Retry Scopes to Enable Robust Workflow Execution in Pervasive Environments*

Hanna Eberle, Oliver Kopp, Tobias Unger, and Frank Leymann

University of Stuttgart, Institute of Architecture of Application Systems
Universitätsstraße 38, 70569 Stuttgart, Germany
lastname@iaas.uni-stuttgart.de

Abstract. Recent workflow languages are designed to serve the needs of business processes running in an unambiguous world based on unambiguous data. In contrast to business processes, processes running in a real world environment have to deal with data uncertainty and instability of the execution environment. Building a workflow language for real world flows based on a workflow language for business processes therefore may need additional modeling elements to be able to deal with this uncertainty and instability. Based on a real world process scenario we analyse and derive requirements for workflow language extensions for real world processes. The contributions provided by this paper are at first to investigate, how a workflow language can be extended properly followed up by the definition of workflow language extensions for real world processes, whereas the extensions are motivated by the real world process scenario. In this paper we use the Business Process Execution Language (BPEL) as extension foundation.

1 Introduction

Processes running in a real world environment usually have to deal with environment instability and data uncertainty. Environment instability is caused by the fact , that people and devices are moving around in space, which affects the execution environment, e.g. a network connection to a device may break down. Data uncertainty is caused by the fact that data representing situations are derived from the interpretation of sensor and other context data and therefore has a probabilistic and uncertain character.

This instability and the probabilistic character of situations and the environment might result in runtime faults, because e.g. assumed situations might turn out wrong. These faults might force the process to terminate followed up by the compensation of work already done. For example, resources crucial for a process execution such as workers and tools are moving around in space. During the process execution it might happen that a necessary resource is currently not available, which results in the fact that an *availability fault* gets thrown. But just in the moment the process starts terminating its faulted execution, the fault causing situation might change, since the missing resource, e.g. a worker arrives at his destination and becomes available for the process. Unfortunately, the process is not able to react to the new situation since it is terminating already.

* This work is partially funded by the ALLOW project. ALLOW (http://www.allow-project.eu/) is part of the EU 7th Framework Programme (contract no. FP7-213339).

A. Dan, F. Gittler, and F. Toumani (Eds.): ICSOC/ServiceWave 2009, LNCS 6275, pp. 358–369, 2010.
© Springer-Verlag Berlin Heidelberg 2010

As we demonstrated above, today's process modeling is not robust enough for real world process requirements. Since the resource availability might change during execution, as shown in the example shown above, an easy way to deal with an *availability fault* is the retry of the faulting activity, which might result in a successful completion of the activity and consequently the process, due to the fact that resources are now available.

A fundamental property of pervasive computing is to support people in their daily actions in an unobtrusive way. This leads to the challenge that first of all the system must be aware of peoples' actions. One attempt to overcome this challenge is to combine the pervasive computing paradigm with the business process management (BPM) paradigm [1]. Business processes are used to capture peoples' daily actions. But as described above existing process modeling languages are lacking in some features required for modeling pervasive processes. For example peoples' behavior is often erroneous [2]. People are starting activities. If they recognize that they are doing wrong they simply restart the execution of the activity. Furthermore, people sometimes are erratic. They jump from one activity to another activity without completing the first. Later on, they jump back to the first activity and have to start the execution of the activity from scratch. This implies that the in the process instance the first activity has to be restarted.

As a result, we have to extend exiting process modeling languages in order to provide a higher flexibility. The flexibility is required in order to enable the modeling of pervasive systems using business processes. As the paradigm of service oriented computing provides inherently a great flexibility, pervasive systems more and more are realized as service based applications. The Business Process Execution Language (BPEL) [3] the most popular process modeling language for modeling business process within service-based applications. However, BPEL has its origin in the BPM domain. It was designed to model and execute business process as service orchestrations. In this case reacting to high dynamic properties like network connections has not got much point during BPEL's design phase. Compared with pervasive environments data centers of enterprises are configured to maintain certain properties like availability using redundancy by default. Hence, a lot of faults arise rarely and consequently are not considered in process models.

