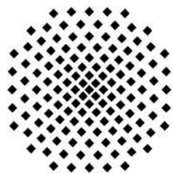
View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Online Publikationen der Universität Stuttgart



Universität Stuttgart

Fakultät Informatik, Elektrotechnik und Informationstechnik

An Event Model for WS-BPEL 2.0

Oliver Kopp, Sebastian Henke,
Dimka Karastoyanova, Rania Khalaf,
Frank Leymann, Mirko Sonntag,
Thomas Steinmetz, Tobias Unger,
Branimir Wetzstein

Report 2011/07

Institut für Architektur von Anwendungssystemen

Universitätsstraße 38
70569 Stuttgart
Germany

CR: D.2.6, H.4.1

Contents

1 Introduction **5**
1.1 Changes from the BPEL4WS 1.1 Event Model 6
1.2 Overview 7

2 Syntactical Notes **9**

3 Process State Model **11**
3.1 States 11
3.2 Outgoing Events 11
3.3 Incoming Events 12
3.4 Changes from the Previous Version 12
3.5 Missing Features 12

4 Process Instance State Model **13**
4.1 States 14
4.2 Outgoing Events 14
4.3 Incoming Events 15
4.4 Changes from the Previous Version 15
4.5 Missing Features 16

5 Activity State Model **17**
5.1 States 18
5.2 Outgoing Events 18
5.3 Incoming Events 19
5.4 Changes from the Previous Version 20
5.5 Missing Features 20

6 Scope State Model **23**
6.1 States 25
6.2 Outgoing Events 25
6.3 Incoming Events 26
6.4 Changes from the Previous Version 26
6.5 Missing Features 27

7 The Invoke Activity **29**
7.1 States 31
7.2 Outgoing Events 31

7.3 Incoming Events 31
7.4 Changes from the Previous Version 32
7.5 Missing Features 32

8 Loop State Model **33**
8.1 States 34
8.2 Outgoing Events 34
8.3 Incoming Events 34
8.4 Note on Parallel forEach Activities 35
8.5 Changes from the Previous Version 35
8.6 Missing Features 36

9 Link State Model **37**
9.1 States 37
9.2 Outgoing Events 38
9.3 Incoming Events 38
9.4 Changes from the Previous Version 39
9.5 Missing Features 39

10 Variable State Model **41**
10.1 States 41
10.2 Outgoing Events 43
10.3 Incoming Events 43
10.4 Changes from the Previous Version 44
10.5 Missing Features 44

11 Information Contained in the Events **45**

12 Implementation and Application **47**

13 Monitoring-only Event Model **49**

14 Conclusion **51**

1 Introduction

The Web Service Business Process Execution Language 2.0 (WS-BPEL 2.0 [20], BPEL for short) is one option for workflow execution in the Web service platform, which realizes a SOA [25]. A BPEL process model defines a process by orchestrating Web services and is exposed as a Web service itself. The process model contains different types of entities for defining the process logic such as *partner links* for linking the Web services involved in the service orchestration, *variables* holding process data, *basic activities* for interacting with Web services and data handling, *structured activities* for defining the control-flow of the process and enabling compensation-based recovery. The process model is deployed on a BPEL engine which executes process instances based on the process model.

Process execution results in *state* changes of the BPEL entities involved in the execution. For example, a process instance is first *instantiated*, is then *running* for a while, and at some point will either be *completed* successfully, *faulted*, or *exited* purposely using a BPEL `exit` activity. The transitions between the states are signaled using *events*. The states and the state transitions are part of the *state model* (or alternatively *event model*) of each BPEL entity. To some extent, state models for the different BPEL entities can be derived from the operational semantics of BPEL. State models, however, are not explicitly defined in the BPEL specification and in particular the specification does not prescribe, which events the BPEL engine has to expose to the outside for *monitoring* purposes. Thus, BPEL engine implementations differ in the *event models* they support.

Events are typically saved in the audit trail or are published on a messaging infrastructure used for *passive* monitoring of process execution. In addition to passive monitoring, it is often desirable to enable external applications to *actively* influence the process execution. That includes, for example, *suspending* and *resuming* the execution of a process instance or *adapting* the process logic of a process instance by skipping some of the process activities. In both cases, an external application (rather than the process engine) triggers state transitions of BPEL entities.

Khalaf et al. [11] present an architecture for an event-based modification of the runtime of a BPEL process. *Outgoing messages* are sent from the BPEL engine to a generic controller signalling state changes in the process. Custom controllers can register at the generic controller to receive outgoing events. To be able to react on events and influence the process execution the state model defines *incoming events*, which control the execution: The execution of the process instance is blocked until explicitly being unblocked by a message (*incoming event*) sent by a custom controller. Such an incoming

event *unblocks* the process instance and *triggers* a state transition from the outside. For instance, a custom controller receiving the event that an activity is **Ready** for execution can send an appropriate incoming event back to the generic controller instructing it to skip that activity. This "adaption action" results in a state transition from **Ready** to **Completing** and a corresponding outgoing event Activity_Executed. To avoid contradicting incoming events, for each incoming event at each activity instance, only one custom controller is allowed to send incoming events. We call this controller *blocking custom controller*.

This report presents an *engine-independent WS-BPEL 2.0 event model*. It supports both *passive monitoring* and *active control* of process execution by external applications. Some of the assumptions in the presented event model are inspired by a particular implementation, e. g. fault handling and compensation; however they are kept as general as possible, so that they can be mapped on other engine-specific approaches to tackle faults and support compensation. In addition, the report draws on the experience of some of the authors in business process management and software development. The overall BPEL event model consists of a set of event models for the different types of BPEL entities that change their states: processes, process instances, general activities, scope activities, invoke activities, loops, links, variables, partner links, and correlation sets. The event model is used by the authors of the report in several projects, all utilizing process life cycle events in different scenarios.

1.1 Changes from the BPEL4WS 1.1 Event Model

An event-model for BPEL4WS 1.1 has been presented by Karastoyanova et al. [8]. WS-BPEL 2.0 is the successor of BPEL4WS 1.1 and introduced new activities. We adapted the event-model by Karastoyanova et al. to reflect these changes and also honored the experiences we made by applying the BPEL4WS 1.1 event-model in cross-partner transaction transactions [18] and cross-process monitoring [27].

The former event model introduced *blocking* events: A state transition was not allowed to be done unless another event produced by an application external to the BPEL processor arrived. In case there were two options (such as skipping an activity or letting the activity execute), an additional state has been introduced. In the current version of the event model, additional states have been introduced, even if the subscriber only has one choice.

The state charts were modified such that a subscriber to the events does not need to know whether there is a blocking subscriber. For instance, the state **Executed** (formerly called **Waiting**) is always passed after the activity has finished execution. A detailed list of changes is provided at the description of each state chart.

1.2 Overview

In the following sections, we present the state models. We begin with a short note on the UML syntax used in Section 2. Process states are presented in Section 3. Activity states are presented in Section 5. Scope states are presented in Section 6. Invoke states are presented in Section 7. Loop states are presented in Section 8. Link states are presented in Section 9. Variable states are presented in Section 10. Minimum information to be contained in the events is presented in Section 11. A reference to an implementation and references to projects building on the presented eventing framework is provided in Section 12. The presented event model is geared towards adaptation. We discuss in Section 13 the modifications necessary for a monitoring-only event model. Finally, Section 14 concludes the report.

2 Syntactical Notes

Each event model is presented in an UML state diagram [19] describing the possible states and state transitions. For each state transition we define (i) the activating trigger, which may be an engine-internal behavior or an external incoming event, and (ii) the outgoing event fired as a result of a state transition. An incoming event written in **bold** denotes the *default* transition. That means, if no custom controller is subscribed as sender of this event, this condition of the transition is immediately treated as true.

The presentation of each state model follows following pattern: First, the *UML state diagram* is given. It is followed by a short description of the contained *states*, the *outgoing events*, and the *incoming events*. Subsequently, the *changes from the previous BPEL 1.1 event model* are described. Each presented state model is concluded by a discussion on *missing features*.

3 Process State Model

A process model has to be deployed on a process engine to enable execution of it. After a BPEL process model is deployed, the process model may be instantiated after the receipt of a message. Usually, there are multiple instances for a process model.

Figure 3.1 presents the event model for the life cycle of a process model.

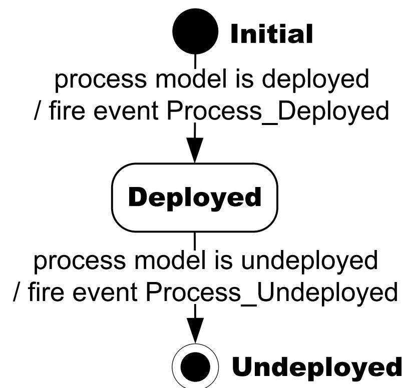


Figure 3.1: Process Model State Chart

3.1 States

Undeployed The process model is not deployed on the engine any more.

Deployed The process model is deployed on the engine.

Initial This is the initial state of the process model. It is not deployed on an engine.

3.2 Outgoing Events

Process_Deployed This event is fired whenever a new BPEL-process is deployed into a BPEL engine. This event is fired once for each process model. The other process events are fired for each process instance.

Process_Undeployed This event is fired when a process model is undeployed.

3.3 Incoming Events

There are no incoming events defined.

3.4 Changes from the Previous Version

- The former event model did not distinguish between process model and process instance.
- The event Process_Undeployed has been introduced.
- The state **Undeployed** has been introduced.

