is a rich language and although its advance language constructs are definitely helpful in rapid prototyping and object oriented software development [19], for the purpose of this project we actually need a subset of the language which is as close as possible to the pure ASMs. For modeling the BPEL semantics, a tight relation between the full DASM model and the executable model is of utmost importance. Though, in order to be executable some changes and additions were inevitable [15].

6.2. The AsmL Model

Refining the DASM model of Chapter 5 to the AsmL executable model requires certain considerations with respect to the translation aspects, refining abstract parts of the model, dealing with the underlying communication model and achieving a useful method of visualization.

Intuitively, the AsmL encoding splits into five separate modules, each of which deals with a basically different aspect: (1) the original model (2) the internal structure (3) the refinement of the original model (4) GUI-related extensions, and (5) the communication model. It is worth mentioning that the current executable model does not cover the complete intermediate model. Our main goal was to establish a minimal, yet principle executable model to reveal the feasibility of achieving such a model through refinement. Through such model we also show the importance of executable specifications in early design stages and prove how simulation and testing using such a model provide useful feedbacks to establish key system attributes. As a result, the current executable model captures the behaviour of main entities of the intermediate model including the inbox manager, the outbox manager, process instances, sequence agents, flow agents, receive and reply. The complete executable model is under development as part of another project in our group.

6.2.1. Original Model

The original model is basically the translation of the intermediate model to AsmL. The DASM model of the BPEL semantics is subject to changes and several iterations due to the dynamics in the development of the BPEL in an industrial setting. Hence, to maintain the tight relation between the DASM model and the executable model, it is necessary to keep the executable model as simple as possible. The advanced constructs of AsmL are beneficial for structuring complex models; however, in this project our main challenge was to keep the executable close to the pure ASM by using a subset of the language. The following provides an example of the translation from the intermediate model to the executable model for the sequence agent.

The sequence agent program is defined as follows in the intermediate model.

```
SEQUENCEPROGRAM ≡
   if ¬busy(self) then
      currentActivity(self) := sequenceCounter(baseActivity(self))
      busy(self) := true
   else
      if currentActivity(self) ≠ undef then
         Execute_Activity(currentActivity(self))
      else
         stop self
         busy(parentAgent(self)) := false
```

In the executable model the sequence agent program is translated to the following.

```
public class SEQUENCE_AGENT extends ACTIVITYAGENT
  var currentActivity as ACTIVITY? = undef
//---------------------------SequenceProgram------------------------------------------
  override Program()
    match baseActivity
    //required type checking in AsmL
      baseAct as SEQUENCE:
        if not busy then
          currentActivity := sequenceCounter(intStr, baseAct)
          busy := true
        else
          if not (currentActivity = null) then
            Execute_Activity(me, currentActivity)
          else
            stop(me)
            parentAgent.busy := false
```

In the executable model a new parameter is introduced for the sequenceCounter function which refers to the internal structure described in the next section.

6.2.2. Internal Structure

The internal structure acts as an interface between our abstract machine model and the BPEL definition of the business process. In order to execute a process instance, we need a way of accessing the definition of the business process. Normally, each process instance is executing an activity as defined in the BPEL process definition and further determined by the history of that specific instance. As discussed in Section 4.2, we assume that the required information on a given business process is extracted from the underlying BPEL document and is formalized as function definitions in the initial state. To make the model executable, we have to resolve the abstractions and define such functions explicitly. In the executable model, the internal structure provides the specific information on a business process based on the BPEL process definition. For instance, consider the sequenceCounter function in the sequence agent program of the previous section (Section 6.2.1). As described in Section 5.9, this function yields the activities specified inside a sequence one by one. Abstractly, it operates on the definition of the business process as provided by the internal structure of the executable model.

The internal structure is defined as an AsmL *interface*[16] [1] which allows different implementations of it. This project uses an array-based implementation of the internal structure where the definition of a business process is hard-coded in the internal structure while other possible implementations can be considered in the future. The definition of the interface and some of the methods provided by the internal structure are presented below. The reader is referred to Appendix D for more details.

```
interface INTERNAL_STR
  sequenceCounter(s as SEQUENCE) as ACTIVITY?
//returns the next activity to be executed in a sequence agent

  processActivity() as ACTIVITY
//returns the main activity of the process

  accept(activity as ACTIVITY, m as MESSAGE) as Boolean
//returns true if the message has the correct type as required by the activity

  taggedWithCorrelation(activity as INPUT_ACTIVITY) as Boolean
//returns true if the activity is associated with a correlatin set

  initiateCorrelation(activity as INPUT_ACTIVITY) as Boolean
//returns true if the correlation set associated with the activity
//must be initiated
  ...
```

6.2.3. Execution-Specific Additions to the ASM Model

In the stepwise refinement of the original model, abstract parts are refined depending on their role in the model, either by introducing non-determinism or assigning clear deterministic behaviour to them. In some cases, complex substructures had to be introduced. For example, in order to model the correlation behaviour in a business process instance, we need a structure for correlation sets mapping properties to their values. This structure completely complies with the definition of the correlation sets in the LRM. In addition, a predicate is defined to check the compatibility of a message to a

[16] AsmL interfaces provide a vocabulary (or type signature) without implementation [1].

correlation set, i.e. to check whether the message contains the required correlation token values or not. The abstract definition of the correlation sets of the intermediate model and the refined definition of the executable model are both presented below. It is worth mentioning that although the intermediate model deals with the correlation sets abstractly, this refinement is necessary in the executable model in order to deal with the correlation values.

Correlation sets are defined by the CORRELATIONSET domain in the intermediate model.

```
domain CORRELATIONSET
```

The definition of the correlation sets in the executable model is as follows.

```
class CORRELATIONSET
  var properties as Map of String to DATA

  messageContainsTokens(m as MESSAGE) as Boolean
//This method checks the compatibility of a message to a correlation set.
Normally, we should check if the message carries the correlation token values.
```

6.2.4. GUI-Related extensions

An executable model needs a GUI that makes it a useful tool for user-controlled simulation and testing. The GUI is written in Visual C# .NET[17]. By utilizing AsmL's APIs with C#, we were able to integrate the model with its GUI, by defining an appropriate interface called *View*. Figure 6-1 shows the current version of GUI, capturing the state of a set of business process instances.

---

[17] Microsoft Visual C# .NET, Microsoft Development Environment

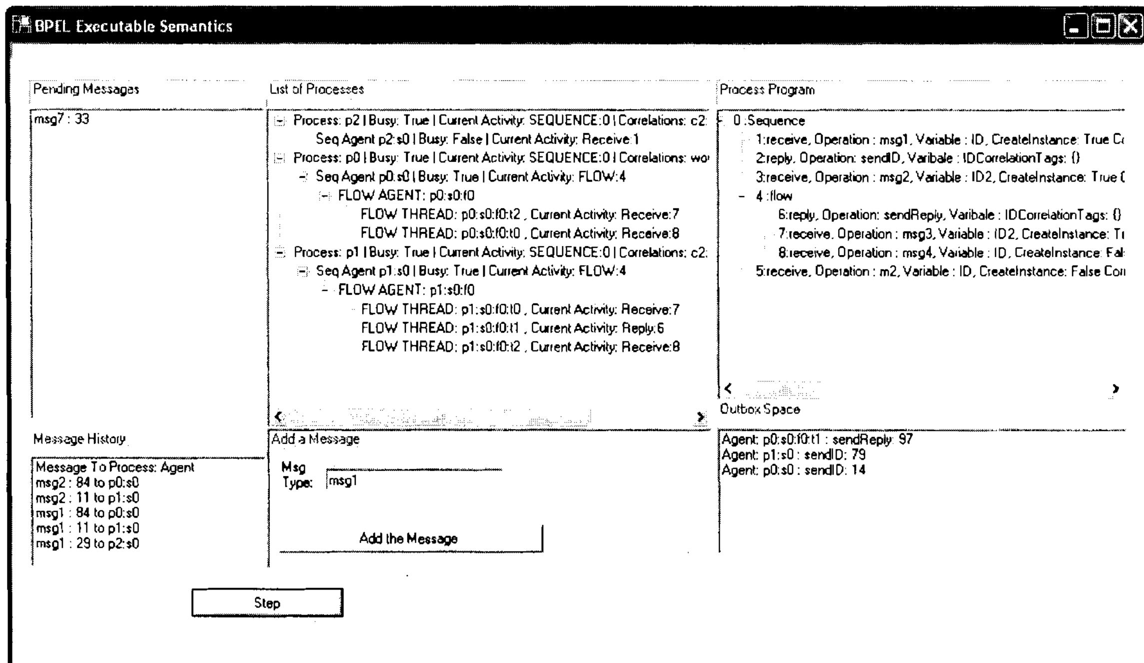


Figure 6-1 Graphical user interface of a sample AsmL model

There is a *list of processes* that shows the process instances together with their internal states. *Pending Messages* shows the messages that had arrived at the Web service and were placed in the inbox space of the inbox manager. *Outbox space* shows a list of output descriptors that are created and left in the outbox space of the outbox manager. Finally, *message history* keeps track of the messages that have been assigned to different instances of the business process.

Obviously, to establish the required connections between the AsmL model and the user interface, certain methods and interfaces are needed. *View* is the class that provides the graphical user interface and is written in C#. Its integration into the AsmL model needs appropriate interfaces, as defined below.

```
 [External]
class View
  public refreshMessageList(mArray as ArrayList)
  public refreshProcessTreeView(mArray as ArrayList)
  public refreshMsgHistoryList(mArray as ArrayList)
  public refreshOutspaceList(mArray as ArrayList)
  public setProgramBox(mArray as ArrayList)

class MODEL
  var view as View
```

The AsmL model uses these methods to interact with the GUI. For example, as mentioned the GUI needs to keep track of the messages that have been assigned to different business process instances. To maintain this list, and to keep it up-to-date, the model records all the assignments done in one step of DASM and informs the GUI at the end of each step. For details of such an interaction please refer to Appendix D.

6.2.5. Communication Model

As mentioned in Section 4.1, this project concentrated on the service model while assuming the abstract network model of [20] as the core of the underlying communication model. The composition of the service model and the network model is well-defined, as the inbox space and the outbox space perform as the interfaces of a Web service with the outside world and are naturally viewed as two mailboxes for the network model. Although an executable version of the network model [20] is being developed in our group, a full composition of the two models requires taking into account the specific message bindings and service bindings as mentioned in Section 4.1. In the current executable model the interactions with the network model, which includes receiving a message and sending a message as well as the corresponding transformations according to WSDL binding rules, are done manually. However, as this problem is addressed by other projects of the group, considering both the network model and the service model, we are confident that the full composition will be accomplished in the near future.

6.3. Experimental Validation

A receive activity is a "*blocking activity in the sense that it will not complete until a matching message is received by the process instance.*" [10, Section 11.4] Therefore, it is implicitly assumed that a matching message will arrive after the corresponding receive activity has been executed. Consider the following activity in a business process:

```
<sequence>
    <activity1>
    <activity2>
    ...
    <receive partnerLink="PL1" portType="PT1" operation="OP1">
</sequence>
```

Suppose that when a process instance is executing activity2, a message arrives from *partnerLink* PL1 using *portType* PT1 and *operation* OP1. Since the process instance has NOT executed the receive activity yet, it is not waiting for this message. It is not clear from the LRM what happens to such a message. Indeed, there could be multiple choices:

- Buffer: The message can be stored in a buffer, so that the receive activity can fetch it later.
- Discard: The message can simply be discarded, when there is no receive activity waiting for it.
- Fault: A fault can be thrown since the Web service has received a message for which no process instance is waiting.

It is certainly important for the LRM to distinguish among these choices, since it will cause inconsistencies in the behaviour of different implementations of the language.

This problem was discovered during experimental validation, when our inbox manager received a message that no process instance was expecting at the time.

Chapter 7. Critical Analysis of BPEL

The goal of the LRM is to establish the key system attributes of the Web services architecture for automated business processes as a basis for the development of e-business applications. However, a careful analysis of the language points out a number of weak points and open issues in the language definition. The goal of the OASIS technical committee is to discover and address these weak points, so that the language becomes more robust and clear. To this date, the technical committee has listed 97 issues and has resolved a number of them. In this chapter, we mention the weak points that were discovered in this project through modelling key system attributes of the language. Some of them have been already acknowledged in the WSPEL TC issue list [34].

7.1. Ambiguities

The definitions and constraints introduced for different constructs of the language are, in many cases, scattered and not centralized. The lack of a formal organization, together with the imprecise nature of the natural language definitions causes inevitable ambiguities and uncertainties in the language. We present two such ambiguities here.

