ASM-based semantics for BPEL: The negative Control Flow

D.Fahland, W. Reisig

Department of Computer Science, Humboldt-Universität zu Berlin

Abstract. BPEL is presently the most prominent language to specify and execute business processes, using Web Services as its technological basis. Particular problems arise when activities are *faulty*: faults have to be propagated, other activities have to be irregularly terminated, etc. We describe the formal semantics of *fault handlers* and *event handlers*, demonstrating that ASMs are most adequate for this purpose.

1 Introduction

Modern software architecture focuses on *service oriented architectures* (SOA). A *service* is an executable piece of software. Services are made to communicate, preferably along the world wide web. Thus *web services* constitute an increasingly important software architecture. The most important applications of web services include *business processes*, i.e. the agreement on and execution of joint business activities of different partners by help of messages over the web.

IBM, Microsoft and BEA established the *Business Process Execution Language for Web Services* (BPEL for short) in 2003. With a consortium of more than 130 partners, BPEL has turned into the de-facto standard of languages for business processes.

The semantics of languages such as BPEL differs fundamentally from the semantics of conventional programming languages: Typical elementary objects and operations are fairly abstract. They include objects such as "this is a reply to your previous request" or operations such as "compensate the last activity". This kind of objects and operations has no canonical representation in terms of conventional data structures. And there is no need for such a representation. The ASM style of specification explicitly supports this kind of abstract objects and operations.

One of the advantages of BPEL is its clear discrimination of *positive* and *negative* control flow: Positive control flow focuses the intended behaviour. [FGV04] and [Fah04] suggested two slightly different versions of an ASM-based semantics for positive control flow. Negative control flow manages faults, in particular *compensation* of activities already executed or started, but later revoked due to failures, missing resources or events outside the process. This case is much more difficult to handle. Objects now include *references* to control flow; compensation may propagate over a sequence of sites, requiring new kinds of messages and operations.

In this paper we show that negative control flow can be represented by fairly natural ASM rules. We present the decisive, albeit typical rules for the *fault handler* and the *event handler*.

This paper is organized as follows: In Section 2 we introduce BPEL at the required degree and explain how BPEL processes are being executed in terms of a distributed, reactive system. In Section 3 we present the ASM rules for the control flow. Introducing the fault handler, we show how negative control flow is triggered in Section 4. In Section 5 we introduce the event handler and show how the concept of positive control flow is extended. We give a brief look on related work in Section 6 and we close with a conclusion and an outlook on future work.

2 Structure of BPEL4WS

In this section we explain the aspects of BPEL relevant for this paper. We show that the control flow of BPEL can be expressed in terms of a distributed multi-agent ASM, as defined in [BS03]. The principle approach is similar to previous formalisations of distributed systems such as SDL-2000 [EGG$^+$01] and the UPnP architecture [GGV02b], [GGV02a].

The structures and behaviour described herein have informally been introduced in [CGK$^+$03].

2.1 Processes in BPEL4WS

Adapting the notion of business processes, a *BPEL process* describes the order of execution of workitems to achieve a commercial goal. In BPEL, *activities* denote the order of execution and workitems. Just as any business process, a BPEL process requires input and generates output. In the web services approach, both is realized by *messages* which are communicated with the process' environment via a defined *interface*.

BPEL defines two types of activities. A *basic activity* corresponds to a workitem like “send a message” (activity: *invoke*) or “manipulate data” (activity: *assign*). A *structured activity* defines the order of execution of other activities like “execute these activities sequentially” (activity: *sequence*) or “repeat the execution of this activity” (activity: *while*).

For each type of activity we introduce a new infinite set in our formal model. The set contains all (thinkable) activities of that type, whether they are used in a given process or not. For two different types of activities, the sets are disjoint. To distinguish the sets syntactically, we introduce a new symbol for each set. Because each of the symbols is interpreted as a subset of the carrier of the ASM, we will declare these symbols with the keyword **Universe**, e.g. **Universe**: $Activity = Sequence \cup While \cup Invoke \cup Assign \cup \ldots$.

The activities constituting a BPEL process are structured in a tree: each leaf is a basic activity, the root of a non-trivial subtree is a structured activity. We call this tree the *activity tree* of the process.

Formally, we declare two typed function-symbols whose interpretation is given in the initial state of the ASM.

$$childActivities : Activity \rightarrow \mathcal{P}(Activity)$$
$$parentActivity : Activity \rightarrow Activity$$

By definition, if act_{root} is the root activity of a process' activity tree then $parentActivity(act_{root}) =_{def} act_{root}$. We also say that the activities in the subtree under an activity are *inside* of it or are *enclosed* by it.

We distinguish the definition of a BPEL process from its execution. The *definition of a process* provides its static structure and is written as an XML-document: For the sake of simplicity, a BPEL process consists of exactly one activity (which may be structured) and the description of its interface is written in WSDL [CCMW01]. The *definition of an activity* is the part of the process definition which defines the static properties of an activity and is given by an XML-tag. Fig. 1 depicts an example, to which we will refer throughout the paper.

```
<scope name="S" ...>
  <sequence name="I" ...>
    <activity name="A" .../>
    <activity name="B" .../>
  </sequence>

  <faultHandlers>
    <catchAll>
      <activity name="C" .../>
    </catchAll
  </faultHandlers>

  <eventHandlers>
    <onMessage ...>
      <sequence name="I2">
        <activity name="D" .../>
        <activity name="E" .../>
      </sequence>
    </onMessage>
  </eventHandlers>
```

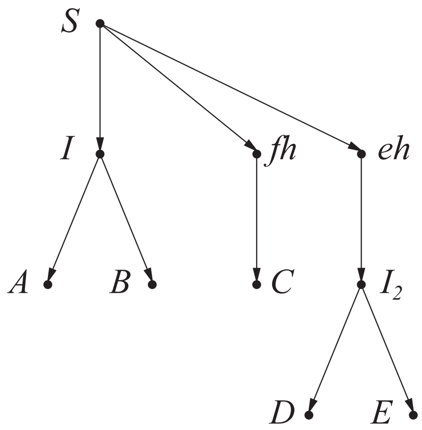

Fig. 1. A part of a BPEL process definition denoted in XML and the corresponding activity tree. The `activity` tags shall denote an arbitrary activity. Since fault handlers and onMessage handlers don't carry a name, we abbreviate them with fh and eh respectively.