3.5 Missing Features

It is not possible to suspend a whole process model using one message. That means, it is impossible to suspend all instances of the process model with one message.

Re-deployment of a process model is not possible. This is a discussion of handling different versions of a process model: Are two process models with the same URI the same process model? We decided to treat them differently. Furthermore, re-deployment of the same process model also leads to proceed from state **Initial**.

4 Process Instance State Model

Figure 4.1 presents the event model for the life cycle of an instance of a process model.

The implicit scope generated by the `process` activity is treated as `scope` activity and not by the process instance model. The `scope` activity is described in Section 6.

The event model allows for rerunning the process itself. Before triggering a rerun, the states of the activities have to be changed as desired.

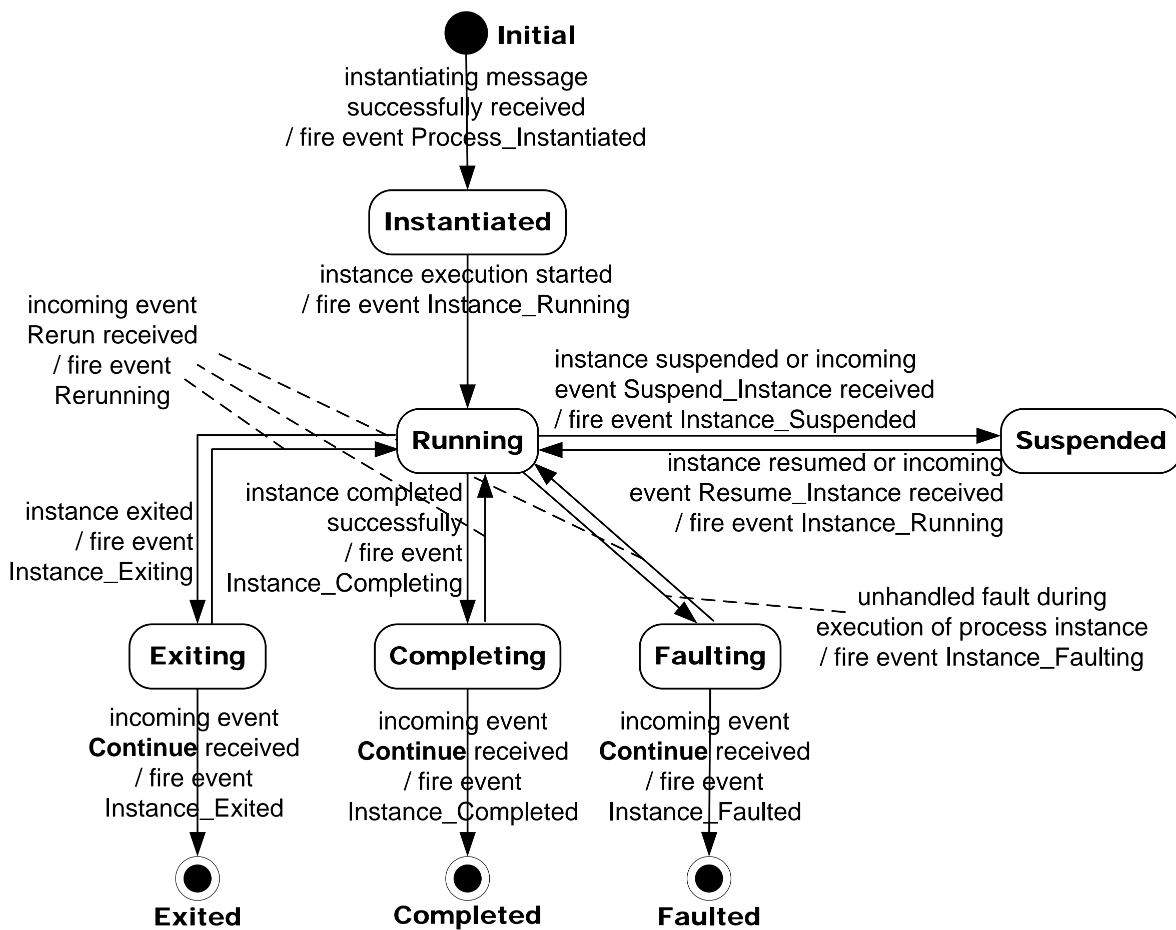


Figure 4.1: Process Instance State Chart

4.1 States

Completed The process instance completed successfully.

Completing The process instance completed. An external subscriber may trigger a rerun of the instance.

Exited The process instance exited successfully.

Exiting The process instance exited. An external subscriber may trigger a rerun of the instance.

Faulted The process instance is faulted: A fault reached the boundary of a `process` activity.

Faulting A fault reached the boundary of a `process` activity. An external subscriber may trigger a rerun of the instance.

Initial The process model is deployed and ready for instantiation.

Instantiated The process model is instantiated.

Running The process instance is running.

Suspended The process instance is suspended. Note that the BPEL specification itself does not make any statements about suspending and resuming a process instance.

4.2 Outgoing Events

Instance_Completing This event is fired, when a process may finish successfully. That means, the process is not terminated by a fault or by an `exit` activity.

Instance_Completed This event is fired, when the process instance is in the state **Completing** and the event **Continue** is received. In case the state **Completing** is non-blocking, the event is fired automatically.

Instance_Exiting This event is fired, when a process instance is terminated through execution of an `exit` activity.

Instance_Exited This event is fired, when the process instance is in the state **Exiting** and the event **Continue** is received. In case the state **Exiting** is non-blocking, the event is fired automatically.

Instance_Faulting This event is fired, when a process should be terminated by a fault that was not handled and propagated to the implicit fault-handler of the root-scope of the process.

Instance_Faulted This event is fired, when the process instance is in the state **Faulting** and the event **Continue** is received. In case the state **Faulting** is non-blocking, the event is fired automatically.

Instance_Running This event is fired, when a process instance starts running after being instantiated or resumed.

Instance_Suspended This event is fired when a process instance is suspended, e. g. a breakpoint is reached or an external request for suspension is received by the engine.

Process_Instantiated The event is fired, when a process is instantiated. This happens if an instance-creating extension activity instantiates a process or if a message is a message is received by a `pick` or `receive` activity with the attribute `createInstance=yes`.

4.3 Incoming Events

Continue This events lets the process instance reach a final state.

Rerun This event triggers a (potentially partial) rerun of the process instance. The concrete rerun effects are dependent on the states of the activities. The states of the activities may be changed during the states **Completing**, **Exiting**, **Faulting**, and **Suspended**.

Resume_Instance This events leads the process instance to resume.

Suspend_Instance This event triggers a suspension of the process instance.

4.4 Changes from the Previous Version

- The event Instance_Terminated has been renamed to Instance_Exited. Accordingly, the state **Terminated** has been renamed to **Exited**. WS-BPEL 2.0 introduced the new concept of termination handling, with a different meaning than Instance_Terminated implied.
- New states **Completing**, **Exiting**, and **Faulting** added. The corresponding events Instance_Completing, Instance_Exiting, Instance_Faulting, Continue, and Rerun. This allows the process modeler to append process logic even if the last activity of the process was executed or to take other corrective actions.

4.5 Missing Features

It is not possible to suspend a whole process *model* using a single message. That means, it is impossible to suspend all instances of the process model with one message.

The event model does not allow for compensating a complete process instance as required by the BPEL sub-process extension [4], but disallowed in the current WS-BPEL 2.0 specification.

There is no possibility to reach a final state from the **Suspended** state. If the process instance should be killed during suspension, it has to be resumed and then killed (via the **Exiting** state).

An explicit triggering of a process exit is not possible.

5 Activity State Model

The state chart presented in Figure 5.1 presents the general state chart for all BPEL activities. The state charts of invokes, scopes, and loops contain additional states and are presented in Section 7, Section 6, and Section 8.