7.1.1. Correlations

The LRM states that "*After a correlation set is initiated, the values of the properties for a correlation set must be identical for all the messages in all the operations that carry the correlation set and occur within the corresponding scope until its completion*" [10, Section 10.2] Logically, the operations that carry the correlation sets can be categorized

into two basically different groups: input activities, including receive, invoke, and pick, and output activities, including reply and invoke. Therefore, we can decompose the above consistency constraint into two separate constraints: (1) the property must hold on all input activities; (2) the property must hold on all output activities.

To see that the first constraint is satisfied is trivial. The LRM clearly specifies that a message must carry the required correlation tokens in order to be accepted by the process instance. This is true for every input activity. In our model, the inbox manager fulfils this duty. A message will be assigned to a process instance only if it "matches" the process instance; thus, it must carry the required correlation token values.

The second property, however, requires a closer investigation. This property can itself be decomposed to two sub-properties: (2.1) the property must hold in all output activities, where the correlation is initiated by the same output activity; (2.2) the property must hold in all output activities where the correlation set is already initiated.

(2.1) is confirmed by the LRM as well. The correlation set will be initiated and the correlation tokens get their values from the message that is to be sent out. For (2.2), the language does not provide enough details to prove or falsify the second property.

In case of incoming messages, the business process is capable of filtering the messages; i.e. it will only pick those messages that match the correlation. On the other hand, in case of outgoing messages, the business process has no responsibility other than sending the message out. Although the LRM defines the semantics of a process that violates this consistency constraint as undefined, it is not precisely mentioned that output activities (like input activities) are blocking activities, and thus the ambiguity leads to further problems as follows.

7.1.2. Synchronous Receive/Reply

According to the LRM *"A reply activity is used to send a response to a request previously accepted through a receive activity. Such responses are only meaningful for*

synchronous interactions." [10, Section 11.4] In order to clarify a request/response interaction, the BPEL LRM states that *"The correlation between a request and the corresponding reply is based on the constraint that more than one outstanding synchronous request from a specific partner link for a particular portType, operation and correlation set(s) MUST NOT be outstanding simultaneously."* [10, Section 11.4] Although the definition of "outstanding" is not elucidated in the LRM, according to its interpretation by WSBPEL TC ([34, issue #26]), one can assume that an outstanding synchronous receive is a receive activity for which the required message has arrived but the reply is not sent out yet. Therefore, the following must be permissible:

```
<receive partnerLink="PL1" portType="P1" operation="O1"
correlation="C1">
<receive partnerLink="PL1" portType="P1" operation="O1"
correlation="C2">
<reply partnerLink="PL1" portType="P1" operation="O1">
```

Assuming that operation O1 is an input-output operation, these two receive activities start two synchronous request/response transactions, and as the correlation sets of these receive activities are different, these two transactions are valid to be outstanding concurrently. The problem arises when a reply message is sent to the same partner without specifying any correlation set. This is a valid reply. The problem in this case is that it is impossible to determine to which receive activity this reply is coupled; it is not clear which request/response is still outstanding and which one is not [15].

7.2. Loose Ends

The BPEL reference manual suffers from the lack of precision caused by natural language definitions. The lack of formalism makes it difficult to delineate the scope of the language and to identify the interfaces of a business process with the outside world. This problem inevitably causes missing points and loose ends. This section presents two examples of such loose points which were discovered through the modelling process.

7.2.1. Partners Communication

The LRM defines the communication between a business process and its partners through partner links (see Section 2.7.1). For example, in the e-Book store example (Section 2.7), the communication between the business process and the shipping company is defined by identifying the port type of the shipping company Web service which is responsible for the communication. Hence, in order to establish a conversation, it is important for the message to carry the required information about the partner link, the port type and the operation to which it belongs. Suppose that a business process invokes a partner service by sending a message using the correct port type and operation of that service. The business process then waits to receive a call-back from that service, which is sent back using the same port type and operation. The inbox manager, or any other entity responsible for assigning the messages to the business process instances, needs a mechanism to distinguish this message from all the other messages that arrive at the business process. The LRM does not specify how such a conversation is established. It is not clear whether there is a mechanism to build a fixed communication channel between two partners or how the underlying messaging protocol must carry the required information along with the messages. The LRM does not specify any requirements for the underlying messaging protocol. It states that *"BPEL4WS depends on the following XML-based specifications: WSDL 1.1, XML Schema 1.0, XPath 1.0 and WS-Addressing. Among these, WSDL has the most influence on the BPEL4WS language."* [10, Section 3] Nevertheless, WSDL is used in conjunction with different messaging protocols (e.g. SOAP 1.1, HTTP GET/POST, and MIME) and does not impose the protocol to carry such information. Thus, this requirement must indeed be clarified by the LRM.

7.2.2. Re-Initiating a Correlation Set

In Section 2.2 we introduced the notion of a start activity and described its role in the life cycle of a business process. The LRM states that *"The only way to instantiate a business process in BPEL4WS is to annotate a receive activity with the* `createInstance` *attribute*

set to "`yes`*" (see 12.4. Pick for a variant). The default value of this attribute is "*`no`*". A receive activity annotated in this way MUST be an initial activity in the process, that is, the only other basic activities may potentially be performed prior to or simultaneously with such a receive activity MUST be similarly annotated receive activities.*"[10, Section 11.4]

Now, consider a Web service that receives a number of commands from different users and then performs them one at a time. A business process instance is created for each user that communicates with the system (and has a unique ID). Once a business process instance is created it handles all the commands from that specific user. The definition of such a business process is presented here in a pseudo-code-like style.

```
PROCESS
    WHILE
          RECEIVE
                MESSAGE(ID, command),
                createInstance="yes",
                CORRELATION(initiate="yes",ID)
```

This is a valid process definition and satisfies the condition of having at least one start activity. However, the problem arises when we want to assign messages to this business process. Consider the following scenario. A message *m1* = (12, "start") arrives and accordingly a new business process instance *p1* is instantiated and a correlation set is initiated (c = {ID: 12}). From now on, the messages with the same correlation token value (12) are expected to be assigned to this process instance, so if a message *m2* with same correlation token value (12) arrives, it is expected to be assigned to the same business process.

To check whether the message can be assigned to the existing process instance, the message must be checked against its correlation set; i.e. *m2* has to carry the required correlation token value. The matching procedure can be defined as follows:

```
if m2 has the correct message type then
    if activity has a correlation set attribute then
        if this correlation set is tagged with initiation then
            if the correlation is not yet initiated then
                m2 matches p1
            else
                SPECIAL-CASE //The matching is done, only if the
                             //waiting activity has certain properties
        else
            if the correlation set is initiated then
                if m2 carries the required correlation token values then
                    m2 matches p1
                else
                    m2 does NOT match p1
            else
                Error. The semantics is undefined
    else // No correlation exists, so the message matches the process
        m2 matches p1
else // The message does not have the required type at all
    m2 does NOT matches p1
```

Now, consider the part in the above procedure which is defined as SPECIAL-CASE. According to the LRM, *"A correlation set can be initiated only once during the lifetime of the scope it belongs to."* [10, Section 10.1] Hence, we can conclude that if receiving a message causes re-initiation of a correlation set, it can not be matched to the current process instance; i.e. a new process instance must be created for it. It is worth mentioning that there is an exception to this rule in case of concurrent start activities[18], but this does not affect our example. Therefore, if the special case happens in this example, it means that the message does not match the process instance, and a new business process instance must be created to handle *m2*.

The above mentioned exception is defined for concurrent start activities and enables them to ignore the fact that the correlation is already initiated. If a set of concurrent start activities is defined for a business process, the LRM states that *"compliant*

---

[18] This is the reason why this case is called a special case.

implementations MUST ensure that only one of the inbound messages carrying the same correlation set tokens actually instantiates the business process (usually the first one to arrive, but this is implementation dependent). The other incoming messages in the concurrent initial set MUST be delivered to the corresponding receive activities in the already created instance." [10, Section 11.4] Though it is not clearly mentioned, one can conclude that once the first message is received and a business process is created, the following messages for the remaining concurrent activities (carrying the same correlation tokens) will not attempt to initiate any new correlation sets, and thus are permissible. In other words, as confirmed by the issue #78 *"the 'initiate='yes" value is only true on the first multi-start activity that fires, the rest are magically transformed into 'initiate='no" once the first multi-start fires."* [34] Such an exception is not introduced for a while activity, making it impossible to define the behaviour of such a business process as we did in this example.

7.3. Inconsistencies

Inconsistencies are another category of defects caused by natural language descriptions. Normally, natural language definitions seem meaningful and reasonable; however, they can (-and sometimes do) introduce contradicting, or confusing meanings for a concept. The inconsistencies presented in this section were discovered during the formal modelling process, where we had to assign a specific meaning (or behaviour) to each entity of the language, and we found some of these meanings were indeed contradictory.

As mentioned before, the LRM permits a business process definition to have multiple concurrent start activities:

"It is permissible to have the `createInstance` *attribute set to* `"yes"` *for a set of concurrent initial activities. In this case the intent is to express the possibility that any one of a set of required inbound messages can create the process instance because the order in which these messages arrive cannot be predicted."* [10, Section 11.4]

Thus, as soon as a message arrives for one of these activities, a new business process instance must be created and the rest of the activities do not create any new process instances as long as they receive messages in the same correlation group. In other words, this special case implies that createInstance = "yes" does not always mean that a new business process instance must be created; it actually depends on the circumstances.

The same problem exists for the correlation initiation attribute. As mentioned in Section 7.2.2, initiate = "yes" does not always mean a new correlation set must be initiated either. In case of concurrent start activities, after the first activity receives a message and initiates a correlation, correlation initiation is disabled in all other activities. The confusion that is caused by this inconsistency is addressed in issue #78: *"The use of the initiate attribute on correlations in multi-start activities can easily lead to misunderstanding. The 'initiate = "yes"' value is only true on the first multi-start activity that fires, the rest are magically transformed into 'initiate = "No"' once the first multi-start fires. One can easily imagine the resulting confusion. To prevent this confusion perhaps we should add a new value for initiate such as 'initiate = "multiStart"'. This shows that the programmer understands the special semantics of correlation sets on multi-start activities."* [34]

Nonetheless, the technical committee has not yet addressed the same problem with the createInstance attribute.

Chapter 8. Conclusion and Future Work

Our formalization of the key semantic aspects of BPEL in terms of a hierarchically defined service abstract machine forms the major building block of the BPEL abstract machine and shows that the asynchronous DASM model is a natural choice for defining a precise semantic foundation. The resulting formal model transforms the abstract language definition in two refinement steps into an executable specification. In combination with inspection by analytical means, e.g. the ability to formally reason about critical language properties, experimental validation through simulation and testing helps establishing coherence and consistence of the semantics, thereby improving the quality of the language definition [15]. An advanced GUI facilitates such tasks.

A prerequisite for feasibility of formalization when applied as a practical instrument in an industrial standardization context is conciseness, intelligibility and robustness [18]. Standardization is an ongoing and potentially open-ended activity which brings a high dynamics into the development and maintenance of a language. Such dynamics demands a formalization framework that also meets the basic pragmatic needs. To this end, the abstract machine concept has already proven to be useful for enhancing conciseness and robustness of the formal model. The proposed hierarchical structuring of this model into three levels of abstraction reflects a clear separation of concerns, enhances intelligibility, and enables a tighter integration of the formal and the informal language description [15].

The current work forms the first building block on which a comprehensive formal model of BPEL can be established. To the author's best knowledge this work is the first published formal model of BPEL [14], [15]. Its orientation toward practical needs in industrial system design may even result in an opportunity to get involved in the design and standardization process of the language. As such, this work is already recognized and

encouraged by the WSBPEL TC in response to the necessity of formalism "*in surfacing ambiguities and irregularities in the process of construction of the formal model*" [34, Issue#42].

The work presented here is being continued as part of different projects in our group. A major revision of the current model is being developed incorporating a two dimensional organization of the model which facilitates extending the service abstract machine model towards modeling and integration of variables, compensation behaviour and fault handling. Moreover, the executable model is being extended to capture the complete service abstract machine model. On the other hand, the availability of an executable network model [20], which has been developed as part of another project in our group, will allow the full composition of the executable model and the network model in the near future. In addition, further expected improvements on the GUI and the underlying visualization tools will definitely be a great asset for performing validation through simulation and testing.

