An abstract machine architecture for web service based business process management

Roozbeh Farahbod,* Uwe Glässer and
Mona Vajihollahi

Software Technology Lab,
School of Computing Science,
Simon Fraser University
Burnaby, BC, Canada
E-mail: rfarahbo@cs.sfu.ca
E-mail: glaesser@cs.sfu.ca E-mail: mvajihol@cs.sfu.ca
*Corresponding author

Abstract: We define an abstract operational model of the Business Process Execution Language for Web Service (BPEL4WS) based on the *Abstract State Machine* (ASM) formalism. That is, we abstractly model dynamic properties of the key language constructs through the construction of a *Business Process Execution Language (BPEL) abstract machine*. Specifically, we present the *process execution model* and the underlying *execution lifecycle* of BPEL activities. The goal of our work is to provide a precise and well defined semantic framework for establishing the key language attributes. To this end, the BPEL abstract machine forms a comprehensive and robust formalisation closely reflecting the view of the informal language definition.

Keywords: web services orchestration; BPEL4WS; abstract operational semantics; abstract state machines; ASM; requirements specification.

Reference to this paper should be made as follows: Farahbod, R., Glässer, U. and Vajihollahi, M. (2006) ‘An abstract machine architecture for web service based business process management’, *Int. J. Business Process Integration and Management*, Vol. 1, No. 4, pp.279–291.

Biographical notes: Roozbeh Farahbod is a PhD candidate of computing science at Simon Fraser University. His research focuses on agile formal methods and applications of such methods in requirement specification and system design. He has been working on formal semantics of the Business Process Execution Language for Web Services since 2003. His main research is now focused on design and implementation of an executable specification language and a tool architecture that supports experimental validation and formal verification of abstract system models.

Uwe Glässer received an MSc, PhD and Habil (postdoctoral qualification) in computer science, from the University of Paderborn, Germany. He is an Associate Professor and director of the School of Computing Science and the founder of the Software Technology Lab at Simon Fraser University. His research interests focus on formal aspects of software technology and their applications in industrial systems design. He is also interested in novel application areas such as Computational Criminology, Public Safety and Computational Security.

Mona Vajihollahi received an MSc in computing science from Simon Fraser University in 2004 and is currently a PhD candidate of computing science. She is interested in formal aspects of software technology and novel applications of agile formal methods. Since 2002, her research has been focused on formal specification of Web Services architectures, computational modelling and analysis of patterns in crime and application of formal modelling techniques in civil aviation security.

1 Introduction

In this paper, we present an abstract operational semantics of the Business Process Execution Language for Web Services (BPEL4WS) (Andrews et al., 2003). BPEL4WS or Business Process Executive Language (BPEL) for short, is an XML-based specification language for automated business processes, proposed by OASIS (WSBPEL-TC, 2004) as a future standard for the e-business world. It provides distinctive expressive means for describing the process interfaces of web-based business protocols and builds on existing standards and technologies for Web services. In particular, it is defined on top of the service interaction model of W3C’s Web Services Description Language (WSDL) (W3C, 2003). Intuitively, a BPEL business process orchestrates the interaction between a collection of abstract WSDL services exchanging messages over a communication network.

Based on the Abstract State Machine (ASM) formalism and abstraction principles, (Börger and Stark, 2003), we define a BPEL abstract machine, called $\mathsf{BPEL}_{\mathcal{AM}}$: a concise and robust semantic framework for establishing the key language attributes in a precise and well defined form. That is, it captures the dynamic properties of the key language constructs defined in the language reference manual (Andrews et al., 2003), henceforth called LRM, including concurrent control structures, dynamic creation and termination of service instances, communication primitives, message correlation, event handling and fault and compensation handling. Dynamic properties of the Web services interaction model of a BPEL business process are modelled in abstract operational terms through finite or infinite machine *runs*. The concurrent and reactive nature of Web services and the need for dealing with time-related aspects in coordinating distributed activities call for an asynchronous execution model together with an abstract notion of real time. Hence, the formal definition of $\mathsf{BPEL}_{\mathcal{AM}}$ is based on the *distributed real-time ASM* model (Börger and Stark, 2003), commonly used for dealing with concurrent and reactive behaviour of complex distributed real-time systems.

The goal of the work presented here, first and foremost, is to provide a firm semantic foundation – a 'blueprint' of the language design – for checking consistency and validity of semantic properties. Formalisation is crucial to identify and eliminate deficiencies, such as ambiguities, loose ends and inconsistencies, that easily remain hidden in the informal language definition of the LRM (WSBPEL-TC 2004, Issue #42):

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

The Web services interaction model is characterised by its concurrent and reactive behaviour, making it particularly difficult to predict dynamic system properties with a sufficient degree of detail and precision under all circumstances. To this end, our formal semantic model of BPEL is meant to complement the LRM by sharpening informal requirements into precise specifications. Beyond inspection by analytical means, we also support experimental validation by making our formal semantics executable.

Secondly, building on practical experience from previous work on industrial standards for system specification and design languages, including the ITU-T language SDL (Glässer et al., 2003) and the IEEE language VHDL (Börger et al., 1995), our formalisation approach addresses vital pragmatic issues. The primary focus in semantic modelling of complex language properties is to establish the consistency and completeness of the language requirements specification. We separate here the concern of specification from that of verification, where the emphasis is on specification rather than verification: a proper and reliable formal specification is a prerequisite for any formal verification attempt and thus is the first and the most important step.

Another observation is that sensible use of formal techniques and supporting tools for practical purposes, such as standardisation, calls for a gradual formalisation of abstract requirements with a degree of detail and precision as needed (Glässer et al., 2003). To avoid a gap between the informal language definition and the formal semantics, the ability to model the language definition *as is* without making compromises is crucial. Consequently, we adopt here the LRM view and terminology, effectively formalising the intuitive understanding of BPEL as directly as possible and in a comprehensible and objectively verifiable form.

$\mathsf{BPEL}_{\mathcal{AM}}$ yields what is called an *ASM ground model* (Börger and Stark, 2003) of the BPEL language definition. Intuitively, a ground model serves as a precise semantic foundation for establishing functional requirements and design specifications in a reliable form without compromising conceivable refinements (Börger, 2003). A ground model can be inspected by analytical means (verification) and empirical techniques (simulation) using machine assistance as appropriate. Constructing such a ground model means a major effort, especially, as, in the case of BPEL, a clearly visible architectural model, which is central for dealing with intricate semantic issues, is a major shortcoming of (Andrews et al., 2003). We make this architecture visible as a direct result of our work presented here.

This paper is organised as follows. Section 2 briefly summarises the formal semantic framework. Section 3 introduces the core of the hierarchically defined $\mathsf{BPEL}_{\mathcal{AM}}$ model and Section 4 addresses important extensions to the $\mathsf{BPEL}_{\mathcal{AM}}$ core. Section 5 discusses related work and Section 6 concludes this paper.

2 Abstract state machines

This section briefly outlines the mathematical framework for semantic modelling at an intuitive level of understanding using common notions and structures from discrete mathematics and computing science. For details, we refer to the existing literature on the theory of ASMs (Gurevich, 2000) and their applications (Börger and Stark, 2003).