In this paper we provide an approach to make process models more flexible and robust in a way, that allows processes to deal with such situations in instable environments as described above. We introduce a new concept which enables to define a set of activities within a process model, which should be retried in case a certain fault happens. We realize this new concept as extension to BPEL. BPEL provides several ways how extensions can be made to the language. Hence, we discuss several approaches how BPEL can be extended with retry semantics.

For that purpose, we extend BPEL with a new modeling element while maintaining BPELs' execution semantic. The modeling element enables and extends scopes to react on faulting situations and falsely assumed situations in a more flexible way than conventional BPEL scopes are able to react today.

The paper is organized as follows. In Section 2 we examine some suitable scenarios in detail, to foster a better understanding of the concept and the problems statement of this paper. This section is followed by a section providing for the understanding

necessary background information such the current BPEL scope semantics in Section 3. Our approach is presented in Section 4. We discuss the related work in Section 5. The achievements of this paper and the outlook of our work are summarized in Section 6.

2 Scenarios

We investigate the pharmaceutical development of a new medication as sample scenario. This medication development may include steps, which may fail, but whose successful completion is inevitable to the successful completion of the medication development. Figure 1 shows a simplified excerpt from a process for mixing and testing a new medication. First, the ingredients to test have to be fetched. Afterwards, the ingredients are mixed and tested. After the test the process has two alternatives to continue execution. First alternative is chosen if the mixture did not fulfill the requirements. Second alternative gets chosen in case the test is successful. The three steps, fetching, mixing and testing are grouped to indicate a desired all-or-nothing behavior. During the execution of the testing, a fault might happen. For example, wrong ingredients were fetched or the ingredients were mixed in the wrong order. In this case, the laboratory has to be cleaned up and the activities restarted. The syntax used to present the process is BPMN 1.2 [4].

A slightly different behavior can be observed in following scenario. A process is designed to support a worker of a delivery service in delivering some packages at some locations in town. All navigation instructions are computed relative to the workers current location. If the delivery man loses his way and also contact to the navigation process he might drive as long as he gets connected again. The navigation service computes the route depending on the delivery mans current location, and the delivery man tries to execute the delivery task once again. In the scenario described above the process for the delivery man to deliver some packages has following characteristics. During delivery the execution of the delivery process something might go wrong, but instead of going the whole way back to the delivery starting location the process gets the new location of the worker as input and ***restarts*** the delivery of the package.

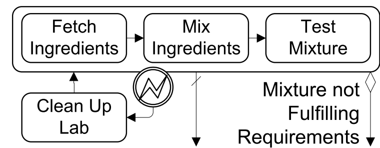

Fig. 1. Medical Test

Both scenarios something unforeseen happens during process execution causing a fault. The first scenario includes some repair actions in case a failure happens, which are performed before the scope gets retried. The process state is repaired to put the process in a state to be able to perform the activities once again. The second scenario restarts the activities of a scope without any repair actions. The state of the process is used as new input to the new execution of the scope.

3 Background Information

In this section we lay the foundation for a better understanding of this paper by introducing the basics of the BPEL's scope concept and having a deeper look at modeling elements, which play an important role in the realization of our concept.

3.1 BPEL Scope Semantics

Most important to foster the understanding of this paper is the BPEL `scope` activity. The scope activity is basically used to model transactional behavior. A scope encompasses a subset of activities. Additionally to these activities a scope enables to annotate business logic, which gets called in case a fault happens or the scope needs to get compensated or other events occur. Fault handler define the process logic to be performed in case a fault happens within a scope, whereas compensation handler define what has to be done to reverse successful work done already in preceding scopes. Event handler process events received by the process on scope or process level and termination handler provide additional control on terminated scopes. As other activity types in BPEL scope activities must be properly nested.