Appendices

Appendix A. BPEL Abstract Syntax Tree

(Sorted Alphabetically)

```
Activity                ::=   BasicActivity
                              | StructuredActivity
                              | ScopeRelatedActivity

ActStandardAttributes  ::=   ActivityName
                              Condition
                              SuppressJoinFailure

ActStandardElements    ::=   Source*
                              |Target*

AssignAct               ::=   ActStandardAttributes
                              ActStandardElements
                              Copy+

BasicActivity          ::=   ReceiveAct
                              |ReplyAct
                              |InvokeAct
                              |AssignAct
                              |ThrowAct
                              |TerminateAct
                              |WaitAct
                              |EmptyAct

Case                    ::=   Condition
                              Activity

CatchFault              ::=   FaultName?
                              VaribleName?
                              Activity

CatchAll                ::=   Activity

CompensateAct           ::=   ScopeName
                              ActStandardAttributes
                              ActStandardElements
```

```
CompensationHandler     ::=   Activity

Condition               ::=   Boolean_Expr

Copy                    ::=   From-Spec
                              To-Spec

CorrelationSets         ::=   CorrelationSetDef+

CorrelationSetDef       ::=   CorrelationName
                              PropertyName+

CorrelationTag          ::=   CorrelationUsage+

CorrelationUsage        ::=   CorrelationName
                              Initiation?
                              Pattern

EmptyAct                ::=   ActStandardAttributes
                              ActStandardElements

EventHandlers           ::=   (OnMesageEvent*
                              OnAlarmEvent*)+

Expression              ::=   Boolean_Expr
                              |Deadline_Expr
                              |Duration_Expr
                              |General_Expr

FaultHandlers           ::=   (CatchFault*
                              CatchAll?)+

FlowActivity            ::=   ActStandardAttributes
                              ActStandardElements
                              Links?
                              Activity+

From_Spec               ::=   From_Variable
                              From_Partner
                              From_Variable_Property
                              From_General_Expr
                              From_LiteralValues
                              From_Opaque

Initiation              ::=   "Yes" | "No"

InvokeAct               ::=   PartnerName
                              PortTypeName
                              OperationName
                              InputVariableName?
                              OutputVariableName?
                              ActStandardAttribues
                              ActStandardElements
```

```
                        CorrelationTag?
                        CatchFault*
                        CatchAll?
                        CompensationHandler?

Links             ::=   LinkName+

OnAlarmEvent      ::=   ForStructure|UntilStructure
                        Activity

OnMessageEvent    ::=   PartnerName
                        PortTypeName
                        OperationName
                        VariableName?
                        CorrelationTag?
                        Activity

Otherwise         ::=   Activity

Pattern           ::=   "in" | "out" | "out-in"

Partners          ::=   ParternDef+

PartnerDef        ::=   PartnerName
                        PartnerLinkName+

PartnerLinks      ::=   PartnerLinkDef+

PartnerLinkDef    ::=   PartnerLinkName
                        PartnerLinkTypeName
                        myRole?
                        partnerRole?

PickActivity      ::=   CreateInstance?
                        ActStandardAttributes
                        ActStandardElements
                        OnMessageEvent+
                        OnAlarmEvent*

ProcessDef        ::=   ProcessName
                        ProcessAttributes
                        PartnerLinks?
                        Partners?
                        Variables?
                        CorrelationSets?
                        FaultHandlers?
                        CompensationHandler?
                        EventHandlers?
                        Activity

Property          ::=   PropertyName
                        TypeName
```

```
PropertyAlias          ::=  PropertyName
                            MessageTypeName
                            PartName
                            Query

ReceiveAct             ::=  PartnerName
                            PortTypeName
                            OperationName
                            VariableName?
                            CreateInstance?
                            ActStandardAttributes
                            ActStandardElements
                            CorrelationTag?

ReplyAct               ::=  PartnerName
                            PortTypeName
                            OperationName
                            VariableName?
                            FaultName?
                            ActStandardAttribues
                            ActStandardElements
                            CorrelationTag?

ScopeAct               ::=  VarAccessSerializable
                            ActStandardAttributes
                            ActStandardElements
                            Variables?
                            CorrelationSets?
                            FaultHandlers?
                            CompensationHandler?
                            EventHandlers?
                            Activity

ScopeRelatedActivity   ::=  ScopeAct
                            |CompensateAct

SequenceAct            ::=  ActStandardAttributes
                            ActStandardElements
                            Activity+

Source                 ::=  LinkName
                            Condition?

StructuredActivity     ::=  SequenceAct
                            |SwitchAct
                            |WhileAct
                            |PickAct
                            |FlowAct
                            |ScopeAct
                            |CompensateAct

SwitchAct              ::=  ActStandardAttributes
                            ActStandardElements
                            Case+
```

```
                                Otherwise?

Target                   ::=    LinkName

TerminateAct             ::=    ActStandardAttributes
                                ActStandardElements

ThrowAct                 ::=    FaultName
                                VariableName?
                                ActStandardAttributes
                                ActStandardElements

To_Spec                  ::=    To_Variable
                                To_Partner
                                To_Variable_Property

Variables                ::=    VariablesDef+

VariableDef              ::=    VariableName
                                MessageTypeName?
                                TypeName?
                                ElementName?

VarAccessSerializable    ::=    "Yes"|"No"

WaitAct                  ::=    ForStructure| UntilStructure
                                ActStandardAttributes
                                ActStandardElements

WhileActivity            ::=    Condition
                                ActStandardAttributes
                                ActStandardElements
                                Activity
```

Appendix B. Abstract Model

B.1. Initial Definitions

```
//Agents
domain PROCESS
domain INBOX_MANAGER
domain OUTBOX_MANAGER

//Activity Agents
domain SEQUENCE_AGENT
domain SWITCH_AGENT
domain WHILE_AGENT
domain PICK_AGENT
domain FLOW_AGENT
domain PICK_ALARM_AGENT      //The agent responsible for the alarms
                             //in a pick activity
domain PICK_MESSAGE_AGENT    //The agent responsible for the onMessage events
                             //in a pick activity
domain FLOW_THREAD_AGENT     //sub agents of a flow agent


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

RUNNING_AGENT ≡ PROCESS U ACTIVITY_AGENT
//RUNNING_AGENT is the set of agents that execute (run) an activity.

AGENT ≡ RUNNING_AGENT U INBOX_MANAGER U OUTBOX_MANAGER

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

EVENT ≡ ONMESSAGE U ONALARM

// Activities
domain REPLY
domain RECEIVE
domain INVOKE
domain WAIT
domain TERMINATE
```

domain EMPTY
domain SEQUENCE
domain SWITCH
domain WHILE
domain PICK
domain FLOW

ACTIVITY ≡ REPLY U RECEIVE U INVOKE U WAIT U TERMINATE U EMPTY
U SEQUENCE U SWITCH U WHILE U PICK U FLOW

domain MESSAGE
domain OUTPUT_DESCRIPTOR

activity: RUNNING_AGENT → ACTIVITY
//Returns the activity that must be executed in a running agent.
//It is derived from the BPEL document and defined in the initial state

busy: RUNNING_AGENT → BOOLEAN
//initial value: false
//An agent is busy while one of its activities is being executed.

chosenActivity: PICK_AGENT → ACTIVITY
//The activity that is chosen by the pick agent to be executed

currentActivity: SEQUENCE_AGENT → ACTIVITY
//Keeps track of the current activity which is being executed

flowActivitySet: FLOW → ACTIVITY-set
//Set of the activities defined inside a FLOW
//It is derived from the BPEL document and defined in the initial state

foundBranch: SWITCH_AGENT → ACTIVITY
//initial value: undef
//The activity associated with the branch that is chosen by switch to be executed

inboxSpace: INBOX_MANAGER → MESSAGE-set
//Keeps the messages that have arrived for a business process and are
//not yet serviced.

match: PROCESS X MESSAGE → BOOLEAN
//Tells whether a messages matches a process instance or not.

outboxSpace: OUTBOX_MANAGER → OUTPUT_DESCRIPTOR-set
//initial value: ∅
//This set keeps the information about all the messages that should go out.

```
receiveMode: RUNNING_AGENT → BOOLEAN
//initial value : false
//Tells whether a runnig agent is waiting to receive a message or not.

startedExecution: PROCESS ∪ FLOW_THREAD_AGENT→ BOOLEAN
//initial value: false
//Tells whether a process or a flow thread agent has started executing its
//activity or not.

synchronous: INOVKE → BOOLEAN
//returns true if the invoke activity contains synchronous interactions;
//i.e. request/response
//It is derived from the BPEL document and defined in the initial state

waiting: PROCESS → BOOLEAN
//Tells wheather a process instance is waiting for a message or not
```

B.2. Programs

Inbox Manager

```
INBOXMANAGERPROGRAM ≡
   if inboxSpace(self) ≠ ∅ then
      choose p ∈ PROCESS, m ∈ inboxSpace(self) with match(p, m) and
                                                         waiting(p)
         ASSIGN_MESSAGE(p, m)
         if p = dummyProcess then
            new newDummy : PROCESS
               dummyProcess := newDummy
```

Outbox Manager

```
OUTBOXMANAGERPROGRAM ≡
   if outboxSpace(self) ≠ ∅ then
      choose od ∈ outboxSpace(self)
         SEND(od)
```

Process

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

Receive Activity

```
Execute_Receive (activity : RECEIVE) ≡
  if ¬receiveMode(self) then
    receiveMode(self) := true   //The running agent waits to receive a message
    ADD_INPUT_DESCRIPTOR_TO_WAITING_SET(activity)
  else
    if message_is_received(activity) then
      receiveMode(self) := false
      busy(self) := false
```

Reply Activity

```
Execute_Reply ( activity : REPLY) ≡
  ADD_OUTPUT_DESCRIPTOR_TO_OUTBOX_SPACE(activity)
  busy(self) := false
```

Invoke Activity

```
Execute_Invoke (activity : INVOKE) ≡
  if ¬receiveMode(self) then
    ADD_OUTPUT_DESCRIPTOR_TO_OUTBOX_SPACE(activity)
    if ¬synchronous(activity)  then
      busy(self) := false

    if synchronous(activity)  then
      receiveMode(self) := true
      ADD_INPUT_DESCRIPTOR_TO_WAITING_SET(activity)

  if receiveMode(self) and message_is_received(activity) then
    receiveMode(self) := false
    busy(self) := false
```

Terminate Activity

```
Execute_Terminate ≡
  STOP_ALL_SUBORDINATE_AGENTS
  stop rootProcess(self)
```

Wait Activity

```
Execute_Wait (activity : WAIT) ≡
   if wait_just_started(activity) then
      RECORD_WAIT_START_TIME(activity)
   else
      if wait_completed(activity) then
         busy(self) := false
```

Empty Activity

```
Execute_Empty (activity : EMPTY) ≡
   busy(self) := false
```

Sequence Activity

```
SEQUENCEPROGRAM ≡
   if ¬busy(self) then
      SET_CURRENT_ACTIVITY_TO_NEXT_ACTIVITY
      busy(self) := true

   if busy(self) then
      if sequence_is_not_completed then  //There are still some activities to execute
         EXECUTE_ACTIVITY(currentActivity(self))
      else          //No more activities
         stop self
         RELEASE_PARENT
```

Switch Activity

```
SWITCHPROGRAM ≡
   if ¬busy(self) and ¬branch_found then //No branch is selected yet
      FIND_BRANCH          //foundbranch(self) is set to the selected branch.
                           //Always successful (because of the default OTHERWISE)
      busy(self) := true
   if busy(self) then
      EXECUTE_ACTIVITY(foundBranch(self))

   if ¬busy(self) and branch_found then
      stop self
      RELEASE_PARENT
```

While Activity

```
WHILEPROGRAM ≡
  if ¬busy(self) and true_while_conditionthen
     busy(self) := true

  if busy(self) then
     EXECUTE_ACTIVITY(activity(self)))
     //Executing the activity inside while; when completed, busy becomes false

  if ¬busy(self) and false_while_condition then
     stop self
     RELEASE_PARENT
```

Pick Activity

```
PICKPROGRAM ≡
  if ¬busy(self) then
     if activity_is_not_chosen then
        CREATE_PICK_ALARM_AGENT              // To manage onAlaram events
        CREATE_PICK_MESSAGE_AGENT            // To manage onMessage events
        busy(self) := true          //The agent is waiting for an event to happen
     else
        RELEASE_PARENT
        stop self

  if busy(self) then
     if activity_is_not_chosen then
       CHOOSE_EARLIEST_HAPPENED_EVENT
       //choose one of the onMessage or onAlarm events that happened first
       //chosenActivity is set to the corresponding activity of that event
     else
       EXECUTE_ACTIVITY(chosenActivity(self))
```