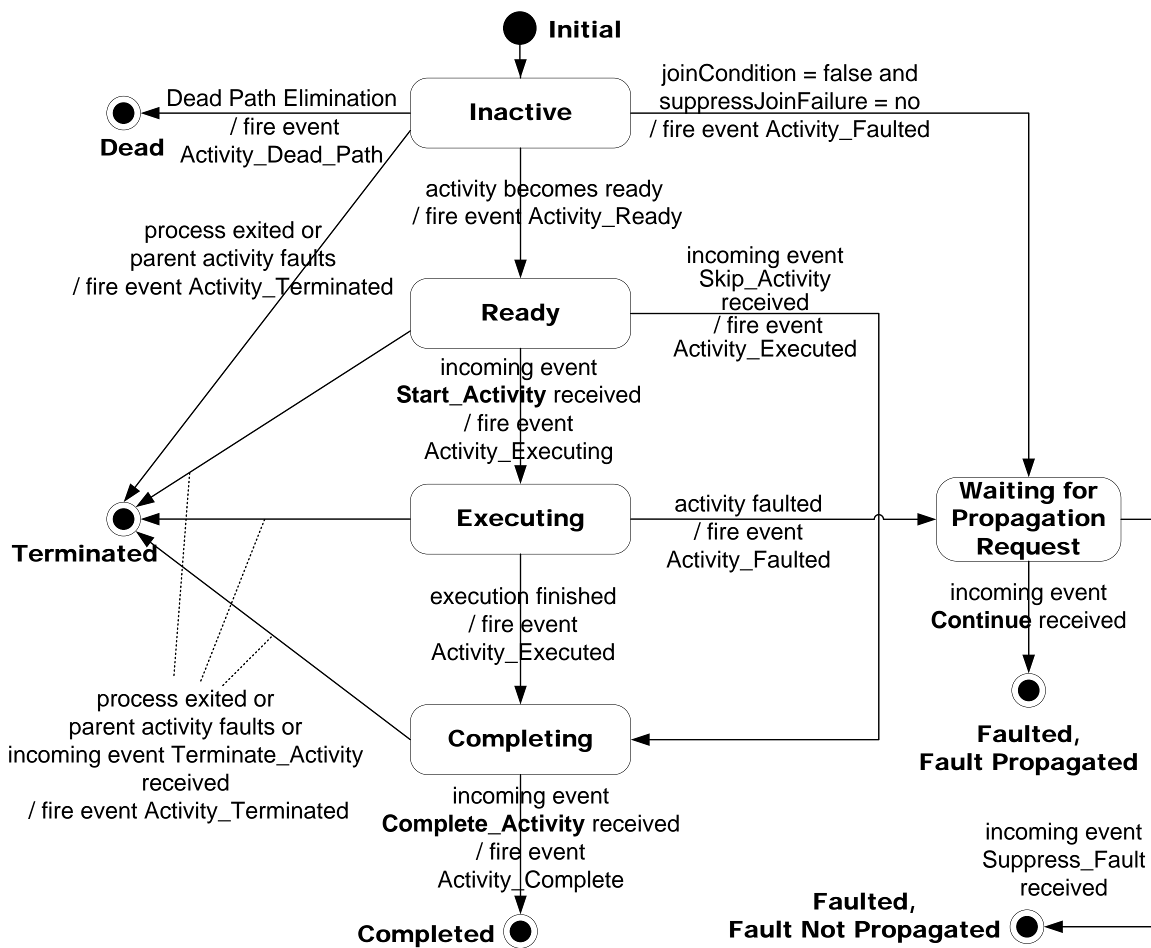


Figure 5.1: Activity State Chart

5.1 States

Completing The activity completed execution, but waits for an external event to fully complete.

Completed The activity completed execution and does not need to wait for an external event. In this state, the engine starts to handle the outgoing links of the activity.

Dead The activity was marked dead by the engine's dead-path-elimination (DPE) mechanism.

Executing The activity executes.

Faulted, Fault Propagated The activity faulted and the fault was propagated to the enclosing scope.

Faulted, Fault Not Propagated The activity faulted and the fault was not propagated to the enclosing scope due to an external event.

Inactive The activity is not executing, but waiting for becoming ready for execution.

Initial The activity's initial state. The transition from **Initial** to **Inactive** is dependent on the concrete engine implementation. For instance, the children of a `sequence` activity may be inactive as soon as the `sequence` activity is inactive or the preceding activity in the `sequence` activity has completed.

Ready The activity is ready for execution.

Terminated The activity is terminated.

Waiting for Propagation Request The activity waits for an external event, which decides whether the fault should be propagated to the enclosing scope or suppressed. The propagation is BPEL's default behavior.

5.2 Outgoing Events

Activity_Completed This event is fired when an activity is in the state **Completing** and the event Complete_Activity is received. In case the state **Completing** is non-blocking, the event is fired automatically.

Activity_Dead_Path This event is fired, when an activity was marked dead by the engine's dead-path-elimination (DPE) mechanism implementation. This is the case if the `joinCondition` evaluates to false and `suppressJoinFailure` is set to `yes`.

Activity_Executing This event is fired, when an activity starts its execution (for activities like pick, receive and onMessage this event is fired, when starting to wait for incoming messages, not when the actual message is received). For instance, in the case of a `receive` activity, this event is fired as soon as the `receive` activity waits for an incoming message. In other words, this event is triggered before the actual message is received.

Activity_Executed This event is fired, when the execution of an activity is finished. The engine changes to the state **Completing**.

Activity_Faulted This event is fired when an activity is skipped or aborted because of a fault that occurred within the activity. It can also be fired if a fault of its child scopes was not handled.

Activity_Ready This event is fired, when an activity becomes ready to execute. This is the case as soon as the activity is inactive, all incoming links are evaluated, and the join-condition is true. In case of a sequence activity, the activity before the current activity has to be completed additionally.

Activity_Terminated This event is fired, when an activity is terminated. It can also be fired if a fault in a preceding activity of the same scope is not handled.

5.3 Incoming Events

Complete_Activity This event unblocks the state **Completing** and leads to a completion of the activity.

Continue This event tells the engine to propagate the fault to the enclosing scope.

Skip_Activity This event may be sent when the activity is in the state **Ready**. The execution of the activity is skipped and the state immediately changes to **Completing**.

Start_Activity This event may be sent when the activity is in the state **Ready** and leads to a normal activity execution.

Suppress_Fault The fault is discarded and not propagated to the enclosing scope.

Terminate_Activity The execution of an activity may be terminated by this event.

In case an activity is not executed, because its `joinCondition` evaluated to `false`, there are two possibilities: (i) `suppressJoinFailure=yes`. Then, the event Activity_Dead_Path is emitted. (ii) `suppressJoinFailure=no`. Then, the event Activity_Faulted is emitted. Besides that, Activity_Faulted is only raised, if a fault occurred at the execution of the activity.

There is no separate state chart for the `exit` activity. As soon as the `exit` activity has finished execution, the whole process is terminated and the exit activity does not reach the state **Completing**.

5.4 Changes from the Previous Version

- State **Waiting** is now called **Executed**.
- State **Dead Path** is now called **Dead**.
- "Parent activity faults or incoming event Terminate_Activity" leads to the state change to **Terminated**. Before, "Process terminated" lead to that state change.
- Renamed "process terminated" to "process exited" in the transitions.
- Renamed event Activity_Complete to Activity_Completed to be consistent with the naming of the final state **Completed**.
- A fault in the process is handled by the transition to **Terminated**. The former model handled that with a transition to **Faulted**.
- In case the join condition of an activity is evaluated to false and $suppressJoinFailure$ is set to `no`, the activity throws a fault. This is reflected in the transition from **Inactive** to **Faulted**.
- Activity_Terminated (instead of Activity_Faulted is sent if a fault of a preceding activity has occurred and was not handled by that activity.
- New incoming events Skip_Activity, Suppress_Fault and Terminate_Activity.
- New state **Waiting for Propagation**. The activity waits for the custom controller to decide whether the activity should propagate or suppress the fault.
- New transition from **Ready** to **Completing**. This transition has been introduced by Schroth [21] to allow skipping activities.

5.5 Missing Features

The event model neither allows for suspending a single activity nor signals such a suspension. Such a partial suspension is motivated by processes offering parts of themselves as subprocesses [16]. When adding such a feature, the state chart has to distinguish between long-running and short-running activities: long-running activities

may be suspended at the states **Ready**, **Executing**, and **Completing**, whereas short-running activities may suspended during the states **Ready** and **Completing**. This is similar to the requirements by the BPEL specification, where short-running activities are not terminated if a fault occurs.

A distinction between long-running and short-running activities also refines the transition to **Terminated**. In the case of short-running activities, the condition on the transition from **Executing** to **Terminated** changes: only “process exited” would be the condition.

The join condition itself and the evaluation result cannot be modified. Skipping an activity is only possible by using the incoming event Skip_Activity. The states of links can be modified by as described in Section 9.

6 Scope State Model

Figure 6.1 presents the state chart for a scope activity. It is a supergraph of the state model of activities. In the following, we describe the additional transitions and states.