The semantics of a conventional scope can be best explained by the means of a state diagram shown in Figure 2. An activity runs through a set of states during its life-cycle. The state diagram shows the states and also what states can be reached after leaving a certain state. The BPEL specification does not impose any state model for activities. Hence, we use a simplified model, which is based on the models presented in [5, 6, 7]. This state diagram is used by the most existing BPEL implementations (e.g. Apache Ode). Depending on the engine implementation the scope activity instance is created either at process creation time or later during process execution. If the process instance gets created during the scope's instance is created, too, the scope's state is set to inactive. The latest point in time a scope instance must be created is, when the first incoming link of the scope activity is evaluated and the scopes state set into state inactive. The state inactive means here that all necessary resources for the scope's execution is allocated and it just need a trigger to set the scope into running. Once all incoming links are evaluated the activity state is set to running. During the running state the execution of the activities contained in the scope is performed and the event handlers attached to the scope get activated. After all contained activities and running event handlers are completed, the scope's state is set to state *complete* as well. Entering the state complete triggers a snapshot to be taken and the compensation handlers to be installed. The snapshot stores all scope specific data, which is needed in case the scopes compensation gets triggered, e.g. variable values at completion time, partnerLink values. If a fault happens within the scope while in state running the scope gets faulted, which means it changes to state faulted and all running activities and scopes surrounded by the faulted scope will terminate. Terminating means here to stop all surrounded running activities and scopes and so on. If all activities and scopes are terminated the scope enters the state terminated and finishes.

3.2 Design Goals for BPEL Extensions

The aim of this paper is to define new modeling elements for BPEL, which allows to model retry and rerun semantics. Hence, one major challenge of this work is, to find a way, how BPEL can be extended with the semantics of retry and rerun scopes in a way that complies with BPEL's extension rules. To be able to evaluate the different possible realization approaches, we need to identify design goals for BPEL extensions, which is done in the following. The design goals are partially derived from the BPEL specification and compliance rules for extensions. Other design goals are determined by usability

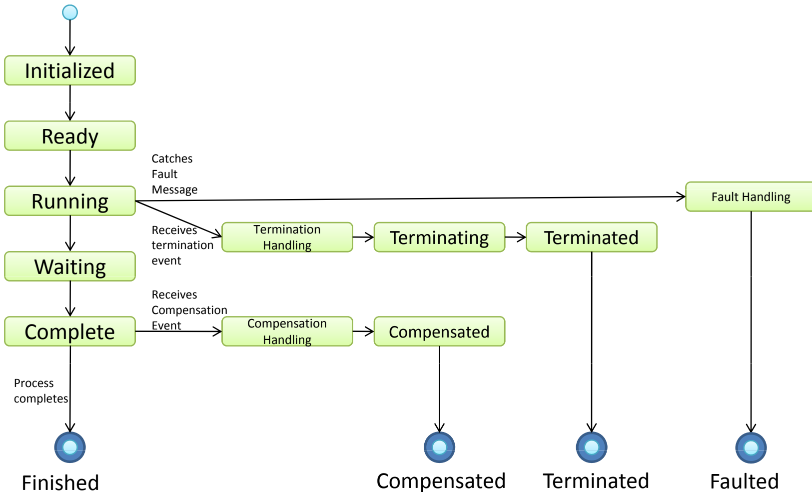

Fig. 2. Scope's Simplified State Diagram

requirements for the new modeling element. BPEL 2.0 specification is designed extensible, which leaves us the possibility to extend any existing BPEL modeling element. Syntactically, there are no restrictions made. BPEL 2.0 specification additionally defines a framework for activity extensions. Although, syntactically any extension can be made, semantically the extensions must not change any of existing BPEL semantics. It is allowed to add functionality to the BPEL specification, but not to change it. Consequently, if a scope activity gets extended, the extension need also to fit into BPEL's **F**ault-**C**ompensation-**T**ermination Handling, e.g. a completed retry/rerun scopes should be compensate-able like any other scope.