Pick Message Agent

```
PICKMESSAGEPROGRAM ≡
  if onAlarm_event_occured then
    REMOVE_ALL_ONMESSAGE_INPUT_DESCRIPTORS_FROM_WAITING_SET
    stop self
  else
    if ¬busy(self) then
      ADD_ALL_ONMESSAGE_INPUT_DESCRIPTORS_TO_WAITING_SET
      busy(self) := true
    else
      CHOOSE_A_COMPLETED_ONMESSAGE_EVENT_AND_INFORM_PICK_AGENT
        stop self
```

Pick Alarm Agent

```
PICKALARMPROGRAM ≡
  if onMessage_event_occured then
    stop self
  else
    if ¬busy(self) then
      RECORD_ALARM_START_TIME
      busy(self) := true
    else
      FORALL_PASSED_ONALARM_EVENTS_INFORM_PICK_AGENT
          stop self
```

Flow Activity

```
FLOWPROGRAM ≡
  if ¬busy(self) then
     //Creates threads to concurrently execute activities grouped inside the flow.
     forall activity ∈ flowActivitySet(self)
       CREATE_A_FLOWTHREAD_AGENT_AND_ADD_TO_FLOWAGENTSET(activity)
    busy(self) := true

  if busy(self) and empty_flowagentset then
     //All threads are done, flow activity is completed.
     RELEASE_PARENT
     stop self
```

Flow Thread Agent

```
FLOWTHREADPROGRAM ≡
   if ¬busy(self) and ¬startedExecution(self) then
      startedExecution(self) := true
      busy(self) := true

   if busy(self) then
      EXECUTE_ACTIVITY(activity(self))

   if ¬busy(self) and startedExecution(self) then
      REMOVE_SELF_FROM_FLOWAGENTSET
      stop self
//Each thread executes one activity. When the execution is completed,
//the thread removes itself from the flow agent set and is terminated.
```

Appendix C. Complete Formal Model

C.1. Initial Definitions

```
//Agents
domain PROCESS
domain INBOX_MANAGER
domain OUTBOX_MANAGER

//Activity Agents
domain SEQUENCE_AGENT
domain SWITCH_AGENT
domain WHILE_AGENT
domain PICK_AGENT
domain FLOW_AGENT
domain PICK_ALARM_AGENT      //The agent responsible for the alarms
                             //in a pick activity
domain PICK_MESSAGE_AGENT    //The agent responsible for the onMessage events
                             //in a pick activity
domain FLOW_THREAD_AGENT     //sub agents of a flow agent


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

RUNNING_AGENT ≡ PROCESS U ACTIVITY_AGENT
//RUNNING_AGENT is the set of agents that execute (run) an activity.

AGENT ≡ RUNNING_AGENT U INBOX_MANAGER U OUTBOX_MANAGER

//Events
domain ONMESSAGE    //OnMessage events of Pick activity
domain ONALARM      //OnAlarm events of Pick activity

EVENT ≡ ONMESSAGE U ONALARM

// Activities
domain REPLY
domain RECEIVE
domain INVOKE
domain WAIT
domain TERMINATE
```

```
domain EMPTY
domain SEQUENCE
domain SWITCH
domain WHILE
domain PICK
domain FLOW

ACTIVITY ≡ REPLY U RECEIVE U INVOKE U WAIT U TERMINATE U EMPTY
           U SEQUENCE U SWITCH U WHILE U PICK U FLOW

//MESSAGE
domain MESSAGE

IN_OPERATION ≡ RECEIVE U INVOKE U ONMESSAGE
OUT_OPERATION ≡ REPLY U INVOKE
OUTPUT_DESCRIPTOR ≡ RUNNING_AGENT X OUT_OPERATION

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

domain SWCASE
//case elements of a switch, it includes conditional cases and otherwise
//otherwise is a special case with an always-true condition

domain PRIORITY
//An ordered domain, with a least element called LEAST_PRIORITY

//CORRELATIONSET
domain CORRELATIONSET

//-----------------------RUNNING_AGENT properties------------------------------
busy: RUNNING_AGENT → BOOLEAN
// initial value: false
// An agent is busy while one of its activities is being executed.

rootProcess: RUNNING_AGENT → PROCESS
//Returns the process agent to which this running agent belongs.

rootProcess(a: RUNNING_AGENT) ≡
                 - a                              : a ∈ PROCESS,
                 - rootProcess (parentAgent(a))   : otherwise.

receiveMode: RUNNING_AGENT→ BOOELAN
//initial value : false
//Tells whether a running agent is waiting to receive a message or not.
```

parentAgent: RUNNING_AGENT → RUNNING_AGENT
//Returns the parent agent (one layer above in the creation tree) of an agent

//---------------------------------- PROCESS properties ------------------------------
inboxManager: PROCESS → INBOX_MANAGER
//An inbox manager is assigned to each process instance

outboxManager: PROCESS → OUTBOX_MANAGER
//An outbox manager is assigned to each process instance

mainActivity: PROCESS → ACTIVITY
//This is the activity which the process should execute
//It is derived from the BPEL document and defined in the initial state

startedExecution: PROCESS ∪ FLOW_THREAD_AGENT → BOOLEAN
//initial value: false
//Tells whether a process or a flow thread agent has started executing its
//activity or not

waitingForMessage: PROCESS → (RUNNING_AGENT X IN_OPERATION)-set
//initial value: ∅
//For each process this set indicates the input activities (or onMessage events)
//waiting for a message

completedInOperations: PROCESS → (RUNNING_AGENT X IN_OPERATION X
TIME)-set
//initial value: ∅
//For each process this set indicates the input activities (or onMessage events)
//that have received a message, together with the receiving time

subordinateAgentSet: PROCESS → ACTIVITY_AGENT-set
//Returns the set of activity agents that have been created and work under control of
//this process.
subordinateAgentSet(p: PROCESS) ≡
{a | a ∈ ACTIVITY_AGENT where rootProcess(a) = p}

//--------------------------------INBOX_MANAGER properties---------------------------------
inboxSpace: INBOX_MANAGER → MESSAGE-set
//initial value: ∅
//Keeps the messages that have arrived for a business process and are
//not yet serviced.

match: PROCESS X IN_OPERATION X MESSAGE → BOOLEAN
//Tells whether a messages matches a specific input operation of
//a process instance or not.

```
//-------------------------------------OUTBOX_MANAGER properties-------------------------
outboxSpace: OUTBOX_MANAGER → OUTPUT_DESCRIPTOR-set
//initial value: ∅
//This set keeps the information about all the messages that should go out.

//-------------------------------------ACTIVITY_AGENT properties--------------------------
baseActivity: ACTIVITY_AGENT→ ACTIVITY
//The activity for which an activity agent is responsible

//-------------------------------ACTIVITY properties------------------------------------------
assignedAgent: ACTIVITY → ACTIVITY_AGENT
// Assumes that every activity that is fetched is unique
// For each structured activity returns the activity agent that is executing it
// returns undef if no agent is assigned yet or the activity is a basic activity
// so there is no need for agents to eliminate this relationship when they end

sourceLinkSet: ACTIVITY → LINK-set
//An activity can be the source of a set of links; returns this set
//It is derived from the BPEL document and defined in the initial state

targetLinkSet: ACTIVITY → LINK-set
//An activity can be the target of a set of links; returns this set
//It is derived from the BPEL document and defined in the initial state

activityJoinCondition: ACTIVITY → BOOLEAN
//default joinCondition: The logical OR of the link status of all the incoming links
//of the activity
//returns true if the joinCondition of the activity is satisfied

//--------------------------------- In Operation Properties -------------------------
initiateCorrelation: IN_OPERATION → BOOLEAN
//indicates whether an input operation initiates a new correlation set or not
//It is derived from the BPEL document and defined in the initial state

//--------------------------------- Invoke Activitiy Properties -------------------------
synchronous: INOVKE → BOOLEAN
//returns true if the invoke activity contains synchronous interactions;
//i.e. request/response
//It is derived from the BPEL document and defined in the initial state

//-------------------------------------Wait Activity Properties-------------------
completionTime: WAIT → TIME
//completionTime returns the time when a wait activity is completed.
//In case of 'until' its trivial, but in case of 'for' it needs the starting time,
//which is accessible through the activity itself.
```

```
startTime: WAIT ∪ PICK_ALARM_AGENT → TIME
//initial value: undef
//startTime keeps the starting time of a wait activity and
//is needed in case of waiting for a duration.
//Pick alarm agent also keeps a starting time.

//-------------------------------------- Sequence Activity Properties-------------------
sequenceCounter: SEQUENCE → ACTIVITY
//Returns the next activity in the sequence
//If there is no more activities in the sequence, returns undef
// It is derived from the BPEL document and defined in the initial state

//-------------------------------------- While Activity Properties ----------------
waCondition: WHILE → BOOLEAN
//Returns the value of the conditional expression of a while activity

innerActivity: WHILE → ACTIVITY
// Returns the activity that is defined inside a while.
//It is derived from the BPEL document and defined in the initial state

//-------------------------------------- Switch Activity Properties ----------------
swCaseSet: SWITCH → SWCASE-set
//returns the list of case elements of the switch plus otherwise
//It is derived from the BPEL document and defined in the initial state

swCaseCondition: SWCASE → BOOLEAN
//Returns the value of the conditional expression of a switch case element
//For otherwise, it always returns true

swCaseActivity: SWCASE → ACTIVITY
//the activity associated with a case element or otherwise
//It is derived from the BPEL document and defined in the initial state

swPrority: SWCASE → PRIORITY
//Each switch case element is assigned a priority,
//resembling the order between cases.
//The lowest priority is assigned to otherwise.
//It is derived from the BPEL document and defined in the initial state

//-------------------------------------- Pick Activity Properties ----------------
onMessageSet: PICK → ONMESSAGE-set
//Set of the onMessage events defined in a pick activity
//It is derived from the BPEL document and defined in the initial state
```

onAlarmSet: PICK → ONALARM-set
//Set of the onAlarm events defined in a pick activity
//It is derived from the BPEL document and defined in the initial state

//--------------------------Event(OnMessage and OnAlaram) Properties---------------------
onEventActivity: EVENT → ACTIVITY
//Returns the activity associated with a specific event
//It is derived from the BPEL document and defined in the initial state

triggerTime: ONALARM X TIME → TIME
//returns the trigger time of an onAlarm activity.
//If onAlarm is defined by a 'for', it uses the second parameter (starting time of the
//alarm agent) to determine the trigger time.

//--------------------------------------- Sequence Agent Properties------------------------
currentActivity: SEQUENCE_AGENT → ACTIVITY
//initial value: ∅
// Keeps track of the current activity which is being executed

//--------------------------------------- Switch Agent Properties-------------------------
foundBranch: SWITCH_AGENT → ACTIVITY
//initial value: undef
//The activity associated with the branch that is chosen by switch to be executed

//--------------------------------------- Pick Agent Properties------------------------
triggeredEvents: PICK_AGENT → (EVENT X TIME)-set
//initial value: Ø
//The set of events that have happened

chosenActivity: PICK_AGENT → ACTIVITY
//initial value: undef
//The activity that is chosen by the pick agent to be executed

//--------------------------------------- Pick Alarm Agent----------------------------------
startTime: PICK_ALARM_AGENT ∪ WAIT→ TIME
//initial value: undef
//startTime keeps the starting time of a pick alarm agent and
//is needed in case of waiting for a duration.
//startTime is also used for the wait activity.