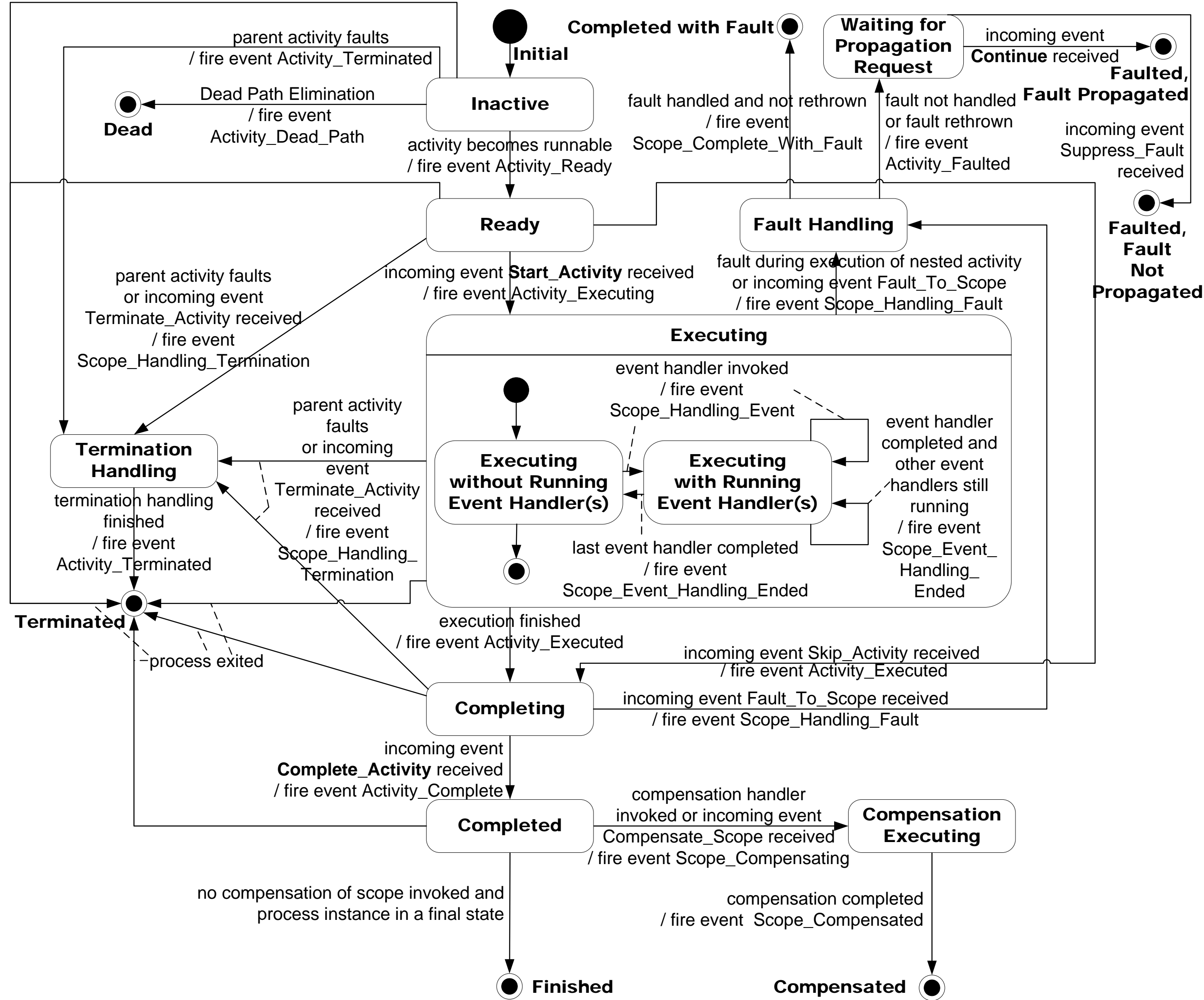


Figure 6.1: Scope State Chart

6.1 States

In the following, the states additional to activity's states are described:

Compensated The scope has been successfully compensated.

Compensation Executing The scope compensates.

Completed with Fault A fault has been handled in a fault handler of scope and has not been rethrown.

Executing with Running Event Handler(s) As soon as an event handler is triggered, the scope executes this event handler *in parallel* to its child activity.

Executing without Running Event Handler(s) The scope executes and has no event handlers running.

Fault Handling The scope handles an encountered fault.

Finished The scope may not be compensated any more. This state is reached if the scope activity has completed and the process instance reached a final state.

Termination Handling The scope handles its termination.

6.2 Outgoing Events

In the following, the outgoing events additional to activity's outgoing events are described:

Scope_Compensating This event is fired, when a completed scope's compensation handler is invoked by the engine or if the incoming event Compensate_Scope is notified.

Scope_Compensated This event is fired as soon as the compensation handler of a scope is finished.

Scope_Complete_With_Fault This event is fired, when a scope completes (see Activity_-Complete) after handling a fault that was not rethrown in the scope's fault handler. In case the fault is rethrown, the event Activity_Faulted is fired.

Scope_Event_Handling_Ended This event is fired, when a scope's event handler finished executing.

Scope_Handling_Event This event is fired, when a scope's event handler (onAlarm or onMessage) starts executing (incoming message received or alarm occurred/fired).

Scope_Handling_Fault This event is fired, when a scope's fault handler is invoked. The fault handler may be an implicit or explicit fault handler.

Scope_Handling_Termination This event is fired upon activation of the scope's termination handler.

Upon a process exit, the state **Terminated** is reached without invoking a termination handler and firing an event from the activity. The subscribers are notified by the event Instance_Terminated fired for the process instance.

There is no event Scope_Termination_Handling_Ended as the Activity_Terminated denotes the same.

Event handling is modeled in the nested states **Executing without Running Event Handler(s)** and **Executing with Running Event Handler(s)**. A scope executes its child activity and processes incoming events in parallel.

6.3 Incoming Events

In the following, the incoming events additional to activity's incoming events are described:

Compensate_Scope This event leads to a compensation of a completed scope.

Fault_To_Scope This event triggers fault handling of the scope. The fault handling is started as if the fault was raised by an activity nested in the scope.

6.4 Changes from the Previous Version

- WS-BPEL 2.0's termination handlers are supported by state **Termination Handling**. The additional event Scope_Handling_Termination signals that the scope started with termination handling.
- Event Compensated renamed to Scope_Compensated.
- **EventHandling** is not a separate any more. A scope activity continues executing its child while handling events. This is now reflected in the nested state **Executing with Running Event Handlers**.
- For symmetry reasons to the states of activities, **Complete** has been renamed to **Completed**.
- The state **FaultHandling_NoHandler** has been removed. This state has been introduced to support fragmented scopes [10]. A semantic equivalent behavior can be achieved when skipping the activity nested in a fault handler by the subscriber.

- Removed the incoming event Compensated. The same behavior can be achieved when sending Compensate_Scope and skipping the activity nested in the compensation handler.
- Renamed **CompletedWithFault** to **Completed with Fault**.

6.5 Missing Features

The event model does not allow to do a more generic state change of a scope. For instance, going from termination back to running is impossible. The same applies for compensation.

There is no possibility to block the execution of the termination handler directly. As the concrete implementation of a termination handler is done via an activity, a skipping or blocking of the termination handler has to be done by skipping or blocking that activity.

Similar to the case of the termination handler, the execution of a compensation handler has to be blocked by handling the nested activity in the compensation handler.

Faults being thrown out of a compensation handler are not treated.

7 The Invoke Activity

An invoke activity may have explicit fault handlers and an explicit compensation handler assigned. Such an invoke activity is semantically equivalent to a scope activity, where the invoke activity is nested. The invoke activity has no handlers assigned. The handlers are assigned to the scope activity. There are two approaches to deal with invokes having handlers attached: (i) Following the "unfolding" and producing events for the scope (with handlers) and the nested invoke activity. (ii) Treating the invoke activity as scope and not producing additional events for the nested invoke activity. (iii) Add a separate state chart for an invoke activity supporting fault and compensation handling.

We follow the second approach to keep the intention that the invoke activity is still a single activity. The state chart is presented in Figure 7.1.

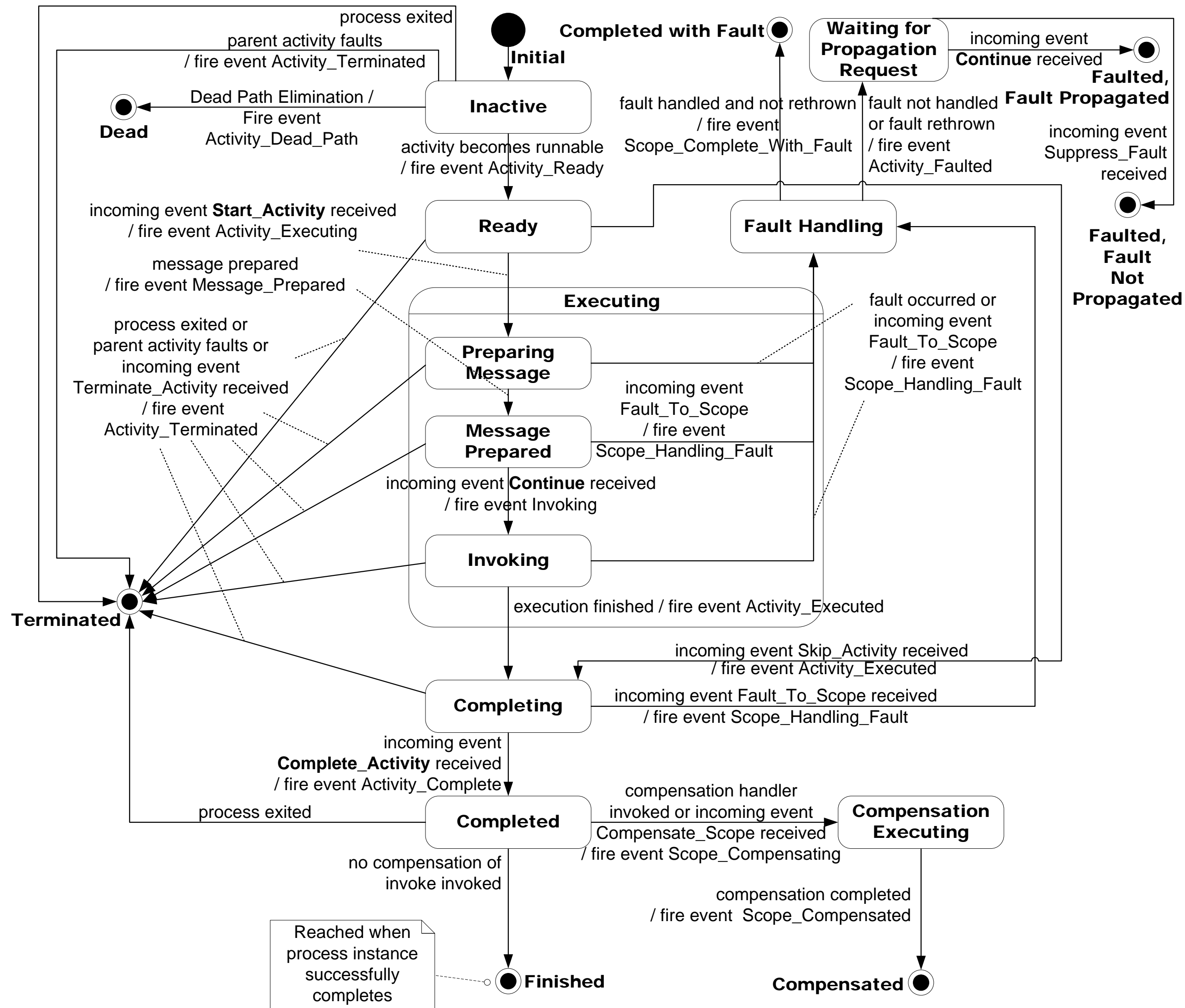


Figure 7.1: Invoke State Chart

As invoke activities may not have termination handlers and event handlers assigned, the corresponding transitions and events have been removed. The state **Executing** has been replaced by the states **Preparing Message**, **Message Prepared**, and **Invoking**. The state **Message Prepared** is important to enable modification of message headers before the actual invocation.

