Specification and Validation of the Business Process Execution Language for Web Services[1]

Roozbeh Farahbod, Uwe Glässer and Mona Vajihollahi

School of Computing Science
Simon Fraser University, Burnaby, B.C., Canada
{rfarahbo/glaesser/mvajihol@cs.sfu.ca}

Abstract. We formally define an abstract executable semantics for the Business Process Execution Language for Web Services in terms of a distributed ASM. The goal of this work is to support the design and standardization of the language. *"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. The language definition assumes an infrastructure for running Web services on some asynchronous communication architecture. A business process is built on top of a collection of Web services performing continuous interactions with the outside world by sending and receiving messages over a communication network. The underlying execution model is characterized by its concurrent and reactive behavior making it particularly difficult to predict dynamic system properties with a sufficient degree of detail and precision under all circumstances.

1 Introduction

In this paper, we formally define an abstract operational semantics for the Business Process Execution Language for Web Services—*BPEL4WS* (or *BPEL*) [6] – in terms of a real-time *distributed abstract state machine* (DASM) [14], [12]. Version 1.1 of the informal BPEL language description [6], henceforth called the language reference manual or LRM, is a forthcoming industrial standard proposed by the OASIS Web Services Business Process Execution Language Technical Committee [21]. Intuitively, BPEL is an XML based formal language for modeling and design of the networking protocols for 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) [20]. 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.

[1] Partly supported through grants from NSERC and SFU President's Research Grant.

Our work on BPEL builds on extensive experience from semantic modeling of various industrial system design languages, including the ITU-T language SDL [11], [7], [8] and the IEEE language VHDL [4], [3]. The goal of this work is twofold. Formalization of language semantics serves two main purposes: (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 lists about seventy basic issues. *"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 [21].

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. In that respect, a pragmatic orientation of formal software models and their use for practical purposes such as standardization demands for a gradual formalization of the key language attributes at different levels of abstraction and with a degree of detail and precision as needed [11].

Our definition of the abstract operational semantics presented here forms a BPEL 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*, specifies the relevant technical details and provides the full DASM model of the core constructs of the language. Finally, the third layer, called *execution model*, provides an abstract executable semantics of BPEL implemented in AsmL [18]. To this end, the BPEL Abstract Machine model forms a hierarchy consisting of three DASM ground models [1], [5] obtained as the result of stepwise refinements of the abstract model. The execution model is complemented by a GUI facilitating experimental validation through simulation and animation of abstract machine runs.

The paper is organized as follows. Section 2 briefly illustrates the concept of Web services architecture and gives an overview of BPEL. Section 3 introduces the abstract model, Section 4 the intermediate model, and Section 5 the execution model. In Section 6, we discuss the verification of key language properties. Section 7 concludes the paper.

2 Web Services Architecture

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-commerce. Each of these standards targets a specific domain within the Web services space. For example, the widely used Simple Object Access Protocol

(SOAP) [19] defines a standard message passing protocol, while WSDL provides a standard way of describing Web services [20].

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 fixed 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).

In April 2003, members of OASIS[2], including IBM and Microsoft among other leading companies in the e-commerce market, formed a Technical Committee [21] in order to continue work on BPEL version 1.1 with the objective to establish a standardized modeling platform and language that enables and accelerates systems design and IP exchange.

2.1 Overview of BPEL

A BPEL process and its partners are defined as abstract WSDL services, and they use abstract messages defined by WSDL model for interaction. Figure 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 interaction between the process and its partners. This definition is just a template for creating business process instances. At least one *start activity*[3] must be defined in the activity of such a template. Whenever a message arrives for a start activity, a new instance of the business process is created and starts its execution. Therefore, process creation in BPEL is always implicit.

2.2 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 callback from the shipping company and upon receiving that callback, it replies back to the customer indicating the order is received and processed successfully.

Figure 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

[2] Organization for the Advancement of Structured Information Standards

[3] 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.

certain *port type* specifying some set of operations. Operations can be either Input-Only, Output-Only, or Input-Output.