//--------------------------------------- Flow Activity Properties ----------------
flowActivitySet: FLOW → ACTIVITY-set
//Set of the activities defined inside a flow
//It is derived from the BPEL document and defined in the initial state

```
flowAgentSet: FLOW_AGENT → FLOW_THREAD_AGENT-set
// initial value: Ø
//The set of alive thread agents that are working under a flow agent

//-----------------------------------Flow Thread Agent-----------------------------
startedExecution: FLOW_THREAD_AGENT ∪ PROCESS → BOOLEAN
//initial value: false
//Tells whether a process or a flow thread agent has started executing its
//activity or not

//----------------------------------- Link --------------------------------------
linkTransitionCondition: LINK → BOOLEAN
//Evaluates the transition condition of a link

linkStatus: LINK → {POSITIVE,NEGATIVE,NOTDEFINED}
//initial value: NOTDEFINED
//returns the status of a link
```

C.2. Programs

Inbox Manager

```
INBOXMANAGERPROGRAM ≡
   if inboxSpace(self) ≠ Ø then
      choose p ∈ PROCESS, m ∈ inboxSpace(self), (agent, op) ∈
               waitingForMessage(p) with match(p, op, m)
         Assign_Message(p, agent, op, m)
         Pick_Activity_Clearance(p, agent, op)

         if p = dummyProcess then
           new newDummy : PROCESS
              dummyProcess := newDummy
```

Assign Message

```
Assign_Message(p : PROCESS, agent : RUNNING_AGENT, op : IN_OPERATION,
                    m : MESSAGE) ≡
   if initiateCorrelation(op)   then
      INITIATE_CORRELATION(p, agent, op, m)

   remove m from inboxSpace(self)
   remove (agent,op) from  waitingForMessage(p)
   add (agent,op,now) to completedInOperations(p)
```

Pick Activity Clearance

```
Pick_Activity_Clearance (p : PROCESS, a : RUNNING_AGENT,
                         op : IN_OPERATION) ≡
   if a ∈ PICK_MESSAGE_AGENT then
      forall (a, op') ∈ waitingForMessage(p) with op' ≠ op
         remove (a,op') from waitingForMessage(p)
```

Outbox Manager

```
OUTBOXMANAGERPROGRAM ≡
   if outboxSpace(self) ≠ ∅ then
      choose  (agent, op) ∈ outboxSpace(self)
         SEND(agent, op)
```

Process

```
PROCESSPROGRAM ≡
  if ¬busy(self) then
    if ¬startedExecution(self) then
      startedExecution(self) := true
      busy(self) := true
    else
      stop self
  else
    Execute_Activity(mainActivity(self))
```

Execute Activity

```
//suppose that busy is set to true before entering this module
Execute_Activity (activitiy: ACTIVITY) ≡
  if ∀x (x ∈ targetLinkSet(activity) → linkStatus(x) ≠ NOTDEFINED) then
    if activityJoinCondition(activity)  then
      if activity in REPLY then
        Execute_Reply (activity)
      if activity in RECEIVE then
        Execute_Receive (activity)
      if activity in INVOKE then
        Execute_Invoke (activity)
      if activity in TERMINATE then
        Execute_Terminate
      if activity in WHILE then
        Execute_While(activity)
      if activity in EMPTY then
        Execute_Empty(activity)
      if activity in SEQUENCE then
        if assignedAgent(activity) = undef then
          new s: SEQUENCE_AGENT
            assignedAgent(activity) := s
            Initialize(s, activity)
      if activity in SWITCH then
        if assignedAgent(activity) = undef then
          new sw: SWITCH_AGENT
            assignedAgent(activity) := sw
            Initialize(sw, activity)
      if activity in WHILE then
        if assignedAgent(activity) = undef then
          new w : WHILE_AGENT
            assignedAgent(activity) := w
```

```
            Initialize(w, activity)
      if activity in PICK then
         if assignedAgent(activity) = undef then
            new p : PICK_AGENT
               assignedAgent(activity) := p
               Initialize(p, activity)
      if activity in FLOW then
         if assignedAgent(activity) = undef then
            new f : FLOW_AGENT
               assignedAgent(activity) := f
               Initialize(f, activity)
   else
      THROW_JOIN_FAILURE
      //JoinCondition is false. A fault (joinFailure) is thrown.
//else
//There are some activities linked to this activity that have not yet finished
//execution. Therefore, the activity can not be executed yet.
```

Initialize

```
Initialize(agent: ACTIVITY_AGENT, activity: ACTIVITY) ≡
   parentAgent(agent) := self
   baseActivity(agent) := activity
```

Receive Activity

```
Execute_Receive (activity : RECEIVE) ≡
   let inputDescriptor = (self, activity) in
      if ¬receiveMode(self) then
         receiveMode(self) := true //The running agent waits to receive a message
         add inputDescriptor to waitingSet
      else
         if inputDescriptor ∉ waitingSet then
            receiveMode(self) := false
            busy(self) := false
            Synchronization(activity)
where waitingSet = waitingForMessage( rootProcess(self) )
```

Reply Activity

```
Execute_Reply ( activity : REPLY) ≡
  let outputDescriptor = (self, activity) in
    add outputDescriptor to outSpace
    busy(self) := false
    Synchronization(activity)
where outSpace = outboxSpace(outboxManager(rootProcess(self)))
```

Invoke Activity

```
Execute_Invoke (activity : INVOKE) ≡
  let ioDescriptor = (self, activity) in
    if ¬recevieMode(self) then    // i.e. if it is the first step
      add ioDescriptor to outSpace
      if ¬synchronous(activity) then   // i.e. if not synchronous invoke
        busy(self) := false
        Synchronization(activity)
      if synchronous(activity) then   // i.e. if synchronous invoke
        receiveMode(self) := true
        add ioDescriptor to waitingSet
    if receiveMode(self) and ioDescriptor ∉ waitingSet then
        receiveMode(self) := false
        busy(self) := false
        Synchronization(activity)
  where
    outSpace = outboxSpace(outboxManager(rootProcess(self)))
    waitingSet = waitingSetForMessage(rootProcess(self))
```

Terminate Activity

```
Execute_Terminate ≡
  forall agent in subordinateAgentSet(rootProcess(self))
    stop agent
  stop rootProcess(self)
```

Wait Activity

```
Execute_Wait (activity : WAIT) ≡
   if startTime(activity) = undef then
      startTime(activity) := now
   else
      if completionTime(activity) ≤  now  then
         busy(self) := false
         Synchronization(activity)
//startTime is associated with each wait activity and its initial value is undef
//startTime is also used separately in pick alarm agent,
//but won't cause any problem here.
```

Empty Activity

```
Execute_Empty (activity : EMPTY) ≡
   busy(self) := false
   Synchronization(activity)
```

Sequence Activity

```
SEQUENCEPROGRAM ≡
   if ¬busy(self) then
      currentActivity(self) := sequenceCounter(baseActivity(self))
      busy(self) := true
   else
      if currentActivity(self) ≠ undef then
         Execute_Activity(currentActivity(self))
      else
         stop self
         busy(parentAgent(self)) := false
         Synchronization(baseActivity(self))
```

Switch Activity

```
SWITCHPROGRAM ≡
  if ¬busy(self) then
     if foundBranch(self) = undef then //No branch is selected yet
        let caseSet = swCaseSet(baseActivity(self)) in
          //caseSet is the set of all cases
          choose c ∈ caseSet with (swCaseCondition(c) ∧
           ∀x ((x ∈ caseSet ∧ swCaseCondition(x))→ swPriority(c) ≥ swPriority(x)))
              foundBranch(self) := swCaseActivity(c)
           //choosing the first [with the highest priority] branch with a true
           //condition. It is always successful, because of the default otherwise
           busy(self) := true
     else //branch is executed and finished
        busy(parentAgent(self)) := false
        stop self
        Synchronization(baseActivity(self))

  if busy(self)   then     //Execute the found branch
     Execute_Activity(foundBranch(self)))
```

While Activity

```
WHILEPROGRAM ≡
  if busy(self)   then
     Execute_Activity(innerActivity(baseActivity(self)))
  else
     if waCondition(baseActivity(self)) then
        busy(self) := true
     else
        busy(parentAgent(self)) := false
        stop self
        Synchronization(baseActivity(self))
```

Pick Activity

```
PICKPROGRAM ≡
  if ¬busy(self) then
    if chosenActivity(self) = undef then
      new a : PICK_ALARM_AGENT
        Initialize(a, baseActivity(self))
      new b: PICK_MESSAGE_AGENT
        Initialize(b, baseActivity(self))
      busy(self) := true //The agent is waiting for an event to happen
    else
      busy(parentAgent(self)) := false
      stop self
      Synchronization(baseActivity(self))

  if busy(self)  then
    if chosenActivity(self) = undef then
      choose (event, time) ∈ triggeredEvents(self) with
                                ∀e∀t ((e,t) ∈ triggeredEvents(self) → time ≤ t)
        chosenActivity(self) := onEventActivity(event)
    else
      Execute_Activity(chosenActivity(self))
```

Pick Message Agent

```
PICKMESSAGEPROGRAM ≡
  if triggeredEvents(parentAgent(self)) ≠ ∅ then
    forall event ∈ onMessageSet(baseActivity(self))
      let inputDescriptor = (self, event) in
        remove inputDescriptor from waitingForMessage(rootProcess(self))
    stop self
  else
    if ¬busy(self) then
      forall event ∈ onMessageSet(baseActivity(self))
        let inputDescriptor = (self , event) in
          add inputDescriptor to waitingForMessage( rootProcess(self))
      busy(self) := true
    else
      choose event ∈ onMessageSet(baseActivity(self)) with
                (self, event, time) ∈ completedInOperations(rootProcess(self))
        add (event, time) to triggeredEvents(parentAgent(self))
        stop self
```

Pick Alarm Agent

```
PICKALARMPROGRAM ≡
  if triggeredEvents(parentAgent(self)) ≠ ∅ then
    stop self
  else
    if ¬busy(self) then
      startTime(self) := now
      busy(self) := true
    else
      forall event ∈ onAlarmSet(baseActivity(self)) with
                              triggerTime(event, startTime(self)) ≤ now
        add (event, triggerTime(event, startTime(self))) to
                                              triggeredEvents(parentAgent(self))
        stop self
```

Flow Activity

```
FLOWPROGRAM ≡
  if ¬busy(self) then
  //Creates threads to concurrently execute activities grouped inside the flow.
    forall activity ∈ flowActivitySet(self)
      new fThread : FLOW_THREAD_AGENT
        Initialize(fThread, activity)
        add fThread to flowAgentSet(self)

    busy(self) := true
  else
    if flowAgentSet(self) =  Ø then
    //All threads are done, flow activity is completed.
      busy(parentAgent(self)) := false
      stop self
      Synchronization(baseActivity(self))
```

Flow Thread Agent

```
FLOWTHREADPROGRAM ≡
  if ¬busy(self) and ¬startedExecution(self) then
    startedExecution(self) := true
    busy(self) := true

  if busy(self) then
    Execute_Activity(baseActivity(self))

  if ¬busy(self) and startedExecution(self) then
    remove self from flowAgentSet(parentAgent(self))
    stop self
//Each thread executes its baseActivity.
//When baseActivity is completed, the thread removes itself from the flow agent set
//and is terminated.
```

Link Semantics

```
Synchronization(activity : ACTIVITY) ≡
  forall link ∈ sourceLinkSet(activity)
    if linkTransitionCondition(link) then
      linkStatus(link) := true
    else
      linkStatus(link) := false
```

Appendix D. Executable Model

D.1. Original Model

Name Space

```
namespace ModelGUI
import System.Collections
import System.Windows.Forms
```

Global Definitions

The inbox Manager, the outbox manager and a dummy process are the DASM agents available in the initial state. agents and processes are two sets representing the corresponding domains: AGENT and PROCESS. The internal structure has replaced the oracle in the ASM model, where it was used to access the BPEL process definition.

```
var inboxManager as INBOX_MANAGER
var outboxManager as OUTBOX_MANAGER = new OUTBOX_MANAGER
var dummy as PROCESS
//domains
var agents as Set of AGENT = {}
var processes as Set of PROCESS = {}
//Execution specific
var intStr as ARRAY_BASED_INT_STR = new ARRAY_BASED_INT_STR
var globalID as Integer = 1
```

Agent

```
public class AGENT
  virtual Program()
```

Running Agent

```
public class RUNNING_AGENT extends AGENT
  var busy as Boolean = false
  var rootProcess as PROCESS?  = undef
  var parentAgent as RUNNING_AGENT?  = undef
  var receiveMode as Boolean = false
  var id as String

  virtual stop()
//------------------------------Initialize------------------------
  Initialize(agent as ACTIVITYAGENT)
    agent.parentAgent := me
    agent.rootProcess := me.rootProcess
    add agent to me.rootProcess.subordinateAgentSet
    add agent to agents
```

`Initialize` maintains two functions `rootProcess` and `subordinateAgentSet`. It also updates the `parentAgent` function. In addition, it maintains the set of agents in the domain AGENT by adding the new agent to agents.