Our model formalises each aspect of a process definition by the help of static functions, given in the initial state of the ASM; [Fah04] provides details. Any element $act \in Activity$ which is reachable via these static functions is a structural element of the process definition that corresponds to an activity of the process.

2.2 Executing Processes in BPEL4WS

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

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

Furthermore, each instance of any activity has a particular dynamic property, its *internal state*. **Universe**: $ActivityState$.

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

A finite state machine (FSM) describes the transitions of the internal state and reflects the *business agreement protocol* [CGK+03]. The FSM is defined equally for each activity.

To simplify reading, "activity" frequently stands for "instance of activity" in case confusion can be excluded.

The *execution of an activity's instance* is coupled to the instance's finite state machine. Depending on its internal state, the activity modifies dynamic properties (including its internal state). Initially, each activity is $disabled \in ActivityState$. It may be set to $enabled \in ActivityState$ to start is execution. Finishing its execution, it enters the terminal state $completed \in ActivityState$ which expresses a *successful completion* of its execution. We say that an activity is *active* after becoming *enabled* but before reaching a terminal, internal state.

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

Execution of an instance of a structured activity means to execute instances of its child activities: Depending on the semantics of the activity and the activity's definition, it may have to execute its child activities in a particular order, execute all of its child activities concurrently or repeatedly execute a child activity, to name a few examples.

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

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

The root's execution is triggered by some mechanism we do not want to discuss in this paper. The execution of a process instance completes successfully if its instance of the root activity completes successfully.

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

It is not necessary to introduce a new symbolic identifier for each instance of an activity. In general, a structured activity executes a child activity at most once – hence no need to discriminate several instances of the same child activity. There are just two exceptions: one is the while-activity (**Universe**: $While$) which iterates the execution of its child activity, the other one is the event handler (**Universe**: $EventHandler$) which we will introduce in Section 5.

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

Defining $parentInstance$ while we execute a BPEL process, we construct a tree of symbolic identifiers induced by the activity tree. We call this tree the *subinstance tree*. We conceive the set of all instances of activities sharing the same symbolic identifier or one in the subtree of the subinstance tree below the identifier as a *subinstance* of the process instance. Consequently, we read the root of the subinstance tree as the identifier of the process instance: **Universe**: $ProcessInstance \subset SubInstance$.

By the activity tree and the subinstance tree, all instances of activities of a process instance are structured in a large tree, its root being the (single) instance of the root of the activity tree.

Activity-to-Activity Communication We will present now the major decision we made for the formalisation of BPEL by introducing a type and an operation which is not explicitly employed in the informal specification [CGK$^+$03]. Speaking about the execution of BPEL processes we briefly mentioned the internal state of an activity's instance and that it has to be set to *enabled* to execute that instance. When we approach the formal definition, we have to model the transition from *disabled* to *enabled* by an unambiguous operational step.

Computer science knows two kinds of operational steps which make sense: The first one is to let the parent activity[1] alter the internal state of the child activity. The second one is to let the child activity do this by itself, triggered by a message from its parent activity. Considering that activities are executed concurrently and that BPEL, although being executable, also may serve as a modelling language where parts of the process may be implemented in a different technology, we favor the second option.

We show now that it suffices to conceive the instance of a BPEL process as a distributed, reactive system to describe the behaviour presented above. Hence we may employ the concept of the *distributed multi-agent ASM*.

Each instance of an activity hides its internal structure and its dynamic properties against other instances of activities. Following the distributed approach, each instances of an activity is executed by its "own" ASM agent, **Universe**: *Agent*. Each agent executes exactly one instance of an activity. The agent needs to know the symbolic identifier of the activity's instance and the definition of the activity:

$$myInstance : Agent \rightarrow SubInstance$$
$$myStatic : Agent \rightarrow Activity$$

As described in [BS03] to make a step, each agent applies the ASM rule ExecuteAgent which is parameterized with its name, using the generic symbol self. The definition of ExecuteAgent is given at the end of this section. Having defined $myInstance(\mathsf{self})$ and $myStatic(\mathsf{self})$ for each agent, we may define an ASM rule ExecuteActivity($\mathsf{sI} \in SubInstance$, $\mathsf{act} \in Activity$) for each type of activity which is applied each time the agent makes a step. By applying the ASM rule

$\underline{\text{CreateAgent}}$
$\quad(\mathsf{sI}_{parent} \in SubInstance, \mathsf{sI}_{new} \in SubInstance, \mathsf{act} \in Activity) \equiv$

let $\mathsf{agent} = \mathbf{new}(Agent)$ **in**
$\quad myInstance(\mathsf{agent}) := \mathsf{sI}_{new}$
$\quad myStatic(\mathsf{agent}) := \mathsf{act}$
$\quad$**if** $\mathsf{sI}_{parent} \neq \mathsf{sI}_{new}$ **then**
$\quad\quad parentInstance(\mathsf{sI}_{new}) := \mathsf{sI}_{parent}$

a new ASM agent for $(\mathsf{sI}_{new}, \mathsf{act})$ is created. If a new symbolic identifier was used to create the agent, the identifier is appended to the subinstance tree.

In our model, two neighboured activities of the activity tree may influence each other's dynamic properties using *signals*. A signal is an internal message, not to be confused with messages exchanged among BPEL processes. We distinguish signals sent from parent activities to their children and vice versa, **Universe**: *UpSignals*, **Universe**: *DownSignals*. Signals are exchanged through a *signal channel*, so we introduce a channel from the parent to each child and one vice versa. Formally, a channel is set of signals. The channel is defined by its static

[1] In this section, "activity" stands for "instance of activity".

endpoints (parent and child) and the subinstance of the child activity. The subinstance of the parent activity is not required since it is unambiguously given by the subinstance tree.

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

Each node is reactive on signals sent to it: it changes its internal state or other dynamic properties. To enable a child activity, a structured activity has to send a signal. If an activity completes, it has to notify its parent using a signal.

Introducing this distributed architecture, we do not want an activity to change a dynamic property of another activity directly, i.e. any changes of dynamic properties of an activity are made by the activity itself due to its internal state or due to signals. Each instance of an activity has to implement the same interface to properly execute a process instance. In [Fah05] we have shown that a small set of fairly natural signals suffices to achieve this aim. We will present most of them in the next sections.

