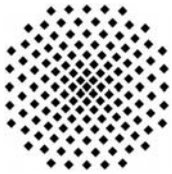

Universität Stuttgart

Fakultät Informatik, Elektrotechnik und Informationstechnik

BPEL Event Model

Dimka Karastoyanova, Rania Khalaf,
Ralf Schroth, Michael Paluszek,
Frank Leymann

Report 2006/10

November, 2006

Institut für Architektur von Anwendungssystemen

Universitätsstr. 38
70569 Stuttgart
Germany

CR: C.2.4, D.2.6, D.2.11, D.2.12, H.4.1

Summary

The document presents an engine-independent BPEL event model. We present two groups of events – events related to the life cycle of BPEL processes that are produced by the process execution environment, and events used to control or influence the life cycle of BPEL processes produced by applications external to the BPEL processor.

The events produced by the BPEL processor are notifying state changes in the life cycle of processes, activities, loops and fragmented loops, scopes and fragmented scopes, BPEL links. Some of the state transitions, depending on the scenario they are used in, may be fired only if a particular action/event is signaled by an external application. This means that a process instance would remain blocked in a particular state if the external event is not notified to the BPEL engine. The external events are meant to control the execution of BPEL processes, in particular to unblock process instances being in particular states, as well as enforce state transitions from the outside.

The event model is used by the authors of the report in several projects, all utilizing process life cycle events in different scenarios. This report represents an attempt to create an event model common to several projects and help reuse of research results and software, and foster cooperation. In general, the model is meant to be independent of BPEL processor implementation. 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.

Contents

1. Introduction 3

2. BPEL Event Model 4

2.1. Process life cycle events 4

2.2. Activity life cycle events 5

2.3. Scope life cycle events 8

2.4. Loop events 10

2.5. Link events 11

2.6. Incoming events 12

2.7. Classification of types of events 13

3. Conclusions 14

4. References 14

1. Introduction

Business Process Execution Language for Web Services (BPEL4WS or BPEL) [1] is the de facto standard for defining processes with Web Services (WS) [7] as participants. The BPEL specification does not put any requirements on the implementation and therefore it does not impose any event model on process execution environments. The process execution environments for BPEL processes are also called BPEL engines and contain a component called the navigator. The navigator is also called a BPEL processor.

In this report we present a *BPEL event model*. It specifies events related to the life cycle of processes, activities, scopes, loops and links. These events are produced by the BPEL processor and are used to drive the state transitions of the elements specified above.

Some of these events are identified as *blocking*, i.e. the state transitions done upon a blocking event result in a state that cannot be left, unless another event produced by an application external to the BPEL processor is notified. The blocking events are dependent on the particular scenario they are used in. The BPEL event model presented here takes into account requirements for blocking imposed by three particular application scenarios: support for coordinated interaction of fragmented processes running on multiple BPEL processors [4], implementation of process compensation based on externalizing the BPEL transaction support to an external coordinator [5], improving the flexibility of processes in an engine-independent manner using language extensions [3] or an aspect-oriented approach [6].

In addition, we identify events that are produced by applications *external* to the BPEL processor and can influence the execution of process instances. External events may be used to unblock process instances in a state reached by a blocking type event.

2. BPEL Event Model

This section presents the event models for BPEL processes, activities, links, scopes and loops in terms of State diagrams [2]. These event models include events required for supporting fragmented scopes and loops [4] as well. The event types we identify and the groups we classify them in are presented later in the document.

2.1. Process life cycle events

The events produced during the life cycle of a process are presented in the next figure.