Inbox Manager

```
class INBOX_MANAGER extends AGENT
  var inboxSpace as Set of MESSAGE = {}
```

Inbox Manager Program

```
class INBOX_MANAGER
  override Program()
    if not (inboxSpace = {}) then
      choose p in processes, m in inboxSpace, (agent, activity) in
      p.waitingForMessage where Match(p,m, activity)
        Assign_Message(p,agent,activity,m)
        MODEL.messageIsAssigned(p,m,agent,activity)

        if p = dummy then //need new dummy
          newDummy = PROCESS.newProcess(globalID,outboxManager)
          dummy := newDummy
          add newDummy to processes
          add newDummy to agents
          globalID := globalID + 1
      ifnone
        WriteLine("No Assign")
```

Assign Message

```
class INBOX_MANAGER
  Assign_Message(p as PROCESS, agent as RUNNING_AGENT, inActivity as
  INPUT_ACTIVITY, m as MESSAGE)
    if initiateCorrelation(intStr, inActivity) then
      Initiate_Correlation(p,getCorrelationSetsToInitiate(intStr,inActivity),m)
    choose (a, act) in p.waitingForMessage where a = agent and act = inActivity
      remove (a, act) from p.waitingForMessage
    remove m from inboxSpace
    add (agent, inActivity) to p.completedInOperation
```

Outbox Manager

```
public class OUTBOX_MANAGER extends AGENT
  var outboxSpace as Set of (RUNNING_AGENT, OUTPUT_ACTIVITY) = {}
```

Process

```
public class PROCESS extends RUNNING_AGENT
  var waitingForMessage as Set of (RUNNING_AGENT, INPUT_ACTIVITY) = {}
  var completedInOperation as Set of (RUNNING_AGENT, INPUT_ACTIVITY) = {}
  var correlations as Set of CORRELATIONSET = {}
  var mainActivity as ACTIVITY = processActivity(intStr)
  var waitingActivity as INPUT_ACTIVITY? = null
  var subordinateAgentSet as Set of ACTIVITYAGENT = {}
  var outboxManager as OUTBOX_MANAGER?
  var startedExecution as Boolean = false
//---------------------------------stop-----------------------
  override stop()
    remove me from agents
    remove me from processes
```

Process Program

```
public class PROCESS
  override Program()
    if not busy then
      if not startedExecution then
        startedExecution := true
        busy := true
      else
        stop(me)
    else
      Execute_Activity(me, mainActivity)
```

Initiate Correlation

`Initiate_Correlation` initiates a number of correlation sets by setting the value of each property in the correlation set to the values of the tokens carried by the message.

```
class PROCESS
  Initiate_Correlation(cSet as Set of CORRELATIONSET, m as MESSAGE)
```

Activities

```
public class ACTIVITY

public class STRUCTURED_ACTIVITY extends ACTIVITY
  var assignedAgent as ACTIVITYAGENT? = undef

public class INPUT_ACTIVITY extends ACTIVITY

public class OUTPUT_ACTIVITY extends ACTIVITY

public class RECEIVE extends INPUT_ACTIVITY

public class REPLY extends OUTPUT_ACTIVITY

public class FLOW extends STRUCTURED_ACTIVITY
  var flowActivitySet as Set of ACTIVITY

public class SEQUENCE extends STRUCTURED_ACTIVITY
```

Activity Agents

```
public class ACTIVITYAGENT extends RUNNING_AGENT
  var baseActivity as ACTIVITY

  override stop()
    remove me from agents
    remove me from me.rootProcess.subordinateAgentSet
```

When an activity agent is terminated, the `subordinateAgentSet` function must also be updated.

Sequence Agent

```
public class SEQUENCE_AGENT extends ACTIVITYAGENT
  var currentActivity as ACTIVITY? = undef
//---------------------------SequenceProgram-----------------------------------
  override Program()
    match baseActivity
      baseAct as SEQUENCE:
        if not busy then
          currentActivity := sequenceCounter(intStr, baseAct)
          busy := true
        else
          if not (currentActivity = null) then
            Execute_Activity(me, currentActivity)
          else
            stop(me)
            parentAgent.busy := false
```

Flow Agent

```
class FLOW_AGENT extends ACTIVITYAGENT
  var flowAgentSet as Set of FLOW_THREAD_AGENT = {}
//-------------------------------FlowProgram--------------------------------
  override Program()
    match baseActivity
      baseAct as FLOW:
        if not busy then
          busy := true

          step foreach activity in baseAct.flowActivitySet
            var fThread as FLOW_THREAD_AGENT = new FLOW_THREAD_AGENT( id + ":t"
+ idCounter,activity)
            idCounter := idCounter + 1
            Initialize(fThread)
            add fThread to flowAgentSet
        else
          if flowAgentSet = {} then
            WriteLine("ALL thread agents finished")
            parentAgent.busy := false
            stop(me)
```

Flow Thread Agent

```
class FLOW_THREAD_AGENT extends ACTIVITYAGENT
  var startedExecution as Boolean = false
//-----------------------------FlowThreadProgram-----------------------------
  override Program()
    if (not busy) and (not startedExecution) then
      startedExecution := true
      busy := true

    if busy then
      Execute_Activity(me, baseActivity)
    if (not busy) and startedExecution then
      match me.parentAgent
        parent as FLOW_AGENT:
          remove me from parent.flowAgentSet
          stop(me)
```

Execute Activity

```
Execute_Activity(self as RUNNING_AGENT, activity as ACTIVITY) =
  match activity
    inActivity as RECEIVE:
      Execute_Receive(self,inActivity)
    inActivity as REPLY:
      Execute_Reply(self,inActivity)

    inActivity as FLOW:
      if inActivity.assignedAgent = null then
        var   fAgent   as   FLOW_AGENT   =   new   FLOW_AGENT(self.id   +   ":f"+
self.idCounter,inActivity)

        self.idCounter := self.idCounter + 1
        inActivity.assignedAgent := fAgent
        Initialize(self, fAgent)

    inActivity as SEQUENCE:
      if inActivity.assignedAgent = null then
        var  sAgent  as  SEQUENCE_AGENT  =  new  SEQUENCE_AGENT(self.id  +  ":s"+
self.idCounter,inActivity)

        self.idCounter := self.idCounter + 1
        inActivity.assignedAgent := sAgent
        Initialize(self, sAgent)

    _:
      WriteLine("Not a valid activity")
```

Execute Receive

```
Execute_Receive(self as RUNNING_AGENT, activity as INPUT_ACTIVITY)
  let inputDescriptor = (self, activity)
  if not self.receiveMode then
    self.receiveMode := true
    add inputDescriptor to self.rootProcess.waitingForMessage
  else
    if not (inputDescriptor in self.rootProcess.waitingForMessage) then
      self.receiveMode := false
      self.busy := false
```

Execute Reply

```
Execute_Reply(self as RUNNING_AGENT, activity as OUTPUT_ACTIVITY) =
  let outputDescriptor = (self, activity)
  add outputDescriptor to self.rootProcess.outboxManager.outboxSpace
  self.busy := false
```

Message

```
public class MESSAGE
```

Correlation Sets

```
class CORRELATIONSET
  name as String
```

D.2. Execution-Specific Additions to the ASM Model

Executable Model

```
class MODEL
  var view as ModelGUI.View
//---------------------------------initialize----------------------------------
  [EntryPoint]
  initialize()
    step
      inboxManager := new INBOX_MANAGER
      agents := {}
      processes := {}
      intStr := new ARRAY_BASED_INT_STR
    step
      intStr.initialize()
    step
      globalID := 1
      dummy := PROCESS.newProcess(0,outboxManager)
    step
      add inboxManager to agents
    step
      add dummy to agents
      add dummy to processes
    step
      showProgram()
//----------------------------------run----------------------------------------
  [EntryPoint]
  run()
    step
      forall a in agents
        showMessages()
        showProcesses()
        showOutspace()
        a.Program()
        informAssignments()
//---------------------------------addMessage----------------------------------
  [EntryPoint]
  addMessage(m as String, p as Integer)
    add new MESSAGE(new DATA(p),m) to inboxManager.inboxSpace
```

Data

```
public class DATA
  var data as Integer
  isEqual(d as DATA) as Boolean
```

```
    if data = d.data then
      return true
    else
      return false
```

Message

```
public class MESSAGE
  var dataField as DATA?
  var msgType as String
```

Correlation Set

```
class CORRELATIONSET
  var properties as Map of String to DATA
//For now we suppose that each correlation set only has ONE property in it.
//Size(properties) = 1
//-----------------------messageContainsTokens-----------------
  messageContainsTokens(m as MESSAGE) as Boolean
    choose i in Indices(properties)
      return m.dataField.isEqual(properties(i))
    ifnone
      WriteLine("ERROR!! This correlation had no property!!")
      return false
```

This method checks the compatibility of a single message to a correlation set. To check this, we normally should check if the message carries the correlation token values. For now, as correlation sets just have one property and messages just carry a single data field, it is implemented as above.

Activity

```
public class ACTIVITY
  var refNumber as Integer
```

Process

New Process Creation

This static function takes care of creating new processes and initializing its properties.

```
class PROCESS
  shared newProcess(id as Integer, oManager as OUTBOX_MANAGER) as PROCESS
    var newP as PROCESS
    step
      newP := new PROCESS("p" + id,oManager)
    step
      newP.rootProcess := newP
      newP.parentAgent := undef
    step
      return newP
```

Correlation Existence

This function checks to see whether any of the correlation sets in *cSet* already exists or not. *cSet* is the set of correlation sets for a specific activity where they are all tagged with initiation.

```
class PROCESS
  correlationExists(cSet as Set of CORRELATIONSET) as Boolean
    var result as Boolean = true
    step foreach cor in cSet
      choose c in correlations where c.name = cor.name
        skip
      ifnone
        result := false
    step
      return result
```

Satisfy

If the correlation set is already initiated the message has to carry the required values for all the business tokens specified by corresponding correlation sets. `satisfy` checks if the message satisfies this condition.

```
class PROCESS
  satisfy(inActivity as INPUT_ACTIVITY?, m as MESSAGE) as Boolean
    let corrs = getCorrelationSetsToSatisfy(intStr, inActivity)
    var flag = true

    if corrs = {} then
      return true
    else
      step foreach c in corrs
        choose c2 in correlations where c.name = c2.name
          if not (messageContainsTokens(c2,m)) then
            flag := false
        ifnone // Trying to acces a correlation that is not initiated yet.
          WriteLine("ERROR! Trying to access a correlation that is not yet
initiated!")
          flag := false
      step
        return flag
```

Initiate Correlation

This function initiates a set of correlation sets by initializing the properties defined in the correlation sets with the token values carried by the message.

```
class PROCESS
  Initiate_Correlation(cSet as Set of CORRELATIONSET, m as MESSAGE)
    forall c in cSet
      forall i in Indices(c.properties)
        c.properties(i) := m.dataField

      add c to correlations
```

Note: Although for now size (properties) = 1, it does not affect this method.

Global Functions

Opaque

```
opaque() as Integer
  let x = any y| y in {1..100}
  return x
```

```
class PROCESS
  satisfy(inActivity as INPUT_ACTIVITY?, m as MESSAGE) as Boolean
    let corrs = getCorrelationSetsToSatisfy(intStr, inActivity)
    var flag = true

    if corrs = {} then
      return true
    else
      step foreach c in corrs
        choose c2 in correlations where c.name = c2.name
          if not (messageContainsTokens(c2,m)) then
            flag := false
        ifnone // Trying to acces a correlation that is not initiated yet.
          WriteLine("ERROR! Trying to access a correlation that is not yet
initiated!")
          flag := false
      step
        return flag
```

Initiate Correlation

This function initiates a set of correlation sets by initializing the properties defined in the correlation sets with the token values carried by the message.

```
class PROCESS
  Initiate_Correlation(cSet as Set of CORRELATIONSET, m as MESSAGE)
    forall c in cSet
      forall i in Indices(c.properties)
        c.properties(i) := m.dataField

      add c to correlations
```

Note: Although for now size (properties) = 1, it does not affect this method.

