Farid Zaid, Rainer Berbner, Ralf Steinmetz: ***Leveraging the BPEL Event Model to Support QoS-aware Process Execution***. In: Kommunikation in Verteilten Systemen 2009 - KiVS 2009: Informatik aktuell, no. 1431-472X, Springer, January 2009.

Leveraging the BPEL Event Model to Support QoS-aware Process Execution

Farid Zaid, Rainer Berbner and Ralf Steinmetz

Multimedia Communications Lab - KOM, TU Darmstadt, Germany
Email: {Farid.Zaid, Berbner, Ralf.Steinmetz}@kom.tu-darmstadt.de

Business processes executed using compositions of distributed Web Services are susceptible to different fault types. The Web Services Business Process Execution Language (BPEL) is widely used to execute such processes. While BPEL provides fault handling mechanisms to handle functional faults like invalid message types, it still lacks a flexible native mechanism to handle non-functional exceptions associated with violations of QoS levels that are typically specified in a governing Service Level Agreement (SLA). In this paper, we present an approach to complement BPEL's fault handling, where expected QoS levels and necessary recovery actions are specified declaratively in form of Event-Condition-Action (ECA) rules. Our main contribution is leveraging BPEL's standard event model which we use as an event space for the created ECA rules. We validate our approach by an extension to an open source BPEL engine.

1 Introduction

One challenge for Service-oriented Architecture (SOA) is to meet the needs of particular processes, not just within the enterprise but across the entire value chain. Research in the context of SOA has shown that the usage of WSDL [8] and SOAP [6] is not sufficient to establish cross-organizational processes in real-world scenarios. Considering Quality of Service (QoS) requirements is crucial for a sustainable success of SOA. Without any guarantee regarding QoS, no enterprise is willing to rely on external Web Services within critical business.

Generally, Quality of Service (QoS) requirements are specified in a Service Level Agreement (SLA) between process partners. The The Web Services Business Process Execution Language (WS-BPEL or shortly BPEL) specification [3] offers support for handling abnormal (functional) situations in form of fault and compensation handlers. However, BPEL has no integrated method to handle QoS degradations, which are more considered as finer non-functional exceptions.

In this paper, we propose an approach that utilizes the BPEL event model to overcome BPEL's limitation in handling QoS exceptions. We also propose to use the Even-Condition-Action (ECA) rules as a logical framework to specify SLA statements and map them to relevant events and recovery actions. The proposed approach leverages BPEL in the following dimensions:

– The approach keeps the alignment with BPEL as a Programming in the Large paradigm. Similar to BPEL's declarative syntax that enables easy

mapping of business strategies to corresponding business processes, the ECA rules have a declarative syntax that can be used to express SLA terms in a logical way.
- The approach leaves BPEL's standard syntax intact while reusing BPEL event model to accommodate proposed recovery actions.
- The approach provides explicit separation-of-concerns between the business logic and business control. Thus it is possible to modify existing ECA rules and/or apply new ones without need to modify the process definition or stop running process instances.

This paper is organized as follows: Section 2 describes relevant basic concepts like the BPEL fault handling, the BPEL event model and the ECA rules. Section 3 presents our approach for handling QoS violations. Section 4 sketches our implementation to validate the concept. Section 5 positions our work among existing literature. Finally, Section 6 wraps up our paper.

2 Basic Concepts

2.1 Fault Handling in BPEL

A BPEL process can handle a fault through one or more fault handlers. Within a fault handler, some activities are defined to recover from the fault, possibly reverse the partial (unsuccessful) work of the activity where the fault has occurred, and/or signal a fault message to the client. BPEL faults can be classified in two categories [3]:

- Business faults are application specific faults and occur when an explicit <throw> activity is executed or an <invoke> activity gets a fault as response. Such faults are denoted by the <fault> element within the <operation> declaration of a WSDL definition. A fault has a qualified name (QName) and an associated message type.
- Runtime faults are not user defined and will not appear in the WSDL of a process or a service. The BPEL specification defines standard faults like "uninitializedVariable", "invalidReply", "mismatchedAssignmentFailure", etc. All these faults are typeless, meaning they do not have associated message types.