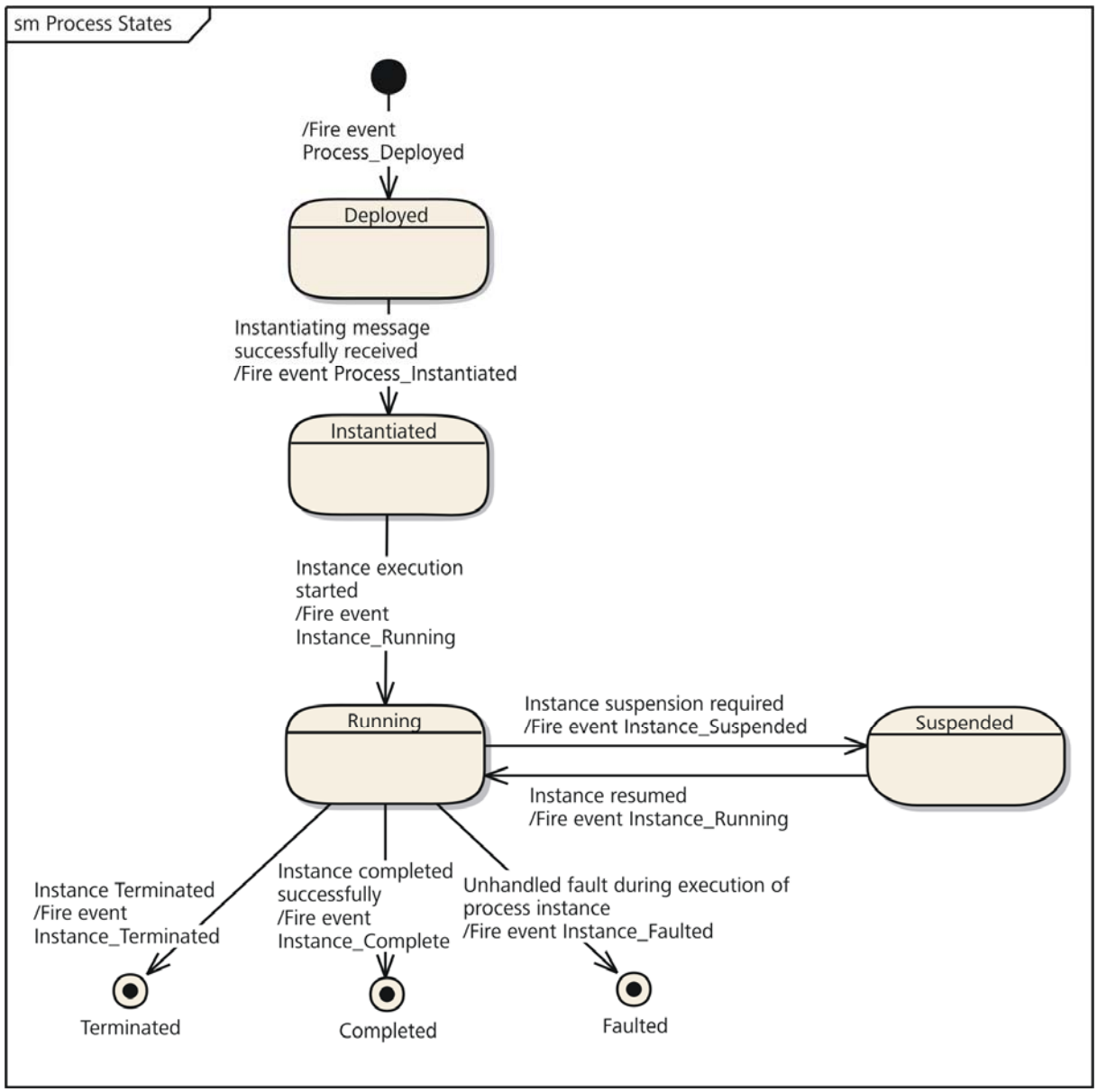

Figure 2.1.1: Process Life cycle events

Description of events:

- **Process_Deployed**
  This event is fired whenever a new BPEL-process is deployed into a BPEL engine. This event is not fired for each instance, but rather per process model.
- **Process_Instantiated**
  The event is fired, when a process is instantiated through a pick, receive or onMessage activity with the attribute "`createInstance=yes`".
- **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.
- **Instance_Terminated**
  This event is fired, when a process instance is terminated though execution of a `<terminate>` activity.
- **Instance_Complete**
  This event is fired, when a process finishes successfully (not terminated by a fault or by a `<terminate>` activity)
- **Instance_Faulted**
  This event is fired, when a process is terminated by a fault that was not handled and propagated to the implicit fault-handler of the root-scope of the process.

2.2. Activity life cycle events

This section describes the general life cycle events for all BPEL activities. Later sections cover special events for scopes and loops, but some of the events typical for scopes and loop, being activities, are presented here as well. The activity life cycle events are presented in Figure 2.2.1.

