

A Formal Semantics for the Business Process Execution Language for Web Services

Roozbeh Farahbod, Uwe Glässer, and Mona Vajihollahi
{rfarahbo/glaesser/mvajihol}@cs.sfu.ca

Software Technology Lab
School of Computing Science
Simon Fraser University
Burnaby, B.C., Canada

Abstract. We define an abstract operational semantics for the Business Process Execution Language for Web Services (BPEL) based on the *abstract state machine* (ASM) formalism. This way, we model the dynamic properties of the key language constructs through the construction of a *BPEL abstract machine* in terms of a distributed real-time ASM. Specifically, we focus here on the *process execution model* and the underlying *execution lifecycle* of BPEL activities. The goal of our work is to provide a well defined semantic foundation for establishing the key language attributes. The resulting abstract machine model provides a comprehensive and robust formalization at three different levels of abstraction.

Keywords: Web Services Orchestration, BPEL4WS, Abstract Operational Semantics, Abstract State Machines, Requirements Specification

1 Introduction

In this paper, we present an abstract operational semantics of the XML based Business Process Execution Language for Web Services (BPEL4WS) [1], a novel Web Services orchestration language proposed by OASIS [2] as a future standard for the e-business world. BPEL4WS, or BPEL for short, provides distinctive expressive means for describing the process interfaces of Web based business protocols and builds on existing standards and technologies for Web Services. It is defined on top of the service interaction model of W3C's Web Services Description Language (WSDL) [3]. 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 [4], we define a *BPEL abstract machine*, called BPEL_{AM} , as a concise and robust semantic framework for modeling the key language attributes in a precise and well defined form. That is, we formalize dynamic properties of the Web Services interaction model of a BPEL business process in terms of finite or infinite abstract machine *runs*. Due to the concurrent and reactive nature of Web Services and the need for dealing with time related aspects in coordinating distributed activities, we combine

an asynchronous execution model with an abstract notion of real time. The resulting computational model is referred to as a *distributed real-time ASM*. Our model captures the dynamic properties of the key language constructs defined in the language reference manual [1], henceforth called the LRM, including concurrent control structures, dynamic creation and termination of service instances, communication primitives, message correlation, event handling, and fault and compensation handling.

The goal of our work is twofold. First and foremost, $\text{BPEL}_{\mathcal{AM}}$ provides a firm semantic foundation for checking the consistency and validity of the language definition by conceptual means and by analytical means. Formalization is crucial for identifying and eliminating deficiencies that otherwise remain hidden in the informal language definition of the LRM [2, 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.*”

Second, we address pragmatic issues resulting from previous experience with other industrial standards, including the ITU-T language SDL¹ [6] and the IEEE language VHDL [7]. An important observation is that formalization techniques and supporting tools for practical purposes such as standardization call for a gradual formalization of abstract requirements with a degree of detail and precision as needed [8]. 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 view and terminology of the LRM, effectively formalizing the intuitive understanding of BPEL as directly as possible in an objectively verifiable form.

The result of our work is what is called an *ASM ground model* [4] of BPEL. Intuitively, ground models serve as ‘blueprints’ for establishing functional software requirements, including their elicitation, clarification and documentation. Constructing such a ground model requires a major effort — especially, as a clear architectural view, which is central for dealing with complex semantic issues, is widely missing in the BPEL language definition.

The paper is organized as follows. Section 2 briefly summarizes the formal semantic framework. Section 3 introduces the core of our hierarchically defined $\text{BPEL}_{\mathcal{AM}}$, and Section 4 then addresses important extensions to the $\text{BPEL}_{\mathcal{AM}}$ core. Section 5 discusses related work, and Section 6 concludes the paper.

2 Distributed Real-time ASM

We briefly outline the formal semantic framework 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 abstract state machines [9] and their applications [4].²

¹ Our ASM semantic model of SDL is part of the current SDL standard defined by the International Telecommunication Union [5].

² See also the ASM Web site at www.eecs.umich.edu/gasm.

We focus here on the asynchronous ASM model, called distributed abstract state machine (DASM), as formal basis for modeling concurrent and reactive system behavior in terms of abstract machine *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 symbols denoting the various semantic objects and their relations 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 yields a valid interpretation of all the symbols in V .