In BPEL functionality is designed as various activity types. There are activity types for data handling operations, activity types to define the control flow and basic activity types. To summarize, what has just been stated; BPEL's design paradigm follows the paradigm to design functionality as activity types, and not e.g. as attributes to a very basic activity type. Making all these functionalities explicit in that way makes it easier to be understood by the modeler. The usability of every extension must be investigated separately. From the usability point of view, we argue, that an extension to BPEL should be easy to understand and use.

Therefore to define an extension to the existing workflow language BPEL is a tightrope walk between following requirements:

1. the modeling element should as far as possible follow BPEL's modeling paradigm
2. the modeling element should support the required semantics as precise and concise as possible
3. the modeling element shall support the modeler's work and should be therefore intuitive comprehensible and usable
4. the modeling element should not define semantics redundant to functionality already defined.

4 Retry/Rerun Scopes: The Concept Extending BPEL with Retry/Rerun Scopes

In the following we define our realization approach for the scenario based on the results of the scenario analysis and the short introduction to BPEL and BPEL scopes. But first we shortly recall the results of the scenario analysis. We extracted the following two types of behavior.

1. Retry behavior says, that some activities need to be redone due to a faulting situation, but before those activities can be redone, some things need to get 'cleaned up'.
2. The restart behavior shows almost the same characteristics, but is a bit simpler due to the fact that conducted activities need not be 'cleaned up'.

The purpose of this paper is to define new modeling elements in BPEL to support the modeler in modeling the behavior as presented in the scenarios and to specify the semantics of these BPEL extensions.

4.1 Discussing Possible Realization Approaches

There are several possible ways to extend BPEL with *retry* and *restart* options. One way to integrate *retry* and *restart* options to BPEL is to introduce two new extension activity types. *retryScope* and *rerunScope* implement each either the rerun or the retry capabilities for scopes. The rerun/retry is scheduled if the scope gets faulted. It has to be defined how these extension activities integrate into the fault and compensation handling, which basically copies the behavior defined for normal scopes. Considering the design goals defined in section 3.2 this approach is not smart in that way , that much functionality must be redefined due to the fact that *retryScope* activity needs to redefine the whole scope semantics. Another critical point in this realization approach is, how these new scope activities can be integrated into the execution. Because all fault messages are send to a normal scope. To integrate the new extension activity types into the execution the fault messaging mechanism has to be bending to suit the desirable behavior properly. Another integration option is to extend the scope activity with an attribute, e.g. *restartOption*. Default setting of this attribute is `none`. The scope is executed as usual. If the *restartOption* is set to rerun, the scope in case of an error is compensated and restarted. If the *restartOption* option is set to restart, the scope is just started again after terminating enclosed running activities. But regarding this modeling approach closer it appears to be inflexible in the way that it is not possible to react with different behaviors to different types of fault messages, which is possible in normal scopes of the BPEL specification. Every time a fault is thrown the scope is rerun/restarted. In this case the we break with the design paradigm that every functionality is implemented as an activity type, because we indicate functionality as attribute to an existing activity type.

Retry/rerun behavior can be also modeled implicitly as extended fault handler reaction. Using this option the rerun option is triggered if the according fault message gets caught. This way we have the opportunity to force the scope to terminate immediately, because the fault is too serious. Other faults, are not that serious, but expected trigger the scope, the fault is caught, to be retried and rerun respectively. We introduce a new *rerun*

activity type. This activity can be used within a catch-block of a fault handler. Using the rerun activity in sequence after a compensate activity implements the retry option. A disadvantage of this approach is that it breaks with the BPEL design restriction that control flow modeled within a fault handler must not have a link, which points into the scope the fault handler is attached to. This approach enables also a differentiating treatment of different types of faults. The *rerun* activity type is modeled into the fault handlers catch-block, where each catch-block handles a different fault message. The retry/restart options are modeled rather implicitly. Additionally it is possible to force the scope to terminate immediately, if the fault is a serious one and gets caught by a corresponding catch block. Other faults are not that serious but expected to trigger the rerun/restart of the scope. A fault handler can define different reaction activities based on the type of fault which is thrown. The modeling restriction for the new activity type is, that it is only allowed to be used within a fault handlers catch-block. To reduce redundant definitions we argue to model the retry behavior option as a sequence, containing a *compensate* activity followed by a *restart* activity. This approach implements functionality as activity type and already defined scope behavior need not to be redefined.