We focus here on situations related to SLA violation, for example, when the respose time of a service invocation exceeds the threshold specified by a governing SLA. We regard such a situation more as an *exception* rather than a *fault*. The difference is that BPEL deals with an occurring fault as abnormality that leads to unsuccessful completion of the scope/activity, whether that fault is handled or not. However, we see an exception as a finer deviation from normal execution and, if resolved, normal execution may be resumed. Fault is function-related while exception is non-functional.

In BPEL, it is possible to raise a timeout exception in an <onAlarm> construct by a <throw> activity and be caught by a <catch> activity. Listing 1.1

```
<scope name="BillCreationScope">
 <eventHandlers>
  <onAlarm for="PT5S">
   <throw faultName="createPDF:Timeout"
          faultVariable="faultVar" />
  </onAlarm>
 </eventHandler>
 <invoke partnerLink="pdfConverterPL"
         operation="createPDF"
         portType="pdfConversionPT"
         inputVariable="urlVar"
         outputVariable="pdfVar"/>
</scope>
```

Listing 1.1. Timeout Handling in BPEL

shows how a "createPDF:Timeout" fault (or exception in our understanding) is thrown 5 seconds after <invoke> has executed. However, because only one alarm can be activated for each scope at a time, if we want to catch different timeouts, then we must enclose each <invoke> with its own scope, which is a tedious and error-prone task for a process designer.

2.2 BPEL Event Model

Execution of a BPEL process is rich with events that fingerprint each state and transition with useful information. Such information can be used in different scenarios like process audit trailing and coordination of fragmented processes that run on multiple BPEL engines. However, the BPEL specification does not impose any event model on the process execution environment (engine). Therefore, different implementations may have different types of events, and even different event structures [9].

The event model in [9] classifies events according to two different criteria: direction and blocking. The direction indicates the source of the event, which can be the BPEL engine itself or an external entity like another BPEL engine or a coordinator. Blocking events block process instances until an incoming event from an external entity is received that unblocks the particular instance. In the rest of this paper we will use the following definition of a BPEL event.

Definition 1. *For a given BPEL activity A, we define a BPEL event fired by A as the triple $e(A, t, s)$ where t is the time when the event was fired and s is the state entered by A after firing this event.*

Process Events These are triggered as a process instance changes its state as shown in Fig. 1(a). These events are described in detail in [9], however of particular interest to our approach is the *ProcessDeployed* event which is fired

whenever a new BPEL process model is deployed into the BPEL engine. This event is not fired for each instance, but rather per process model, therefore we use this event to define variables with process scope, i.e. variables that are accessible from all process instances. The *ProcessInstantiated* event signals when a new process instance is loaded. The *InstanceCompleted* event signals when the process instance finishes successfully. Therefore we can use these last two events to mark overall QoS associated with each process instance.

Activity Events Fig. 1(b) shows the general life cycle events for all BPEL activities. Events at the activity level can be combined to mark QoS metrics for single activities and or scope of activities.

The *ActivityReady* event is fired when an activity becomes ready to execute. This event may be marked optionally as blocking, in this case an incoming *StartActivity* event is needed to start the actual activity execution. When the activity starts execution, it fires an *ActivityExecuting* event. The execution itself depends on the work to be done by that activity, for e.g. an <invoke> activity places a call to a web service, while the <receive> activity waits for an incoming request from another service. If the activity finishes execution successfully, it either fires the *ActivityExecuted* event and enters the *Waiting* state if an event listener is registered, or it fires the *ActivityCompleted* event and enters the *Completed* state directly if no event listener is registered. In the *Waiting* state the activity is blocked and can be only completed by an incoming *CompleteActivity* event. At any state in its life cycle, the activity can change to the *Faulted* state if a fault is encountered in the activity itself or in a preceding activity or when a fault in a child scope is not handled. A faulty situation is signaled via the *ActivityFaulted* event.

Incoming events External sources like a coordinator of fragmented processes or a monitoring tool can influence a process execution by sending events. The *StartActivity* event causes an activity that is blocked in the *Ready* state to be continued. The *CompleteActivity* event unblocks an activity that is blocked in the *Waiting* state (thus causing the transition to the *Completed* state) or to fire the direct transition from the *Ready* state to the *Completed* state (thus causing an activity to be skipped). The *Continue* event simply unblocks activities [9].