Concurrent control threads in an execution of P_M are modeled by a dynamic set AGENT of autonomously operating *agents*. Agents of M interact with each other by reading and writing shared locations of global machine states, where the underlying semantic model regulates such interactions so that potential conflicts are resolved according to the definition of *partially ordered runs* [4].

P_M consists of a statically defined collection of agent programs, each of which defines the behavior of a certain *type* of agent in terms of state transition rules. The canonical rule consists of a basic update instruction of the form $f(t_1, t_2, \dots, t_n) := t_0$, where f is an n -ary dynamic function symbol and the t_i 's ($0 \leq i \leq n$) are terms. Intuitively, one can conceive a dynamic function as a *function table* where each row associates a sequence of argument values with a function value. An update instruction specifies a pointwise function update, i.e., an operation that replaces an existing function value by a new value to be associated with the given arguments.

Finally, M models the embedding of a system into a given environment — the *external world* — through actions and events as observable at interfaces. The external world affects operations of M through externally controlled or *monitored* functions. Such functions change their values dynamically over runs of M , although they cannot be updated by agents of M . A typical example is the representation of time by means of a nullary monitored function *now* taking values in a linearly ordered domain TIME. Intuitively, *now* yields the time as measured by some external clock.

3 BPEL Abstract Machine

This section introduces the core components of BPEL_{AM} architecture and the underlying abstraction principles starting with a brief characterization of the key language features as defined in [1]. We then present BPEL's process execution model and its decomposition into *execution lifecycles* of basic and structured activities. As a concrete example of a structured activity dealing with concurrency and real-time aspects, we consider the *pick* activity. The architectural view, the decomposition into execution lifecycles, and the model of *pick* are new and not contained in [10].

BPEL introduces a stateful model of Web Services interacting by exchanging sequences of messages between business partners. A BPEL process and its part-

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

Dynamic Process Creation A BPEL process definition works as a template for creating business process instances. Process creation is implicit and is done by defining a *start activity*, which is 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. BPEL introduces a generic mechanism for dynamic binding of messages to process instances, called *correlation*.

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 the effects of previous activities in case of an exception enables so-called Long-Running (Business) Transactions (LRTs).

3.1 Abstract Machine Architecture

Logically, BPEL_{AM} consists of three basic building blocks referred to as *core*, *data handling extension*, and *fault and compensation extension* (Figure 1). 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 behavior. Rather these aspects are treated as extensions to the core (see Section 4). Together with the *core* these extensions form the complete BPEL_{AM} .

The vertical organization of the machine architecture consists of three layers, called *abstract* model, *intermediate* model and *executable* model. The abstract model formally sketches the behavior of the key BPEL constructs, while the intermediate model, obtained as the result of the first refinement step, provides a complete formalization. Finally, the executable model provides an abstract executable semantics implemented in AsmL [8]. A GUI facilitates experimental validation through simulation and animation of abstract machine runs.

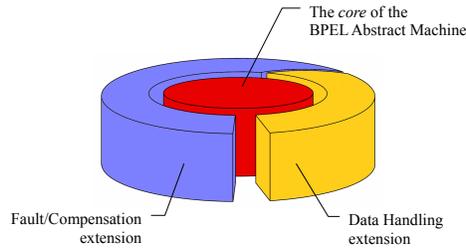


Fig. 1. BPEL_{AM} Behavioural Decomposition

Figure 2 shows an abstract view of the underlying 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.

Inbox manager, outbox manager, and process instances are modeled 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.