With this decision in the design of the operational semantics of BPEL, all structures and the entire behaviour of the activities can naturally be translated from the informal description to the formal ASM specification.

Control Flow The *Control flow* in BPEL is just the order of execution of its activities, caused by exchanging signals. The *positive* control flow is the intended behaviour of the process in the "all-well-case". The *negative* control flow is the well-defined behaviour in case of an error. We will cover both in the forthcoming sections.

2.3 Scope

We want to close this section with the introduction of a special activity, the *scope*. A scope is a structured activity with exactly one *primary* child activity. Following the positive control flow, a scope is executed by executing its primary activity and it completes after its primary activity completed. But the scope provides behaviour beyond positive control flow. There may be *handlers* attached to the scope. The execution of the handlers is not mandatory.

These handlers are: the *fault handler* to supervise each fault occurring inside the scope, the *event handler* to supervise incoming messages and timer events while the part of the process, which is enclosed by the scope, is running, and the *compensation handler* to undo actions of activities within the scope *after* the positive control flow of the scope completed successfully. Execution of a compensation handler may be triggered by a fault handler or a compensation handler attached to a scope which encloses the current scope.

By definition, the root activity of every BPEL process is a scope and each scope always has exactly one fault handler which is given implicitly at least. Additionally, a scope allows to declare *variables* which are accessible to any activity inside the scope. A variable in BPEL is like a variable in any other imperative programming language.

In the forthcoming sections we will focus the fault handler and focus the event handler. We skip the compensation handler here as its proper formalization requires some extensions within the positive control flow which we do not want to motivate here.

3 Control Flow for Activities

In Section 2.2 we shortly introduced our distributed multi-agent ASM model where control flow is realized by activities exchanging signals. To understand the central result of this paper – the mechanisms of fault handling and event handling – we need to formalise what kind of signals are exchanged and how and when they are exchanged.

In this section, we show that all activities which participate in the positive control flow share the same set of rules and functions to exchange signals.

3.1 Positive Control Flow

We now show, how the finite state machine of an activity's internal state and the exchange of signals together constitute the positive control flow in BPEL.

An activity is enabled by its parent activity: The activity's instance is created and it waits for the signal $signalEnable \in DownSignals$ from its parent activity to enter its internal state *enabled*. The agent executing the parent activity $(\mathsf{sl}, \mathsf{act})$ has to create the agent using CreateAgent and send $signalEnable$ by applying the following rule.

$$\begin{aligned}&signalChannel_{down}(\mathsf{sl}, \mathsf{act}, childActivity(\mathsf{act}))\\&\qquad := signalChannel_{down}(\mathsf{sl}, \mathsf{act}, childActivity(\mathsf{act})) \cup \{signalEnable\}\end{aligned}$$

We exploit the macro concept of ASM and abbreviate this rule using an infix-style notation for ASM rules with n parameters as used in [Far04].

$$\textbf{signal } signalEnable \textbf{ to } childActivity(\mathsf{act}) \textbf{ in } \mathsf{sl} \textbf{ via } signalChannel_{down}.$$

In turn the child activity has to attend the channel for signals:

$$\begin{aligned}&\textbf{if } signalEnable \in signalChannel_{down}(\mathsf{sl}, parentActivity(\mathsf{act}), \mathsf{act}) \textbf{ then}\\&\quad signalChannel_{down}(\mathsf{sl}, parentActivity(\mathsf{act}), \mathsf{act})\\&\qquad := signalChannel_{down}(\mathsf{sl}, parentActivity(\mathsf{act}), \mathsf{act}) \setminus \{signalEnable\}\\&\quad \textbf{if } activityState(\mathsf{sl}, \mathsf{act}) = disabled \textbf{ then}\\&\qquad activityState(\mathsf{sl}, \mathsf{act}) := enabled\end{aligned}$$

for which we shortly write

$$\begin{aligned}&\textbf{onSignal } signalEnable \textbf{ from } parentActivity(\mathsf{act})\\&\qquad \textbf{in } \mathsf{sl} \textbf{ via } signalChannel_{down} \textbf{ do}\\&\quad \textbf{set } activityState \textbf{ from } disabled \textbf{ to } enabled\end{aligned}$$

Now being in *enabled*, the activity is executed according to its semantics. Having finished, it enters the state *completed* and sends $signalCompleted \in UpSignals$ to its parent. The parent has to wait for this signal in order to complete its execution. Obviously, any child activity may be parent activity of another activity. If so, the ASM rules for this activity include another in **signal** .. **to** .. -rule. Hence, this model defines the hierarchical execution of the positive control flow of a BPEL process.

As an example, fig. 2 shows the execution of the positive control flow of the example process in fig. 1.

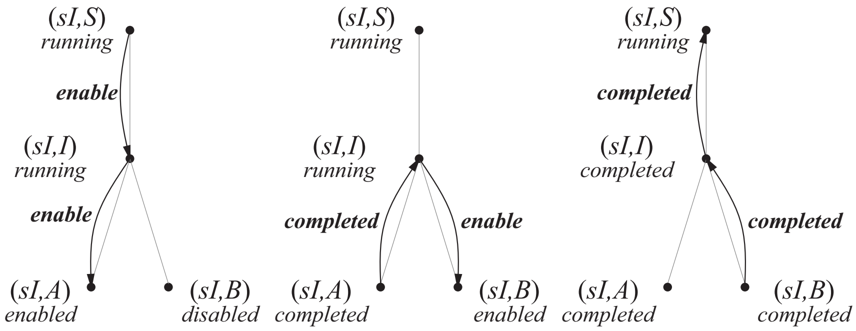

Fig. 2. The execution of the positive control flow from the example process in fig. 1. The inscription of each node describes the instance of the activity and its current internal state. A directed arcs denotes a signal sent from one activity to another, its inscription describes the signal: The scope S enables the sequence I, which enables its first child A. If the first child completes, A sends *signalCompleted* to its parent I, which enables its second child B. Upon completion of B, the sequence completes.

3.2 Propagating Faults

At any time during the execution of a process instance, its environment may alter the state of the system. A step of the environment may result in a state where the semantics of an activity is undefined. In such a case, a *fault* is generated (or *thrown*), **Universe**: $Fault$. In BPEL, a fault is a named message with an explicit *fault name* classifying the fault, **Universe**: $FaultName$. Formally, we declare the typed function symbol $faultName : Fault \rightarrow FaultName$.