4.2 Realization

Considering all possible realization approaches as discussed above we argue for the most flexible and less redundant approach, which uses a *restart* activity within a fault handler's catch-block. In the following we introduce and define the detailed semantics of the restart activity type, which triggers the re-execution of its scope.

Restart Activity Type. The restart activity has a syntax of the following kind: `<restart times="5" />`. The number of restarts of a fault handler for a certain fault type is restrained by the value of the `<times>` attribute. This attribute is needed to avoid causing an endless loop, which gets restarted over and over again by the restart activity. Note here, that the counter is not increased on each restart of the scope activity, but on the execution of the restart activity, where the restart activity is part of the fault handler. The counter is initialized during the initialization of the activity. Every execution of the restart activity with the same activity ID and process ID increases the counter.

```
<faultHandlers>
    <catch faultName="NoUserFound"?
        faultVariable="BPELVariableName">
        <sequence>
            <compensate />
            <wait />
            <restart times="5" />
        </sequence>
    </catch>
</faultHandlers>
```

Listing 1. Fault Handler with Restart Activity and Retry Semantic

In case the retry/rerun scope gets called by the same failure exceeding the boundary set by the `<times>` attribute, a fault is thrown. Therefore, we introduce new standard fault called *RestartScopeFault*. Using this fault implies that the extension must be declared with `mustUnderstand="true"`.

Listing 1 shows an example on how to use the restart activity to model *retry* behavior. The catch-block consists of the sequenced activities, compensate, wait and restart. The wait is used here to delay the restart of the scope activity. The *restart* activity must be the last activity in a branch of a catch block. The respective catch-block of a fault handler must not have any links pointing into another part of the process.

Retry/Rerun scope semantics — Fitting Retry/Rerun scopes into standard BPEL semantics. The introduction of the restart activity influences the conventional scope semantics since a scope must be able to change from state fault handling to state running.

A walk-through of example of Listing 1: First of all, if a fault gets caught by a fault handler the running child scopes are terminated. The `<compensate/>` activity triggers the compensation of completed child scopes. After compensation we wait as long as defined in the wait activity. This wait activity is followed by a restart activity. The restart activity sends a restart event to its corresponding scope. After this the restart activity completes. In the meantime the scope receives this event and changes its state from *faulted* to *running*. Therefore the scope state diagram must be extended by a transition to from state from *faulted* to *running*.

In case a fault is thrown during execution of the fault handlers catch block, the BPEL standard behavior is terminate the current execution of the fault handler and to catch the fault if possible by a fault handler of the same scope or a surrounding scope. Enclosed scopes are treated no differently as enclosed in conventional scopes, e.g. in case a fault is thrown, enclosed running scopes is terminated and enclosed already completed scopes is compensated.

We can distinguish two cases, on how the `<restart/>` activity can be used. Either it is used after a `<compensate/>` activity or without a `<compensate/>` modeled in the fault handler's catch block. In case the scope is not compensated, we need to make sure that data of the faulted execution is not lost, as described in the following section.

Data Handling Concerning Restart Activities. The data handling of retried scopes in case of a restart without compensation can be distinguished into two cases.

1. The restart of the scope is done without compensation and without taking over any computed knowledge during the former execution of the scope.
2. The restart of the scope is done without compensation, but with taking over of computed knowledge during the former execution of the scope. Where computed knowledge stands for data values of *variables* of the scope and *partnerLinks* respectively.

To enable option two we extend the restart activity type with a Boolean attribute *keepState*. The default value of *keepState* is `no`. If *keepState* is set to `yes`, (`<restart <keepState="yes"/>`) the values of the scopes' local data, like local `<variables>` and local `<partnerlinks>` is saved before the restart is triggered by the restart activity. This additional snapshot is needed, because the restarted scope has not been completed yet,

therefore no normal snapshot has been taken, when changing state from faulted to running. This *keepState*-knowledge is injected into the new scope instance.