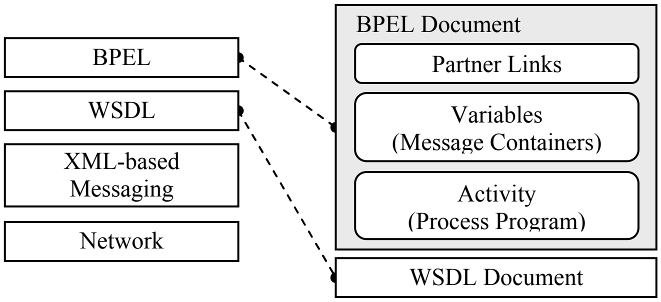

Fig. **1**. BPEL is defined on top of WSDL.

An abstract schema of the *e*-Book Store business process can also be found in Fig 2. 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).

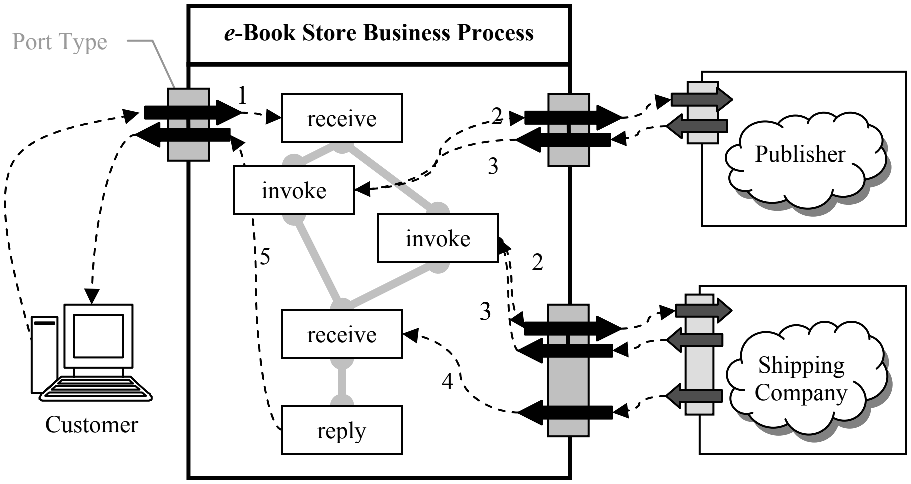

Fig. 2. Inside the *e*-Book Store business process

2.3 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 attributed *abstract syntax tree* [11]. 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 defined our own abstract syntax, which is presented in [9].

2.4 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 [6].

The need for such a mechanism can be easily 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 the messages to the correct process instances. One standard approach to this problem is to carry a business token (e.g. an order number) in all transactions between *e*-Book Store and the publisher. In this way, all the messages that arrive for a specific process instance should carry the desired business token.

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 correlation tokens are identical for all the messages in that correlation group. In this way, an application-level conversation between business process instances is identified.

2.5 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 basic activities. It is important that structured activities can be nested. Structured activities include *sequence*, *switch*, *flow, pick* and *while*. *Sequence* structures a collection of activities to take place one after another. *Switch* provides the ability to choose among a collection of activities. *Flow* enables concurrent execution of a set of activities. *Pick* waits on a set of events for one of them to occur and executes its corresponding activity. Finally, *while* executes an activity repeatedly until its condition is no longer true.

2.6 Long-running business process, compensation behavior

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 larger unit of work that is being abandoned"* [6, 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). Further information on the LRTs and their formal specification is beyond the scope of this paper. Nevertheless, we have considered this concept as an important extension to the core model as is described in [9].

3 Formalization of the Web Services Architecture

We formalize the key functional attributes of the BPEL Web services architecture based on the asynchronous computation model of distributed ASMs [14]. The primary focus is on the concurrent and reactive behavior of Web services and their interaction through communication networks. This includes concurrent control structures, communication primitives, event handling mechanisms, compensation handling, 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 behavior of our BPEL abstract machine.

