SH-BPEL - A Self-Healing plug-in for Ws-BPEL engines

ABSTRACT

Self-Healing is an emerging exigence for Information Systems where processes are more and more complicated and where many autonomous actors are involved. Self-healing mechanisms can be viewed as a set of automatic recovery actions fired at run-time according to the detected fault. These actions can be at infrastructure level, i.e., transparently to the process, or they can be defined in the workflow model and executed by the workflow engine. Standard recovery mechanisms provided by Ws-BPEL are not enough to implement with reasonable effort lots of suitable recovery actions. The aim of this paper is to present a Self-Healing plug-in for a Ws-BPEL engine that enhances the ability of a standard engine to provide process-based recovery actions.
Terms: H4.1 Workflow systems, business processes

1. INTRODUCTION

In Service Oriented Architectures languages and framework for managing processes are becoming mature and quite stable. Several engines [1, 2, 3] are available for executing processes written in Ws-BPEL, the de-facto standard for defining Web-Service based processes [7].

An emerging need for research in this field is an advanced management of faults to realize Self-Healing Information Systems able to choose and implement proper recovery actions. A Self Healing Information System needs a monitoring part for capturing faults, a diagnosis part to detect what/where is the source of the problem and to decide which recovery action has to be fired, and a recovery part that actually implements the recovery action.

One of the basic parts in a Self-Healing architecture is the process engine. Thus the engine has to provide some recovery actions, implementing both the possibility of having an internal module with the recovery logic and the possibility of exposing a management interface that gives to external actors, having the proper privileges, the possibility of asking specific recovery actions. The idea of writing a new web-service based workflow language has been discarded for sake of generality and for providing something an easy add-on to existing business solutions. In the approach presented in this paper, no modification on the standard Ws-BPEL language is required, and only some annotations, used to generate a standard and thus executable Ws-BPEL schema, are needed for enabling recovery actions.

Three kinds of recovery actions are available:

- *S*tandard Ws-BPEL recovery mechanisms. These mechanisms are provided by the Ws-BPEL standard language and are fully specified by the designer. They are automatically executed by the Process Engine.
- *P*re-Processing based Ws-BPEL recovery mechanisms. These mechanisms are realized by combining standard Ws-BPEL activities and handlers. To insert these mechanisms in the original Ws-BPEL process specification, a pre-processing phase interprets the annotated Ws-BPEL modifying it in a way that preserves the business logic, but enables specific recovery actions. They cover a wide range of possibility: i) the ability of modifying the value of process variables by means of external messages, ii) the ability of redoing a single Task or an entire *Scope*, iii) the possibility of specifying alternative paths to be followed, in the prosecution of the execution, after the reception of an enabling message, iv) the possibility of going back in the process to a point defined as safe for redoing the same set of tasks or for performing an alternative path.
- *E*xtended recovery mechanisms. These mechanisms are executed without the intervention of the Process Engine, therefore are hidden to the process execution and are managed by a specific management module. Extended recovery mechanisms allow, for instance, substitution of dynamically selected services in case they do not respond within a timeout threshold.

In this paper we present Sh-BPEL, a Self-Healing plug-in to realize the above mentioned recovery mechanisms. The available Ws-BPEL engines usually provide a set of APIs for directly interacting with the engine. Our approach uses this kind of APIs and, specifically, it is based on the ActiveBPEL engine [3]; a specific interface manager is devoted to translate general mechanisms into commands for the specific engine.

The paper is structured as follows: in Section 2 we discuss recovery mechanisms; Section 3 shows the SH-BPEL architecture; Section 4 presents Recovery Mechanisms realization in SH-BPEL. Section 5 discusses related work.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...$5.00.

Fault handler	Compensation handler	Event handler
Its aim is to undo the partial and unsuccessful work of a *Scope*. The first thing it does is to terminate all the activities contained within the *Scope*. If a fault is not consumed in the current *Scope* it is recursively forwarded to the enclosed ones. If termination of instance has not been invoked, after consuming the exception the normal flow restarts at the end of the *Scope* associated with the fault.	This is basically a wrapper for compensation activities. A Compensation Handler is available only after the related *Scope* has been completed correctly. It represents a compensation process of already executed activities. Activities see a snapshot of all the variables of the *Scope* the Compensation Handler is attached to and they cannot update live data. It can be used from within fault-handlers or Compensation Handlers.	The whole process as well as each *Scope* can be associated with a set of Event Handlers that are invoked concurrently if the corresponding event occurs. The actions taken within an Event Handler can be any type of activity. Event Handler is considered a part of the normal processing of the *Scope*, i.e., active Event Handlers are concurrent activities within the *Scope*. Events can be: i) incoming messages; ii)temporal alarms.