7.1 States

In the following, the states additional to scope's states are described:

Invoking The external service is invoked. In case of a synchronous invoke, the activity remains in this state until the reply message is received.

Message Prepared The message is prepared. Now, it can be inspected and modified externally.

Preparing Message The `invoke` activity prepares the message to be sent.

7.2 Outgoing Events

In the following, the outgoing events additional to scope's outgoing events are described:

Invoking The activity begins with invoking the external service.

Message_Prepared The outgoing message is prepared.

7.3 Incoming Events

In the following, the incoming events additional to scope's incoming events are described:

Continue This event is used to unblock the state **Message Prepared** after required message modifications have been done.

7.4 Changes from the Previous Version

The explicit state model for an invoke activity did not exist in the previous event model.

7.5 Missing Features

The invoke activity state model inherits the missing features of the scope activity (cf. Section 6.5).

8 Loop State Model

The state chart provided in Figure 8.1 presents the general state chart for loops in BPEL processes. Loops are the sequential `forEach` activity, the `repeatUntil` activity, and the `while` activity.

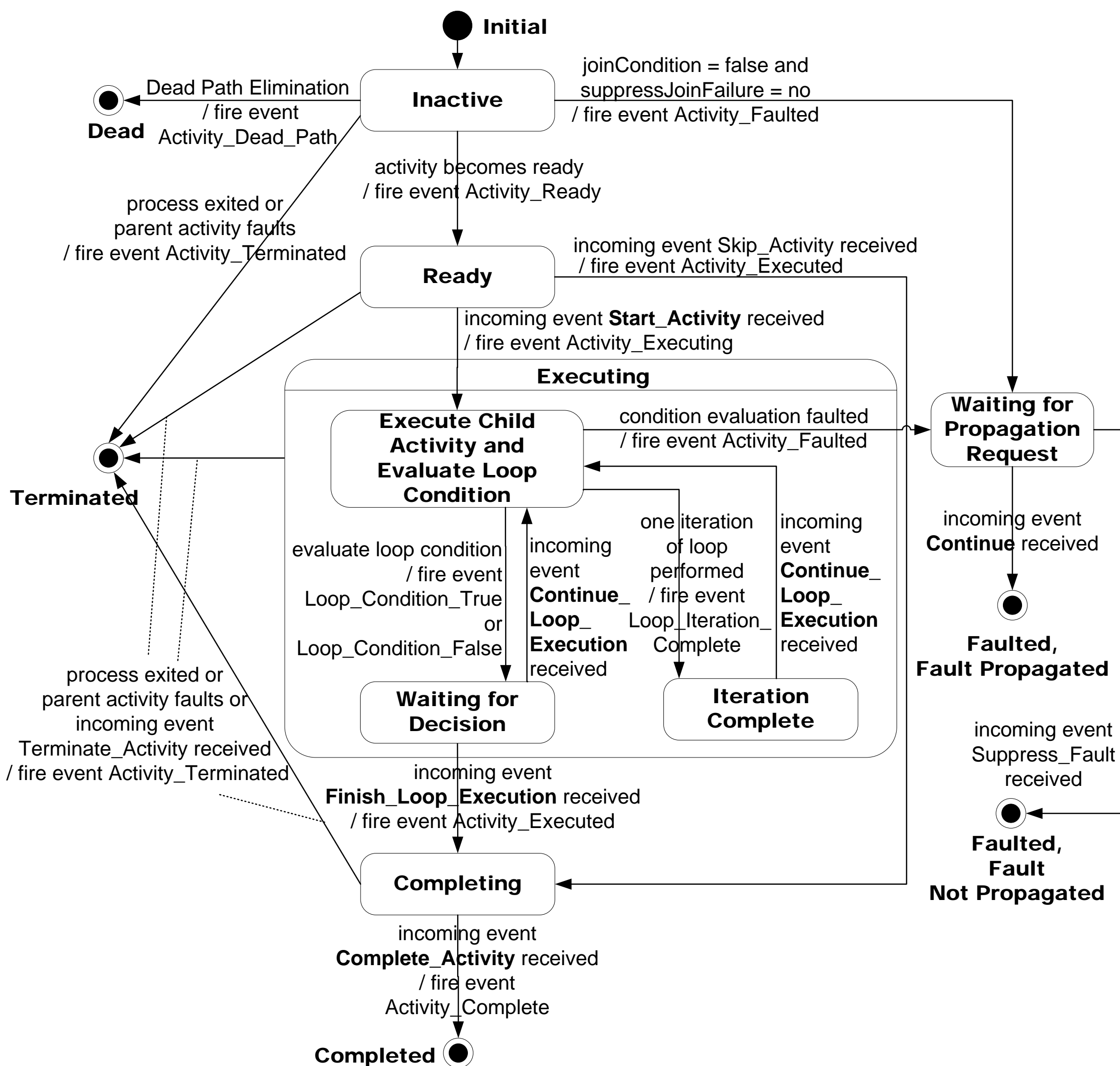


Figure 8.1: Loop State Chart

8.1 States

In the following, the states additional to activity's states are described:

Execute Child Activity and Evaluate Loop Condition The child activity is executed.

Iteration Complete The loop iteration is complete. A blocking custom controller signals the start of the next loop iteration. This state has been introduced to support fragmented loops [10].

Waiting for Decision The activity waits for the custom controller to decide whether the loop should do another run or the loop iteration should finished. In case there is no blocking custom controller subscribed, the result of the evaluation is used directly.

8.2 Outgoing Events

In the following, the outgoing events additional to activity's outgoing events are described:

Loop_Condition_False This event is fired when the loop condition has been evaluated to false. We need this event for fragmented loops to tell other fragments the loop condition, because only one fragment is able to evaluate the loop condition for all.

Loop_Condition_True This event is fired when the loop condition has been evaluated to true. This event is needed for the fragmented loops scenario to tell other fragments the loop condition, because only one fragment is able to evaluate the loop condition for the loop.

Loop_Iteration_Complete This event is fired when an iteration of a loop is complete and (in the case of a `while` activity) before the loop-condition is re-evaluated. This event is especially important for fragmented loops [10], where it is necessary to synchronize the individual fragments.

8.3 Incoming Events

In the following, the incoming events additional to activity's incoming events are described:

Continue_Loop_Execution This event lets the loop continue its execution. For instance, in the case of the `repeatUntil` loop in the transition from **Waiting for Decision**, this leads to an execution of the activity nested in the loop.

Finish_Loop_Execution This event forbids an additional loop run. The loop has finished looping and now is in the state **Completing**.

Similar to `repeatUntil` and `while` activities, the sequential `forEach` activities signal by the events Loop_Condition_False and Loop_Condition_True whether the `forEach` loop is complete. This can be overridden by the events Continue_Loop_Execution and Finish_-Loop_Execution.

8.4 Note on Parallel forEach Activities

The *parallel* `forEach` activity is *not* treated as loop. The parallel `forEach` activity determines the number of children before the child activity starts. Therefore, the events Loop_Condition_True and Loop_Condition_False are unnecessary. The parallel `forEach` activity does not (sequentially) iterate over the children. Therefore, the event Loop_-Iteration_Complete is misleading and is dropped. A completion of one iteration can be observed by the Activity_Complete of a child activity. A The looping condition is evaluated once at the start of the loop. A fault in that case is propagated using the event Activity_Faulted. These facts lead to the state model presented for plain activities in Section 5.

8.5 Changes from the Previous Version

- The transitions from **Executing** to **Waiting** and **Complete** have been removed: A loop may only be left through the state **Waiting for Decision**.
- The external event Continue_Loop has been used in the condition from **Iteration Complete** to **Executing**. This event has been renamed to Continue_Loop_Execution, which in turn is used from **Check Condition** to **Execution**.
- State **Waiting for Decision** has been renamed to **Waiting for Decision** to reflect that the condition has already been evaluated by the engine, but the custom controller has the decision to override the evaluation.

8.6 Missing Features

Currently, there is no way to modify the `startCounterValue`, `finalCounterValue`, and `completionCondition`. Their values are determined at the beginning of **Executing**. Using the current event model, a `forEach` loop may be set as completed or may be continued using the external events Finish_Loop_Execution or Continue_Loop_Execution. A setting of specific start values and final values is not possible. A future version of the event model should introduce an additional state to separate the determination of their values and beginning of execution of the child activity.

Similar to the case of `startcounterValue` etc., it is not possible to change the looping condition itself.