Logically, the 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 behavior of the network from the behavior 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.

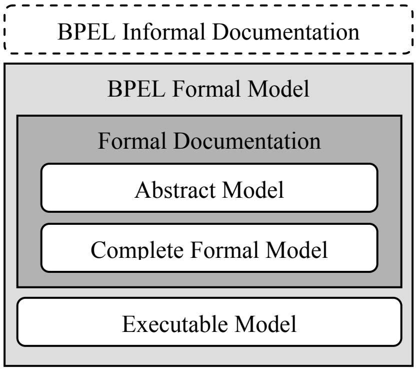

Fig. 3. A Three Level Approach: From formal documentation to the executable model

In this paper, we concentrate on the *service abstract machine* model, whereas a *network abstract machine* model is defined in [12]. The composition of these two machine models 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, i.e. the ports at the communication endpoints via which services send and receive messages.

The overall organization of the BPEL abstract machine splits into three different layers as illustrated in Figure 3. The abstract model is introduced below; the complete formal model and the executable model are presented in Section 4 and Section 5.

3.1 DASM computation model

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. Intuitively, M consists of a collection of autonomously operating *agents* from some finite set *AGENT*. This set changes dynamically over runs of M as required to model varying computational resources. The behavior of an agent a, in a given state S of M, is defined by the program $program_S(a)$. An agent a can be terminated by resetting $program_S(a)$ to *undef* (not representing a valid program). To introduce a new agent b in state S, a valid program has to be assigned to $program_S(b)$.

The creation and the termination of an agent a is stated by the following two operations which, at the same time, also update the (sub-)domain of agents to which a belongs.

new *a* : ⟨domain⟩ // creates a new agent *a* of type ⟨domain⟩ and sets *program(a)*

stop *a* // discards agent *a* from the domain of agents and resets *program(a)*

To cope with partial updates of sets, we follow the solution proposed in [16] and use the following operations for adding/removing an element to/from a set.

add *a* **to** S //Adds element *a* to set *S*

remove *a* **from** *S* //removes element *a* from set *S*

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

$$AGENT_S \equiv \{\, x \in X : program_S(x) \neq undef \,\}$$

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

Concurrency and Real Time. Intuitively, the agents of M model the concurrent control threads in the execution of Π_M. Agents interact with each other by reading and writing shared locations of global machine states. The underlying semantic model regulates such interactions so that potential conflicts are resolved according to the definition of *partially ordered runs* [14]. Real time behavior imposes additional constraints on

DASM runs ensuring that the agents react instantaneously [15]. For details see our technical report [9].

3.2 BPEL Abstract Model

The top layer of the BPEL abstract machine, called abstract model, provides an overview of the architecture and defines the underlying modeling 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 (i.e., the communication network) through two interface components, called *inbox manager* and *outbox manager*, as shown in Fig. 4

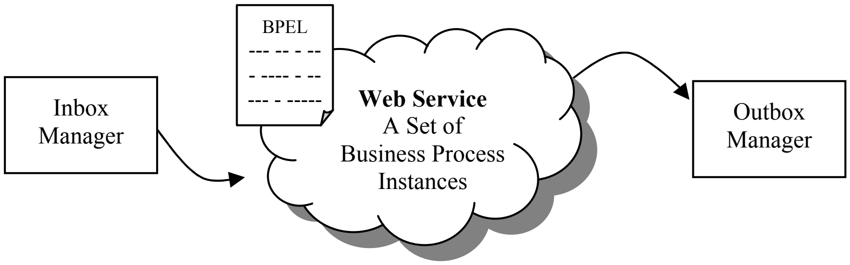

Fig. 4. High-level structure of our BPEL model.

The inbox manager 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 output messages from process instances to the network. Inbox managers, outbox managers, and process instances are modeled by three different types of DASM agents. Additionally, we introduce two further agent types, *activity agents* and *handler agents*. Each process agent is responsible to execute a single process instance; it uses dynamically created activity agents for executing complex (structured) activities. Handler agents are responsible for compensation handling or fault handling during the execution of a process instance.