ASMs are best known for their versatility in semantic modelling of algorithms, architectures, languages, protocols and virtually all kinds of sequential, parallel and distributed systems. Widely recognised applications include semantic foundations of popular industrial system design languages, like SDL (Glässer et al., 2003), VHDL (Börger et al., 1995) and SystemC (Müller et al., 2003), programming languages, like JAVA (Stärk et al., 2001) and C# (Börger et al., 2005), communication architectures (Glässer and Gu, 2005), embedded control systems (Börger et al., 2000), et cetera.[1]

Leaning toward practical applications of formal methods, the above work resulted in a solid methodological foundation for building ASM ground models, where the role and nature of ground models, as discussed by Börger and Stark (2003), leads itself to the conclusion that the concept of ground model is inevitably present in every system design, but often not in an explicit form. The desire to make ground models visible has been a key factor in the development of ASM specification, validation and verification techniques (Börger and Stärk, 2003; Farahbod and Glässer, 2006).

2.1 Distributed real-time ASM

The asynchronous computation model of Distributed Abstract State Machine[2] (DASM) defines concurrent and reactive behaviour as observable in distributed computations performed by autonomously operating computational agents, in terms of *partially ordered runs*.

A DASM M is defined over a given vocabulary V by its program P_M and a non-empty set I_M of initial states. V consists of a finite collection of symbols denoting mathematical objects and their relation in the formal representation of M, where we distinguish *domain symbols*, *function symbols* and *predicate symbols*. Symbols that have a fixed interpretation regardless of the state of M are called *static*; those that may have different interpretations in different states of M are called *dynamic*. A state S of M results from a valid interpretation of all the symbols in V and constitutes a variant of a first-order structure, one in which all relations are formally represented as Boolean-valued functions.

Concurrent control threads in an execution of P_M are modelled by a dynamic set AGENT of computational *agents*. This set may change dynamically over runs of M, as required to deal with varying computational resources. Agents of M interact with one another and possibly also with the operational environment of M, by reading and writing shared locations of a global machine state, where the underlying semantic model regulates such interactions so that potential conflicts are resolved according to the definition of partially ordered runs.

P_M consists of a statically defined collection of agent programs $P_{M_1}, ..., P_{M_k}$, $k \geq 1$, each of which defines the behaviour of a certain *type* of agent in terms of state transition rules. The canonical rule consists of a basic update instruction of the form

$$f(t_1, t_2, \ldots, t_n) := t_0$$

where f is an n-ary dynamic function symbol and the $t_i\,'s$ $(0 \leq i \leq n)$ are terms. An update instruction specifies a pointwise function update, that is, an operation that replaces an existing function value by a new value to be associated with the given function arguments. Complex rules are inductively defined by a number of simple rule constructors allowing the composition of rules in various ways (as will be presented in the examples of Sections 3 and 4).

A computation of an individual agent of M, executing program P_{M_j}, is modelled by a finite or infinite sequence of state transitions of the form

$$S_0 \overset{\Delta_{S_0}(P_{M_j})}{\longrightarrow} S_1 \overset{\Delta_{S_1}(P_{M_j})}{\longrightarrow} S_2 \overset{\Delta_{S_2}(P_{M_j})}{\longrightarrow} \cdots$$

such that S_{i+1} is obtained from S_i, for $i \geq 0$, by firing $\Delta_{S_i}(P_{M_j})$ on S_i, where $\Delta_{S_i}(P_{M_j})$ denotes a finite set of updates computed by evaluating P_{M_j} over S_i. Firing an update set means that all the updates in this set are fired simultaneously in one atomic step. The result of firing an update set is defined if and only if the set does not contain conflicting updates (attempting to assign different values to the same location).

2.2 Reactivity and time

A DASM M models interactions with a given operational environment, the part of the external world with which M interacts, through actions and events as observable at external interfaces, formally represented by externally controlled functions. Of particular interest are *monitored functions*. Such functions change their values dynamically over runs of M, although they cannot be updated internally by agents of M. A typical example is the abstract representation of global system time.

In a given state S of M, the global time (as measured by some external clock) is given by a monitored nullary function *now*, taking values in a linearly ordered domain TIME. Values of *now* increase monotonic over runs of M. Additionally, $'\infty'$ represents a distinguished value of TIME, such that $t < \infty$ for all $t \in \text{TIME} \setminus \{\infty\}$. Finite time intervals are given as elements of a linearly ordered domain DURATION.

2.3 Notational convention

In our formal model we use a signalling method for agents to communicate with each other. For increased readability and a clear separation of concerns, we introduce syntactical abbreviations to employ this method. Precise semantic definitions of these abbreviations are provided in an Appendix to this paper.

Identical indentions are used in our formalisation to imply a parallel composition of rules. Function names are in *italic*, rule names are in SansSerif and domain names are in SMALLCAPS shape.

3 BPEL Abstract machine

This section introduces the core $\text{BPEL}_{\mathcal{AM}}$ architecture and underlying abstraction principles. Starting with a brief characterisation of the key language features defined by Andrews et al. (2003), we describe the process execution model and its decomposition into *execution lifecycles* of basic and structured activities. Based on this abstract architectural view, we model the *pick* activity as a concrete example of a structured activity involving concurrency and real time among other aspects. Beyond the work presented by Farahbod et al. (2004, 2006), the model as presented here, also addresses fault handler agents and the agent interaction model. Additionally, it explores in more detail the fault handling behaviour of scope agents and structured activity agents.

BPEL introduces a stateful model of Web services interacting with one another by exchanging sequences of messages. A business process and its partners are defined by a collection of abstract WSDL services based on the WSDL model for message interaction. The major parts of a BPEL process definition consist of

1 *partners* of the business process, that is, Web services that this process interacts with,

2 a set of *variables* that keep the state of the process and,

3 an *activity* defining the logic of interactions between the process and its partners.

Activities that can be performed by a business process are categorised into *basic* activities, *structured* activities and *scope-related* activities. Basic activities perform simple operations like *receive*, *reply*, *invoke* and others. Structured activities impose an execution order on a collection of activities and can be nested. Scope-related activities serve for defining logical units of work and encapsulating the reversible behaviour of each such unit.

Dynamic process creation: a BPEL process definition serves as a template for creating business process instances. Process creation is implicit and is done by defining a *start activity* – either a *receive* or a *pick* activity that is annotated with *'createInstance = yes'* – causing a new process instance to be created upon receiving a matching message. That is, when a new instance of a business process is created, it starts its execution by receiving the message that triggered its creation.

Correlation and data handling: a web service consists of a number of business process instances; thus, the messages arriving at a specific port must be delivered to the correct process instance according to the state of each process instance. BPEL introduces a generic mechanism for dynamic binding of messages to process instances, called *correlation*. The data handling features of BPEL facilitate dealing with stateful interactions by providing the ability to keep track of the internal state of each business process instance.

Long running business transactions: business processes normally perform transactions with non-negligible duration involving local updates at business partners. When an error occurs, it may be required to reverse the effects of some or even all of the previous activities. This is known as *compensation*. The ability to compensate for the effects of previous activities in case of an exception enables so-called Long-Running (Business) Transactions (LRTs).

In the process of building the $\text{BPEL}_{\mathcal{AM}}$ model, we have extracted the key language requirements from the LRM in form of *requirement lists* making these requirements accessible for further extensions, validation and verification of the model and also to facilitate finding inconsistencies and ambiguities in the LRM. For a complete list of these requirements see Farahbod (2004).