Table 1: Overview of Ws-BPEL recovery mechanisms

2. RECOVERY MECHANISMS

Ws-BPEL provides a set of basic constructs to define both exceptions and recovery strategies. The rationale is that designers, who have a perfect knowledge of the process, are able to define a process flow that contains all the strategies for recovering from faulty states. In this way, at runtime the Ws-BPEL engine recovers from errors.

Unfortunately, the optimistic assumption that designers have a complete knowledge of the *world* cannot be considered true. This implies that very often, Ws-BPEL processes have a flow that does not contain all the required repair strategies, but instead throw exceptions that have to be handled by external actors in charge of solving faulty situations. Furthermore, even if the assumption could be considered true, the definition of recovery strategies in complex processes is a hard design task, which could easily lead to the creation of too complex, unmanageable, and hardly executable process flows.

Another weak point of using basic Ws-BPEL constructs is that they are quite simple and do not allow designers to define sophisticated recovery actions that could involve, for example, rollbacks, execution of alternative patterns, or Web service substitutions

These limitations can be overcome using two different strategies: augmenting the number and quality of the Ws-BPEL constructs or extending the Ws-BPEL engine with one or more modules to execute sophisticated recovery strategies.

In the following, we describe the Ws-BPEL standard exception handling and illustrate the details of our idea of augmenting fault recovery capabilities combining annotations associated to Ws-BPEL constructs together with ad-hoc recovery modules which realize management operations associated with the process.

2.1 Standard Ws-BPEL recovery mechanisms

In the Service Oriented Computing world Ws-BPEL [7] is the most used language to specify Web-Service Orchestration. It defines a language to design workflow processes by specifying the process tasks and their interaction with the external world. Ws-BPEL has standard activities (*invoke*, *receive*, *assign*, *wait*, *reply*, *terminate* etc.) called simply tasks, and structured activities (*loop*, *pick*, *sequence* etc.) that can be combined to define the process.

An important concept is the *Scope* that defines a portion of workflow upon which variables and handlers can be defined. Each *Scope* has a primary activity that defines its normal behavior. The primary activity can be a complex structured activity, with many nested activities with an arbitrary depth. The *Scope* is shared by all the nested activities. Each *Scope* has unique entry and exit points.

Ws-BPEL provides standard mechanisms for managing exceptions. Specific handlers (fault, compensation and event) are associated with a single action or with a *Scope*, that is, a set of actions. The major problems of handlers is their lack of flexibility. Those handlers are enabled at different time during execution: fault and event handlers are enabled only during running time (of a Task or *Scope*); Compensation Handler is enabled when the status is “completed” and therefore the execution point is ahead of the *Scope*. The fault and compensation can be defined both at Task or *Scope* level, but Event Handler exists only at *Scope* level. Table 1 shows a brief description of each handler.

Using these three handlers as they are, it is possible to realize very basic “recovery” mechanism. In fact Ws-BPEL provides the three handles as standard mechanisms and leaves to the designer any other specification about the tasks actually executed when a handler is fired. Therefore more powerful and flexible instruments could be built, but this effort is currently fully in charge of the designer. For instance, no native method is available for redoing an action, designer could use the Compensation Handler to do this, but it is not so easy.

Moreover in developing advanced recovery mechanisms, the designer has to consider the side effect of each specific handler. For instance when *Fault Handler* is invoked, it terminates all the activities in the *Scope* upon which it is defined, therefore if the designer wants to provide a repair action on the process without killing a part of it (or all), he needs to catch the fault with the *Event Handler* that is executed concurrently to the normal flow. It does not kill any activity, but, at the same time, it does not stop the process flow.

From our perspective all faults related to a specific operation and internally managed by a fault handler are not really faults. In the following, we will focus instead on recovery mechanisms which are not part of the process specified at design time and which allow automatic or semiautomatic recovery from unexpected faults.

2.2 Pre-Processing based Ws-BPEL recovery mechanisms

Pre-Processing based Ws-BPEL recovery mechanisms are based on a transformation of the original Ws-BPEL process. They could be specified by the designer, but the effort required becomes too cumbersome. For this reason, following

the approach presented in [15] designers can annotate Ws-BPEL file for requiring a preprocessing phase generating a standard Ws-BPEL supporting specific, and more powerful recovery actions exported through a Management Interface. The annotations are based on the XML nature of Ws-BPEL, actually new tags are inserted in the XML during the design phase and then removed during the preprocessing phase.