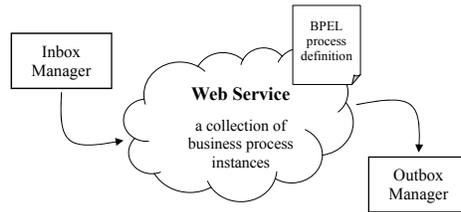


Fig. 2. High-level Structure of BPEL_{AM}

3.2 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

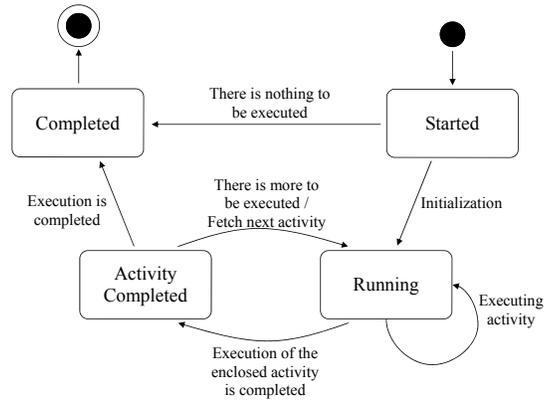


Fig. 3. Activity Execution Lifecycle: BPEL_{AM} core

agents for dealing with concurrent control threads (like in *flow* and *pick*³). For instance, to concurrently execute a set of activities, a flow agent assigns each enclosed activity to a separate *flow thread agent* [10]. 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 sub-agents monitors the execution of its child agents (if any) and notifies its parent agent in case of normal completion or fault. This structure provides a general framework for 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 3 illustrates the normal activity execution lifecycle of kernel agents in the BPEL_{AM} core. When created, a kernel agent is in the *Started* mode. After initialization, the kernel agent starts executing its assigned task by switching its mode to *Running*. Upon completion, the agent switches its mode to *Activity-Completed* and decides (based on the nature of the assigned task) to either return to the *Running* mode or finalize 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 *Activity-Completed* and *Running*.

3.3 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

³ One may argue that *pick* is not a concurrent control construct, but as we will see in Section 3.3, it can naturally be viewed as such.

any other event.⁴ 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 BPEL_{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 formalize 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.

Pick Agent

PickProgram \equiv
case *execMode(self)* **of**
 Started \rightarrow **PickAgentStarted**
 Running \rightarrow **PickAgentRunning**
 ActivityCompleted \rightarrow **FinalizePickAgent**
 Completed \rightarrow **stop self**

When created, the pick agent is in the *Started* mode and initializes 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 arrived or a timer expired.

Pick Agent

PickAgentRunning \equiv
if *normalExecution(self)* **then**
 onsignal *s* : AGENT_COMPLETED
 execMode(self) := *ActivityCompleted*
 otherwise
 if *chosenAct(self)* = *undef* **then**
 choose *dsc* \in *occurredEvents(self)* **with** *MinTime(dsc)*
 chosenAct(self) := *onEventAct(edscEvent(dsc))*
 // *onEventAct* is the activity associated with an event
 else
 ExecuteActivity(*chosenAct(self)*)

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

⁴ 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.” [1]

chosen activity is completed as indicated by a predicate *chosenActCompleted*. It then switches its execution mode to *Activity-Completed*.

Finalizing 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`, `FinalizePickAgent`, as well as the programs of the pick message and the pick alarm agents, but refer to [11, 12] for a complete description.

4 Extensions to the $\text{BPEL}_{\mathcal{AM}}$ Core

For a clear separation of concerns and also for robustness of the formal semantic model, the aspects of data handling, fault handling and compensation behavior are carefully separated from the core of the language. To this end, the core of $\text{BPEL}_{\mathcal{AM}}$ 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 [1].

Compensation and fault handling behavior is a fairly complex issue in the definition of BPEL. An in-depth analysis in fact shows that the semantics of fault and compensation handling, even when ignoring all the syntactical issues, is related to more than 40 individual requirements spread out all over the LRM. These requirements (some of them comprise up to 10 sub-items) address a variety of separate issues related to the core semantics, general constraints, and various special cases (see [2]). A thorough treatment of the extensions is beyond the space limitations of this paper. Thus, we present an overview of the fault handling behavior in the following sections and refer to [11] for a comprehensive description.

4.1 Scope activity

The *scope* activity is the core construct of data handling, fault handling, and compensation handling 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 behavior of the scope. This activity can be any basic or structured activity. BPEL allows scopes to be nested arbitrarily. In $\text{BPEL}_{\mathcal{AM}}$, we model scopes by defining a new type of activity agents, called *scope agents*.

Fault handling in BPEL can be thought of as a mode switch from the normal execution of the process [1]. When a fault occurs in the execution of an activity, the fault is thrown up to the innermost enclosing scope. If the scope handles the fault successfully, it sends an *exited* signal to its parent scope and ends gracefully,

but if the fault is re-thrown from the fault handler, or a new fault 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 [1].

The normal execution lifecycle of the process execution model (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.