Global Functions

Opaque

```
opaque() as Integer
  let x = any y| y in {1..100}
  return x
```

Random

```
random() as Boolean
  let x = any y| y in {1..100}
  if x >= 50 then
    return true
  else
    return false
```

Match

Match specifies whether message *m* can be matched to process instance *p* or not. If the corresponding waiting activity in *p* follows a correlation, the message can be matched to the process instance if and only if it carries the required correlation tokens.

```
Match(p as PROCESS, m as MESSAGE, inActivity as INPUT_ACTIVITY) as Boolean
  if accept(intStr,inActivity,m) then //checks the message source,destination
    if taggedWithCorrelation(intStr, inActivity) then
      if initiateCorrelation(intStr, inActivity) then
        if correlationExists(p,
                    getCorrelationSetsToInitiate(intStr, inActivity)) then
          return false
          //According to the LRM, the matching can only be performed in case
          //of pick activity, which is not covered in the current version.
        else
          return true
      else
        if satisfy(p, inActivity, m) then
          return true
        else
          return false
    else
      return true
  else
    return false
```

D.3. GUI-Related Additions

Model

```
class MODEL
  shared var assignments as Set of HISTORY_TUPLE = {}

//--------------------------messageIsReceived---------------------------------
  shared messageIsAssigned(p as PROCESS,m as MESSAGE,agent as RUNNING_AGENT,activity as ACTIVITY)
    t = new HISTORY_TUPLE(p,m,agent,activity)
    add t to assignments
//--------------------------showProgram---------------------------------------
  showProgram()
    var b as ArrayList
    step
      b := mapToArrayList(intStr.program)
    step
      view.setProgramBox(b)
//--------------------------showMessages--------------------------------------
  showMessages()
    var b as ArrayList
    step
      b := setToArrayList(inboxManager.inboxSpace)
    step
      view.refreshMessageList(b)
//--------------------------showOutspace--------------------------------------
  showOutspace()
    var b as ArrayList
    var newSet as Set of OUTBOX_TUPLE ={}
    step foreach (a,act) in outboxManager.outboxSpace
      add new OUTBOX_TUPLE(a,act) to newSet
    step
      b := setToArrayList(newSet)
    step
      view.refreshOutspaceList(b)
//--------------------------showProcesses-------------------------------------
  showProcesses()
    var b as ArrayList
    var nodes as Map of String to TreeNode = {->}
    step
      nodes := buildTree()
    step
      b := setToArrayList(values(nodes))
    step
      view.refreshProcessTreeView(b)
```

```
//------------------------informAssignments--------------------------------
  informAssignments()
    var b as ArrayList
    step
      b := setToArrayList(assignments)
    step
      view.refreshMsgHistoryList(b)
//-----------------------buildTree---------------------------------------
  buildTree() as Map of String to TreeNode
    var temp as Map of String to TreeNode = {->}
    var newAgentSet as Set of ACTIVITYAGENT = {}
    var ags as Set of ACTIVITYAGENT = {}
    var aliveProcesses as Set of PROCESS = processes
    step
      remove dummy from aliveProcesses
    step foreach p in aliveProcesses
      WriteLine("For process" + p.id)
      step
        temp(p.id) := new TreeNode(p.ToString())
      step //to find the leaves
        foreach a in p.subordinateAgentSet
          choose b in p.subordinateAgentSet where b.parentAgent = a
            skip
          ifnone //it means a is a leaf
            WriteLine(a.ToString() + "is leaf")
            temp(a.id) :=  new TreeNode(a.ToString())
            add a to newAgentSet
      step until newAgentSet = {}
        step
          ags := newAgentSet
          newAgentSet := {}
        step
          WriteLine("ags" + ags)
        step foreach a in ags
          choose b in p.subordinateAgentSet where a.parentAgent = b
            step
              if not(b.id in Indices(temp)) then
                temp(b.id) := new TreeNode(b.ToString())
            step
              temp(b.id).Nodes.Add(temp(a.id))
              remove temp(a.id)
              add b to newAgentSet
          ifnone  //a is a direct child
            skip
      step foreach a in ags
        temp(p.id).Nodes.Add(temp(a.id))
        remove temp(a.id)
    step
      return temp
```

Agent

```
public class AGENT
  virtual ToString() as String
```

Running Agent

```
public class RUNNING_AGENT
  var idCounter as Integer = 0
  public getId() as String
    return id
```

Process

```
public class PROCESS extends RUNNING_AGENT
  override ToString() as String
    step
      var s1 as String = "Process: " + id + " / Busy: " + busy + " / Current
Activity: "
    step
        s1 := s1 + mainActivity.ToString()
    step
      s1 := s1 + " / Correlations: "
    step foreach c in correlations
      s1 := s1 + c.ToString() + " // "
    step
      return s1

  getBusy() as Boolean
    return busy

  getCurrActivity() as ACTIVITY?
    return mainActivity
```

Activity Agents

Sequence Agent

```
class SEQUENCE_AGENT extends ACTIVITYAGENT
  override ToString() as String
    step
      var s1 as String = "Seq Agent " + id + " / Busy: " + busy + " / Current
Activity: "
```

```
    step
      if not(currentActivity = null) then
         s1 := s1 + currentActivity.ToString()
      else
         s1 := "Null"
    step
      return s1
```

Flow Agent

```
class FLOW_AGENT extends ACTIVITYAGENT
  override ToString() as String
    return "FLOW AGENT: " + id
```

Flow Thread Agent

```
class FLOW_THREAD_AGENT extends ACTIVITYAGENT
  override ToString() as String
    return "FLOW THREAD: " + id + " , Current Activity: " +
baseActivity.ToString()
```

Activities

```
public class ACTIVITY
  virtual ToString() as String
  virtual getInfo() as String
```

Sequence

```
public class SEQUENCE extends STRUCTURED_ACTIVITY
  override ToString() as String
    return "SEQUENCE" + ":" + refNumber
```

Flow

```
public class FLOW extends STRUCTURED_ACTIVITY
  override ToString() as String
    return "FLOW" + ":" + refNumber
```

Receive

```
public class RECEIVE extends INPUT_ACTIVITY
  override ToString() as String
    return "Receive" + ":" + refNumber
```

Reply

```
public class REPLY extends OUTPUT_ACTIVITY
  var msgType as String

  override ToString() as String
    return "Reply" + ":" + refNumber

  override getInfo() as String
    return msgType
```

Additions to Other Classes

Message

```
public class MESSAGE
  getMsgType() as String
    return msgType

  ToString() as String
    if dataField = null then
      return msgType
    else
      return msgType + " : " +  dataField.ToString()
```

Data

```
class DATA
  ToString() as String
    return "" + data
```

Correlation Set

```
class CORRELATIONSET
  ToString() as String
    step
      var s as String = name + ": "
    step foreach a in Indices(properties)
      s := s + a + ": " + properties(a).ToString() + " , "
    step
      return s
```

New Classes

Outbox Tuple

```
public class OUTBOX_TUPLE
  var agent as RUNNING_AGENT
  var activity as OUTPUT_ACTIVITY

  getAgent() as RUNNING_AGENT
    return agent

  getActivity() as OUTPUT_ACTIVITY
    return activity
```

History Tuple

```
public class HISTORY_TUPLE
  var proc as PROCESS
  var msg as MESSAGE
  var agent as RUNNING_AGENT
  var activity as ACTIVITY

  getProc() as PROCESS
    return proc

  getMsg() as MESSAGE
    return msg

  getAgent() as RUNNING_AGENT
    return agent

  getActivity() as ACTIVITY
    return activity
```

Interfaces

External View

```
[External]
class View
  public refreshMessageList(mArray as ArrayList)
  public refreshProcessTreeview(mArray as ArrayList)
  public refreshMsgHistoryList(mArray as ArrayList)
  public refreshOutspaceList(mArray as ArrayList)
  public setProgramBox(mArray as ArrayList)
```

Auxiliary Methods

```
// ------------------------setToArrayList-----------------------------------
setToArrayList(s as Set of Object) as ArrayList
  var mArray as ArrayList = new ArrayList()
  step foreach a in s
    mArray.Add(a)
  step
    return mArray
//-----------------------mapToArrayList---------------------------------------
mapToArrayList(m as Map of Integer to ACTIVITY) as ArrayList
  WriteLine("size" + Size(m))
  var mArray as ArrayList = new ArrayList(Size(m))
  step foreach i in Indices(m)
    mArray.Add(m(i))
  step
    return mArray
//------------------------mapToArrayList-------------------------------
mapToArrayList(m as Map of Integer to INT_ACTIVITY) as ArrayList
  WriteLine("size" + Size(m))
  var mArray as ArrayList = new ArrayList(Size(m))
  step foreach i in Indices(m)
    mArray.Add(m(i))
  step
    return mArray
```

D.4. Internal Structure

```
interface INTERNAL_STR
  sequenceCounter(s as SEQUENCE) as ACTIVITY?
//returns the next activity to be executed in a sequence agent

  processActivity() as ACTIVITY
//returns the main activity of the process

  accept(activity as ACTIVITY, m as MESSAGE) as Boolean
//returns true if the message has the correct type as required by the activity

  taggedWithCorrelation(activity as INPUT_ACTIVITY) as Boolean
//returns true if the activity is associated with a correlatin set

  initiateCorrelation(activity as INPUT_ACTIVITY) as Boolean
//returns true if the correlation set associated with the activity
//must be initiated

  getCorrelationSetsToInitiate(activity as INPUT_ACTIVITY?) as Set of
CORRELATIONSET
//returns the set of correlation set associated with the activity that
 //must be initiated

  getCorrelationSetsToSatisfy(activity as INPUT_ACTIVITY?) as Set of
CORRELATIONSET
//returns the set of correlation set associated with the activity that must
//be followed (satisfied)
```

Internal Activities

```
public class INT_ACTIVITY extends ACTIVITY
  var activityType as String
  virtual getActivity() as ACTIVITY
  getRefNumber() as Integer
    return refNumber
```

Internal Output Activity

```
public class INT_OUTPUT_ACTIVITY extends INT_ACTIVITY
  var opMsgType as String
  var correlationTags as Set of CORRELATION_USAGE

  override getActivity() as ACTIVITY
    if activityType = "reply" then
```

```
      return new REPLY(refNumber, opMsgType)
    else
      return new OUTPUT_ACTIVITY(15)

  override ToString() as String
    return refNumber +":" + activityType + ", Operation: " + opMsgType + "
CorrelationTags: " + correlationTags
```

Internal Input Activity

```
public class INT_INPUT_ACTIVITY extends INT_ACTIVITY
  var opMsgType as String
  var createInstance as Boolean
  var correlationTags as Set of CORRELATION_USAGE

  override getActivity() as ACTIVITY
    if activityType = "receive" then
      return new RECEIVE(refNumber)
    else
      return new INPUT_ACTIVITY(15)

  override ToString() as String
    return refNumber + ":" + activityType + ", Operation : " + opMsgType +
", CreateInstance: " + createInstance + " CorrelationTags: " + correlationTags
```

Internal Structured Activities

```
public class INT_STRUCTURED_ACTIVITY extends INT_ACTIVITY
  virtual getInsideActivities() as ArrayList
```

Internal Sequence Activity

```
public class INT_SEQUENCE_ACTIVITY extends INT_STRUCTURED_ACTIVITY
  var activitySet as Map of Integer to INT_ACTIVITY

  override getInsideActivities() as ArrayList
    return mapToArrayList(activitySet)

  override getActivity() as ACTIVITY
    return new SEQUENCE(refNumber)

  override ToString() as String
    return refNumber + " :" + activityType
```

Internal Flow Activity

```
public class INT_FLOW_ACTIVITY extends INT_STRUCTURED_ACTIVITY
  var activitySet as Set of INT_ACTIVITY

  override getInsideActivities() as ArrayList
    return setToArrayList(activitySet)

  override getActivity() as ACTIVITY
    if activityType = "flow" then
      var temp as Set of ACTIVITY = {}
      step foreach act in activitySet
          add act.getActivity() to temp
      step
        return new FLOW(refNumber, temp)

  override ToString() as String
    var s as String = refNumber + " :" + activityType + " "
    step
      return s
```

Array-Based Internal Structure

```
class ARRAY_BASED_INT_STR implements INTERNAL_STR
  structure CORRELATION_USAGE
    name as String
    initiate as Boolean
    pattern as Boolean

  var activities as Map of Integer to INT_ACTIVITY = {->}
  var program as Map of Integer to INT_ACTIVITY = {->}
  var sequenceMap as Map of SEQUENCE to Integer ={->}
```

Initialize

```
class ARRAY_BASED_INT_STR
  initialize()
    step
      activities(1)    :=    new    INT_INPUT_ACTIVITY(1,"receive","msg1",true,
{CORRELATION_USAGE("wow",true,false)})
      activities(2) := new INT_OUTPUT_ACTIVITY(2,"reply","sendID",{})
      activities(3)    :=    new    INT_INPUT_ACTIVITY(3,"receive","msg2",false
,{CORRELATION_USAGE("wow",false,false)})
      activities(5) := new INT_INPUT_ACTIVITY(5,"receive","m2",false , {})
```

```
    step
      activities(6) := new INT_OUTPUT_ACTIVITY(6,"reply","sendReply", {})
      activities(7)                                          :=                                     new
INT_INPUT_ACTIVITY(7,"receive","msg3",false,{CORRELATION_USAGE("c3",true,false)
})
      activities(8) := new INT_INPUT_ACTIVITY(8,"receive","msg4",false, {})
    step
      activities(4)     :=     new     INT_FLOW_ACTIVITY(4,"flow",{activities(6),
activities(7),activities(8)})
    step
      activities(0)      :=      new      INT_SEQUENCE_ACTIVITY(0,"Sequence",{0->
activities(1), 1-> activities(2), 2-> activities(3), 3-> activities(4), 4 -
>activities(5)})
    step
      program(0) := activities(0)
```

Process Activity

```
class ARRAY_BASED_INT_STR
  processActivity() as ACTIVITY
    return getActivity(program(0))
```