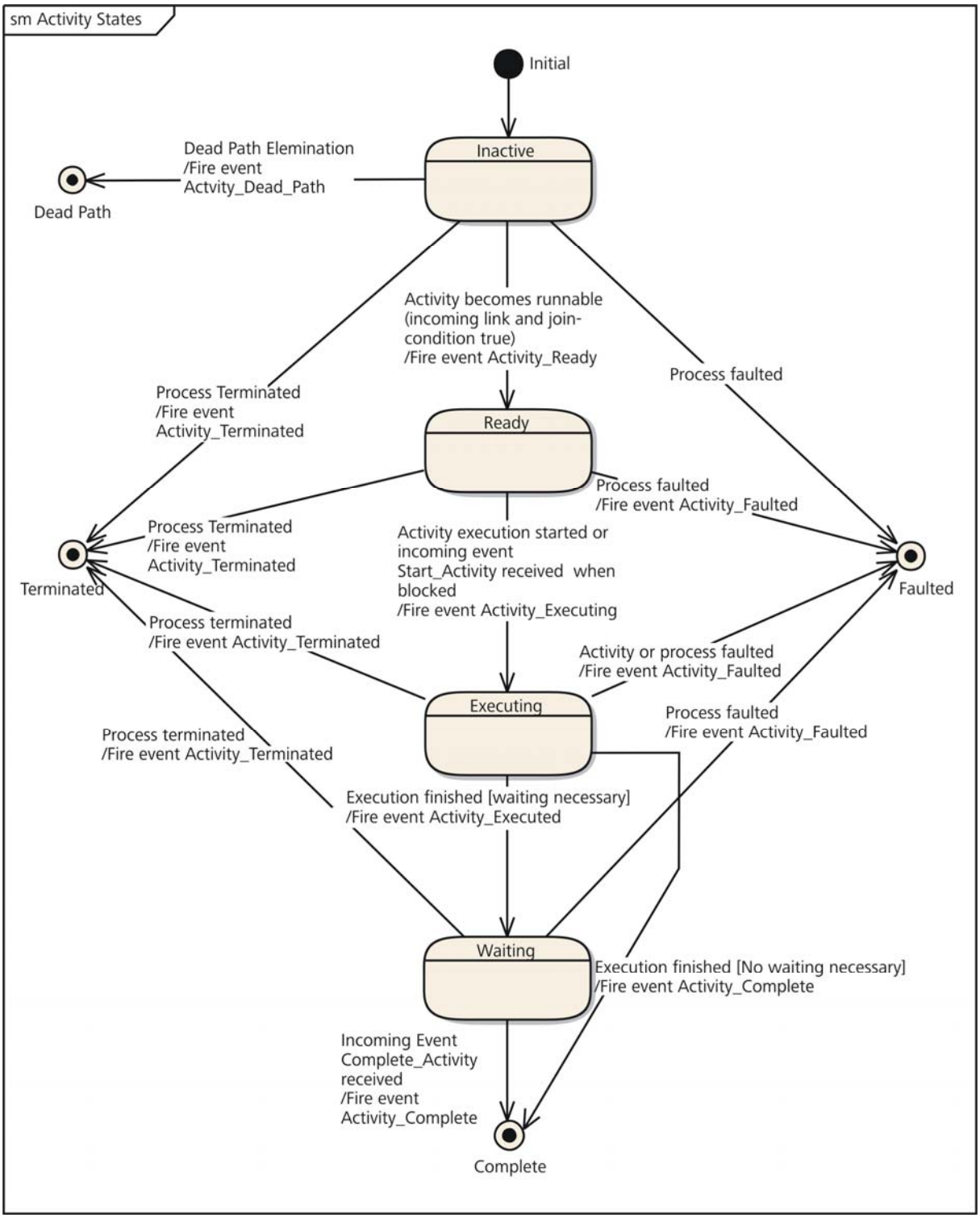

Figure 2.2.1: Activity life cycle events

Description of events:

- **Activity_Ready**
  This event is fired, when an activity becomes ready to execute (i.e. all incoming links are evaluated and the join-condition is true). In case of fragmented scopes, the engine is waiting (blocking) for a response from the coordinator before starting the execution, e.g. to synchronize the start of execution of fragments with different incoming

links. Further execution is triggered by the incoming event Start_Activity which is sent by the coordinator. This blocking only occurs when the event is marked as blocking for the current activity.