As the repeat until loop and the while loop are represented in one diagram, the state transitions between **Execute Child Activity and Evaluate Loop Condition**, **Iteration Complete**, and **Waiting for Decision** are not exact. In case different state charts for the different loop types were introduced, the transitions would be more precise.

9 Link State Model

The state chart provided in Figure 9.1 presents the general state chart for links in BPEL processes.

9.1 States

Evaluated The value of the link has been determined. This value can be overridden by the external event Set_Link_State. In case there is no blocking custom controller, the outgoing transitions depend on the evaluation result. In case the link evaluated to true, the transition to state **True** is taken, otherwise the transition to state **False** is taken.

False The link status is false.

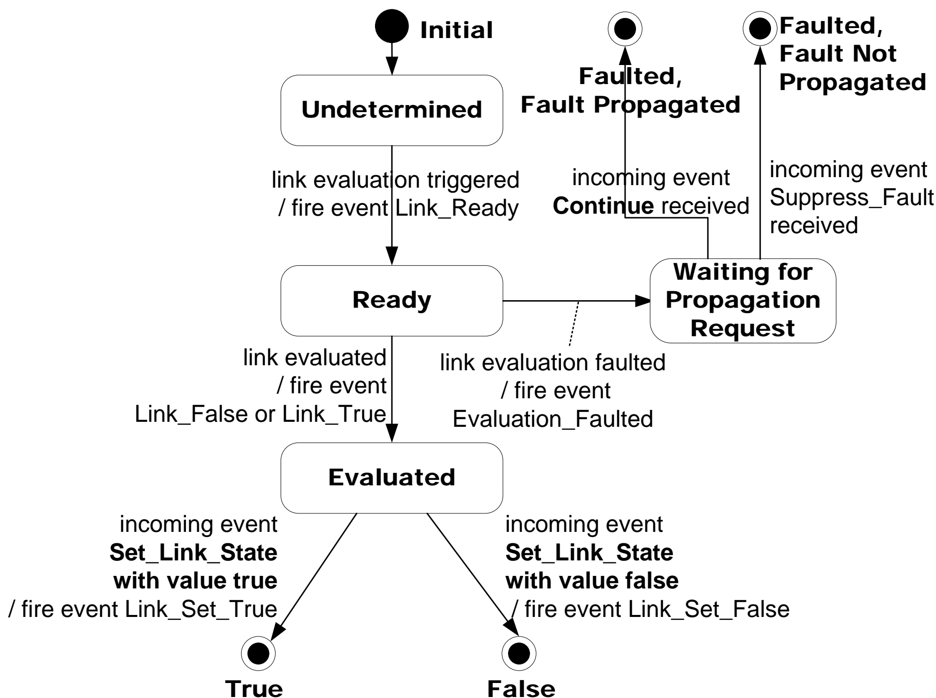


Figure 9.1: Link State Chart

Faulted, Fault Not Propagated Link evaluation faulted and the fault has not been propagated to the enclosing scope.

Faulted, Fault Propagated Link evaluation faulted and the fault has been propagated to the enclosing scope.

Initial The link is not instantiated.

Ready The link is ready for evaluation.

True The link status is true.

Undetermined The status of the link is not determined.

Waiting for Propagation Request In case a fault occurred during evaluation, a custom controller may suppress this fault. The link has to wait for this decision, modeled by this state.

9.2 Outgoing Events

Evaluation_Faulted This event is fired if the evaluation of the link lead to a fault.

Link_False The event is fired when the transition condition on a link has been evaluated to false.

Link_Ready This event is fired when a Link is ready for evaluation, i. e. immediately after its source activity has been completed.

Link_Set_False This event is fired if the link status is set to false.

Link_Set_True This event is fired if the link status is set to true.

Link_True The event is fired when the transition condition on a link has been evaluated to true.

In the case of dead-path-elimination, the sequence Link_Ready, Link_False, Link_Set_False is emitted.

9.3 Incoming Events

Set_Link_State This events indicates that the result of the evaluation should be ignored and the link status should be set to a value contained in this event.

9.4 Changes from the Previous Version

- Events Link_Set_True and Link_Set_False introduced to signal the concrete status of the links.
- The event Link_Evaluated has been split to the events Link_False and Link_True to be consistent to the loop events Loop_Condition_False and Loop_Condition_True.
- New event Evaluation_Faulted with the succeeding states.

9.5 Missing Features

The transition condition of a link cannot be modified externally.

10 Variable State Model

This section describes the states of variables (Figure 10.1), variables holding a message (Figure 10.2), partner links (Figure 10.3), and correlation sets (Figure 10.4). The BPEL specification does not treat message headers. In practice, message headers contain information needed for a proper workflow execution. Therefore, events are added to handle message headers.

10.1 States

Active The variable is available.

Deleted The variable is not available any more.

Initial The variable is not available.

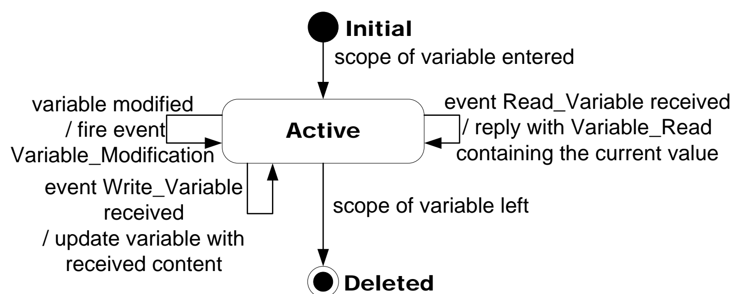


Figure 10.1: Variable State Chart

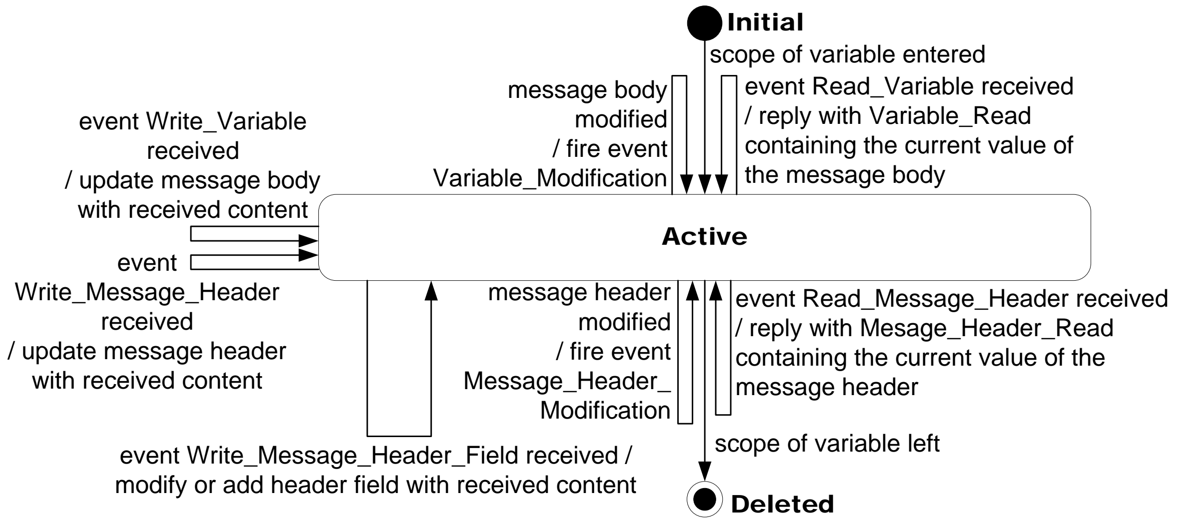


Figure 10.2: Message Variable State Chart

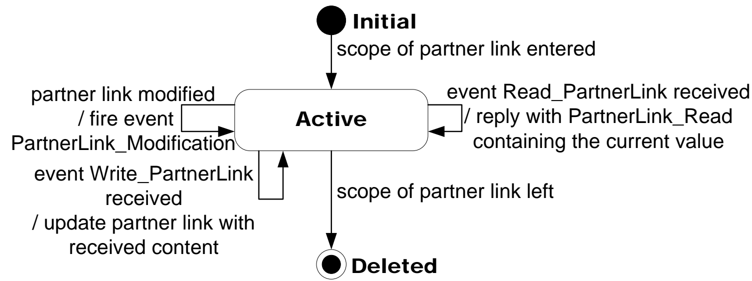


Figure 10.3: Partner Link State Chart

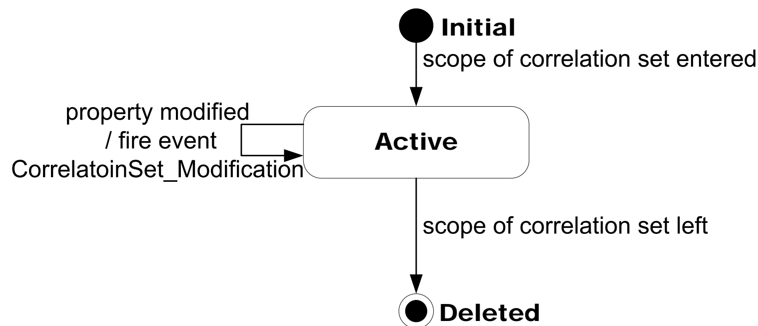


Figure 10.4: Correlation Set State Chart

10.2 Outgoing Events

In the following, the outgoing events additional to activity's outgoing events are described:

CorrelationSet_Modification In case a property of a correlation set changes, this event is fired.