The idea upon which this pre-processing phase is based is that combining in a proper way existing Ws-BPEL constructs it is possible to obtain specific behaviors realizing specific recovery actions. The mechanisms are fired by an external module that, according to a recovery logic, asks the process to perform some recovery actions.

The output of the pre-processing phase is a standard Ws-BPEL, that is a Ws-BPEL executable on a normal Ws-BPEL engine. After the deploy of this enriched Ws-BPEL, the process will export in its WSDL both the operations related to the "business logic" (functional interface) and the operations related to the "management logic" (management interface). The idea, described in Section 3.1, is to maintain on the standard Ws-BPEL engine a unified interface, using modules of our plug-in for splitting Business methods and Management methods in their presentation to the external world. Management methods will also be enriched by other features. In this way we follow the proposal of WSDM [4] which allows differentiation among functional and management operations and focuses on providing an infrastructure to support management operations.

Proposed operations provided by the Management Interface cover a wide range of possibility:

- *m*odifying the value of process variables by means of external messages. The basic idea behind this mechanism is to use several Event Handlers with a simple activity each one modifying the corresponding variable.

- *r*edoing a single Task or an entire *Scope*. In this mechanism redo is viewed as a compensation activity. The basic idea is to define a behavior that syntactically is inserted in a Compensation Handler, but actually is the repetition of the "normal" behavior. To activate this compensation, an Event Handler fired by the specific redo message is used.

- *s*pecifying alternative paths to be followed, in the prosecution of the execution, after the reception of an enabling message. The idea behind this mechanism is to have some alternative behaviors available along the process and related to several portions of it. The preprocessing phase uses a *Switch* activity to store all the possible alternative behaviors in the corresponding places, specific variable drives one and only one of this kind of *Switch*. Each alternative behavior is fired (or killed) by a specific message managed by an Event Handler.

- *g*oing back in the process to a point defined as safe for starting redoing the same set of tasks or for performing an alternative path. A common recovery action in workflow exception management is to compensate a part of the executed task rolling back the state to a "safe point". By using the simple Compensation Handler provided by standard Ws-BPEL it is possible to compensate a *Scope*, but the execution flow could proceed only ahead. This mechanism combines the side effect of fault handler with a *While* construct and a set of *Switch* for driving the going back and restarting of the flow.

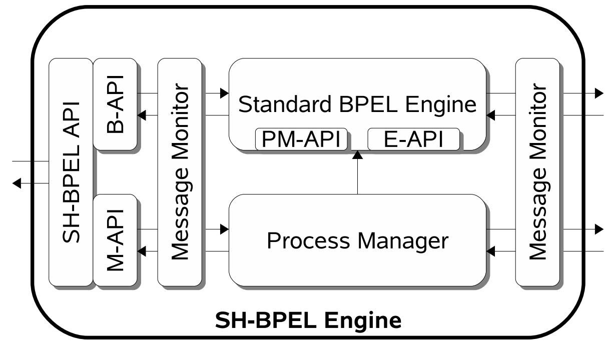

Figure 1: The architecture of the SH-BPEL Engine

2.3 Extended recovery mechanisms

The third possibility for performing recovery actions is exploited using an external module with respect to the Ws-BPEL engine. This choice overcomes both the Ws-BPEL model and engine limitations. A typical situation where it is used is when a retry operation has to be performed as reaction to a timeout. In this situation it can try to substitute the invoked service with another one, and, under certain condition, this operation can be totally hidden for the process engine, since it is not part of the process logic but only linked o the dynamic binding logic.

3. THE SH-BPEL ENGINE

This section describes the architecture of our enhanced Ws-BPEL engine. The objective is to give an overview of the main architectural modules and introduce the functionalities that each module provides. The description is divided into two parts: in the first one the focus is on the architecture of the SH-BPEL Engine, while in the second one we enter into the details of the module responsible for the enactment of the management actions.

3.1 The Architecture

Figure 1 shows the SH-BPEL Engine, where SH stands for *Self Healing*. The core modules are the *Standard Ws-BPEL Engine* and the *Process Manager*, which realize the main functionalities. The other components are interfaces to export the engine functionalities and filters to monitor exchanged messages.

In detail, there are four main components in the SH-BPEL Engine architecture: the *Standard BPEL Engine*, the *Process Manager*, the *Message Monitor*, and the *SH-BPEL API*.