2.3 ECA Rules

ECA Rules form a natural candidate for systems where reactive functionality is needed. Each rule is characterized by the events that can trigger it; once a rule is triggered and its condition holds, then rule action is executed. Rules have languages for event, query, action and condition testing. Each of these languages has metadata and ontology which are associated with a processor [2]. In this paper, we do focus on the paradigm rather than on the specific syntax of the rules. Listing 1.2 shows the pseudo syntax for an ECA rule we will be using in rest of this paper.

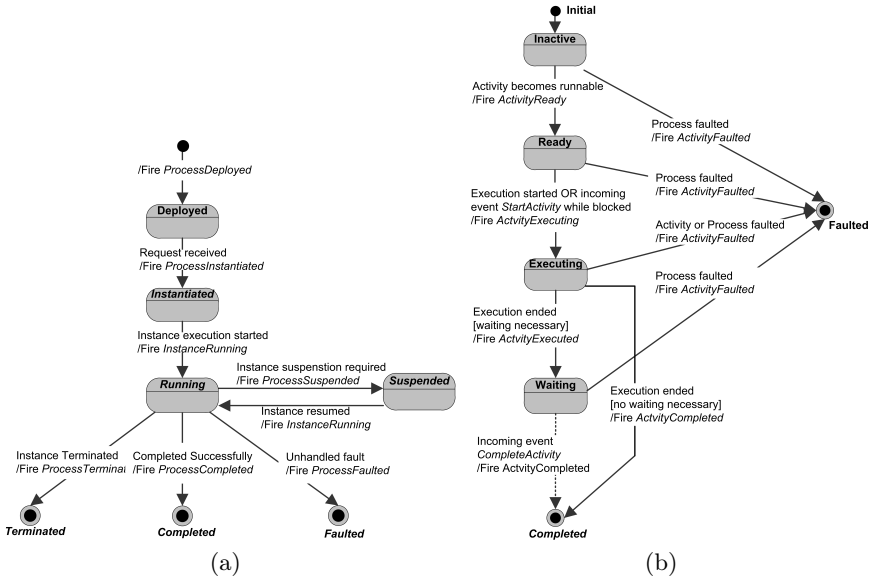


Fig. 1. Process events (a) and activity events (b) [8].

```
Rule ruleID
 Event triggering event
 Condition guarding statement
 Action recovery action
```

Listing 1.2. ECA rule pseudo syntax

3 Approach

We propose a flexible and declarative approach to handle QoS exceptions in BPEL processes by applying ECA rules to events generated during a process execution. Fig. 2 shows an overview of the approach. In the Logic Plane, a standard BPEL process will be deployed to a BPEL engine for execution. In the Control Plane, a set of ECA rules are registered with a rule engine (event manager). The BPEL engine sends subscribed events to the rule engine and reacts to recovery actions issued by the rule engine when some rules are triggered.

This clear distinction between logic and control planes permits creation of rules to modify QoS constraints without interrupting running process instances.

3.1 Detecting QoS Exceptions

It is obvious from Section 2 that BPEL events represent atomic events, in the sense that an event conveys information about a volatile situation at some point

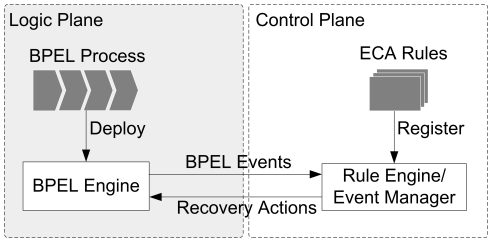


Fig. 2. Approach Overview

in time. Therefore applying ECA rules to these events directly will trigger real-time and instant actions, which may not be relevant to detect QoS exceptions which typically require interval-based rule processing. To handle this, we apply a formalism based on Temporal Event Algebra to express QoS exceptions as composite events formed from atomic BPEL events. Then we apply ECA rules to the mapped composite events. Next we show how to detect response time and availability violations from primitive BPEL events.