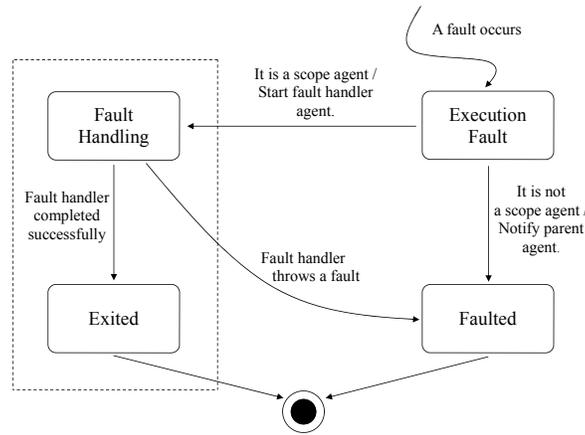


Fig. 4. Activity Execution Lifecycle: Fault Handling

In $BPEL_{AM}$, whenever a sub-process agent encounters a fault, the agent leaves its normal execution mode and enters the *Execution-Fault* mode. If this agent is not a scope agent, it informs its parent agent of the fault and stays in the *Execution-Fault* mode until it receives a notification for termination. On the other hand, if the faulted agent is a scope agent, it terminates its enclosing activity, creates a fault handler, assigns the fault to that handler, and switches to the *Fault-Handling* 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.

4.2 Pick activity: extended

The structured activities of the *core* (activity agents) are also refined to capture the fault handling behavior of BPEL. The well-defined activity execution life-cycle of BPEL_{AM} (Figures 3 and 4) along with the fact that the fault handling behavior of BPEL is mostly centered in the *scope* activity, enable us to generally extend the behavior of structured activities by defining two new rules: *HandleExceptionsInRunningMode* and *WaitForTermination*. As an example, the pick agent program of Section 3.3 is refined as follows:

Pick Activity Extended

PickProgram \equiv
 PickProgram_{core}
 case *execMode*(*self*) of
 Running \rightarrow *HandleExceptionsInRunningMode*
 ExecutionFault \rightarrow *WaitForTermination*
 Faulted \rightarrow **stop** *self*

Activity agents react to a fault by informing their parent agent of the fault and stay in the *Execution-Fault* 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 sub-process 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. If a sub-process agent receives a termination notification while in its normal execution mode, it first enters the *Execution-Fault* mode and then reacts as if it were waiting for the notification.

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

Structured Activity Extended

HandleExceptionsInRunningMode \equiv
 if *faultExtensionSignal*(*self*) then
 onsignal *s* : AGENT_EXITED
 execMode(*self*) := *ActivityCompleted*
 otherwise
 onsignal *s* : AGENT_FAULTED
 TransitionToExecutionFault(*fault*(*s*))
 otherwise
 onsignal *s* : FORCED_TERMINATION
 faultThrown(*self*) := *fault*(*s*)
 PassForcedTerminationToChildren(*fault*(*s*))
 execMode(*self*) := *emExecutionFault*

In the *Execution-Fault* mode, if a termination notification is received, the pick agent terminates its enclosing activity and goes to the *Faulted* mode. Analogously to the *Completed* mode, sub-process agents terminate their execution in the *Faulted* mode. For the complete extended pick agent program see [12].

5 Related work

There are various research activities to formally define, analyze, and verify Web Services orchestration languages. A group at Humboldt University is working on formalizations of BPEL for analysis, graphics and semantics [13]. Specifically, they use Petri-nets and ASMs to formalize the semantics of BPEL. However, the pattern-based Petri-Net semantics of BPEL [14] does not capture fault handling, compensation handling, and timing aspects; overall, the feasibility of verifying more complex business processes is not clear and still subject to future work. The ASM semantic model in [15] closely follows what we had presented in [16] with minor technical differences in handling basic activities and variables.

Formal verification of Web Services is addressed in several papers. The SPIN model-checker is used for verification [17] by translating Web Services Flow Language (WSFL) descriptions into Promela. [18] uses a process algebra to derive a structural operational semantics of BPEL as a formal basis for verifying properties of the specification. In [19], BPEL processes are translated to Finite State Process (FSP) models and compiled into a Labeled Transition System (LTS) which is used as a basis for verification. [20] presents a model-theoretic semantics (based on situation calculus) for the DAML-S language which facilitates simulation, composition, testing, and verifying compositions of Web Services.

6 Conclusions

We formally define a BPEL abstract machine in terms of a distributed real-time ASM providing a precise and well defined semantic foundation for establishing the key semantic concepts of BPEL. Transforming informal requirements into precise specifications facilitates reasoning about critical language attributes, exploration of different design choices and experimental validation. As a result of our formalization, we have discovered a number of weak points in the LRM [12].