- **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).
- **Activity_Executed**
  This event is fired, when the execution of an activity is finished and a subscriber for this event is registered. The engine changes to the state "waiting" and waits (blocking) for signals from external sources (e.g. completion of child-activities, notification from coordinator etc.) before completing the activity. When no subscriber is present this event is not fired and the engine transfers directly from state "Executing" to "Complete" (see "Continue" incoming event).
- **Activity_Complete**
  This event is fired when an activity is completed and received signals from external sources (e.g. completion of child-activities, notification from coordinator)
- **Activity_Dead_Path**
  This event is fired, when an activity was marked dead by the engine's dead-path-elimination (DPE) mechanism implementation.
- **Activity_Terminated**
  This event is fired, when an activity is marked terminated or aborted because the associated process instance is terminating.
- **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 in a preceding activity of the same scope is not handled or if a fault of its child-scopes was not handled.

2.3. Scope life cycle events

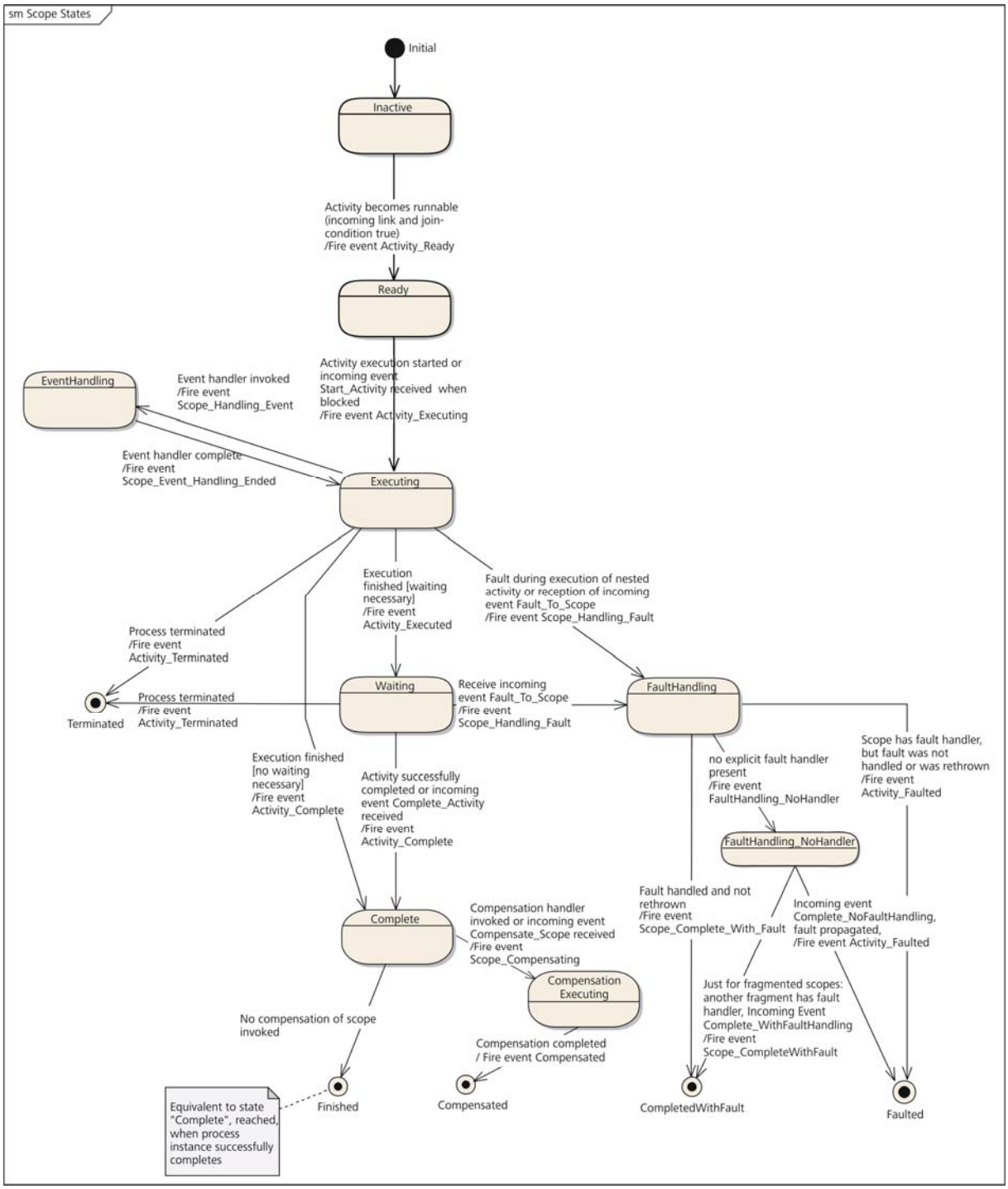

Figure 2.3.1: Life cycle events of Scopes

Description of events:

- **Scope_Handling_Fault**
  This event is fired, when a scope's fault-handler (explicit as well as implicit) is invoked.
- **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.
- **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_Event_Handling_Ended**
  This event is fired, when a scope's event handler finished executing.
- **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.