Response Time Exceptions Listing 1.3 shows the events and rule needed to detect a violation of response time threshold when executing an activity *A*. The rule engine needs to subscribe to two events fired by the invocation activity *A*: event *e1* with type *ActivityExecuting* which marks when *A* starts to execute, and *e2* with type *ActivityExecuted* which marks when *A* finishes execution. We use the temporal event composer *AND THEN* to indicate that the rule *rtRule* triggers each time an *e1* event is followed by *e2* event [2].

```
Subscribe
 e1(invoke(A), t, ActivityExecuting)
 e2(invoke(A), t, ActivityExecuted)
Rule rtRule
 Event e1 AND THEN e2
 Condition (e2.pInstance == e1.pInstance) &&
           (e2.t-e1.t > rtThreshold)
 Action replaceOnNext(A, a1, a2)
```

Listing 1.3. Events and rule to handle a response time exception

We measure the response time as the time elapsed between the two events. Of course, this definition of response time is an approximation of reality as it counts communication delay to the response time of the partner service. The condition component assures that these events are fired within life cycle of the same process instance (pInstance) and that the estimated response time does exceed the threshold value. If the condition holds, the action *replaceOnNext* is executed (explained in Section 3.2).

BPEL's <invoke> activity represents the actual interaction between a process and external services. However, as the event model described earlier applies

to all BPEL activities, we can apply similar approach to handle QoS violations for composition activities (constructs). For example, if two invoke activities are composed via a <sequence> construct, then we can use the *ActivityExecuting* and *ActivityExecuted* events fired by the sequence activity to estimate the aggregate response time of the two activities, rather than filtering all events for individual invoke activities.

Availability Exceptions Listing 1.4 shows how different BPEL events can be combined to detect an availability exception. Here, the rule engine subscribes to *e1* event fired from the process where activity *A* is defined. This event is used to to trigger *initRule* that initializes two variables with process-wide scope: *all*, to count the number of times *A* was scheduled for execution, and *failures* to count times of failed executions. The *schedRule* uses event *e1* to increment *all*, while the *failRule* increments *failures* when an *e2 AND THEN e3* pattern occurs. An *e3* event identifies a faulty invocation, which as discussed in Section 2 can have different reasons, therefore the condition part of *failRule* assures that the fault (fault_type) is really related to partner's availability, and is not due to a local reason. When its condition holds, *failRule* will fire the *avEvaluate* which is an internal event we define to trigger the *avRule* rule. This in turn uses the *failures* to *all* ratio to estimate the availability [7].

```
Subscribe
 e1(process(A),t,ProcessDeployed)
 e2(invoke(A), t, ActivityExecuting)
 e3(invoke(A), t, ActivityFaulted)
Rule initRule
 Event e1
 Condition true
 Action all=0, failures=0
Rule schedRule
 Event e2
 Condition true
 Action all++
Rule failRule
 Event e2 AND THEN e3
 Condition (e3.fault_type == remoteFault)
 Action failures++, fire (avEvaluate)
Rule avRule
 Event avEvaluate
 Condition (1-failures/all < avThreshold)
 Action replaceOnNext (A, a1, a2)
```

Listing 1.4. Events and rules to handle availability exception

In contrary to active monitoring, which implies probing partner's availability with periodic pings, the shown mechanism resembles a kind of passive monitoring, which counts the times of successful and unsuccessful processing of requests sent to the partner service.

3.2 Recovery Actions

Basic Actions Actions are recovery commands that attempt to repair process execution in favor of better fulfillment of QoS levels.

As mentioned in Section 2, situations that can be handled by the standard fault handling are excluded here, although our approach can be elaborated to cover such situations. Initially, we identify the following basic actions:

- $Ignore\,(A)$ causes execution of activity A to proceed normally. This is the default action when no QoS deterioration is detected. It steps the activity from *Waiting* state to *Completed* state.
- $Skip\,(A)$ causes execution of activity A to be skipped. It causes the activity to pass over from $READY$ state to *Completed* state.
- $ReplaceOnNext\,(A,\, a_1,\, a_2)$ instructs activity A to change its binding from partner a_1 to partner a_2 for execution of next process requests (instances). This is useful when a partner service violates QoS levels repeatedly and better be replaced by another service that meets the same functional requirements. This command affects service binding during next process instances and has no effect on current activity state.

Mapping Actions to Event Model The target of an action can be the same activity firing the event that triggered the ECA rule, or it can be another activity. However, external recovery actions should be enacted in such a way that they do not cause conflicts to the states of the executing activities. We make use of the incoming events defined in Section 2 to apply external actions to BPEL activities. However, to preserve consistency of the event model, we assume that an activity enters the *Waiting* state after leaving the *Executing* state. With this assumption, we can map the basic recovery actions to BPEL event model as follows:

- The *Ignore* command is mapped to the *CompleteActivity* incoming event.
- The *Skip* command is mapped to the *Continue* incoming event.
- The *ReplaceOnNext* is mapped to the *StartActivity* incoming event. This command is applied to next execution of the activity and not to current execution.