The contents of the fault describes the error. Here, the term *message* is identical to the messages which are exchanged among BPEL processes.

As mentioned in sec. 2.3, each activity is enclosed by a scope, whose fault handler is in charge of handling any faults thrown inside the scope. To handle the fault, the fault handler has to receive (or *catch*) the fault first.

We follow the principles of the activity-to-activity communication. When a fault is thrown by an activity, it is passed to its parent activity through a channel

$faultChannel_{up}$. This channel is different from $signalChannel_{up}$ to distinguish between faults and signals.

$$faultChannel_{up} : SubInstance \times Activity \times Activity \rightarrow \mathcal{P}(Fault)$$

Each activity with the exception of scopes, receiving a fault through the $faultChannel_{up}$ forwards the fault to its parent activity using the designated channel. Hence each fault eventually reaches a scope. A fault which reaches a scope is forwarded to the scope's fault handler by the same means.

In principle, the rules for propagating faults have been presented in the previous subsection. For this paper, we assume that they are subsumed in the ASM rule PropagateFaultsActivity($\mathsf{sI} \in SubInstance, \mathsf{act} \in Activity$).

3.3 Stopping Parts of a Process Instance: The Stop Concept

Whenever a fault is caught by a fault handler, the propagation of the positive control flow has to be stopped in that part of the process which is enclosed by the scope of the fault handler.

Similar to enabling activities, a structured activity may stop its child activities by sending a dedicated signal $signalStop \in DownSignals$. Upon receiving this signal each activity immediately stops its current execution regardless of its internal state and enters the internal state $stopping \in ActivityState$. For this paper we assume that this transition is formalised in the ASM rule StopActivity($\mathsf{sI} \in SubInstance, \mathsf{act} \in Activity$).

If the activity is a basic activity, it confirms its stopping by sending the signal $signalStopped \in UpSignals$ to its parent and enters the internal state *stopped*. If the activity is structured, it forwards *signalStop* to all of its active child activities and awaits their confirmation. Finally, the fault handler receives a *signalStopped* which indicates that the enclosed part of the process stopped its execution. We call this behaviour the *stop concept*.

The formal rules for this behaviour provide nothing new and therefore are skipped.

3.4 General Behaviour of BPEL4WS Activities

We will now present an abstract ASM rule including each aspect of control flow for an activity. We write 'abstract' because the concrete ASM rule for a concrete activity is a refinement of that rule.

In addition to the the positive control flow, the propagation of faults and the stop concept, BPEL includes two additional types of information to influence the control flow.

The first one is the immediate termination of the entire process instance. For this paper we assume that the rules for this behaviour subsume in the ASM rule PropagateTerminate. The second one deals with the concept of *links* and the *dead path elimination*. We also skip a detailed definition and assume

PROPAGATEDPE. Both rules wouldn't provide anything new. The detailed definition and explanations can be found in [Fah05].

Up to now, each BPEL activity executes the mentioned behaviour together with its specific behaviour, defined by a finite state machine, which we assume to be defined in ACTIVITYSTATEMACHINE. Hence the rule for each activity abstractly reads as follows:

<u>EXECUTEACTIVITY</u>
    (sI $\in$ *SubInstance*, act $\in$ *Activity*) $\equiv$

PROPAGATEDPE(sI, act)
PROPAGATETERMINATE(sI, act)
onSignal *signalStop* **from** *parentActivity*(act)
    in *sI* **via** $signalChannel_{down}$ **do**
  STOPACTIVITY(sI, act)
otherwise
  onSignal *signalEnable* **from** *parentActivity*(act)
      in *sI* **via** $signalChannel_{down}$ **do**
    set *activityState* **from** *disabled* **to** *enabled*
  PROPAGATEFAULTSACTIVITY(sI, act)
  ACTIVITYSTATEMACHINE(sI, act)

The rule above shows that the macro concept provides excellent means to subsume semantically corresponding operational steps in a single rule. ASM's principle of parallel composition allows us to show mutual dependency and independency of aspects of the behaviour of an activity: The positive control flow and the execution of the activities is dominated by the stop-concept. Dead path elimination, propagation of termination and stop-concept are independent of each other. The same holds for the execution of an activity and the propagation of faults. The ASM-style of specifying systems reveals the key aspects of the behaviour already on the syntactical level.

3.5 The distributed ASM model

We will close this section with a description of the distributed ASM as a whole. For each type of activity in BPEL, there exists a refinement of EXECUTEACTIVITY and ACTIVITYSTATEMACHINE to specific rules, e.g. for the sequence, we may obtain EXECUTESEQUENCE and SEQUENCESTATEMACHINE respectively via refinement. The ASM rule which is applied by each ASM agent of the distributed ASM model distinguishes the behaviour of the agent by its static type and its subinstance using *myInstance* and *myStatic* as defined in Section 2.2.

<u>EXECUTEAGENT</u>
    (self $\in$ *Agent*) $\equiv$

let sI = *myInstance*(self),
    act = *myStatic*(self) **in**
  if *act* $\in$ *Scope* **then**

EXECUTESCOPE(sI, act)
if $act \in Sequence$ **then**
EXECUTESEQUENCE(sI, act)
...

The entire distributed multi-agent ASM has a (quite large) signature including all function symbols and set symbols introduced throughout this paper (and some more symbols, which are omitted here). The ASM defines a dynamic set of ASM agents. Each agent applies EXECUTEAGENT each time it makes a move.

The initial state of the ASM has an infinite, countable carrier. Its functions are given by a large specification φ_{init} in first order logic using terms of its signature. The specification is a conjunction of two specifications: $\varphi_{init} \equiv \varphi_{stat} \wedge \varphi_{dyn}$. The first specification φ_{stat} is specifically given for each specific BPEL process, defining the static structure of the process. φ_{stat} can be constructed from the source-code of a BPEL process, [Fah04] provides details. The second specification φ_{dyn} is equally given for any BPEL process. It defines the initial values of the dynamic functions and the subsets of the carriers, i.e. the universes.

4 Fault Handler

In this section we show that by using the functions and rules defined in the previous sections, we may formalise the fault handler's behaviour using just the structures and operations introduced in the informal specification.

As mentioned in Section 2.3, a fault handler is in charge of handling any faults occurring within the scope it is attached to. Using the mechanisms presented in Section 3, the fault handler's semantics is to use positive control flow, fault propagation and the stop-concept properly.