Of course in the case that transactional behavior is wanted, both options described above are not suitable.

Defining Default Compensation Semantics of Restart Scopes. The default compensation of retry/rerun scopes diverges from the compensation of normal scopes. BPEL standard behavior takes a snapshot after a scope has completed. This snapshot is taken directly after the scope gets completed and is needed for compensation purposes. Therefore the snapshot saves the values of the scopes' local data, like local `<variables>` and local `<partnerlinks>`. This data is needed in case the scope gets the request to be compensated. A snapshot is taken after the scope enters the state *completed*. The default compensation of scopes is done by executing the compensation handlers of enclosed activities ordered by the backward control dependencies and executed instance traces. Control dependencies are not enough information due to the fact, that a scope might be enclosed by a loop activity. Every executed instance of the enclosed scope must be compensated. In case a compensation handler is defined the business logic in the compensation handler is executed, not caring about control dependencies and instance traces. The implications of the modeling options described above to compensation do also differ, the way compensation has to be performed. In the first case, where the `<restart/>` activity is sequenced after a `<compensate/>`, the compensation steps do not diverge from ordinary compensation steps, because, the state and the execution history of this scope does not diverge from an ordinary scope execution. Every executed step was compensated and therefore does not exist in the execution history any more. This does not happen to be the case, if no compensation is executed before a restart of a scope, which holds for both cases without snapshot and with snapshot. A possible execution history of a scope S sequencing the activities A, B, C and D might look like: A, B, C, A, A, B, A, B, C, D; scope S gets completed after completion of the enclosed scope activity D. The enclosed scope activity A got executed four times, scope B three times, C two times and D only one time. All instances in the execution history are compensated ordered by reverse control flow and instance history, like the scope instances created by a loop-activity.

Navigation Implications to Cross Boundary Links. Ordinary scopes in BPEL allow links to break through the boundary defined by activities belonging to the scope and activities, who do not. Links, which connect activities from within the scope with activities, not belonging to the scope, are called cross boundary links. In BPEL loop activities do not allow links to cross the loops boundary neither from the inside nor from the outside.

We allow cross boundary links to be modeled for rerun/retry scopes. But this has some serious implications on the navigation of the scopes enclosed activities. First we describe the problems that appear, if we do not adept the navigation behavior, which is followed by our solution approach. If the scope has to be restarted, but some links crossing the boundary from scope to the outside scope were evaluated already and execution might have proceeded already. What happens with the second evaluation of the cross boundary link? If a scope gets executed twice, the cross boundary link needs to

be executed twice. Now, two cases can be distinguished. Either the incoming links are evaluated before the second execution of a scope is performed, which indicates, that the process has proceeded with its execution in this branch, or the other incoming links have not been evaluated so far, which indicates that the second value of the transition is taken into account by the *joinCondition* evaluation. We propose following approach in this paper. There are no modeling constraints for cross boundary links. Cross boundary links are not evaluated until the scope completes successfully. This way the cross boundary link receives the proper input values to its evaluation. Evaluation of cross boundary links get delayed until the scope completes. Note that the delay of link evaluation might cause deadlocks, if cyclic dependencies appear.

An example of a cross boundary link is shown in Figure 3. The retry scope consists of the activities, *Fetch Ingredients*, *Mix Ingredients* and *Test Mixture*. The mixture shall go into production as soon as possible. Therefore, the ingredients for production shall be ordered as soon as the ingredients are determined. The cross boundary link semantics is defined as follows. The evaluation of the link needs to wait till the scope has completed successfully, which means, that the link will be evaluated after the successful completion of *Test Mixture*.