- **Standard BPEL Engine**. This module is the standard BPEL engine to be enhanced. For us, a standard engine is a a BPEL4WS 1.1 compatible engine that has the ability of executing precesses defined using that version of the language. The interaction with this module is possible using the *B-API*, the *PM-API*, and the *E-API*. The B-API (Business API) contains the business methods of the deployed BPEL process. These methods depends on how the process has been designed and are usually invoked by the users. The PM-API (Process Management API) contains the management methods that can be used to control a process

instance. The implementation of these methods is provided by our architecture but it is a design task deciding whether to use those methods inside a process or not. The E-API (Engine API) contains the methods that allow controlling the BPEL engine. Different BPEL engine implementations (e.g., ActiveBPEL or BPEL4J) may have different E-API;

- **Process Manager**. This module is responsible for enacting the management actions over the Standard BPEL Process. Management actions can influence a single process instance, several instances of the same process, but also instances of different processes. The Process Manager can work in three different ways: using the PM-API, using the E-API, or acting over the messages that flow between the BPEL engine and the invoked Web services. The difference is that, using the third solution, the Process Manager performs management actions that are transparent to the Standard BPEL Engine. The Process Manager can be controlled via the *M-API* (Management API).

- **Message Monitor**. The SH-BPEL Engine has two monitoring modules. One for the incoming messages and the other for the outcoming ones. The purpose of those filters is to analyze exchanged messages and generate events when defined patterns are recognized. The configuration of the filters (i.e., the definition of the patterns they recognize) and the subscription to receive the desired events is performed using the *M-API*.

- **SH-BPEL API**. This API is the functional interface of the SH-BPEL Engine. As described in Figure 1, this interface is derived combining the B-API with the M-API. The SH-BPEL API provides a homogeneous way to access the SH-BPEL Engine. Since it would not be a good practice to let generic users invoke management methods, the SH-BPEL API comes with access right management functionalities that restrict the access to the M-API.

3.2 The Process Manager

As introduced in the previous Section, the Process Manager (see Figure 2) is responsible for the enactment of the management actions.

The *Management Engine* is the component that executes management actions and the available management operations are published via its *Management Interface*, which corresponds to the *M-API* presented in Section 3.1 The Management Engine can work in both an *active* and a *passive* mode.

In the passive mode, the Management Engine does not perform any management action unless explicitly ordered via the M-API. On the contrary, when in the active mode, the Management Engine may decide to perform management actions without the need of receiving explicit orders. The way in which the Management Engine selects and executes management actions while it is in the active mode is defined in its configuration. Using the M-API it is possible to modify such a configuration.

As an example, consider a situation in which an invoked Web service has failed sending to the Standard BPEL engine a response message formatted with a wrong XSD Schema.

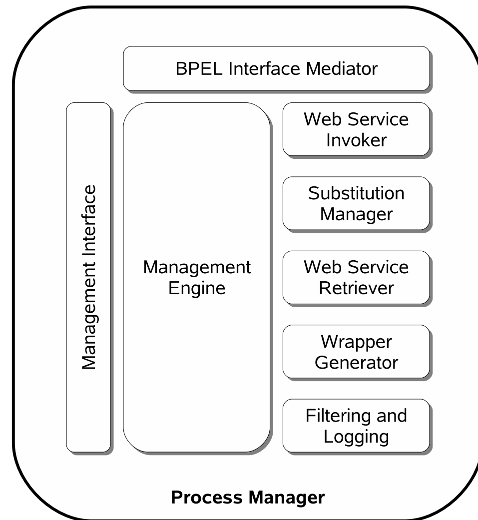

Figure 2: The architecture of the Process Manager

Currently the standard ActiveBPEL engine will raise an exception at level of engine whereas in our architecture the Message Monitor notices the problem and generates a notification message that is sent both to the Management Engine. Therefore it would be possible to retry the invocation or to substitute the failed Web service. If the Management Engine is in the passive mode, it simply ignores the notification and waits for an external decision which can be a request for a retry or for a substitution. Otherwise, if it is in the active mode, the Management Engine executes the management action connected to the received notification (e.g., the service substitution).

In our architecture, the Management Engine does not perform management actions directly but, instead, uses a set of *support modules*. The fact that the Management Engine relies on support modules implies that its management capabilities depends on the functionalities of the used modules. The advantage of this solution is having a system in which single management modules can be updated, added or removed without having to implement the Management Engine from scratch every time. The support modules are:

- **Web Service Invoker**. This module provides functionalities to dynamically invoke Web services. The invocation is performed independently from the WSDL interfaces of the Web services and guarantees flexibility, since it avoids the creation of stubs;

- **Substitution Manager**. This module allows the substitution of Web services. It provides functionalities to substitute Web services that are already used by the BPEL engine with new ones;

- **Web Service Retriever**. This modules provides operations to search Web services. In order to have a dynamic substitution of Web services it is mandatory to have a module that enables the dynamic discovery of Web services;