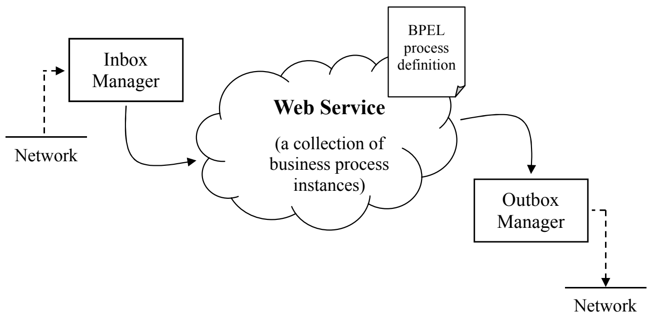
3.1 Abstract machine architecture

The $\text{BPEL}_{\mathcal{AM}}$ architecture is composed of three basic building blocks, referred to as *core*, *data handling extension* and *fault and compensation extension*. The *core* handles dynamic process creation/termination, communication primitives, message correlation, concurrent control structures, as well as the following activities: *receive*, *reply*, *invoke*, *wait*, *empty*, *sequence*, *switch*, *while*, *pick* and *flow*. The *core* does not consider data handling, fault handling and compensation behaviour; rather these aspects are treated as extensions to the core (see Section 4). Together with the *core*, the extensions form the complete $\text{BPEL}_{\mathcal{AM}}$.

The vertical organisation of the machine architecture consists of three layers, called *abstract* model, *intermediate* model and *executable* model. The abstract model formally sketches the behaviour of the key BPEL constructs and introduces the overall organisation of the abstract machine architecture. The intermediate model, obtained as the result of the first refinement step, provides a comprehensive formalisation as required for establishing of and reasoning about key language properties. Finally, the executable model provides an abstract executable semantics implemented in AsmL (Glässer et al., 2004). A GUI facilitates experimental validation through simulation and animation of abstract machine runs.

Figure 1 illustrates the overall view of the Web services interaction model. A BPEL document abstractly defines a web service consisting of a collection of business process instances. Each such instance interacts with the external world through two interface components, called *inbox manager* and *outbox manager*. The inbox manager handles all the messages that arrive at the web service. If a message matches a request from a local process instance waiting for that message, it is forwarded to this process instance. Additionally, the inbox manager also deals with new process instance creation. The outbox manager, on the other hand, forwards outbound messages from process instances to the network.

Figure 1 $\text{BPEL}_{\mathcal{AM}}$ view of the Web services interaction model



Inbox manager, outbox manager and process instances are modelled by three different types of DASM agents: the *inbox manager agent*, the *outbox manager agent* and one uniquely identified *process agent* for each of the process instances.

In the following sections, we model the behaviour of the inbox manager and the process instances in terms of the execution lifecycles of basic and structured BPEL activities. For a comprehensive definition of the formal model see (Farahbod, 2004; Vajihollahi, 2004).

3.2 Inbox manager

The LRM does not explicitly address the mechanism for assigning inbound messages to matching business process instances but provides only loose guidelines basically leaving this problem to the engine (or implementation). We contend that the message assignment mechanism is essential for defining the semantics of activities that receive messages, such as *receive* and *pick*. Hence, we collect the scattered LRM requirements on inbound messages (see the requirement lists in Farahbod, (2004), Appendix A) and combine them to model the behaviour of the inbox manager. In our model, the inbox manager is the entity responsible for assigning inbound messages to matching process instances.

The inbox manager agent operates on the *inbox space*, a possibly empty set of inbound messages. In each computation step, it attempts to assign a message to a matching process instance. The predicate *correspond*(p, d, m) holds if message m can be assigned to process instance p according to the information specified by the input descriptor d. The predicate *match*(p, op, m) holds if message m can be assigned to operation *op*, which is running in the process instance p, according to the information specified by the input descriptor. The inbox manager uses this predicate to find an appropriate message that matches a waiting process instance. Correlation initialisation and message matching are left abstract in the LRM. Consequently, the *match* predicate abstracts the details on how the correlation information is maintained and how the matching is actually performed based on the underlying data structure.

An *input descriptor*[3] contains information on the waiting input operation and the waiting agent. If the matching is successful, the message is assigned to the process instance by the AssignMessage rule which is further defined in the intermediate model (Farahbod, 2004; Vajihollahi, 2004).

Another major issue deserving attention is process creation. The LRM states (Andrews et al., 2003, Section 6.4) that

> "the creation of a process instance in BPEL4WS is always implicit; activities that receive messages (that is, receive activities and pick activities) can be annotated to indicate that the occurrence of that activity causes a new instance of the business process to be created. [...] When a message is received by such an activity, an instance of the business process is created if it does not already exist."

Therefore, the execution of such an input activity (called start activity) is accompanied by the creation of the corresponding process instance. In other words, a new process instance is created by execution of its first activity. So the question is, how can an activity of a process instance be executed before the process instance is created? Although this approach is somewhat unconventional, the LRM does not further clarify process creation. However, because of the importance of process creation in the lifecycle of business processes, we capture this requirement *as is* in the formal model. This is done by introducing the notion of a *dummy process instance* in the inbox manager.

Basically, the dummy process instance is not different from other process instances in its nature. However, there is always one and only one such process instance which is waiting on its start activity. The inbox manager creates a new process instance whenever a matching message arrives for a start activity of the dummy process. Modelling process instance creation is simplified by introducing a nullary function *dummy* identifying the dummy process instance. 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 updating the value of *dummy* accordingly.

The DASM program given below specifies the behaviour of the inbox manager, where *self* refers to an inbox manager agent.

```
InboxManagerProgram ≡
  if $inboxSpace(self) \neq \emptyset$ then
    choose $p \in PROCESS, m \in inboxSpace(self)$,
      $d \in waitingSetForInput(p)$ with $correspond(p, d, m)$
      AssignMessage$(p, d, m)$
      if $p = dummyProcess$ then  // process instance creation
        new $newDummy$ : Process
          $dummyProcess := newDummy$
where
  $correspond(p, d, m) \equiv$
    $match(p, dscOperation(d), m)$
    $\wedge\ waitingOnIO(dscAgent(d), p)$
    //waitingOnIO confirms the agent is still waiting
```

The behaviour of the inbox manager also addresses a loose end in the LRM. According to the LRM, 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." (Andrews et al., 2003, Section 11.4) Therefore, it is implicitly assumed that a matching message will arrive *after* the corresponding receive activity has been executed and it is not clear what happens when a message arrives *before* the corresponding receive activity is executed. Indeed, such a message can be regarded pessimistically (e.g. discarded) or optimistically (e.g. stored in a buffer), each of which giving rise to a different implementation of the language. Thus, it is certainly important for the LRM to provide a comprehensive description of the message assignment mechanism. For a more detailed discussion of the inbox manager and the associated issues of the LRM, the reader is referred to (Vajihollahi, 2004). In BPEL$_{\mathcal{AM}}$, all incoming messages are buffered before being processed.

3.3 Activity execution lifecycle

Intuitively, the execution of a process instance is decomposed into a collection of execution lifecycles for the individual BPEL activities. We therefore introduce *activity agents*, created dynamically by process agents, for executing structured activities. Each activity agent dynamically creates additional activity agents for executing nested, structured activities. Similarly, it creates auxiliary activity agents for dealing with concurrent control threads (like in *flow* and *pick*[4]). For instance, to concurrently execute a set of activities, a flow agent assigns each enclosed activity to a separate *flow thread agent* (Farahbod et al, 2004). At any time during the execution of a process instance, the DASM agents running under control of this process agent form a tree structure where each of the subagents monitors the execution of its child agents (if any) and notifies its parent agent in case of normal completion or fault. This structure provides a general framework for execution of BPEL activities. The DASM agents that model BPEL process execution are jointly called *kernel agents*. They include process agents and subprocess agents. In the *core*, however, subprocess agents are identical to activity agents. Figure 2 sketches the process execution tree of the BPEL$_{\mathcal{AM}}$.