A fault handler (**Universe**: $FaultHandler \subset Activity$) is enabled when the scope's primary activity is enabled. Being enabled, a fault handler performs three stages of execution: Firstly, to wait for a fault to be caught. Secondly, if a fault is caught, to stop the execution of the process inside the fault handler's scope. Thirdly, to execute an activity to handle the fault. A fault handler catches at most one fault.

We model these stages as special internal states: **Universe**: $FHStage = \{awaitFault, stopScope, executeCatch\}$, $faultHandlerStage : SubInstance \times FaultHandler \rightarrow FaultHandlerStage$.

The ASM rule, which is applied by the fault handler's agent if the fault handler is active, represents just another finite state machine. Its definition is straight forward. We will explain the fault handlers behaviour and its formal definition for each of the three stages in the following.

<u>RUNFAULTHANDLER</u>
$(\mathsf{sI} \in SubInstance, \mathsf{fh} \in FaultHandler) \equiv$

if $faultHandlerStage(\mathsf{sI}, \mathsf{fh}) = awaitFault$ **then**
WAITFORFAULT(sI, fh)

if $faultHandlerStage(\mathsf{sI}, \mathsf{fh}) = stopScope$ **then**
  $\textsc{StopScope}(\mathsf{sI}, \mathsf{fh})$
if $faultHandlerStage(\mathsf{sI}, \mathsf{fh}) = executeCatch$ **then**
  $\textsc{ExecuteCatch}(\mathsf{sI}, \mathsf{fh})$

Being in *awaitFault*, the fault handler just has to wait for its scope to send a fault, as we assume the distributed propagation of faults presented in Section 3.2. Since the fault is sent downwards the activity tree to the fault handler, we need a special channel to pass faults from a scope to its fault handler, $faultChannel_{down} : SubInstance \times Scope \times FaultHandler \rightarrow \mathcal{P}(Fault)$.

BPEL provides means to execute different activities to handle a fault depending on its type. Remember from Section 3.2 that a fault is a named message. Both, the fault name and the data-type of its contents, are used to discriminate faults and to supply the most appropriate activity for a fault.

```
<faultHandlers>?
    <catch faultName="qname"? faultVariable="ncname"?>*
       activity
    </catch>
    <catchAll>?
       activity
    </catchAll>
</faultHandlers>
```

Fig. 3. BPEL syntax to denote a fault handler

In the formal semantics we slightly alter the terminology in order to sharpen the semantics: Instead of letting a scope have multiple fault handlers – one for each type of fault, as the informal specification [CGK+03] describes – we let a scope have exactly one fault handler with one or more *catch nodes*, cf. Fig. 3. A catch node is *not* an activity, but a sub-structure of the fault handler. We do so because a fault handler handles no more than one fault and – by the informal specification – at most one fault may be handled the scope's fault handlers.

A catch node (**Universe**: $CatchNode$) has a fault name, references a variable (**Universe**: $Variable$) and encloses an activity. Each fault handler has a most general catch node. Its activity is in charge of handling unexpected faults. We called it the *catch all*-node.

$$\begin{aligned} faultHandlerCatchNodes &: FaultHandler \rightarrow \mathcal{P}(CatchNode) \\ catchNodeFault &: \quad CatchNode \rightarrow FaultName \cup \{catchAll\} \\ catchNodeActivity &: \quad CatchNode \rightarrow Activity \\ catchNodeVariable &: \quad CatchNode \rightarrow Variable \end{aligned}$$

The fault handler assumes that the activity of a catch node is appropriate to handle a fault if the fault names match and if the data-types of the fault and of the referenced variable match.

In our semantics we decided to chose that appropriate activity as soon as the fault has been caught. The fault handler has to remember its choice. We provide the function

$$faultHandlerActiveCatch : SubInstance \times FaultHandler \rightarrow CatchNode$$

for this purpose.

Let fault be caught by a fault handler's instance $(\mathsf{sI}, \mathsf{fh})$. Then for the sake of simplicity, we assume that applying $\textsc{ChooseCatchBranch}(\mathsf{sI}, \mathsf{fh}, \mathsf{fault})$ redefines $faultHandlerActiveCatch(\mathsf{sI}, \mathsf{fh})$ as the most appropriate catch node for the given fault. The same rule shall also store the contents of the fault at the catch node's variable. Hence the behaviour of a fault handler in the stage *awaitFault* formally reads as follows:

$\underline{\textsc{WaitForFault}}$
$\quad(\mathsf{sI} \in SubInstance, \mathsf{fh} \in FaultHandler) \equiv$

if $faultChannel_{down}(\mathsf{sI}, parentActivity(\mathsf{fh}), \mathsf{fh}) \neq \emptyset$ **then**
  choose $\mathsf{fault} \in faultChannel_{down}(\mathsf{sI}, parentActivity(\mathsf{fh}), \mathsf{fh})$ **in**
    $faultHandlerStage(\mathsf{sI}, \mathsf{fh}) := stopScope$
    $\textsc{ChooseCatchBranch}(\mathsf{sI}, \mathsf{fh}, \mathsf{fault})$

Now being in *stopScope*, the fault handler has to stop the execution of activities within the scope. To do so, we employ the stop-concept (cf. Section 3.3). Since every activity behaves properly according to the stop concept, the fault handler just has to send *signalStop* to its associated scope and to await a *signalStopped* in return. From the perspective of the fault handler, this involves two stages, for which we introduce **Universe**: $FHStopMode =_{def} \{sendingStop, awaitingStopped\}$ and

$$fHStopMode : SubInstance \times FaultHandler \rightarrow FHStopMode$$

used in:

$\underline{\textsc{StopScope}}$
$\quad(\mathsf{sI} \in SubInstance, \mathsf{fh} \in FaultHandler) \equiv$

if $faultHandlerStopMode(\mathsf{sI}, \mathsf{fh}) = sendingStop$ **then**
  signal *signalStop* **to** $parentActivity(\mathsf{fh})$ **in** sI **via** $signalChannel_{up}$
  $faultHandlerStopMode(\mathsf{sI}, \mathsf{fh}) := awaitingStopped$

if $faultHandlerStopMode(\mathsf{sI}, \mathsf{fh}) = awaitingStopped$ **then**
  onSignal *signalStopped* **from** $parentActivity(\mathsf{fh})$
    in sI **via** $signalChannel_{down}$ **do**
    $faultHandlerStage(\mathsf{sI}, \mathsf{fh}) := executeCatch$

Hereby we modelled the scope to listen for *signalStop* from its fault handler and to forward it to its child activity and to do the reverse for *signalStopped*. Fig. 4 shows a scenario of propagating and catching a fault together with the application of the the stop-concept.