Another assumption that is specific to applying the *Skip* command is that functional requirements are still met. This implies activity can not be *skipped* if its output parameters form a required input for a successive activity, even if the *Skip* command was issued by a triggered rule.

4 Implementation

To demonstrate our approach, we implemented the Event Manager (EM) as an extension to the ActiveBPEL engine [1]. Consistent with how ActiveBPEL engine manages various functions using managers, EM plugs into the engine by

implementing the "IAeManager" interface which defines all methods a manager has to support, most importantly, the create, start, and stop methods. Once EM is created, it registers itself as a listener for following event types:

- AeEngineEvent which marks when a process instance is created, started and terminated.
- AeProcessEvent which marks the different states of a BPEL activity.
- EmDeploymentEvent which is an extra event we defined to signal deployment of new business processes.

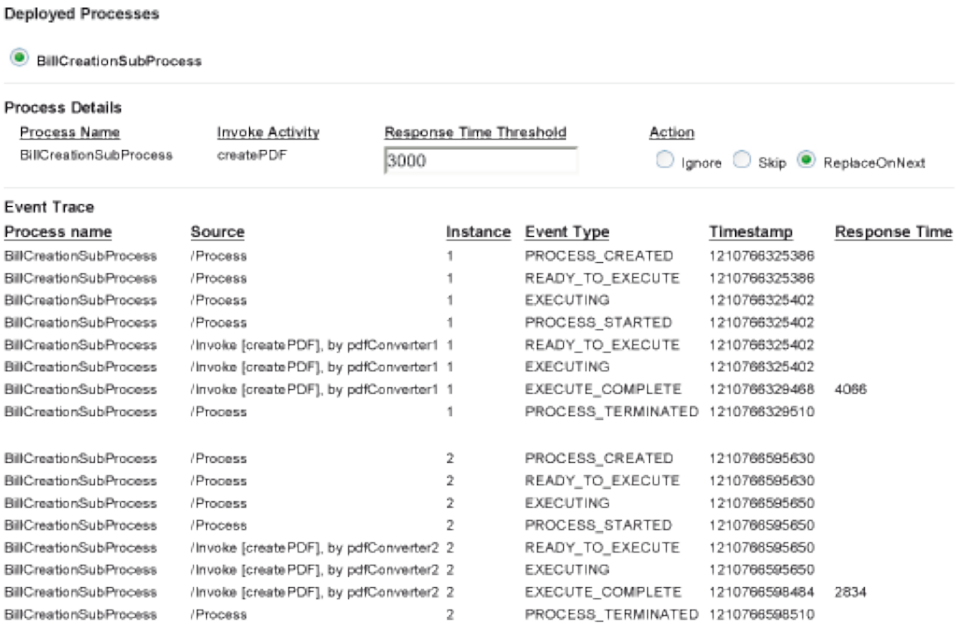
EM maintains several registries: an event registry where BPEL events are stored, a rule registry where ECA rules are stored and execution registry where an execution record for each process instance is stored. An execution record is identified by the pInstance as a key, and it accompanies a process instance all through its lifetime and holds information about QoS levels to be monitored and non-functional status of execution for each activity in that process. EM has also an event component which detects event patterns based on a simple event-matching algorithm, a condition component which evaluates condition part of an ECA rule and an action component which applies recovery commands to process execution.

Fig. 3 shows the web frontend we use to manage EM. The "Deployed Processes" pane shows all processes deployed to the engine (information extracted by the deployment event). The "Process Details" pane shows information about the process currently selected in the "Deployed Processes". This information includes a list of invoke activities for that process. In this pane, it is also possible to configure a simple ECA rule per invoke activity. We restrict our proof-of-concept implementation to detecting response time exceptions as it requires less effort to implement. The bottom pane displays breakdown of execution records related to the instances of the selected process. In the example shown, a *replaceOnNext* command causes the "pdfCreate" invoke in second process instance to bind to the backup "pdfConverter2" service, because the response time of the "pdfConverter1" service exceeded the set threshold of 3 seconds. Services "pdfConverter1" and "pdfConverter2" have similar implementations, however they have different response times, which we configured by introducing different artificial delays. This way, we simulated partner services that meet same functional requirements, but with different QoS levels.