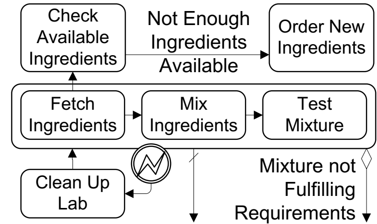

Fig. 3. Medical Test with Automatic Ordering of Ingredients for Production

5 Related Work

In this paper we realize these concepts as BPEL modeling options and extend the BPEL semantics. A concept of rerun and retry scopes can be found in [8]. The idea of retrying activities until they succeed is also mentioned in [9]. [10] presents a retry solution for BPEL based on ECA rules: ECA rules are attached to BPEL constructs. These rules are then transformed to a standard BPEL process. Thus, retrying is not a first-class modeling construct, whereas the approach shown in this paper smoothly integrates in BPEL.

Reruns are also supported in a platform for scientific workflows called *Kepler* [11]. Scientific workflows are very data and computing power intense processes used for simulation [12]. Simulations usually have a quite respectable amount of input parameters. If a simulation with a certain parameter configuration appears to show the expected result, it is possible to rerun the process with a slighly different parameter setting. The rerun in Kepler supports 'smart' rerun capabilities especially to reuse already computed data which does not need to be recomputed. Kepler uses a special kind of snapshot mechanism. The approach used in Kepler focuses on the reuse of already computed data, whereas our approach focuses some actions to be redone in case a failure happens. An assessment of BPEL to scientific workflows is done in [13]. There, the need of rerunning activities is also identified. Some other workflow specifications provide a modeling element for repeats as reaction to failures. MQWF [14] FDL defines for each activity an exit condition, which is evaluated after the execution of the activity. If the exit condition does not hold, the activity is restarted. Also the ADOME [15] framework offers

the possibility to re-execute activities similar to FDL. ADOME distiguishes several activity types, among others also a repeatable activity type. Once an activity is selected for execution but fails, it can be re-executed by the same agent or alternate agents, or using alternate resources; but the WFMS should not try other alternate paths because of some reasons. (E.g. due to set up overhead for a production line or a process etc.) Both the MQWF and the ADOME framework do not support repeats over more than one activity: It is not possible to model. In MQWF the decision whether an activity is restarted is triggered by the evaluation of a condition over the input and output data of the activity after completion and not immediately on the occurrence of a fault.

Using Business Process Management Notation (BPMN) [4] restart behavior can be modeled using a work around. All activities which should be restarted can be placed in a `Collapsed Sub-process`. An `Intermediate Error` triggers the fault handling and eventually the compensation actions. Thereafter a `Data-based Exclusive` can be used to determine whether the process goes on or the activities are restarted by forming a loop which triggers the `Collapsed Sub-process` another time.

BPMN does not provide native modeling elements to model the scenarios described in this paper. The modeler has to map restart semantics to normal BPMN constructs, which does not support the modelers work in an optimal way. Same applies if using [16].

6 Conclusions

The crucial question of extending a Turing-complete process modeling language with new modeling elements is the question of the benefit it provides.

In our work, the benefit of our extension constitutes in the fact, that implementing the very scenarios using our extensions, the process modeler can directly model what he has in mind, whereas modeling the same behavior with a modeling workaround using a loop activity. Furthermore, during runtime we can directly monitor the executed retries and do not have to count back them from the loop cycles. As a consequence of extending BPEL also the execution engines have to be extended, which can be seen as a disadvantage. Also the modelers have to learn and use the new modeling elements. However, in our opinion the pro-language-extension-arguments are stronger than the arguments against. In particular, that the extensions allow the modeler to model what he has in his mind, argues in favor of the extension solution.