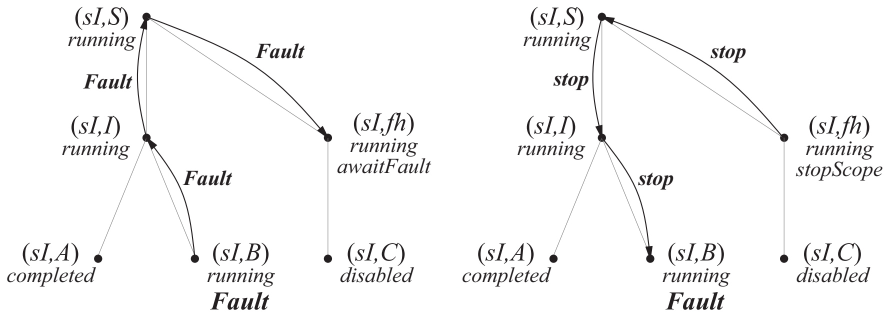

Fig. 4. The execution of the negative control flow in the example process of fig. 1. During the execution of B, a $Fault$ was thrown. It is propagated to the enclosing scope S and sent to fh. The fault handler switches from stage $awaitFault$ to stage $stopScope$ and initiates the stop-concept at S. Then fh waits for $signalStopped$ from S.

Having reached $executeCatch$, the execution of the chosen, appropriate activity to handle the fault is fairly simple. Just send $signalEnable$ to it and await $signalCompleted$. Once again we require two stages **Universe**: $FHExecMode = \{sendingEnable, awaitingCompleted\}$ and the corresponding function

$$fHExecuteMode : SubInstance \times FaultHandler \rightarrow FHExecMode.$$

Hence the ASM rule reads as follows:

<u>ExecuteCatch</u>
$(\mathsf{sI} \in SubInstance, \mathsf{fh} \in FaultHandler) \equiv$

let act $= catchNodeActivity(faultHandlerActiveCatch(\mathsf{sI}, \mathsf{fh}))$**in**
 if $fHExecuteMode(\mathsf{sI}, \mathsf{fh}) = sendingEnable$ **then**
  CreateAgent(sI, sI, act)
  signal $signalEnable$ **to** act **in** sI **via** $signalChannel_{down}$
  $fHExecuteMode(\mathsf{sI}, \mathsf{fh}) := awaitingCompleted$

 if $fHExecuteMode(\mathsf{sI}, \mathsf{fh}) = awaitingCompleted$ **then**
  onSignal $signalCompleted$ **from** act **in** sI **via** $signalChannel_{up}$ **do**
   $activityState(\mathsf{sI}, \mathsf{fh}) := completed$
   signal $signalCompleted$ **to** $parentActivity(\mathsf{fh})$
    in sI **via** $signalChannel_{up}$

Upon reaching the last line of the above rule, the fault handler completes its execution.

We consider a fault handler to be an activity. Hence the full formal definition of the fault handler's behaviour ExecuteFaultHandler is a refinement of ExecuteActivity (cf. sec. 3.4). The ASM rule RunFaultHandler which we introduced in this section is referenced inside of FaultHandlerStateMachine that refines the abstract model of an activity's state machine.

The refined rules include the propagation of faults occurring within the fault handler as well. We can model this quite important case for the complete semantics of BPEL in a straight forward manner. Any such fault is forwarded to the associated scope of the fault handler. To achieve compliance with the informal specification, we just require a slight modification of PROPAGATEFAULTSACTIVITY for the scope. Details are given in [Fah05].

5 Event Handler

In this section, we present the natural formalisation of the event handler as it raises from the informal specification and the distributed ASM-model which we introduced in the preceding sections.

We briefly mentioned the event handler in Section 2.3. An event handler is enabled to react on events as long as the scope's primary activity is active. BPEL defines two types of *events*.

OnAlarm events are timed events. They *occur* if the clock of the process reaches a certain moment of time (the intuitive understanding of clock shall suffice for this paper.) An onAlarm event may occur at most once. An *onMessage* event occurs each time a message arrives from a remote web service at a designated interface of the process. An onMessage event may occur several times.

Each event handler (**Universe**: $EventHandler \subset Activity$) has exactly one child activity and *handles* exactly one event. An event handler handles its event by executing its child activity. Upon an *onAlarm event handler* starts handling its event, it ignores any further occurrence. It completes when its child activity completes. Its formal semantics provides no surprise and is skipped therefore in this paper. In contrast, an *onMessage event handler* stays active to handle further onMessage events: Each message that generates an event has to be received and has to be handled by the handler.

The interesting aspect of the onMessage event handler's behaviour is, that it handles each onMessage event concurrently to previous events: for each event, the onMessage event handler creates a new instance of its child activity which is in charge of processing the contents of the received message. In the following we will formalise the operational steps of executing multiple instances of the fault handlers child activity.

For the sake of simplicity we do not model how a message (**Universe**: $BPELMsg$) is received in this paper. Instead we will consider the rule which is finally applied, each time a message msg arrives at the corresponding event handler $(\mathsf{sl}, \mathsf{eh})$: HANDLEMESSAGEEVENT$(\mathsf{sl}, \mathsf{eh}, \mathsf{msg})$.

The message generating the event has some contents. This contents must be made available to the new instance which was just created to handle the event. The contents is copied to a variable local to the new instance; the variable is given by $eventHandlerVariable : EventHandler \rightarrow Variable$. We omit the full specification and simply reference the rule COPYMSGTOVARIABLE.

Creating multiple instances of its child activity, the event handler has to keep track of the running instances of its child activity. A set suffices to achieve this:

$eHRunningInstances : SubInstance \times EventHandler \rightarrow \mathcal{P}(SubInstance)$. In all other aspects, the behaviour is identical to any other structured activity. The rule to handle an onMessage event reads as follows.