Sequence Counter

```
class ARRAY_BASED_INT_STR
  sequenceCounter(s as SEQUENCE) as ACTIVITY?
    match (activities(s.refNumber))
      seq as INT_SEQUENCE_ACTIVITY:
        if s in Indices(sequenceMap)
          if Size(seq.activitySet ) > sequenceMap(s) + 1 then
          //there is still some activities there
            sequenceMap(s) := sequenceMap(s) + 1
            return getActivity(seq.activitySet(sequenceMap(s)+1))
          else //process is ended
            return null
        else
          sequenceMap(s) := 0
          return getActivity(seq.activitySet(0))
      _:
        WriteLine("ERROR")
        return null
```

Accept

```
class ARRAY_BASED_INT_STR
  accept(activity as ACTIVITY, m as MESSAGE) as Boolean
    a = getIntActivity(activity)
    match a
      inputA as INT_INPUT_ACTIVITY:
        if m.msgType = inputA.opMsgType then
          return true
        else
          return false
      _:
        WriteLine("ERROR")
        return false
```

Tagged With Correlation

```
class ARRAY_BASED_INT_STR
  taggedWithCorrelation(activity as INPUT_ACTIVITY) as Boolean
    a = getIntActivity(activity)
    match a
      inputA as INT_INPUT_ACTIVITY:
        if not (inputA.correlationTags = {}) then
          WriteLine("HEY! THERE IS CORRELATION!")
          return true
        else
          return false
      _:
        WriteLine("ERROR")
        return false
```

Initiate Correlation

```
class ARRAY_BASED_INT_STR
  initiateCorrelation(activity as INPUT_ACTIVITY) as Boolean
    a = getIntActivity(activity)
    match a
      inputA as INT_INPUT_ACTIVITY:
        choose c in inputA.correlationTags where c.initiate
          return true
        ifnone
          return false
      _:
        WriteLine("ERROR")
        return false
```

Get Correlation Sets to Initiate

```
class ARRAY_BASED_INT_STR
  getCorrelationSetsToInitiate(activity   as   INPUT_ACTIVITY?)   as   Set   of
CORRELATIONSET
    var cSet as Set of CORRELATIONSET = {}
    if activity <> null then
      a = getIntActivity(activity)
      match a
        inputA as INT_INPUT_ACTIVITY:
          step foreach c in inputA.correlationTags where c.initiate
            add new CORRELATIONSET(c.name,{"id"->new DATA(0) }) to cSet
          step
            return cSet
        _:
          return cSet
     else
       WriteLine ("ERROR")
       return cSet
```

Get Correlation Sets to Satisfy

```
class ARRAY_BASED_INT_STR
  getCorrelationSetsToSatisfy(activity   as   INPUT_ACTIVITY?)   as   Set   of
CORRELATIONSET
    var cSet as Set of CORRELATIONSET = {}
    if activity <> null then
      a = getIntActivity(activity)
      match a
        inputA as INT_INPUT_ACTIVITY:
          step foreach c in inputA.correlationTags where not(c.initiate)
            add new CORRELATIONSET(c.name,{"id"->new DATA(0) }) to cSet
          step
            return cSet
        _:
          return cSet
     else
       WriteLine("ERROR")
       return cSet
```

Get Internal Activity

```
class ARRAY_BASED_INT_STR
  getIntActivity(activity as ACTIVITY) as INT_ACTIVITY
    return activities(activity.refNumber)
```

Random Internal Structure

```
class RANDOM_INT_STR implements INTERNAL_STR
  var num as Integer = 0
//-----------------------------processActivity-------------------------------
  processActivity() as ACTIVITY
    return new RECEIVE(0)
//------------------------------sequenceCounter-------------------------------
  sequenceCounter(s as SEQUENCE) as ACTIVITY?
    WriteLine("Current activity is a RECEIVE")
    return new RECEIVE(0)
//-----------------------------------accept-----------------------------------
  accept(activity as ACTIVITY, m as MESSAGE) as Boolean
    let result = random()
    WriteLine("The message has correct type for the waiting activity " +
result)
    return result
//--------------------------taggedWithCorrelation-----------------------------
  taggedWithCorrelation(activity as INPUT_ACTIVITY) as Boolean
      let result = random()
      WriteLine("The activity is tagged with correlation: " + result)
      return result
//---------------------------initiateCorrelation------------------------------
  initiateCorrelation(activity as INPUT_ACTIVITY) as Boolean
      let result = random()
      WriteLine("Correlation must be initiated: " + result)
      return result
//--------------------------getCorrelationSetsToInitiate----------------------
  getCorrelationSetsToInitiate(activity as INPUT_ACTIVITY?) as Set of
CORRELATIONSET
    var cSet as Set of CORRELATIONSET = {}
    add new CORRELATIONSET("wow",{"id"->new DATA(0)}) to cSet
    if activity <> null then
      return cSet
    else
      WriteLine ("ERROR")
      return cSet
//--------------------------getCorrelationSetsToSatisfy-----------------------
  getCorrelationSetsToSatisfy(activity as INPUT_ACTIVITY?) as Set of
CORRELATIONSET
    var cSet as Set of CORRELATIONSET = {}
    add new CORRELATIONSET("wow",{"id"->new DATA(0)}) to cSet

    if activity <> null then
      return cSet
    else
      WriteLine ("ERROR")
      return cSet
```

References

[1] *The Abstract State Machine Language* [online]. Microsoft Research, Foundations of Software Engineering [cited June 2003]. Available from: <www.research.microsoft.com/foundations/AsmL>.

[2] A. Benczur, U. Glässer and T. Lukovszki. "Formal Description of a Distributed Location Service for Ad Hoc Mobile Networks." In *Abstract State Machines 2003 - Advances in Theory and Practice*, eds. E. Börger, A. Gargantini, E. Riccobene. Vol. 2589 of LNCS, pages 204-217, Springer, 2003.

[3] A. Blass and Y. Gurevich. "Background, Reserve, and Gandy Machines." In *Proceedings of CSL'2000*, eds. Peter Clote and Helmut Schwichtenberg. Vol. 1862 of LNCS, pages 1-17, Springer, 2000.

[4] E. Börger. "A Logical Operational Semantics for Full Prolog. Part I: Selection Core and Control." In *CSL'89. 3rd Workshop on Computer Science Logic*, eds. E. Börger, H. Kleine Büning, M. M. Richter, and W. Schönfeld. Vol. 440 of LNCS, pages 36–64. Springer, 1990.

[5] E. Börger. "A Logical Operational Semantics of Full Prolog. Part II: Built-in Predicates for Database Manipulation." In *Mathematical Foundations of Computer Science*, ed. B. Rovan. Vol. 452 of LNCS, pages 1–14. Springer, 1990.

[6] E. Börger. "The Origins and the Development of the ASM Method for High Level System Design and Analysis." *Journal of Universal Computer Science*, Vol. 8, no. 1, pages 2-74, 2003.

[7] E. Börger, U. Glässer and W. Müller. "The Semantics of Behavioral VHDL'92 Descriptions." In *Proc. of EURO-VHDL'94*, pages 500-505, Grenoble, France, Sep. 1994.

[8] E. Börger, U. Glässer and W. Müller. "Formal Definition of an Abstract VHDL'93 Simulator by EA-Machines." In *Formal Semantics for VHDL*, eds. C. Delgado Kloos and Peter T. Breuer, pages 107-139, Kluwer Academic Publishers, 1995.

[9] E. Börger and R. Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer, 2003.

[10] F. Curbera, Y. Goland, J. Klein, F. Leymann, D. Roller, S. Thatte, and S. Weerawarana, *Business Process Execution Language for Web Services Version 1.1*, BEA Systems, International Business Machines Corporation, Microsoft Corporation, SAP AG, Siebel Systems, May 2003.

[11] A. van Deursen, P. Klint, and J. Visser. "Domain-Specific Languages: An Annotated Bibliography." In *ACM SIGPLAN Notices*, 35(6):97-105, June 2000.

[12] R. Eschbach , U. Glässer, R. Gotzhein, M. von Löwis and A. Prinz. "Formal Definition of SDL-2000 —Compiling and Running SDL Specifications as ASM Models." In *Journal of Universal Computer Science*, 7 (11): 1025-1050, Springer Pub. Co., Nov. 2001.

[13] R. Eschbach, U. Glässer, R. Gotzhein and A. Prinz. "On the Formal Semantics of SDL-2000: a Compilation Approach Based on an Abstract SDL Machine." In *Abstract State Machines — Theory and Application*, eds. Y. Gurevich, P.W. Kutter, M. Odersky and L. Thiele. Vol. 1912 of LNCS, pages 244-265, Springer-Verlag, 2000.

[14] R. Farahbod, U. Glässer, M. Vajihollahi. *Specification and Validation of the Business Process Execution Language for Web Services.* SFU-CMPT-TR-2003-06, Sep. 2003

[15] R. Farahbod, U. Glässer and M. Vajihollahi. "Specification and Validation of the Business Process Execution Language for Web Services." To appear in *Proc. of the 11th International Workshop on Abstract State Machines (ASM'2004)*, Germany, May 2004.

[16] Foundations of Software Engineering Group at Microsoft [online, cited June 2003]. Available from: <http://research.microsoft.com/fse>

[17] N. E. Fuchs. "Specifications are (Preferably) Executable." In *Software Engineering Journal*, pages 323-324, September 1992.

[18] U. Glässer, R. Gotzhein and A. Prinz. "Formal Semantics of SDL-2000: Status and Perspectives." In *Computer Networks*, Vol. 42, Issue 3, pages 343-358 (June 2003), ITU-T System Design Languages (SDL), Elsevier, 2003

[19] U. Glaesser, Y. Gurevich and M. Veanes. *An Abstract Communication Model.* Technical Report MSR-TR-2002-55, Microsoft Research.

[20] U. Glässer, Y. Gurevich, and M. Veanes. "An Abstract Communication Architecture for Modeling Distributed Systems." Submitted to *IEEE TSE*, 2003.

[21] U. Glässer, M. Vajihollahi, "Engineering Concurrent and Reactive Systems with Distributed Real-Time Abstract State Machines." Submitted to *IFIP World Computer Congress*, France, August 2004.

[22] U. Glässer and M. Veanes. "Universal Plug and Play Machine Models: Modeling with Distributed Abstract State Machines." In *Design and Analysis of Distributed Embedded Systems*, eds. B. Kleinjohann, K. H. Kim, L. Kleinjohann, A. Rettberg. Kluwer Academic Publishers, 2002.

[23] Y. Gurevich. "Evolving Algebras 1993: Lipari Guide." In *Specification and Validation Methods*, ed. E. Börger. pages 9-36, Oxford University Press, 1995.

[24] Y. Gurevich and J. Huggins. "The Semantics of the C Programming Language." Vol. 702 of LNCS, 1993, pages 274-308.

[25] Y. Gurevich and J. Huggins. "The Railroad Crossing Problem: An Experiment with Instantaneous Actions and Immediate Reactions." In *Computer Science Logic*, ed. H.K. Büning. Vol. 1092 of LNCS, pages 266-290, Springer, 1996.

[26] Y. Gurevich and N. Tillmann. "Partial Updates: Exploration." In *Journal of Universal Computer Science*. Vol. 7, no. 11 (2001): 918-952, Springer.

[27] I. J. Hayes, and C.B. Jones. "Specifications are not (necessarily) executable." In *Software Engineering Journal*, no. 6 (1989): 330-338.

[28] ITU-T Recommendation Z.100 Annex F (11/00), "SDL Formal Semantics Definition," In *International Telecommunication Union*, Geneva, 2001.

[29] W. Mueller, J. Ruf, D. Hofmann, J. Gerlach, T. Kropf, and W. Rosenstiehl. "The Simulation Semantics of SystemC." In *Proc. of DATE 2001*. IEEE CS Press, March 2001.

[30] *SOAP Version 1.2 Part 0: Primer, W3C Recommendation 24* [online, cited June 2003]. Available from: <www.w3c.org/TR/soap12-part0>.

[31] R. Stärk, J. Schmid and E. Börger. *Java and the Java Virtual Machine: Definition, Verification, Validation.* Springer, 2001.

[32] *Web Services Description Language (WSDL) Version 1.2 Part 1: Core Language* [online, cited June 2003]. Available from: <www.w3c.org/TR/wsdl12>.

[33] *Web Services Transaction (WS-Transaction)* [online, cited August 2002]. BEA Systems, International Business Machines Corporation, Microsoft Corporation, Inc.
Available from: <www.ibm.com/developerworks/library/ws-transpec>.

[34] WSBPEL TC at the Organization of Advancement of Structured Information Standards (OASIS) [online]. Available from: <www.oasis-open.org>.