Figure 3 illustrates the normal activity execution lifecycle of kernel agents in the BPEL$_{\mathcal{AM}}$ core. When created, a kernel agent is in the *Started* mode. After initialisation, the kernel agent starts executing its assigned task by switching its mode to *Running*. Upon completion, the agent switches its mode to *ActivityCompleted* and decides (based on the nature of the assigned task) to either return to the *Running* mode or finalise the execution and become *Completed*. Activity agents that may execute more than one activity (like *sequence*) or execute one activity more than once (like *while*) can switch back and forth between the two modes *ActivityCompleted* and *Running*.

Figure 2 Process execution tree

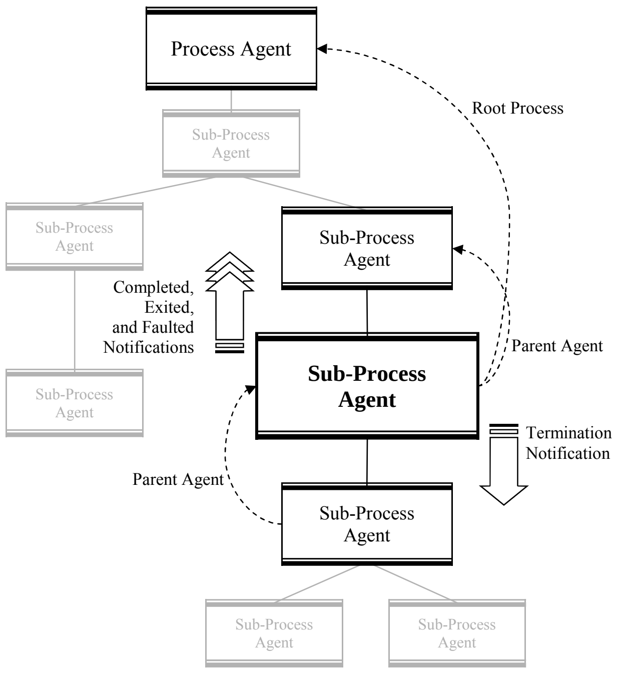
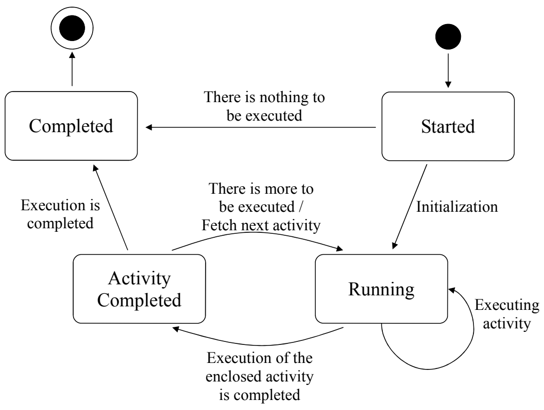


Figure 3 Activity execution lifecycle: $\mathsf{BPEL}_{\mathcal{AM}}$ core


Completed
There is nothing to be executed
Started
Execution is completed
There is more to be executed / Fetch next activity
Initialization
Activity Completed
Running
Executing activity
Execution of the enclosed activity is completed

As an example, *sequence* is a simple structured activity that enforces a sequential execution order on a collection of activities (henceforth called *subactivities*). The sequence programme which models the sequence activity in $\mathsf{BPEL}_{\mathcal{AM}}$ starts in the *Started* mode, loads the first subactivity of the sequence and switches to the *Running* mode to execute that subactivity. When the execution of a subactivity is completed, the programme switches to the *Activity Completed* mode. If there are more subactivities to be executed, the next one is loaded and the mode is switched back to the *Running* mode. When all the subactivities are executed, the mode is changed to the *Completed* mode where the programme ends. For brevity, we refer to (Farahbod, 2004) for the formal definition of the sequence programme.

3.4 Pick activity

A *pick* activity identifies a set of events and associates with each of these events a certain activity. Intuitively, it waits on one of the events to occur and then performs the respective activity; thereafter, the *pick* activity no longer accepts any other event.[5] There are basically two different types of events: *onMessage* events and *onAlarm* events. An onMessage event occurs as soon as a related message is received, whereas an onAlarm event is triggered by a timer mechanism waiting *'for'* a certain period of time or *'until'* a certain deadline is reached.

In $\mathsf{BPEL}_{\mathcal{AM}}$, each *pick* activity is modeled by a separate activity agent, called *pick agent*. A pick agent is assisted by two auxiliary agents, a *pick message agent* that is waiting for a message to arrive and a *pick alarm agent* that is watching a timer. We formalise the semantics of the *pick* activity in several steps, each of which addresses a particular property and then compose the resulting DASM program, called PickProgram, in which *self* refers to a pick agent executing the program.

```
PickProgram ≡
  case execMode(self) of
    Started → PickAgentStarted
    Running → PickAgentRunning
    ActivityCompleted → FinalizePickAgent
    Completed → stop self
```

When created, the pick agent is in the *Started* mode and initialises its execution by creating a pick alarm agent and a pick message agent. It then switches its mode to *Running* and waits for an event to occur – either a message arrives or a timer expires.

Depending on the event type, either the pick message agent or the pick alarm agent notifies the pick agent by adding an *event descriptor* to the *occurredEvents* set of the pick agent. An event descriptor contains information on the event such as the time of its occurrence. When an event occurs, the pick agent updates the function *chosenAct* (with initial value *undef*) with the activity associated with the event. Once the activity is chosen (*chosenAct(self)* $\neq$ *undef*), the pick agent performs the chosen activity and remains *Running* until the execution of the chosen activity is completed as indicated by receiving an agent-completed signal. It then switches its execution mode to *Activity-Completed*.

```
PickAgentRunning ≡
  if normalExecution(self) then
    onsignal s : agentCompleted
      execMode(self) := ActivityCompleted
    otherwise
      if chosenAct(self) = undef then
        choose dsc ∈ occurredEvents(self) with MinTime(dsc)
          chosenAct(self) := onEventAct(edscEvent(dsc))
          //onEventAct of an event descriptor is the activity
          //that is associated with that event descriptor
      else
        ExecuteActivity(chosenAct(self))
```

Finalising a running pick agent includes informing its parent agent that the execution is completed and changing the execution mode to *Completed*. As illustrated in Figure 3, the *Completed* mode leads to the agent's termination.

Due to the space limitations, we do not show here the definitions of PickAgent-Started, FinalisePickAgent, as well as the programs of the pick message and the pick alarm agents, but refer to (Farahbod, 2004) for a complete description.