To implement the *replaceOnNext* functionality, we modified the AeInvokeHandler which handles invocations of endpoint references on behalf of the business process engine. For each Invoke activity, the handler places a SOAP call (org.apache.axis.client.Call) to the web service located at specified AeEndpointReference address [1]. The address URL can be obtained from two different sources, dependent on the addressing scheme used:

- SERVICE: here the address URL is extracted from the <port> element in the WSDL interface of the target endpoint.
- ADDRESS: here the address URL is retrieved from a WS-Addressing endpoint reference, usually sent by another web service for callback.

We implemented the CurrentBindings class which is a singletone object to store active bindings (i.e. address URL of the partner link) for each invoke activity. The bindings are updated by the replaceOnNext command. The InvokeHandler itself is modified to check first if a binding exists in CurrentBindings. If no binding is available, then one of the two schemes mentioned above will be used to resolve the address URL.



Deployed Processes

BillCreationSubProcess

Process Details

Process Name	Invoke Activity	Response Time Threshold	Action
BillCreationSubProcess	createPDF	3000	Ignore Skip ReplaceOnNext

Event Trace

Process name	Source	Instance	Event Type	Timestamp	Response Time
BillCreationSubProcess	/Process	1	PROCESS_CREATED	1210766325386	
BillCreationSubProcess	/Process	1	READY_TO_EXECUTE	1210766325386	
BillCreationSubProcess	/Process	1	EXECUTING	1210766325402	
BillCreationSubProcess	/Process	1	PROCESS_STARTED	1210766325402	
BillCreationSubProcess	/Invoke [createPDF], by pdfConverter1	1	READY_TO_EXECUTE	1210766325402	
BillCreationSubProcess	/Invoke [createPDF], by pdfConverter1	1	EXECUTING	1210766325402	
BillCreationSubProcess	/Invoke [createPDF], by pdfConverter1	1	EXECUTE_COMPLETE	1210766329468	4066
BillCreationSubProcess	/Process	1	PROCESS_TERMINATED	1210766329510	
BillCreationSubProcess	/Process	2	PROCESS_CREATED	1210766595630	
BillCreationSubProcess	/Process	2	READY_TO_EXECUTE	1210766595630	
BillCreationSubProcess	/Process	2	EXECUTING	1210766595650	
BillCreationSubProcess	/Process	2	PROCESS_STARTED	1210766595650	
BillCreationSubProcess	/Invoke [createPDF], by pdfConverter2	2	READY_TO_EXECUTE	1210766595650	
BillCreationSubProcess	/Invoke [createPDF], by pdfConverter2	2	EXECUTING	1210766595650	
BillCreationSubProcess	/Invoke [createPDF], by pdfConverter2	2	EXECUTE_COMPLETE	1210766598484	2834
BillCreationSubProcess	/Process	2	PROCESS_TERMINATED	1210766598510	

Fig. 3. A snapshot of the Event manager frontend

5 Related Work

In [5], a component-based architecture is proposed to manage execution of process-oriented applications. Binding to a service is performed at runtime, after services are ranked based on their up-to-date QoS attributes.

In [12] SLA conditions are classified in soft and hard constraints. Violation of hard constraints leads to abnormal execution and is handled using constraint violation handlers, while soft constraints violation does not lead to erroneous state and is handled with event handlers. Composite events are detected by applying semantic matching of primitive events. Finally, recovery actions like *replaceBy* and *Retry* are used to fix problems manifested by the fault occurrence.

The authors in [11] follow a top-down approach to annotate a BPEL process with QoS assertions. At the top level, they specify a WSCDL description of the

process partners and the messages to be exchanged. The WSCDL descriptor is annotated with references to one or more SLAs at the same level. The SLAs define obligations and guarantees among the participants. The abstract WSDL description is then transformed into executable BPEL processes, while SLAs are transformed to QoS assertions that are directly attached to the corresponding partner links in BPEL to so that they cab be enforced by the BPEL engine.

Similarly, [4] represents dynamic service compositions with BPEL and provides assertions to check if involved services adhere to the contracted QoS levels. Assertions are verified with monitors which can be automaticallly defined as additional services and linked to the composite service.