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

In the initial DASM state, there are only three DASM agents: the inbox manager, the outbox manager and a dummy process. This dummy process instance simplifies the method of creating 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 behavior of the inbox manager agent.

```
domain MESSAGE
inboxSpace: INBOX_MANAGER → MESSAGE-set

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
```

The predicate match (p**:** PROCESS, m**:** MESSAGE) checks whether message m can be delivered to process p or not, trying to match message type and correlation information between the waiting process and the incoming message.

In general, a BPEL program combines two different types of activities: basic activities and structured activities. Structured activities impose an execution order to a collection of basic activities. The execution of each structured activity inside a process instance is modeled by a single DASM agent of type activity agent. Figure 5 shows the control structure of DASM activity agents where one can associate one branch from the root to a leaf with each single process instance.

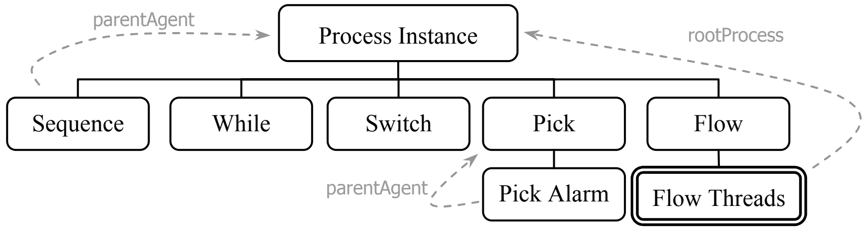

Fig. 5. Control structure of DASM activity agents

Below is the DASM program that abstractly specifies the behavior of process agents. In this abstract model, we do not provide the definition of Execute_Activity.

```
RUNNING_AGENT ≡ PROCESS U ACTIVITY_AGENT
//RUNNING_AGENT is the set of agents that execute (run) an activity.
startedExecution: PROCESS → BOOLEAN              //initial value: false
// Tells whether the process has started its execution or not.
suspended: RUNNING_AGENT → BOOLEAN            //initial value: false
// An agent is suspended while one of its activities is being executed.

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

4. Complete Formal Model

By refining the abstract model of Section 3.3, we obtain the intermediate model, which provides the full DASM model of the core constructs of BPEL. The intermediate model forms the basis for deriving the executable model in Section 5.

The previous section described how a PROCESS agent executes its main activity, but we did not define Execute_Activity at that level. Following the definition in BPEL, an activity can be any of the structured or basic activities, as follows:

```
domain REPLY
domain RECEIVE
domain FLOW
domain SEQUENCE
ACTIVITY ≡ REPLY ∪ RECEIVE ∪ FLOW ∪ SEQUENCE ∪ …
// … and all other activity domains
```

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.

```
domain FLOW_AGENT
domain SEQUENCE_AGENT
domain FLOW_THREAD_AGENT
ACTIVITY_AGENT ≡ FLOW_AGENT ∪ SEQUENCE_AGENT ∪ FLOW_THREAD_AGENT ∪ …
// … and all other agents for structured activities
```

```
// In the following rule, the predicates linkStatusDefined and joinCondition state
// synchronization dependencies between concurrent activities. Their definition is
// captured in the complete formal model by checking certain conditions before
// executing an activity. For brevity, these conditions are left abstract here.

//suspended is set to true before entering this module
Execute_Activity(activitiy: ACTIVITY) ≡
    if  linkStatusDefined then
        if joinCondition then
            if  activity ∈ RECEIVE then
                Execute_Receive(activity)

            … //and all other basic activities

            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
            … //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.
    where
        linkStatusDefined ≡ …, joinCondition ≡ …
```

In connection with structured activities, we define a function parentAgent for linking the parent agent and the subordinate activity agent. A process instance may have a number of subordinate agents that handle the structured activities inside the process instance. 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. These functions are defined as follows.