4 Extensions to the $BPEL_{\mathcal{AM}}$ core

Our two-dimensional refinement approach (Farahbod, 2004) facilitates refinements of the *core* to capture additional aspects of BPEL through incremental extensions (Börger, 2003) and enables step by step elucidation of the extensions through a combination of data refinement and procedural refinement approaches (Börger, 2003; Farahbod, 2004). For a clear separation of concerns and also for robustness of the formal semantic model, the aspects of data handling, fault handling and compensation behaviour are carefully separated from the core of the language. To this end, the core of $\mathsf{BPEL}_{\mathcal{AM}}$ provides a basic, yet comprehensive, model for *abstract processes* in which data handling focuses on protocol relevant data in the form of correlations while payload data values are left unspecified (Andrews et al., 2003).

Compensation and fault handling behaviour is a fairly complex issue in the definition of BPEL. An in-depth analysis in fact shows that the semantics of fault and compensation handling, even when ignoring all the syntactical issues, is related to more than 40 individual requirements spread out all over the LRM. These requirements (some of them comprise up to 10 subitems) address a variety of separate issues related to the core semantics, general constraints and various special cases (see (Farahbod, 2004, Appendix A)). While most of these requirements are defined with painstaking accuracy, such definitions are not free of ambiguities and imprecisions inherent to natural languages. Consequently, complementary formal descriptions are vital for turning abstract requirements into precise specifications. Specifically, they are beneficial since:

1 analysing a requirement to construct a formal specification often provides a different view to the requirement (and to the system under inspection) potentially uncovering possible problems, such as ambiguities, inconsistencies and loose ends;

2 formalisation of requirements along with their informal description (the idea of *literate specifications* (Johnson, 1996)) provides a sensible way of gaining precision without loosing intelligibility.

A thorough treatment of the extensions is beyond the space limitations of this paper. We present an overview of the fault handling behaviour in the following sections and refer to Farahbod (2004) for a comprehensive description of other non-trivial issues.

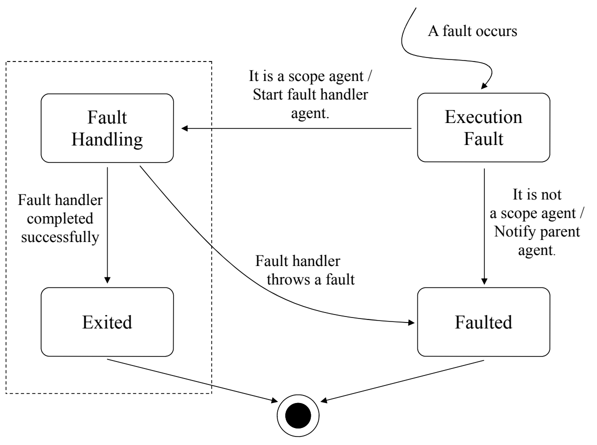
4.1 Scope activity and fault handling

The *scope* activity is the core construct of data handling, fault handling and compensation behaviour in BPEL. A *scope* activity is a wrapper around a logical unit of work (a block of BPEL code) that provides local variables, a fault handler and a compensation handler. The fault handler of a scope is a set of *catch* clauses defining how the scope should respond to different types of faults. A compensation handler is a wrapper around a BPEL activity that compensates the effects of the execution of the scope. Each scope has a primary activity which defines the normal behaviour of the scope. This activity can be any basic or structured activity. BPEL allows scopes to be nested arbitrarily. In $\mathsf{BPEL}_{\mathcal{AM}}$, we model scopes by defining a new type of activity agents, called *scope agents*.

Fault handling in BPEL can be conceived as a mode switch from the normal execution of the process (Andrews et al., 2009). When a fault occurs in the execution of an activity, the fault is thrown up to the innermost enclosing scope. If the scope handles the fault successfully, it sends an *exited* signal to its parent scope and ends gracefully, but if the fault is rethrown from the fault handler or a new fault has occurred during the fault handling procedure, the scope sends a *faulted* signal along with the thrown fault to its parent scope. The fault is thrown up from scopes to parent scopes until a scope handles it successfully. A successful fault handling switches the execution mode back to normal. If a fault reaches the global scope, the process execution terminates (Andrews et al., 2003).

The normal execution lifecycle of kernel agents (Figure 3) needs to be extended to comprise the fault handling mode of BPEL processes. The occurrence of a fault causes the kernel agent (be it an activity agent or the main process) to leave its normal execution lifecycle and enter a fault handling lifecycle. Figure 4 illustrates the extended execution lifecycle of BPEL activities.

Figure 4 Activity execution lifecycle: fault handling


A fault occurs
It is a scope agent / Start fault handler agent.
Fault Handling
Execution Fault
Fault handler completed successfully
It is not a scope agent / Notify parent agent.
Fault handler throws a fault
Exited
Faulted

In $\mathsf{BPEL}_{\mathcal{AM}}$, whenever a subprocess agent encounters a fault, the agent leaves its normal execution mode and enters the *ExecutionFault* mode. If this agent is neither a scope agent nor a process agent, it should also notify its parent agent of the fault. This behaviour is performed by the TransitionToExecutionFault rule. For every kernel agent, the dynamic function *faultThrown* (defined on kernel agents) keeps the current fault which is thrown in the execution of the agent. The default value of *faultThrown* is *undef*.

```
TransitionToExecutionFault(fault) ≡
  execMode(self) := ExecutionFault
  faultThrown(self) := fault
  if ¬isScopeAgent(self) ∧ ¬isProcessAgent(self) then
    trigger s : agentFaulted for parentAgent(self)
      fault(s) := fault
```

A scope agent, in the *ExecutionFault* mode, terminates its enclosed activity, creates a fault handler assigns the fault to that handler and switches to the *FaultHandling* mode. If the fault handler finishes successfully, the scope agent enters the *Exited* mode indicating that this agent exited its execution with a successful fault handling process. The difference between a *scope* which has finished its execution in the *Completed* mode and a *scope* that has finished in the *Exited* mode is reflected by the way scopes are compensated, which we do not further address in this paper. The behaviour of scope agents in the *Execution-Fault* mode is specified by the following rule:

```
ScopeAgentExecutionFault ≡
  TerminateEnclosedActivity(self)
  CreateAndExecuteFaultHandler(self)
  execMode(self) := FaultHandling
```

Termination of enclosed activities (of any subprocess agent) is modelled by the TerminateEnclosedActivity rule. If the enclosed activity of an agent is a basic activity, it needs to be terminated according to the requirements addressed by the LRM (Andrews et al., 2003, Section 13.4.2). These requirements are modelled by the TerminateBasicActivity rule. To terminate structured activities, the enclosing agent (e.g. a scope agent) sends a *forcedTermination* signal to its child agent(s).

```
TerminateEnclosedActivity(agent) ≡
  TerminateBasicActivity(agent)
  forall child in childAgents(agent)
    trigger s : forcedTermination for child
      fault(s) := bpwsForcedTermination
```

For the complete specification of TerminateBasicActivity we refer to (Farahbod, 2004).

4.2 Pick activity: extended

The structured activities of the *core* (activity agents) are also refined to capture the fault handling behaviour of BPEL. The well-defined activity execution lifecycle of $\text{BPEL}_{\mathcal{AM}}$ (Figures 3 and 4) along with the fact that the fault handling behaviour of BPEL is mostly centered in the *scope* activity, enable us to generally extend the behaviour of structured activities by defining two new rules: HandleExceptionsInRunningMode and WaitForTermination. As an example, the pick agent program of Section 3.4 is refined as follows:

```
PickProgram ≡
  PickProgram_core
  case execMode(self) of
    Running → HandleExceptionsInRunningMode
    ExecutionFault → WaitForTermination
    Faulted → stop self
```

Activity agents react to a fault by informing their parent agent of the fault and stay in the *ExecutionFault* mode until they receive a notification for termination. If the parent agent is not a scope agent, the parent agent reacts in the same way and the fault is passed upwards until it reaches a scope agent. The scope agent handles the fault as described in Section 4.1 and sends a termination notification to its child agent. Upon receiving the notification, a subprocess agent that is waiting for a termination notification in turn passes it to its child agents (if any) and enters the *Faulted* mode, where it then terminates.