PartnerLink_Modification This event is fired in case a partner link changes.

Variable_Modification This event is fired in case a variable is modified.

10.3 Incoming Events

Read_PartnerLink Using this event, the concrete value of a partner link can be read. As reply to this event, the event PartnerLink_Read is sent. This event is not distributed to all subscribers, but directly sent to the requester.

Read_Variable Using this event, the concrete value of a variable can be read. As reply to this event, the event Variable_Read containing the content of the variable is sent. In case the variable is a message variable, the content of the message body is sent. This event is not distributed to all subscribers, but directly sent to the requester.

Write_PartnerLink Using this event, a subscriber may modify the value of a partner link. Upon completion of the write, the event PartnerLink_Modification is generated.

Write_Message_Header Using this event, a subscriber may modify the value of the complete header of a message variable. Upon completion of the write, the event Message_Header_Modification is generated.

Write_Message_Header_Field Using this event, a subscriber may modify the value of an existing header field. In case the addressed header field does not exist, a new header field is added. Upon completion of the write, the event Message_Header_Modification is generated.

Write_Variable Using this event, a subscriber may modify the value of a variable. In case the variable is a message variable, the message body is updated. Upon completion of the write, the event Variable_Modification is generated.

In case a correlation set has to be modified, the respective variables have to be modified using Write_Variable.

An alternative solution to update message variables is to introduce separate events for the message body and let the variable events tackle the whole variable including header

and body. We do not follow this approach as the BPEL specification treats the message body as variable and lets the engines internally handle the message header.

10.4 Changes from the Previous Version

Events for variables did not exist in the previous event model.

10.5 Missing Features

If the scope of the variable is left, it is possible that the variable is still available for reading operations. This is not reflected in the model.

11 Information Contained in the Events

An event has to contain information to enable assignment to the corresponding process instance, activity instance, link instance, or variable instance.

In the following, we discuss the minimal set of IDs in case the events arrive in order as created by the engine (event stream semantics) and the whole event stream is available. For instance, this is the case if the BPEL engine directly stores the events in an audit trail DB. If, however, events are sent to a pub/sub infrastructure, in general it is not guaranteed that they will arrive in order to a subscriber (e. g., an event correlation engine). In that case, either an event reordering has to take place first or the event correlation statements cannot assume event stream semantics (but event cloud semantics instead) and can get more complicated. For the latter case, we discuss which additional information has to be added to the events by the engine to simplify event correlation.

To uniquely identify a *process model*, following information is required:

- QName of the process model.
- version number of the process model.

The version number is optional: Certain engines allow for handling different versions of a process model with the same QName. Apache ODE is one example.

To uniquely identify a *process instance*, following information is required:

- globally unique instance ID of the process instance.

Usually, the unique instance ID is generated by the BPEL engine. This instance ID should be unique across BPEL engine installations to enable custom controllers to be subscribed to different BPEL engines. The instance ID has to be sent at least once together with the unique ID of the corresponding process model. Typically, this happens as part of the event Process_Instantiated, which is the first event triggered for a new process instance. In practice, both IDs can be sent at each event.

To uniquely identify an *activity instance*, following information is required:

- an XPath expression, which uniquely identifies the construct in the process model.
- globally unique instance ID of the process instance.
- unique instance ID of the innermost scope instance, where the activity is nested in

- unique instance ID of the activity instance

An activity can be uniquely identified in the process model by using an XPath expression. In case the activity is nested in loops or event handlers, several instances my exist in a single process instance.

In case the activity is nested in a `repeatUntil` loop, a `while` loop, or in a sequential `forEach` activity, activity instances (of the same activity in the model) are executed once per iteration in a sequential manner. As a consequence, they can be distinguished and ordered based on the event creation timestamps. Alternatively, a loop iteration number can be added to the events. Consequently, the event Activity_Executed for loop activities should contain the overall number of iterations performed.

In case an activity is nested in a parallel `forEach` construct or an event handler, it is also always nested in a scope as this is required by the BPEL specification. To distinguish *parallel* instances of such a scope, each scope instance has to be assigned an instance ID. Events for activities nested in such a scope have to include the instance ID of the (innermost) scope. This enables assignment of an activity instance to the corresponding scope instance. In case such a nested activity is again a scope (from which several scope instances can run in parallel), in addition its own activity instance ID has to be included into the event.

As we do not want to distinguish between different types of activities and scenarios (e. g., whether the model contains parallel for each statements or not), we simply assign all of the above four identifiers to each activity instance.

To uniquely identify *links*, *variables*, *partner links*, *correlation sets*, following information is required:

- unique instance ID of the process instance.
- unique instance ID of the scope where the element is declared.
- an XPath expression uniquely identifying the construct in the process model.

Each outgoing event has to contain a unique event ID to be able to correlate incoming events driving the element state.

Incoming events not related to a preceding outgoing event have to contain information to enable identification of elements (activities, variables, partner links) where the corresponding action should be taken. This information is the same as stated above for the case of outgoing events.

12 Implementation and Application

The pluggable framework for extended BPEL behavior [11] shows an architecture where the presented event model can be used. This concept has been implemented in the Apache ODE engine v1.3.4 and is available at `http://www.iaas.uni-stuttgart.de/forschung/projects/ODE-PGF/`. A detailed description of the implementation is provided by Steinmetz [24].

In the following, we present a short overview on projects using the eventing framework. Details can be found in the provided references.

Karastoyanova et al. use the eventing framework to add flexibility to workflows. Concrete service implementations can be bound during the runtime of the workflow within the workflow enginge [6, 7].

Karastoyanova and Leymann [5] use the framework to enable aspect-oriented programming with BPEL. Sonntag and Karastoyanova [23] extended this approach to support compensation of aspects.

Sonntag and Karastoyanova [22] use the framework for enabling the explorative development of scientific workflows, an approach called "Model-As-You-Go". A process instance is kept alive if it runs out of work. Further process logic can be appended and the instance resumed. Completion of process instances is done explicitly by the scientists themselves.

Khalaf [13] uses the framework to coordinate split loops and scopes. Parts of the split loops and scopes are called fragments. The idea is presented by Khalaf and Leymann [9]. Concrete coordination protocols are presented by Khalaf and Leymann [10]. Khalaf et al. [12] show how data dependencies across fragments can be kept.

Kopp et al. [14] discuss the interplay of an external transaction and the transaction of a BPEL scope. Henke [3] uses the eventing framework to coordinate such an interplay.

Kopp and Leymann present the idea of a choreography sphere [15]. This sphere spans multiple BPEL processes and introduces a fault and compensation dependency. Kopp et al. [18] extend the concept to support scientific simulations. An implementation of the concept based on the eventing framework is presented by Bors [2] and Steinmetz [24]. Bischof [1] extended the concept of a choreography sphere to interaction choreographies and presents an implementation using the eventing framework.

Kopp et al. [17] present an approach to monitor and prevent violations of choreography models on the enterprise service bus. The implementation is based on Apache Service Mix, Apache ODE, and the presented eventing framework.

Wetzstein et al. [26] use the eventing framework to measure performance of a BPEL process. Wetzstein et al. [27] extend this approach to monitor the execution and performance of choreographies.

Zaid et al. [28] use the eventing framework to monitor the QoS of the process and to skip or replace activities upon a QoS violation.

13 Monitoring-only Event Model

There are scenarios requiring passive monitoring only. In this case, an engine may offer a simplified event model by removing states and state transitions, which have been introduced into the model for supporting active control. This simplified event model can be created as follows: (i) all incoming events are removed from state transition triggers; (ii) if the removed incoming event was the only trigger of the state transition, then that state transition is removed and the target state also can be removed if it now has no incoming transitions any more; (iii) if the removed incoming event denoted a default transition (see Section 2) then effectively this transition does not have a trigger anymore and the two states A → B with such a default transition can be merged as follows: A is removed together with its outgoing transitions (they are in our case either a subset of outgoing transitions of B or B is an end state and thus they are not needed any more); incoming transitions of A become incoming transitions of B whereby the fired events on those incoming transitions are adapted to reflect that now B is reached.

Consider as an example the activity state model presented in Section 5. The transition from **Ready** to **Executed** can be removed as it is only triggered via an incoming event. The transition from **Ready** to **Executing** is a default transition. Thus, **Ready** can be removed and a transition is added from **Inactive** to **Executing** whereby the event Activity_Executing is fired. In the same manner the states **Completing** and **Waiting for Propagation Request** can be removed. An engine implementing the overall event model can expose the simplified event model to subscribers by implementing a corresponding event filter. Figure 13.1 presents the result.