The dynamic nature of standardization calls for flexibility and robustness of the formalization approach. To this end, we feel that the ASM formalism and abstraction principles offer a good compromise between practical relevance and mathematical elegance — already proven useful in other contexts [6]. Our model can serve as a starting point for formal verification (considering formal specification as a prerequisite for formal verification). Beyond inspection by analytical means, we also support experimental validation by making our abstract machine model executable using the executable ASM language *AsmL* [21].

References

1. Andrews, T., et al.: Business process execution language for web services version 1.1 (2003) Last visited Feb. 2005, <http://www-106.ibm.com/developerworks/webservices/library/ws-bpel>.
2. Organization for the Advancement of Structured Information Standards (OASIS): WS BPEL issues list. (2004) <http://www.oasis-open.org>.

3. W3C: Web Services Description Language (WSDL) Version 1.2 Part 1: Core Language. (2003) Last visited May 2004, <http://www.w3.org/TR/2003/WD-wsd112-20030303>.
4. Börger, E., Stärk, R.: Abstract State Machines: A Method for High-Level System Design and Analysis. Springer-Verlag (2003)
5. ITU-T Recommendation Z.100 Annex F (11/00): SDL Formal Semantics Definition. International Telecommunication Union. (2001)
6. Glässer, U., Gotzhein, R., Prinz, A.: The formal semantics of sdl-2000: status and perspectives. *Comput. Networks* **42** (2003) 343–358
7. Börger, E., Glässer, U., Müller, W.: Formal Definition of an Abstract VHDL'93 Simulator by EA-Machines. In Delgado Kloos, C., Breuer, P.T., eds.: *Formal Semantics for VHDL*. Kluwer Academic Publishers (1995) 107–139
8. Glässer, U., Gurevich, Y., Veanes, M.: An abstract communication architecture for modeling distributed systems. *IEEE Trans. on Soft. Eng.* **30** (2004) 458–472
9. Gurevich, Y.: Sequential Abstract State Machines Capture Sequential Algorithms. *ACM Transactions on Computational Logic* **1** (2000) 77–111
10. Farahbod, R., Glässer, U., Vajihollahi, M.: Specification and Validation of the Business Process Execution Language for Web Services. In: *Proc. of the 11th Int'l Workshop on Abstract State Machines*, Springer-Verlag (2004)
11. Farahbod, R.: 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 (2004)
12. Farahbod, R., Glässer, U., Vajihollahi, M.: Abstract Operational Semantics of the Business Process Execution Language for Web Services. Technical Report SFU-CMPT-TR-2005-04, Simon Fraser University (2005) Revised version of SFU-CMPT-TR-2004-03, April 2004.
13. Martens, A.: Analysis and re-engineering of web services. To appear in 6th International Conference on Enterprise Information Systems (ICEIS'04) (2004)
14. Schmidt, K., Stahl, C.: A petri net semantic for BPEL4WS - validation and application. In Kindler, E., ed.: *Proceedings of 11th Workshop on Algorithms and Tools for Petri Nets*. (2004)
15. Fahland, D.: Ein Ansatz einer formalen Semantik der Business Process Execution Language for Web Services mit Abstract State Machines. Technical report, Humboldt-Universität zu Berlin (2004)
16. Farahbod, R., Glässer, U., Vajihollahi, M.: Specification and Validation of the Business Process Execution Language for Web Services. Technical Report SFU-CMPT-TR-2003-06, Simon Fraser University (2003)
17. Nakajima, S.: Model-checking verification for reliable web service. In: *OOPSLA 2002: Workshop on Object-Oriented Web Services*. (2002)
18. Koshkina, M., van Breugel, F.: Verification of Business Processes for Web Services. Technical Report CS-2003-11, York University (2003)
19. Foster, H., Uchitel, S., Magee, J., Kramer, J.: Compatibility verification for web service choreography. In: *Proceedings of the IEEE International Conference on Web Services (ICWS'04)*, IEEE Computer Society (2004) 738–741
20. Narayanan, S., McIlraith, S.A.: Simulation, verification and automated composition of web services. In: *Proceedings of the eleventh international conference on World Wide Web*, ACM Press (2002) 77–88
21. Farahbod, R., Gervasi, V., Glässer, U.: CoreASM: An extensible ASM execution engine. In: *Proc. of the 12th Int'l Workshop on Abstract State Machines*. (2005)