The normal execution of activity agents in the *Running* mode is extended by the following rule:

```
HandleExceptionsInRunningMode ≡
  if faultExtensionSignal(self) then
    onsignal s : agentExited
      execMode(self) := ActivityCompleted
    otherwise
      onsignal s : agentFaulted
        TransitionToExecutionFault(fault(s))
      otherwise
        onsignal s : forcedTermination
          faultThrown(self) := fault(s)
          execMode(self) := ExecutionFault
```

The *faultExtensionSignal* predicate holds only if the agent has received a signal related to fault and compensation handling. The *normalExecution* predicate in PickAgentRunning (Section 3.4) is defined as the negation of this predicate which facilitate conservative refinement of the core model. If a subprocess agent receives a termination notification[6] while in its normal execution mode, it enters the *ExecutionFault* mode, where it terminates its enclosed activity and goes to the *Faulted* mode.

In the *ExecutionFault* mode, if a termination notification is received, the pick agent terminates its enclosed activity and goes to the *Faulted* mode. Analogously to the *Completed* mode, subprocess agents terminate their execution in the *Faulted* mode. For the complete extended pick agent program see (Farahbod, 2004).

```
WaitForTermination ≡
  if isForcedTerminated(self) then
    execMode(self) := Faulted
    TerminateEnclosedActivity(self)
  else
    onsignal s : forcedTermination
      faultThrown(self) := fault(s)
      execMode(self) := Faulted
      TerminateEnclosedActivity(self)
where
  isForcedTerminated(a) ≡
              faultThrown(a) = bpwsForcedTermination
```

The LRM does not precisely specify how activity termination (due to a fault) takes place. It states that when a fault occurs in a *scope*, the fault handler begins by implicitly terminating all activities inside the *scope*. Further, in (Andrews et al., 2003, Appendix A) on standard faults, the LRM states that *forcedTermination* is used to terminate activities enclosed in a scope. However, it is not clear how the *forcedTermination* fault is used to terminate enclosed activities: it is not stated whether the faulted activities should wait for the *forcedTermination* fault after they encounter a fault or they should terminate spontaneously. See Farahbod (2004) for an example and more details on this discussion.

4.3 Fault Handlers

We use DASM agents to also model fault handlers. The behaviour of fault handler agents is formally defined by the following program.

```
FaultHandlerProgram ≡
  case execMode(self) of
    Started → FaultHandlerStarted
    Running →
      FaultHandlerRunningNormal
      FaultHandlerRunningExtended
    ActivityCompleted → FinalizeKernelAgent
    Completed → stop self
    ExecutionFault → FaultHandlerExecutionFault
    Faulted → stop self
```

According to the LRM, the normal behaviour of a fault handler begins with selecting a *catch* clause that matches the fault that is being handled. The function *faultHandlerCatchSet* is defined in BPEL$_{\mathcal{AM}}$ to provide the set of *catch* clauses in the fault handler of a *scope* activity. The abstract predicate *matchingCatch* defined on catch clauses is used to find the matching *catch* clause of a fault. The chosen *catch* clause is then stored in *executingCatch* for further processing.

```
FaultHandlerStarted ≡
  execMode(self) := Running
  choose c ∈ faultHandlerCatchSet(handlerScope(self))
       with matchingCatch(c, faultThrown(self))
    executingCatch(self) := c
```

To model the main behaviour of fault handlers (i.e. fault handler agents in the *Running* mode), analogous to structured activities in BPEL, we can split their behaviour into two parts: normal execution and fault-handling extended execution. The normal behaviour of a fault handler in the *Running* mode (*normalExecution*(*self*) = *true*) is similar to other structured activities. If it receives an agent-completed signal, it switches to the *ActivityCompleted* mode and finishes its execution; otherwise, it executes the selected *catch* clause. However, according to the LRM (Andrews et al., 2003, Section 13.4), if no *catch* clause is selected, the fault is rethrown. This is done by executing a predefined *catch* clause, called *rethrowCatchClause*.[7]

```
FaultHandlerRunningNormal ≡
  if normalExecution(self) then
    onsignal s : agentCompleted
      execMode(self) := ActivityCompleted
    otherwise
      if executingCatch(self) = undef then
        executingCatch(self) := rethrowCatchClause
      else
        ExecuteActivity(catchActivity(executingCatch(self)))
```

While in the *Running* mode, a fault handler may receive fault handling signals (*faultExtensionsSignal*(*self*) = *true*) generated due to an internal fault in its enclosed activity. If the internal fault is already handled properly, an agent-exited signal is received by the fault handler agent, otherwise an agent-faulted signal is received. According to the LRM (Andrews et al., 2003, Section 13.4.2), "if the scope has already experienced an internal fault and invoked a fault handler, then [...] the forced termination has no effect." Thus, fault handler agents, while in the *Running* mode, do not process forced-termination signals sent by enclosing scopes.

```
FaultHandlerRunningExtended ≡
  if faultExtensionSignal(self) then
    onsignal s : agentExited
      execMode(self) := ActivityCompleted
    otherwise
      onsignal s : agentFaulted
        TransitionToExecutionFault(fault(s))
```

Occurrence of an unhandled internal fault in the execution of a fault handler changes the execution mode of the fault handler to *ExecutionFault*. In this mode, according to the LRM (Andrews et al., 2003, Section 13.4.2), the fault handler must terminate its execution prematurely. The FaultHandlerExecutionFault, presented below, models this behaviour by terminating the execution of the enclosed activity and changing the execution mode of the fault handler to *Faulted*, which leads to the termination of the fault handler.

```
FaultHandlerExecutionFault ≡
  TerminateEnclosedActivity(self)
  execMode(self) := Faulted
```

For a complete specification of fault handling and compensation behaviour in BPEL$_{\mathcal{AM}}$, we refer the reader to Farahbod (2004).

5 Related work

There are various research activities applying formal methods to define, analyse and verify Web services orchestration languages. A group at Humboldt University is working on formalisations of BPEL for analysis, graphics and semantics. Petri-net models of Web services are used in Martens (2005) to analyse essential properties like usability, soundness and compatibility, which is a starting point for deciding the equivalence of two Web services. Furthermore, the group uses both Petri-nets and ASMs to formalise the semantics of BPEL. A pattern-based Petri-net semantics for BPEL activities is provided by Stahl (2004) (in German), however, it fails to provide a comprehensive model for handling the data and process instantiation. In Schmidt and Stahl (2004), a small business process is translated into a Petri-net model without addressing fault handling, compensation handling and timing aspects. The ultimate goal is verification of business processes; however, the feasibility of verifying larger business processes is still subject to future work. The ASM semantic model in Fahland and Reisig (2005) closely follows our work in Farahbod (2004)with a slightly different architecture and minor technical differences in handling basic activities and variables. The model presented by Fahland (2005) extends our work in order to address issues like dead-path-elimination and correlation handling, which we do not cover in much detail. However, Fahland (2005) does not provide all the means for building an executable model whereas our model in Farahbod (2004) is directly executable. Thus, by collaborating closely with the group at Humboldt University, we hope to integrate both models into a complete and executable formal model of BPEL. Along this line, our notion of inbox and outbox manager was adopted in Fahland (2005) to bring the two models closer to each other and serve as a basis for a joint work on BPEL.