```
parentAgent: ACTIVITY_AGENT → RUNNING_AGENT
// Parent agent (one layer above in the creation tree) of an agent
rootProcess: RUNNING_AGENT→ PROCESS
// Returns the process agent to which this running agent belongs.
subordinateAgentSet: PROCESS → ACTIVITY_AGENT-set
// Returns the set of activty agents that have been created and work
// under control of this process.
```

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

$$\text{subordinateAgentSet(p: PROCESS)} \equiv \{\text{a} \mid \text{a} \in \text{ACTIVITY\_AGENT where rootProcess(a) = p}\}$$

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

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

Going entirely through the complete formal model is outside of the space limitations of this paper. Sections 4.1 and 4.2 thus focus on two representative examples for illustrating the BPEL abstract machine model, a basic activity and a structured activity. For further details and complementary parts of the model see [9].

4.1 A Basic Activity: Receive

As is described in [6, Section 11.4], receive activity plays an important role for a business process both in its life cycle[4] and in its service providing to partners.

In order to execute a receive activity for a given process instance, the inbox manager has to be informed that this process instance (or one of its subordinate agents) is waiting for a message. This is done by adding an inputDescriptor to the waitingForMessage set of the root process. inputDescriptor contains sufficient information about the required message and the agent that is waiting for that message. In this way the inbox manager can inspect this list and check whether any of the desired messages

[4] “The only way to instantiate a business process in BPEL is to annotate a receive [or pick] activity with the createInstance attribute set to "yes".” [6, section 11.4]

is received, and if so, assigns it to the matching process instance. Therefore, the agent has to wait until the inbox manager assigns a message to it. The Boolean function receiveMode is used to distinguish between the initialization mode and the waiting mode. The inputDescriptor is removed from the set as soon as a message is assigned to its corresponding activity. Thus, the agent will be informed about the assignment and can proceed with processing the message.

```
receiveMode: RUNNING_AGENT→ BOOELAN //initial value: false
waitingForMessage: PROCESS → <RUNNING_AGENT, ACTIVITY>-set
// For each process, this set indicates the activities waiting for a message.

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
                // Input descriptor is removed, message is received.
                receiveMode(self) := false
                suspended(self) := false //Releasing self.
    where waitingSet = waitingForMessage( rootProcess(self) )
```

4.2 A Structured Activity: Flow

A flow activity groups a set of activities and enables their concurrent execution. A flow completes when all the activities in the flow have completed [6].

For each structured activity, there is an activity agent for executing it. *Flow agent* is responsible for executing a flow activity. To concurrently execute the activities declared inside the flow activity, the flow agent creates a set of *flow thread agents*. Each flow thread agent is responsible for executing one such activity. When all the threads have finished, the flow agent releases its parent and terminates itself.

```
flowActivitySet: FLOW → ACTIVITY-set
// The set of all concurrent activities grouped inside a FLOW activity
flowAgentSet: FLOW_AGENT → FLOW_THREAD_AGENT-set // initial value: ∅

FLOWPROGRAM ≡
    if ¬suspended(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)
        suspended(self) := true
    else
        if flowAgentSet(self) = ∅ then // All threads are done, flow activity is completed.
            suspended(parentAgent(self)) := false // The parent agent is released.
            stop self
```

A flow thread agent executes a single activity. Thus, its program is very similar to a process agent, except that when the execution of the activity is completed, the flow thread agent informs the flow agent by removing itself from the flow agent set.