2.4. Loop events

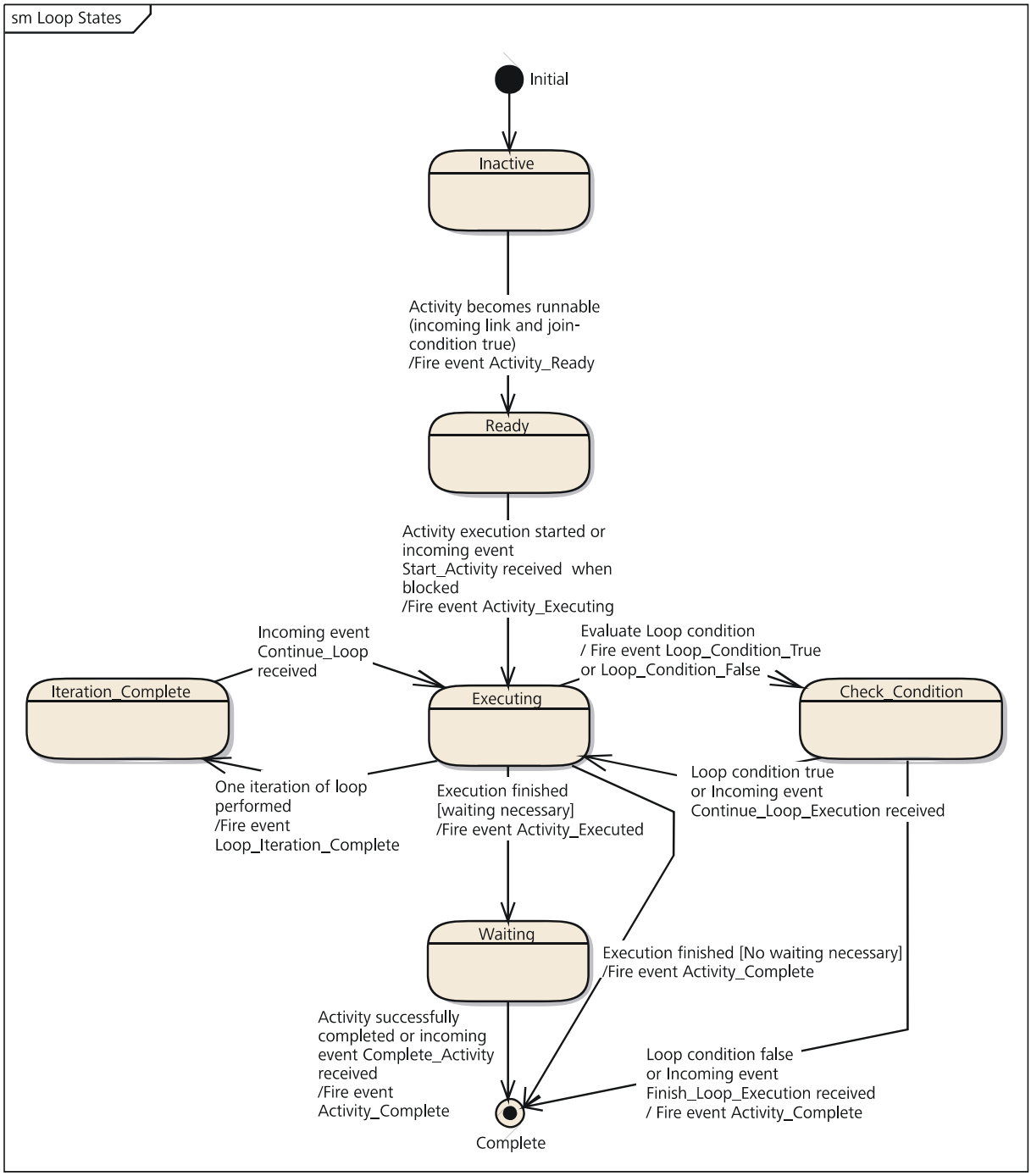

Figure 2.4.1: State-transition Diagram - Loops

Description of events:

- **Loop_Iteration_Complete**
  This event is fired when an iteration of a while loop is complete and before the loop-condition is re-evaluated. This event is especially important for fragmented loops, where it is necessary to synchronize the individual fragments.

- **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_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.

2.5. Link events

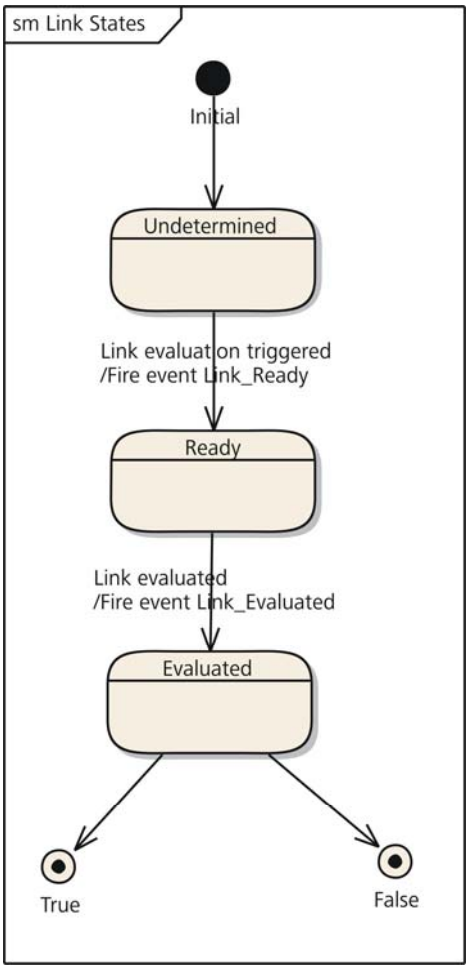

Figure 2.5.1: State-transition Diagram - Links

Description of events:

- **Link_Ready**
  This event is fired when a Link is ready for evaluation, i.e. immediately after its source activity has been completed.
- **Link_Evaluated**
  The event is fired when the transition condition on a link has been evaluated.

2.6. Incoming events

The engine reacts to events that are sent to it by the external entities (applications), e.g. a coordinator of fragmented processes [4], an external monitoring tool [3], an external coordinator supporting BPEL transactions [5], or an aspect weaver [6].