In [10] also a top-down approach is used, however based on an extensible set of fault handling patterns defined as ECA rules. Before a process is deployed and executed, a generator is used to transform the fault patterns into BPEL code snippets (variables and activities like <if>, <catch>, etc.) that collectively give equivalent fault handling functionality. Although no change to BPEL syntax is needed, the code of transformed fault handlers mixes with code of the process logic and makes process maintenance more difficult. Besides, applying new rules mandates redeployment of the process definition.

In our approach, we also proposed the use of ECA rules as they possess a declarative syntax and map logically to the exception handling problem. However, we keep the creation and processing of these rules independent from process creation and deployment. This means that these rules can be created or edited without need to re-deploy the process definition. This is especially important for processes with long running instances. The key enabler to our approach is BPEL's standard event model which serves as a rich event base for detecting different execution anomalies.

6 Summary and Future Work

We have introduced in this paper an approach that leverages BPEL event model to provide handling of situations that are considered as non-functional exceptions rather than hard functional faults. The approach focuses on separating the business logic from SLA handling, thus enhancing BPEL's fault handling while keeping it intact. A process designer will typically specify ECA rules to handle QoS exceptions such as violations of response time and availability thresholds.

Although we validated the concept with a prototype for basic commands, we still have to address several open issues. Some of these issues are:

- The incorporation of commands that can affect the execution flow. So far, we addressed actions that affect the execution of a single activity, mainly the <invoke> activity.
- Monitoring of other QoS attributes. So far, the proof-of-concept is limited to respose time, due to its ease of implementation.
- We still need to study the computaional complexity of the approach and performance overhead introduced by instrumentation of rule processing.

These questions, once answered will much better outline the flexibility of our approach towards supporting QoS for BPEL processes.

Acknowledgement

Parts of this research have been supported by the German Research Foundation (DFG) within the Research Training Group 1362 "Cooperative, adaptive and responsive monitoring in mixed mode environments".

References

1. The ActiveBPEL Community Edition Engine. http://www.activevos.com/community-open-source.php.
2. J. J. Alferes et al. A First Prototype on Evolution and Behaviour at the XML-Level. Technical report, REWERSE, 2006. http://rewerse.net/deliverables/m30/i5-d5.pdf.
3. A. Alves et al. Web Services Business Process Execution Language Version 2.0. OASIS Standard, 2007. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html.
4. L. Baresi, C. Ghezzi, and S. Guinea. Smart monitors for composed services. In *ICSOC '04: Proceedings of the 2nd international conference on Service oriented computing*, pages 193–202. ACM Press, 2004.
5. R. Berbner, M. Spahn, N. Repp, O. Heckmann, and R. Steinmetz. Heuristics for QoS-aware Web Service Composition. *IEEE International Conference on Web Services (ICWS)*, pages 72–82, Sept. 2006.
6. D. Box et al. Simple Object Access Protocol (SOAP) 1.1. W3C Note, 2000. http://www.w3.org/TR/2000/NOTE-SOAP-20000508/.
7. J. Cardoso, A. Sheth, J. Miller, J. Arnold, and K. Kochut. Quality of Service for Workflows and Web Wervice Processes. *Journal of Web Semantics*, 1:281–308, Apr. 2004.
8. E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web Services Description Language (WSDL) 1.1. W3C Note, 2001. http://www.w3.org/TR/wsdl.
9. D. Karastoyanova, R. Khalaf, R. Schroth, M. Paluszek, and F. Leymann. BPEL Event Model. Technical report, Universität Stuttgart, 2006.
10. A. Liu, Q. Li, L. Huang, and M. Xiao. A Declarative Approach to Enhancing the Reliability of BPEL Processes. *IEEE International Conference on Web Services (ICWS)*, pages 272–279, Jul. 2007.
11. F. Rosenberg, C. Enzi, A. Michlmayr, C. Platzer, and S. Dustdar. Integrating Quality of Service Aspects in Top-Down Business Process Development Using WS-CDL and WS-BPEL. *Enterprise Distributed Object Computing Conference (EDOC)*, pages 15–15, Oct. 2007.
12. R. Vaculín, K. Wiesner, and K. Sycara. Exception Handling and Recovery of Semantic Web Services. *The Fourth International Conference on Networking and Services (ICNS)*, 0:217–222, Mar. 2008.