<u>HANDLEMESSAGEEVENT</u>

$(\mathsf{sI} \in SubInstance, \mathsf{eh} \in EventHandler, \mathsf{msg} \in BPELMsg) \equiv$

let $\mathsf{sI}_{child} = \mathbf{new}(SubInstance)$ **in**
  CREATEAGENT(sI_{child}, $childActivity(\mathsf{eh})$)
  $eHRunningInstances(\mathsf{sI}, \mathsf{eh}) := eHRunningInstances(\mathsf{sI}, \mathsf{eh}) \cup \{\mathsf{sI}_{child}\}$
  signal $signalEnable$ **to** $childActivity(\mathsf{eh})$
    in sI_{child} **via** $signalChannel_{down}$
  COPYMSGTOVARIABLE(sI_{child}, msg, $eventHandlerVariable(\mathsf{eh})$)

BPEL's informal specification requires an event handler to handle events as long as the scope's primary activity is active. Nonetheless, activities handling events that occurred earlier may still be active when the scope's primary activity completes. These activities are allowed to complete their execution. The handling of new events is then not allowed.

This requires a new signal to be sent from the scope to its handlers, a "request for completion": $signalComplete \in DownSignals$. Upon receiving $signalCompleted$ from its primary activity, the scope sends $signalComplete$ to its event handlers. Then, the scope waits for $signalCompleted$ from each of its handlers in turn. The operational steps are known.

In our model, the event handler stops to listen for events by entering the internal state $completing \in ActivityState$ upon receiving $signalComplete$ from its scope. The following rule is applied as long as the event handler is active, i.e. in its internal state *running*:

<u>FINISHEVENTHANDLER</u>

$(\mathsf{sI} \in SubInstance, \mathsf{eh} \in EventHandler) \equiv$

onSignal $signalComplete$ **from** $parentActivity(\mathsf{eh})$ **in** sI
    via $signalChannel_{down}$ **do**
  $activityState(\mathsf{sI}, \mathsf{eh}) := completing$

Being in state *completing*, the event handler may complete if there are no running instances of its child activity. The rule COMPLETEEVENTHANDLER is applied according to the activity's state machine model.

<u>COMPLETEEVENTHANDLER</u>

$(\mathsf{sI} \in SubInstance, \mathsf{eh} \in EventHandler) \equiv$

if $eHRunningInstances(\mathsf{sI}, \mathsf{eh}) = \emptyset$ **then**
  signal $signalCompleted$ **to** $parentActivity(\mathsf{eh})$ **in** sI **via** $signalChannel_{up}$
  $activityState(\mathsf{sI}, \mathsf{eh}) := completed$

To round out the semantics, we present the rule which removes an instance from the set *eHRunningInstances* if the corresponding child activity's instance of the handler completes. This rule is applied in *every* state of the handler's state machine.

$\underline{\text{FINISHMESSAGEEVENT}}$
$(\mathsf{sI} \in SubInstance, \mathsf{eh} \in EventHandler) \equiv$

forall $\mathsf{sI}_{child} \in eventHandlerRunningInstances(\mathsf{sI}, \mathsf{eh})$ **do**
onSignal $signalCompleted$ **from** $childActivity(\mathsf{eh})$
in sI_{child} **via** $signalChannel_{up}$ **do**
$eHRunningInstances(\mathsf{sI}, \mathsf{eh}) := eHRunningInstances(\mathsf{sI}, \mathsf{eh}) \setminus \{\mathsf{sI}_{child}\}$

These four ASM rules formalise the key aspects of the positive control flow of event handling. Its execution is optional and subject to events. The completion of the execution is initiated by the completion of the normal positive control flow in the part of the process enclosed by the event handler's scope.

Like the fault handler, the event handler is an activity of which the behaviour is formally defined by the rule EXECUTEEVENTHANDLER obtained by refining EXECUTEACTIVITY. All rules presented in this section are denoted inside of EVENTHANDLERSTATEMACHINE, a refinement of the abstract model of an activity's state machine.

We close this section with a short discussion of the relation of the negative control flow and event handling. Since we defined an event handler as an activity it supports fault propagation and the stop concept. Faults being thrown within a scope's handler are treated like faults being thrown inside the scope's primary activity. Furthermore, stopping the execution of positive control flow also includes stopping the execution of event handlers and their children. For both, all relevant rules have already been presented in this paper. In fig. 5, we show an example for the negative control flow within an event handler.

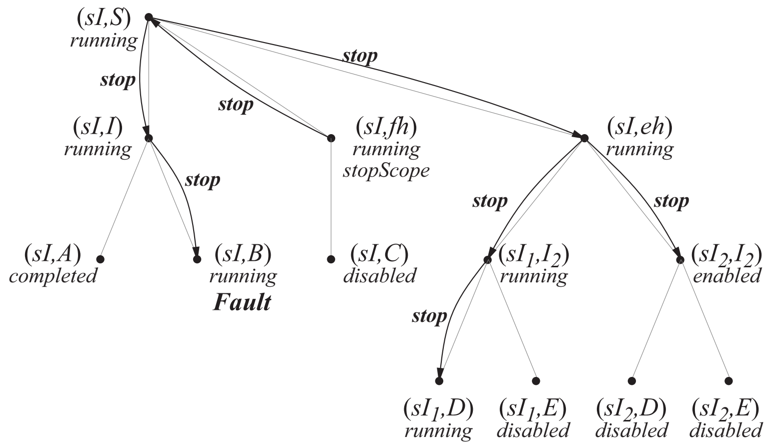

Fig. 5. The execution of the negative control flow within an event handler in the example process of fig. 1. Like in fig. 4, B throws a $Fault$ which is caught by the fault handler fh. Because of the presence of eh, stopping the scope S includes stopping all running instances of I_2. The current scenario depicts the propagation of *signalStop* through the tree of instances of activities.

At this point it is obvious, that a scope's fault handler must not complete before each of the scope's event handlers completed, even if the primary activity already did. The reason is that a fault still may be thrown inside an event handler in that situation. So upon receiving *signalCompleted* from each of its event handlers, the scope initiates the completion of the fault handler in the same way. After receiving *signalCompleted* from the fault handler, the scope may complete.

Although the informal specification lacks details about the operational steps of event handling, it was no problem for us to extend the formal definition of control flow on this special activity within the ASM model. In effect, the ASM-style of specifying distributed systems allowed a seamless integration of the event handling into the existing model. Hence, we may conceive the event handling as an extension of the model just as it is meant to be in the informal specification.