```
FLOWTHREADPROGRAM ≡
    if ¬suspended(self) and ¬startedExecution(self) then
        startedExecution(self) := true
        suspended(self) := true
    if suspended(self) then
        Execute_Activity(baseActivity(self))
    if ¬suspended(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 as is terminated.
```

5 Execution 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 requirements specifications provides us with an effective instrument to further eliminate undesirable behavior and hidden side effects already in early design stages [11]. In combination with analytical techniques, simulation and testing can provide valuable feedback for establishing key system language attributes and exploring alternative design choices. In our project, we use AsmL [18] for this purpose.

5.1 AsmL

AsmL is a rich language and its advanced language constructs are definitely helpful in rapid prototyping and object oriented software development. For the purpose of our project, however, we have deliberately chosen a subset of the language, which is as close as possible to 'pure ASMs.' To facilitate modeling of the BPEL semantics, a tight relation between the full DASM model and the derived execution model is of utmost importance. Though, in order to be executable, some changes and additions were inevitable. A main weakness of AsmL is its lack of direct support for dealing with concurrency. There are no built-in constructs for simulating concurrent control threads; rather such an execution model needs to be hand coded. Ultimately, one would even expect a distributed runtime system allowing to perform truly distributed computations of DASM models encoded in AsmL.

5.2 The Model in AsmL

Intuitively, the AsmL encoding splits into four separate modules, each of which deals with a basically different aspect: *(1) the original model (2) the internal environment (3) the refinement of the original model,* and *(4) GUI-related extensions.*

The original model is basically the translation of the intermediate model to AsmL, where the main challenge was to keep it close to the pure ASMs.

The internal environment 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 running an activity as defined in the BPEL process definition and determined by the history of that specific instance. One option is to encapsulate all the relevant information inside the respective entities of the model. For example, we can keep partner, port type, operation, variable and correlation sets of a receive activity inside it. Abstractly, we assumed that there is an oracle that provides this information whenever we ask for it. In the execution model, we replace this oracle with the internal environment.

In the stepwise refinement of the original model, abstract parts are refined depending on their role in the model, either by non-determinism or assigning clear deterministic behavior to them. In some cases, complex substructures had to be introduced. For example, in order to model the correlation behavior 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 BPEL. Besides, a predicate is defined to check the compatibility of a message to a correlation set, i.e. to check whether the message contains the required correlation tokens or not.

```
class CORRELATIONSET
  var properties as Map of PROPERTY to DATA
  messageContainsTokens(m as MESSAGE) as Boolean
// This method checks the compatibility of a message to a correlation
// set. It should check if the message carries the correlation tokens.
```

Finally, 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[5]. By utilizing AsmL's APIs with C#, we were able to integrate the model with its GUI, by defining an appropriate interface called *View*. For details of the execution model see [9].

5.3 Experimental Validation Results

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.*" [6, 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:

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

```
<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, on *portType* PT1 and for *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 behavior of different implementations of the language.

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

6 Verification aspects

In the current language definition, there are a number of open issues on how to establish certain key system attributes of Web services for business processes. Among those are several abstract language properties that justify formal reasoning either to prove that those properties indeed are implied by the language definition or to clarify the resulting implicit constraints on implementations of the language, the construction of Web services, and the logic design of business processes. Two examples are discussed below.

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*” [6, 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 correlation tokens.

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 instantiated by the same output activity; (2.2) the property must hold in all output activities where the correlation set is already instantiated.

(2.1) is confirmed by the LRM as well. The correlation set will be instantiated 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 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 loose end leads to further problems as follows.

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.”* [6, Section 11.4]. In order to clarify a request/response interaction, 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.”*[6, Section 11.4]. Although the definition of “outstanding” is not elucidated in the LRM, according to its interpretation by WSBPEL TC ([22, 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” cor=”C1”>
<receive partnerLink=”PL1” portType=”P1” operation=”O1” cor=”C2”>
<reply partnerLink=”PL1” portType=”P1” opr=”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.

7 Conclusions and Future Work

Our formalization of the key semantic aspects of BPEL in terms of a hierarchically defined BPEL Abstract Machine shows that the asynchronous DASM model indeed is a natural choice for defining a precise semantic foundation. The resulting formal model transforms the abstract language definition in two consecutive 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) clearly helps establishing coherence and consistence of the semantics, thereby improving the quality of the language definition. An advanced GUI facilitates such tasks (see also [9]).