Formal verification of Web services is also addressed in several works (Arias-Fistens et al., 2005; Ferrara, 2004; Foster et al., 2005; Martens, 2003; Narayanan and McIlraith, 2002; Ouyang et al., 2005b; van Breugei and Koshkina, 2005). The approach in Martens (2003) is based on Petri-nets, while in van Breugal and Koshkina (2005), a process algebra is used to derive a structural operational semantics of BPEL as a formal basis for verifying properties of the specification. It focuses on the main control flow constructs and abstracts from data and time related issues, as well as fault and compensation behaviour. The authors point out the superiority of this approach to Petri-net based approaches in capturing advanced synchronisation patterns like Dead-Path-Elimination (DPE). The approach presented by Ferrara (2004) also uses process algebra and corresponding tools to provide a framework for design and verification of BPEL processes. While issues regarding process instantiation are not captured, it provides support for fault handling and compensation behaviour. WofBPEL is a verification tool for performing static analysis on BPEL processes translated into Petri-nets (Ouyang et al., 2005b) using the formal semantics of (Ouyang et al., 2005a). LTSA WS-Engineer is another verification tool which aims at BPEL composition (Foster et al., 2005). Scenario-based designs of business processes in Message Sequence Charts (MSCs) and BPEL implementations are both translated to Finite State Process (FSP) models and compiled to a Labeled Transition System (LTS) which is used as a basis for verification. The framework presented by Arias-Fisteus et al., (2005) called VERBUS, is also based on finite state machines. The main objective is to achieve a modular and extensible framework by using a common formal model layer for translating BPEL processes and as a basis for verification. In Narayanan and McIlraith (2002), a model-theoretic semantics for the DAML-S language (based on situation calculus) is presented, which facilitates simulation, composition, testing and verifying compositions of Web services.

A critical analysis of BPEL based on workflow data and control-flow patterns, as well as a comparative summary of formalisations of BPEL, is provided in van der Aalst et al., (2005).

6 Conclusion

We propose a BPEL abstract machine as a well-defined and robust computational framework for establishing the key language attributes of BPEL in the form of a comprehensive abstract operational semantics based on the ASM formalism and abstraction principles. Our model provides a concise and precise formalisation of all the dynamic semantic properties that are characteristic for BPEL. At the same time, it reflects the abstract operational view and terminology of the informal language definition in a direct and intuitive way. The hierarchical organisation and the modular structure of the abstract machine architecture support a gradual formalisation of complex requirements and enhances a clear separation of concerns. As a result of building this ASM ground model, we actually discovered a number of deficiencies in the language definition, some of them are addressed in this paper (Section 3.2 and 4.2); for a comprehensive list, as well as a proposal for a new activity providing synchronous request-response services (addressing difficulties discussed by WSBPEL-TC (2004, Issue #26, #49, #50, #120 and #123), the reader is referred elsewhere (Farahbod, 2004; Vajihollahi, 2004).

The focus in this paper is on formal specification rather than on formal verification, understanding the former as the main prerequisite for the latter. Since there is no way to prove correctness or completeness of the initial transformation from the language designers' intuitive understanding to a proper mathematical representation, this formalisation step, which forms the foundation for any subsequent refinement steps, deserves particular attention to avoid misconceptions with most fatal consequences.

Beyond reasoning about the language design and checking consistency and validity of semantic properties, our BPEL abstract machine also serves as a platform for experimental validation through simulation and testing. As a result of the final refinement step, we obtain an abstract executable semantics encoded in *AsmL*, an industrial design language (Glässer et al., 2004). An AsmL model of the core of an earlier version of the $\mathsf{BPEL}_{\mathcal{AM}}$ has been used for simulation purposes (Vajihollahi, 2004). Experimental validation of high-level design specifications clearly offers additional benefits in design exploration and for eliminating deficiencies prior to low-level coding. We plan to develop an advanced executable model of our BPEL abstract machine using *CoreASM* (Farahbold et al., 2005), a novel open source ASM tool environment (under development). Additionally, we also plan to explore the use of ASM model checking techniques (Tang, 2006) for the formal verification of certain key properties of the abstract machine model.

Finally, the dynamic nature of standardisation, being an ongoing and potentially open-ended activity, calls for flexibility and robustness of the formalisation approach. To this end, we feel that the ASM formalism and abstraction principles offer a sensible compromise between mathematical elegance and practical relevance – already proved useful for practical purposes in other standardisation contexts (Glässer et al., 2003).

Acknowledgements

Our sincere appreciation to the five anonymous reviewers of previous versions of this paper for their detailed and valuable comments and suggestions for improvements, which we have incorporated into this paper. Part of the research was supported by NSERC discovery grant No. 611365.

References

Andrews, T. et al. (2003) 'Business process execution language for web services version 1.1', Last visited February 2005, Available at: http://ifr.sap.com/bpel4ws/.

Arias-Fisteus, J., Fernández, L.S. and Kloos, C.D. (2005) 'Applying model checking to bpel4ws business collaborations', H. Haddad et al. (Eds). '*SAC*', *ACM*, pp.826–830.

Börger, E. (2003) 'The ASM refinement method', *Formal Aspects of Computing*, pp.237–257.

Börger, E., Fruja, N.G., Gervasi, V. and Stärk, R.F. (2005) 'A high-level modular definition of the semantics of C#', *Theoretical Computer Science*, Vol. 336, Nos. 2/3, pp.235–284.

Börger, E., Glässer, U. and Müller, W. (1995) 'Formal definition of *an* abstract VHDL'93 simulator by EA-machines, *in* C. Delgado Kloos and P.T. Breuer (Eds). *Formal Semantics for VHDL*, Kluwer Academic Publishers, pp.107–139.

Börger, E., Riccobene, E. and Schmid, J. (2000) 'Capturing requirements by abstract state machines: the light control case study', *Journal of Universal Computer Science* Vol. 6, No. 7, pp.597–620.

Börger, E. and Stärk, R. (2003) *Abstract State Machines: A Method for High-Level System Design and Analysis*, Springer-Verlag.

Fahland, D. (2005) 'Complete abstract operational semantics for the web service business process execution language', *Technical report*, Humboldt-Universität zu Berlin, Informatik-Berichte, p.190.

Fahland, D. and Reisig, W. (2005) 'Asm-based semantics for BPEL: the negative control flow', *Proceedings of the 12th International Workshop on Abstract State Machines*.

Farahbod, R. (2004) 'Extending and refining an abstract operational semantics of the web services architecture for the business process execution language', Master's thesis, Simon Fraser University, Burnaby, Canada.

Farahbod, R. and Glässer, U. (2006) 'Semantic blueprints of discrete dynamic systems: challenges and needs in computational modeling of complex behaviour', *in* F. Meyer auf der Heide and B. Monien (Eds). *New Trends in Parallel and Distributed Computing*, Heinz Nixdorf Institute, pp.81–95.

Farahbod, R., Glässer, U. and Vajihollahi, M. (2004) 'Specification and validation of the business process execution language for web services', in W. Zimmermann and B. Thalheim, (Eds). *Abstract State Machines 2004. Advances In Theory And Practice: 11th International Workshop (ASM 2004)*, Germany, Springer-Verlag.