6 Related Work

Several papers have been published, contributing to the development of formal semantics and analysis of BPEL processes.

A group at the Simon Fraser University (SFU) already published a subsequent series of technical reports formalising parts of BPEL using a distributed real-time Abstract-State Machine model [FGV04], [Far04]. The model captures the three levels of abstraction of BPEL, with the finest model an executable AsmL specification. Currently, this model does not include a detailed definition of the variable extension of BPELand event handling. This limits a comparison of their contribution with ours.

The aspects of BPEL which have been modelled at SFU *and* in [Fah04], [Fah05] are very similar. The differences are rather philosophical than pragmatical: The SFU model reduces the number of ASM agents required for an execution to the bare minimum, i.e. each activity of a sequence is subsequently executed by the same agent where we introduce one agent for each activity. The models differ in their degree of concurrency.

There are a number of further approaches to formalise the operational semantics of BPEL and other web service orchestration languages or parts of them. Complete operational semantics for BPEL using Petri nets as defined in [Sta04], allows automated verification of BPEL processes using model-checking technologies [SS04]. A similar approach for formal semantics and verification of web services written in DAML-S has been published in [NM03].

[vBK03] presents a detailed formalization and analysis of the dead-path elimination of BPEL using a process algebraic approach.

7 Conclusion and Future Work

In this paper we explained and formalised the principles of positive and negative control flow in BPEL using a distributed multi-agent ASM. We have shown, that if conceiving each instance of an activity of a BPEL process as an independent agent, the entire control flow of BPEL can be expressed by the means of asynchronous communication between activities and an equally-shaped finite state machine inside of each activity.

The semantics relies on signal channels between neighboured activities and a reactive behaviour on signals by each activity. Each activity sends and forwards signals according to its specific semantics. As each activity implements the same interface for signal-based communication, the specific semantics for each activity may abstract from the semantics of its neighbours and hence may operate on the interface only.

In particular we have shown that non-standard control flow, like negative control flow caused by fault handlers and optional positive control flow caused by event handlers, requires no changes at the concept and can be formalised as a conservative extension of the positive control flow.

The choice in the design of the formal semantics to introduce signals is covered by the field of application of BPEL. The ASM formalism supports our choice of the model's architecture just as it would support any other design. The formal definition of structures and behaviour by functions and ASM rules arises naturally from the informal specification and the understanding of the language. At no point in our work, we had to compromise between the ASM-formalism to formally express structures and the intended behaviour of BPEL.

The full formal semantics for BPEL using this concept of control flow, also formalises the specific semantics for each activity can be found in [Fah05]. The surrounding field of BPEL requires further work. Besides a verification of the presented semantics, BPEL's relation to underlying techniques, such as WS-Addressing [CB+03] needs to be formalised. We suggest the ASM-method as the decisive formalism to do so.

Acknowledgments

We'd like to thank Christian Stahl for his useful comments during the creation of the formal semantics and Roozbeh Farahbod from whom we obtained the idea how to denote the signal concept.

References

[BS03] E. Börger and R. Stärk. *Abstract State Machines – A Method for High-Level System Design and Analysis.* Springer-Verlag, 2003.

[CB+03] A. Curbera, D. Box, et al. Web Services Addressing (WS-Addressing). Specification, BEA, IBM, Microsoft, March 2003. http://msdn.microsoft.com/ws/2003/03/ws-addressing/.

[CCMW01] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web Services Description Language (WSDL) 1.1. Specification, Ariba, IBM, Microsoft, March 2001. http://www.w3.org/TR/wsdl.html.

[CGK+03] F. Curbera, Y. Goland, J. Klein, F. Leymann, D. Roller, and S. Weerawarana. Business Process Execution Language for Web Services Version

1.1. Specification, BEA Systems, IBM, Microsoft, SAP, Siebel, 05 May 2003. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbiz2k2/html/bpel1-1.asp.

[EGG$^+$01] R. Eschbach, U. Glässer, R. Gotzhein, M. von Lövis, and A. Prinz. Formal Definition of SDL-2000 - Compiling and Running SDL Specifications as ASM Models. *Journal of Universal Computer Science*, 7(11):1024–1049, November 2001.

[Fah04] Dirk Fahland. Ein Ansatz einer formalen Semantik der Business Process Execution Language for Web Services mit Abstract State Machines. Studienarbeit, Humboldt-Universität zu Berlin, June 2004.

[Fah05] Dirk Fahland. Formal Operational Semantics of BPEL4WS. Technical report, Humboldt-Universität zu Berlin, 2005. to appear.

[Far04] Roozbeh Farahbod. Extending and Refining an Abstract Operational Semantics of the Web Services Architecture for the Business Process Execution Language. Master thesis, Simon Fraser University, Burnaby B.C. Canada, July 2004.

[FGV04] Roozbeh Farahbod, Uwe Glässer, and Mona Vajihollahi. Abstract Operationial Semantics of the Business Process Execution Language for Web Services. Technial report, Simon Fraser University, Burnaby B.C. Canada, April 2004. SFU-CMPT-TR-2004-03.

[GGV02a] U. Glässer, Y. Gurevich, and M. Veanes. An Abstract Communication Model. Technical report, Microsoft Research, May 2002. MSR-TR-2002-55.

[GGV02b] U. Glässer, Y. Gurevich, and M. Veanes. High-level executable specification of the universal plug and play architecture. In *HICSS '02: Proceedings of the 35th Annual Hawaii International Conference on System Sciences (HICSS'02)-Volume 9*, page 283. IEEE Computer Society, 2002.

[NM03] Srini Narayanan and Sheila A. McIlraith. Analysis and simulation of Web Services. *Computer Networks*, 42(5):675–693, 2003.

[SS04] Karsten Schmidt and Christian Stahl. A Petri net Semantic for BPEL4WS - Validation and Application. In Ekkart Kindler, editor, *Proceedings of the 11th Workshop on Algorithms and Tools for Petri Nets (AWPN'04)*, pages 1–6. Universität Paderborn, october 2004.

[Sta04] Christian Stahl. Transformation von BPEL4WS in Petrinetze. Diplomarbeit, Humboldt-Universität zu Berlin, April 2004.

[vBK03] Franck van Breugel and Mariya Koshinka. Does Dead-Path-Elimination have Side Effects. Technial report, York University, Canada, April 2003. TR-2003-04.