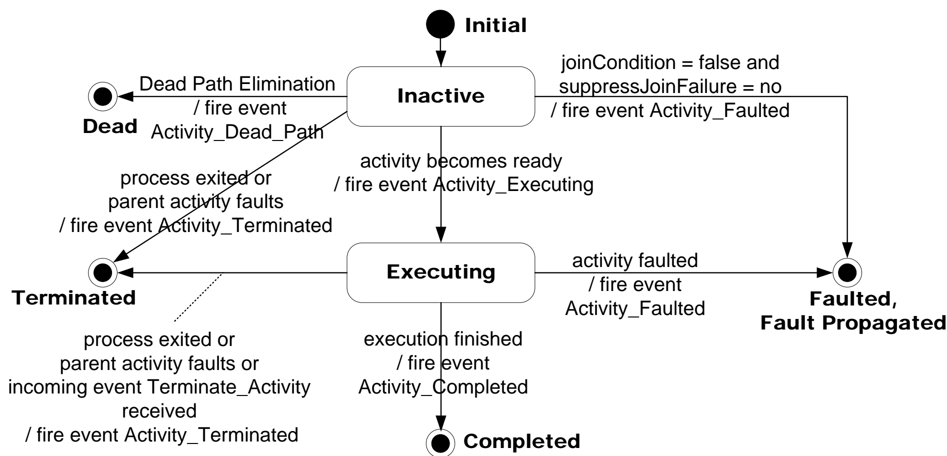


Figure 13.1: Activity Monitoring-Only State Chart

Both event models cannot be used at the same time for a process instance (by different clients), because a blocking subscriber may force state transitions, which are not valid in the simplified event model (e. g., skipping an activity results in Activity_Executed without prior Activity_Executing).

14 Conclusion

This report has presented a generic event model for WS-BPEL 2.0. The event model can be used to monitor running processes. Additionally, the event model offers clients to change the behavior of the process according to their needs. The event model has been implemented using the open source Apache ODE engine.

The presented event model supports both *passive monitoring* and *active control* of process execution by external applications via incoming events. We discussed in Section 13 a way to derive a monitoring-only event model out of the presented event model. The drawback of such a monitoring-only event model is that it cannot be used together with the presented event model as the presented event model allows for skipping events being mandatory in the monitoring-only event model.

Acknowledgments

This report is based on the results of the diploma thesis no. 2729 by Thomas Steinmetz [24], which in turn extended the work by Karastoyanova et al. [8]. The authors thank Tobias Anstett and Daniel Schleicher for their valuable feedback on an earlier version of this report.

The research leading to these results has received funding from the European Community's 7th Framework Programme under the Network of Excellence S-Cube[1] (Grant Agreement no. 215483) and the ALLOW project[2] (contract no. FP7-213339).

The authors M. S. and D. K. would like to thank the German Research Foundation (DFG) for financial support of the project within the Cluster of Excellence in Simulation Technology (EXC 310/1) at the University of Stuttgart.

[1] http://www.s-cube-network.eu/
[2] http://www.allow-project.eu/

Bibliography

[1] Marc Bischof. Modeling and Runtime Support of Faults in Interaction Choreography Models. Diploma thesis 2885, University of Stuttgart, Faculty of Computer Science, Electrical Engineering, and Information Technology, Germany, July 2009.

[2] Sergej Bors. A Runtime for BPEL4Chor Cross-Partner-Scopes. Diploma thesis 2290, University of Stuttgart, Faculty of Computer Science, Electrical Engineering, and Information Technology, Germany, 2010.

[3] Sebastian Henke. Unterstützung für externe Transaktionen in Apache ODE. Diploma thesis 3006, University of Stuttgart, Faculty of Computer Science, Electrical Engineering, and Information Technology, Germany, July 2010.

[4] *WS-BPEL Extension for Sub-processes – BPEL-SPE*. IBM, SAP, 2005.

[5] Dimka Karastoyanova and Frank Leymann. BPEL'n'Aspects: Adapting Service Orchestration Logic. In *Web Services, IEEE International Conference on*, Los Alamitos, CA, USA, 2009. IEEE Computer Society. doi: 10.1109/ICWS.2009.75.

[6] Dimka Karastoyanova, Alejandro Houspanossian, Mariano Cilia, Frank Leymann, and Alejandro P. Buchmann. Extending BPEL for Run Time Adaptability. In *Proceedings of the 9th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2005)*. IEEE Computer Society, September 2005.

[7] Dimka Karastoyanova, Frank Leymann, and Alejandro P. Buchmann. An Approach to Parameterizing Web Service Flows. In *Proceedings of the 3rd International Conference on Service Oriented Computing (ICSOC 2005)*. Springer, November 2005.

[8] Dimka Karastoyanova, Rania Khalaf, Ralf Schroth, Michael Paluszek, and Frank Leymann. BPEL Event Model. Technical Report 2006/10, Institute of Architecture of Application Systems, University of Stuttgart, 2006.

[9] Rania Khalaf and Frank Leymann. Role-based Decomposition of Business Processes using BPEL. In *International Conference on Web Services (ICWS 2006)*, pages 770–780. IEEE Computer Society, September 2006. ISBN 0-7695-2669-1. doi: 10.1109/ICWS.2006.56.

[10] Rania Khalaf and Frank Leymann. Coordination for Fragmented Loops and Scopes in a Distributed Business Process. In *8th International Conference on Business Process Management (BPM 2010)*. Springer, September 2010.

[11] Rania Khalaf, Dimka Karastoyanova, and Frank Leymann. Pluggable Framework for Enabling the Execution of Extended BPEL Behavior. In *Proceedings of the 3rd International Workshop on Engineering Service-Oriented Application (WESOA 2007)*. Springer, September 2007.

[12] Rania Khalaf, Oliver Kopp, and Frank Leymann. Maintaining Data Dependencies Across BPEL Process Fragments. *International Journal of Cooperative Information Systems (IJCIS)*, 17(3):259–282, September 2008. doi: 10.1142/S0218843008001828.

[13] Rania Y. Khalaf. *Supporting Business Process Fragmentation While Maintaining Operational Semantics: A BPEL Perspective*. Dissertation, Institute of Architecture of Application Systems, University of Stuttgart, Juni 2008.

[14] Oliver Kopp, Ralph Mietzner, and Frank Leymann. The Influence of an External Transaction on a BPEL Scope. In *CoopIS 2009 (OTM 2009)*. Springer, November 2009. doi: 10.1007/978-3-642-05148-7_27.

[15] Oliver Kopp, Matthias Wieland, and Frank Leymann. Towards Choreography Transactions. In *Proceedings of the 1st Central-European Workshop on Services and their Composition, ZEUS 2009, Stuttgart, Germany, March 2–3, 2009*, volume 438 of *CEUR Workshop Proceedings*, pages 49–54, Stuttgart, March 2009. CEUR-WS.org.

[16] Oliver Kopp, Hanna Eberle, Frank Leymann, and Tobias Unger. The Subprocess Spectrum. In *Proceedings of the Business Process and Services Computing Conference: BPSC 2010*, volume P-177 of *Lecture Notes in Informatics*, pages 267–279. Gesellschaft für Informatik e.V. (GI), 2010.

[17] Oliver Kopp, Lasse Engler, Tammo van Lessen, Frank Leymann, and Jörg Nitzsche. Interaction Choreography Models in BPEL: Choreographies on the Enterprise Service Bus. In *Subject-Orientation as Enabler for the Next Generation of BPM Tools and Methods – Second International Conference S-BPM ONE 2010*, volume 138 of *Communications in Computer and Information Science*. Springer, 2010.

[18] Oliver Kopp, Katharina Görlach, and Frank Leymann. Extending Choreography Spheres to Improve Simulations. In *International Organization for Information Integration and Web-based Application and Services 2010 (iiWAS 2010)*. ACM, 2010.

[19] *OMG Unified Modeling Language (OMG UML), Superstructure, V2.3*. Object Management Group, May 2010.

[20] *Web Services Business Process Execution Language Version 2.0*. Organization for the Advancement of Structured Information Standards (OASIS), April 2007.

[21] Ralf Schroth. Konzeption und Entwicklung einer AOP-fähigen BPEL Engine und eines Aspect-Weavers für BPEL Prozesse. Master's thesis, Institute of Architecture of Application Systems, University of Stuttgart, November 2006.

[22] Mirko Sonntag and Dimka Karastoyanova. Next Generation Interactive Scientific Experimenting Based On The Workflow Technology. In *Proceedings of the 21st IASTED International Conference on Modelling and Simulation (MS 2010), 2010.* ACTA Press, 2010.

[23] Mirko Sonntag and Dimka Karastoyanova. Compensation of Adapted Service Orchestration Logic in BPEL'n'Aspects. In *International Conference on Business Process Management*. Springer, 2011.

[24] Thomas Steinmetz. Ein Event-Modell für WS-BPEL 2.0 und dessen Realisierung in Apache ODE. Diploma thesis 2729, University of Stuttgart, Faculty of Computer Science, Electrical Engineering, and Information Technology, Germany, August 2008.

[25] Sanjiva Weerawarana, Francisco Curbera, Frank Leymann, Tony Storey, and Donald F. Ferguson. *Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging, and More*. Prentice Hall PTR, 2005.

[26] Branimir Wetzstein, Steve Strauch, and Frank Leymann. Measuring Performance Metrics of WS-BPEL Service Compositions. In *Fifth International Conference on Networking and Services, 2009. ICNS '09*, 2009. doi: 10.1109/ICNS.2009.80.

[27] Branimir Wetzstein, Dimka Karastoyanova, Oliver Kopp, Frank Leymann, and Daniel Zwink. Cross-Organizational Process Monitoring based on Service Choreographies. In *Proceedings of the 25th Annual ACM Symposium on Applied Computing (SAC 2010)*, pages 2485–2490. ACM, March 2010. doi: 10.1145/1774088.1774601.

[28] Farid Zaid, Rainer Berbner, and Ralf Steinmetz. Leveraging the BPEL Event Model to Support QoS-aware Process Execution. In *Kommunikation in Verteilten Systemen (KiVS)*, Informatik aktuell. Springer Berlin Heidelberg, 2009. doi: 10.1007/978-3-540-92666-5_8.