Farahbod, R., Glässer, U. and Vajihollahi, M. (2006) 'An abstract machine architecture for web service based business process management', in C. Bussler et al. (Eds). *BPM 2005 Workshops, LNCS 3812*, Springer-Verlag, pp.144–157.

Farahbod, R., et al. (2005) *The CoreASM Project*, Available at: http://www.coreasm.org.

Ferrara, A. (2004) 'Web services: a process algebra approach', *ICSOC'04: Proceedings of the Second International Conference on Service Oriented Computing*, ACM Press, pp.242–251.

Foster, H., Uchitel, S., Magee, J. and Kramer, J. (2005) 'Tool support for model-based engineering of web service compositions', *ICWS, IEEE Computer Society*, pp.95–102.

Glässer, U., Gotzhein, R. and Prinz, A. (2003) 'The formal semantics of SDL-2000: status and perspectives', *Computer Networks* Vol. 42, No. 3, pp.343–358.

Glässer, U. and Gu, Q-P. (2005) 'Formal description and analysis of a distributed location service for mobile ad hoc networks', *Theoretical Computer Science*, Vol. 336, pp.285–309.

Glässer, U., Gurevich, Y. and Veanes, M. (2004) 'Abstract communication model for distributed systems', *IEEE Transactions on Software Engineering*, Vol. 30, No. 7, pp.458–472.

Gurevich, Y. (2000) 'Sequential abstract state machines capture sequential algorithms', *ACM Transactions on Computational Logic*, Vol. 1, No. 1, pp.77–111.

Johnson, C.W. (1996) 'Literate specifications', *Software Engineering Journal*, Vol. 11, No. 4, pp.225–237.

Martens, A. (2003) 'Verteilte Geschtsprozesse – Modellierung und Verifikation mit Hilfe von Web Services', PhD thesis, Humboldt University of Berlin, Germany.

Martens, A. (2005) 'Analyzing web service based business processes', in M. Cerioli,(Ed). *FASE*, Vol. 3442 of *LNCS*, Springer, pp.19–33.

Müller, W., Ruf, J. and Rosenstiel, W. (2003) 'An ASM based SystemC simulation semantics', in W. Müller, J. Ruf and W. Rosenstiel, (Eds). *SystemC – Methodologies and Applications*, Kluwer Academic Publishers.

Narayanan, S. and McIlraith, S.A. (2002) 'Simulation, verification and automated composition of web services', *Proceedings of the Eleventh International Conference on World Wide Web*, ACM Press, pp.77–88.

Ouyang, C., et al. (2005a) 'Formal semantics and analysis of control flow in WS-BPEL', *Technical Report BPM-05-13*, BPMcenter.org.

Ouyang, C., et al. (2005b) 'WofBPEL: a tool for automated analysis of BPEL processes', in B. Benatallah et al. (Eds). *ICSOC*, Vol. 3826 of *LNCS*, Springer, pp.484–489.

Schmidt, K. and Stahl, C. (2004) 'A petri net semantic for BPEL4WS – validation and application', in E. Kindler, (Ed). *Proceedings of 11th Workshop on Algorithms and Tools for Petri Nets*.

Stahl, C. (2004) 'Transformation von BPEL4WS in Petrinetze', Master's thesis, Humboldt-Universität zu Berlin, Germany.

Stärk, R., Schmid, J. and Börger, E. (2001) *Java and the Java Virtual Machine: Definition, Verification, Validation*, Springer-Verlag.

Tang, C. (2006) 'Model checking abstract state machines with answer set programming', Master's thesis, Simon Fraser University, Burnaby, Canada.

Vajihollahi, M. (2004) 'High level specification and validation of the business process execution language for web services', Master's thesis, Simon Fraser University, Burnaby, Canada.

van Breugel, F. and Koshkina, M. (2005) 'Dead-path-elimination in BPEL4WS', *ACSD, IEEE Computer Society*, pp.192–201.

van der Aalst, W., Dumas, M., ter Hofstede, A., Russell, N., Verbeek, H. and Wohed, P. (2005) 'Life after BPEL?', in M. Bravetti et al. (Eds). *EPEW/WS-FM*, Vol. 3670 of *LNCS*, Springer, pp.35–50.

W3C (2003) *Web Services Description Language (WSDL) Version 1.2 Part 1: Core Language*, Last visited May 2004, Available at: http://www.w3.org.

WSBPEL-TC (2004) *WS BPEL Issues List*, Organization for the Advancement of Structured Information Standards (OASIS), Available at: http://www.oasis-open.org.

Notes

[1]See also the ASM website at www.eecs.umich.edu/gasm.

[2]See (Farahbod and Glässer, 2006) for a tutorial introduction.

[3]In $\mathsf{BPEL}_{\mathcal{AM}}$, input activities (such as *receive* and *pick*) add an input descriptor to the *waitingSetForInput* for every message they expect to receive.

[4]One may argue that *pick* is not a concurrent control construct, but as we will see in Section 3.4, it can naturally be viewed as such.

[5]Regarding the case that several events occur at a time, the LRM is somewhat loose declaring that the choice "is dependent on both timing and implementation" (Andrews et al., 2003).

[6]Such a notification would come from a parent agent as a result of a fault in a concurrent activity.

[7]While we were modelling the default fault handling behaviour of scopes, we identified the need for a special activity to allow a *catchall* clause to rethrow its original fault to its parent scope. Such an activity is missing in the LRM. At the same time, this issue was addressed by the OASIS WSBPEL Technical Committee and was resolved using a similar approach by introducing a `<rethrow/>` construct.

Appendix

Agent interaction model

To avoid changing the state of an agent by its child agent(s) and to make the model more flexible for future changes and extensions, we provide a simple yet elegant framework for agents to communicate with each other. This framework is first introduced by (Farahbod, 2004). Here we present a slightly refined version of the original idea.

Agents communicate through exchange of *signal*s. Every kernel agent can send a signal to another agent using the following operation:

```
trigger $s$ : signal-type for agent
  $Rule_1$
```

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

```
onsignal $s$ : signal-type
  $Rule_1$
otherwise
  $Rule_2$
```

The two rule constructs **trigger** and **onsignal** form an interface to our agent communication framework. To each process instance p, we assign a set of signals *signalSet*(p) which acts as a container of the signals sent to p or any subprocess agent of p. For every signal, *signalSource* and *signalTarget* indicate the source and the target agents of that signal.

When an agent triggers a signal for another agent, a new signal element is created, its type, source and target agents are assigned and the signal is added to the signal set of the target agent (which is the signal set of its root process). The following syntactical transformation provides this behaviour:

```
trigger $s$ : signal-type for agent
  $Rule$
$\equiv$
extend SIGNAL with $s$
  $signalType(s) :=$ signal-type
  $signalSource(s) := self$
  $signalTarget(s) :=$ agent
  add $s$ to $signalSet(rootProcess(self))$
  $Rule$
```

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

```
onsignal $s$ : signal-type
  $Rule_1$
otherwise
  $Rule_2$
$\equiv$
choose $s \in signalSet(rootProcess(self))$ with
      $signalType(s) =$ signal-type $\wedge$ $signalTarget(s) = self$
  remove $s$ from $signalSet(rootProcess(self))$
  $Rule_1$
ifnone
  $Rule_2$
```