A prerequisite for the feasibility of formalization when applied as a practical instrument in an industrial standardization context is conciseness, intelligibility and robustness [11]. 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 for a robust formalization framework that serves pragmatic needs. To this end, our 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 so that they effectively complement each other.

Our future work will concentrate on extending the BPEL Abstract Machine model towards modeling and integration of compensation behavior and fault handling.

Acknowledgements

We thank the anonymous reviewers for their valuable comments and suggestions for improvements. The idea of modeling design languages for automated business processes in terms of distributed real-time ASMs originates from joint work with Margus Veanes while one of the authors visited Microsoft Research in 2001.

References

1. A. Benczur, U. Glässer and T. Lukovszki. *Formal Description of a Distributed Location Service for Ad Hoc Mobile Networks*. In E. Börger, A. Gargantini, E. Riccobene (Eds.): Abstract State Machines 2003 - Advances in Theory and Practice, vol. 2589 of LNCS, pages 204-217, Springer, 2003.
2. 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.
3. 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.

4. E. Börger, U. Glässer and W. Müller. *Formal Definition of an Abstract VHDL'93 Simulator by EA-Machines*. In C. Delgado Kloos and Peter T. Breuer (Eds.): Formal Semantics for VHDL, Kluwer Academic Publishers, 1995, 107-139.
5. E. Börger. and R. Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer, 2003.
6. *Business Process Execution Language for Web Services Version 1.1*, BEA Systems, International Business Machines Corporation, Microsoft Corporation, SAP AG, Siebel Systems, May 2003.
7. 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*. Journal of Universal Computer Science, 7 (11): 1025-1050, Springer Pub. Co., Nov. 2001.
8. 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 Y. Gurevich, P.W. Kutter, M. Odersky and L. Thiele (Eds.): Abstract State Machines — Theory and Application, vol. 1912 of LNCS, pages 244-265, Springer-Verlag, 2000.
9. 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
10. N. E. Fuchs. *Specifications are (Preferably) Executable*. Software Engineering Journal, September 1992, pp323-324
11. U. Glässer, R. Gotzhein and A. Prinz. *Formal Semantics of SDL-2000: Status and Perspectives*. Computer Networks, Volume 42, Issue 3, pages 343-358 (June 2003), ITU-T System Design Languages (SDL), Elsevier, 2003
12. U. Glässer, Y. Gurevich, and M. Veanes. *An Abstract Communication Architecture for Modeling Distributed Systems*. Submitted to IEEE TSE, 2003.
13. U. Glässer and M. Veanes. *Universal Plug and Play Machine Models: Modeling with Distributed Abstract State Machines*. In B. Kleinjohann, K. H. Kim, L. Kleinjohann, A. Rettberg (Eds.): *Design and Analysis of Distributed Embedded Systems*, Kluwer Academic Publishers, 2002
14. Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9-36. Oxford University Press, 1995.
15. Y. Gurevich and J. Huggins. *The Railroad Crossing Problem: An Experiment with Instantaneous Actions and Immediate Reactions*. In H.K. Büning, editor, *Computer Science Logic*, Springer LNCS volume 1092, pages 266-290, 1996.
16. Y. Gurevich and N. Tillmann. *Partial Updates: Exploration*. Springer J. of Universal Computer Science. vol. 7, no. 11 (2001), pages 918-952.
17. I.J. Hayes, and C.B. Jones. *Specifications are not (necessarily) executable*. Software Engineering Journal, 1989, 4, (6), pp. 330-338
18. Microsoft Research: AsmL, www.research.microsoft.com/foundations/AsmL
19. *SOAP Version 1.2 Part 0: Primer*, W3C Recommendation 24 June 2003, www.w3c.org/TR/soap12-part0/
20. *Web Services Description Language (WSDL) Version 1.2 Part 1: Core Language*, W3C Working Draft 11 June 2003, www.w3.org/TR/wsdl12
21. WSBPEL TC at the Organization of Advancement of Structured Information Standards (OASIS), www.oasis-open.org.