In this paper we decided to define a new modeling element and its semantics as extension to BPEL. This new modeling element can be directly motivated by the pervasive scenarios described in Section 2. The modeling element helps to model processes closer to pervasive scenarios, whereas it would be possible to model the processes anyhow without the new modeling element as well. We argue that our modeling element eases the modeling of regarded scenarios. The modeling element supports to model and comprehend process models for real world processes more intuitively. Additionally the modeling element increases the flexibility of a scope definition by adding the modeling element `<restart/>` activity. The modeling element can also consider as adaptation support. Adaptation is triggered by the fault message. Adaptation is computed during the re-execution of the scope by choosing more suitable paths. In contrast to loops, the restart option of repeatable scopes is triggered exclusively by faulting situations, which

appear to be a more suitable modeling option to some scenarios. Further research needs to be done concerning the cross boundary link semantics. In this paper we provided the simplest way on how to deal with cross boundary link evaluation at runtime. Especially we will regard the question, how can cross boundary link evaluation be done with respect to performance concerns. As there are lots of possible way to do this, this was out of scope of this paper.

References

[1] Herrmann, K., Rothermel, K., Kortuem, G., Dulay, N.: Adaptable Pervasive Flows - An Emerging Technology for Pervasive Adaptation. In: Workshop on Pervasive Adaptation (PerAda) (October 2008)
[2] Norman, D.: The Design of Everyday Things. Owner inscription on fep edn. Doubleday Business (February 1990)
[3] Organization for the Advancement of Structured Information Standards (OASIS): Web Services Business Process Execution Language Version 2.0 (March 2007)
[4] Object Management Group (OMG): Business Process Modeling Notation (BPMN) Version 1.2 (January 2009), http://www.bpmn.org/
[5] Kloppmann, M., Koenig, D., Leymann, F., Pfau, G., Rickayzen, A., von Riegen, C., Schmidt, P., Trickovic, I.: WS-BPEL Extension for Sub-processes – BPEL-SPE. In: IBM, SAP (2005)
[6] Steinmetz, T.: Ein Event-Modell für WS-BPEL 2.0 und dessen Realisierung in Apache ODE. Diplomarbeit, University of Stuttgart, Faculty of Computer Science, Electrical Engineering, and Information Technology, Germany (August 2008)
[7] Karastoyanova, D., Khalaf, R., Schroth, R., Paluszek, M., Leymann, F.: BPEL Event Model. Technical Report 2006/10, University of Stuttgart, Faculty of Computer Science, Electrical Engineering, and Information Technology, Germany (2006)
[8] Leymann, F.: Supporting Business Transactions Via Partial Backward Recovery In Workflow Management Systems. In: BTW, pp. 51–70 (1995)
[9] Greenfield, P., Fekete, A., Jang, J., Kuo, D.: Compensation is Not Enough. In: EDOC 2003: Proceedings of the 7th International Conference on Enterprise Distributed Object Computing, Washington, DC, USA, p. 232. IEEE Computer Society, Los Alamitos (2003)
[10] Liu, A., Li, Q., Xiao, M.: A declarative approach to enhancing the reliability of bpel processes. In: IEEE International Conference on Web Services (ICWS 2007), pp. 272–279. IEEE Computer Society, Los Alamitos (2007)
[11] Altintas, I., Barney, O., Jaeger-Frank, E.: Provenance Collection Support in the Kepler Scientific Workflow System. In: Moreau, L., Foster, I. (eds.) IPAW 2006. LNCS, vol. 4145, pp. 118–132. Springer, Heidelberg (2006)
[12] Bharathi, S., et al.: Characterization of Scientific Workflows. In: Proceedings of the 3rd Workshop on Workflows in Support of Large-Scale Science, WORKS (2008)
[13] Akram, A., Meredith, D., Allan, R.: Evaluation of bpel to scientific workflows. In: CC-GRID 2006: Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid, Washington, DC, USA, pp. 269–274. IEEE Computer Society, Los Alamitos (2006)
[14] IBM: MQSeries Workflow
[15] Chiu, D.K.W., Li, Q.: A Meta Modeling Approach for Workflow Management System Supporting Exception Handling. Information Systems 24, 159–184 (1999)
[16] Russell, N., van der Aalst, W.M.P., ter Hofstede, A.H.M.: Workflow Exception Patterns. In: Dubois, E., Pohl, K. (eds.) CAiSE 2006. LNCS, vol. 4001, pp. 288–302. Springer, Heidelberg (2006)