- **Compensate_Scope**
  This event starts the execution of the compensation handler of a scope that is in the "Complete" state.
- **Fault_To_Scope**
  This event causes the specified fault to be thrown in the context of specified scope. The fault is propagated in the scope-hierarchy like regular faults. It is consumed by a scope that is in the "Executing" or "Waiting" state.
- **Compensated**
  This event tells the engine, that the scope's state must change to "Compensated", i.e. all other fragments of a scope have also finished compensating and are ready to change to the "Compensated" state.
- **Start_Activity**
  This event causes the specified activity that is blocked in the state "Ready" to be continued.
- **Complete_Activity**
  This event unblocks the specified activity that is blocked in the state "Waiting" fires the transition to the "Completed" state. In addition, the event is used to fire the transition of activities that are in the "Ready" state to the state "Completed" (doing so activities can be skipped).
- **Continue_Loop**
  This event unblocks a loop that is blocked in the state "Iteration_Complete" and enforces reevaluation of the loop-condition.
- **Continue_Loop_Execution**
  This event unblocks a waiting loop in the state "Check_Condition" after the loop condition was evaluated to true. A fragment can be forced in this way to execute a new loop iteration from a fragment that evaluates the loop condition.
- **Finish_Loop_Execution**
  This event unblocks a waiting loop in the state "Check_Condition" after the loop condition was evaluated to false by another fragment of the loop**.**
- **Set_Link_State**
  This incoming event is used to set the state of a link that is blocked in a state "Evaluated" to a value commanded from outside the engine. Depending on the value of the link's transition condition calculated outside the engine, the Link goes in either state "True" or "False".
- **Continue**
  The event simply unblocks activities; it is used whenever a state is declared as blocked when using aspects [6] but there is no subscribed aspect for the particular activity.

2.7. Classification of types of events

We classify events according to two different criteria: blocking and direction.
The direction indicates whether the event is generated by the engine as a source or by an external entity. Blocking events block process instances (in particular activities/links in a process instance) until an incoming event from an external entity is received that unblocks the particular instance.

Table 1. Classification of types of events

Event	Blocking	Source
Process_Deployed	-	engine
Process_Instantiated	-	engine
Instance_Running	-	engine
Instance_Suspended	-	engine
Instance_Terminated	-	engine
Instance_Completed	-	engine
Instance_Faulted	-	engine
Activity_Ready	X	engine
Activity_Executing	-	engine
Activity_Executed	X	engine
Activity_Complete	-	engine
Activity_Dead_Path	-	engine
Activity_Terminated	-	engine
Activity_Faulted	X	engine
Scope_Compensating	X	engine
Scope_Handling_Event	-	engine
Scope_Event_Handling_Ended	-	engine
Scope_Complete_With_Fault	-	engine
Scope_Handling_Fault	-	engine
Compensate_Scope	-	engine
Loop_Condition_True	X	engine
Loop_Condition_False	X	engine
Loop_Iteration_Complete	X	engine
Link_Ready	-	engine
Link_Evaluated	X	engine
Compensate_Scope	-	external
Fault_To_Scope	-	external
Compensated	-	external
Start_Activity	-	external
Complete_Activity	-	external
Continue_Loop	-	external
Continue_Loop_execution	-	external
Finish_Loop_Execution	-	external
Set_Link_State	-	external
Continue	-	external

3. Conclusions

The BPEL Event model presented here is meant to be independent of any BPEL processor implementation. A corresponding mapping of the model to implementation-specific events is necessary, and it allows for implementations to include additional events if necessary.

This BPEL event model is defined with the purpose of creating the basis for implementing a common infrastructure for notifying life cycle events. Such an infrastructure will be presented in future publications of the authors.

4. References

1. Andrews, T. et al.: Business Process Execution Language for Web Services (BPEL4WS) Version 1.1, May 2003. ftp://www6.software.ibm.com/software/developer/library/ws-bpel.pdf
2. Fowler, M., Scott, K.: UML Distilled, Second Edition. Addison-Wesley, 2000.
3. Karastoyanova, D., Leymann, F., Nitzsche, J., Wetzstein, B., Wutke, D.: Parameterized BPEL Processes: Concepts and Implementation. In Proceedings of BPM 2006. Vienna, Austria, September 2006.
4. Khalaf, R., Leymann, F.: "E Role-based Decomposition of Business Processes using BPEL," In Proceedings of IEEE International Conference on Web Services (ICWS'06), 2006. pp. 770-780
5. Mietzner, R.: Extraction of WS-Business Activity from BPEL 1.1. Diploma Thesis 2444, University of Stuttgart, 2006.
6. Schroth, R.: Specification, Design and Implementation of a BPEL Engine with AOP Support and an Aspect Weaver for BPEL Processes. Diploma Thesis 2523. University of Stuttgart, 2006.
7. Weerawarana, S., Curbera, F., Leymann, F., Storey, T., Ferguson, D.F.: Web Services Platform Architecture; Prentice Hall; March 2005.