- **Wrapper Generator**. Sometimes, substituted Web services do not have same WSDL interface of the substituted ones. This could imply that the standard

BPEL engine is not able to communicate with the new Web services because it cannot recognize their messages. The solution is the Wrapper Generator, which has the capability of transforming hiding the differences between the old and the new WSDL documents;

- **Filtering and Logging**. Very often, during the execution of management actions it is necessary to keep trace of what it is happening. This module provides functionalities to trace and filter events and messages exchanged during the management operations. Furthermore, this module represents the bridge between the Message Monitor and the Message Monitors introduced in Section 3.1

- **BPEL Interface Mediator**. It is used by the Management Engine to access the PM-API and the E-API of the Standard Ws-BPEL Engine. The mediator is necessary since these interfaces may change according to both the particular process instance and the particular Ws-BPEL engine implementation, and thus an adaptation layer is needed.

4. RECOVERY STRATEGIES & SH-BPEL ENGINE

This section is devoted to present how the recovery strategies depicted in Section 2 can be leveraged using the Sh-BPEL architecture. As already pointed out, recovery strategies directly expressed in a fault handler by the designer are treated as normal activities during the process. Their management does not involve self-Healing plug-in.

4.1 Process Manager

In this case the Process Manager has the logic for choosing the specified recovery action and then, after the Message Monitor captures the needed information about the fault, it implements the right strategy for performing the recovery action. It will interact with the standard Ws-BPEL engine by the E-API and the PM-API. The Process Manager comes with a built-in set of recovery strategies that can be personalized via programmable interface.

4.2 External With Privileges

Business scenarios are quickly moving from a centralized and orchestrated approach to a choreographed one. In a choreographed environment several workflow engines cooperate for realizing a global business process. It is possible to envisage a scenario where many actors able to implement recovery strategies communicates at different levels: for business, for diagnosis and for repair. We call this scenario a Self-Healing environment. Figure 3 shows a scenario where three actors are able to communicate for business and for diagnosis, this is the situation where we are going to test and exploit the Sh-BPEL engine. There, each actor is composed of three different modules. The choice of having different modules is for portability and flexibility, because some actors in a scenario of this kind could not be equipped with all modules. In Figure 3 a specific module for each actor is in charge of choosing the proper recovery action in according to the result of diagnosis.

To support this possibility SH-BPEL engine exposes a set of management API (the M-API) that allows external actor (In Figure 3 they are the recovery and diagnosis modules of the same actor) to interacts with the engine for performing recovery action. The external module has to have the right privileges for firing recovery methods that are interpreted by the process manager for translating them in the proper set of commands for the PM-API and the E-API of the standard Ws-BPEL engine.

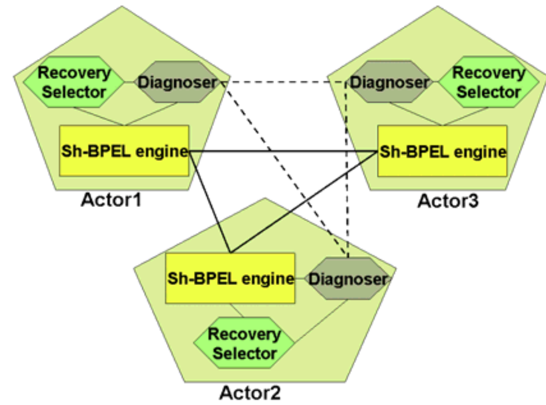

Figure 3: SH-BPEL Engines in a Self-Healing Environment

The concept of active and passive mode will be extended also at this situation where each actor can have its own recovery logic or can be a "passive" actor that is involved in recovery strategies decided outside by someone else.

4.3 Instance & Class Repair Actions

As already pointed out the use of plug-in allows to define both "Instance" and "Class" repair actions, being the former related to a single run of a Business Process and the latter the set of all instances (already performed, running, and future) of a specific process of a specific provider. For this reason the M-API are divided in two different groups, the first one related to Instance repair action and the other one to Class repair actions, the latter will affect all the running and future instances of the specified process of a given provider. The firing of Class repair actions is in charge of externals modules that, according to their policies, at a given time, choose to enable a recovery action valid for all the instances of a given provider.

5. EXISTING APPROACHES

Recovery actions for workflow systems have been widely studied in the past. The work in [6, 8, 17] present specific workflow models that widely support recovery actions; in [12] the authors focus on the analysis, prediction, and prevention of exceptions in order to reduce their occurrences. The model presented in [13] focuses on the handling of expected exceptions and the integration of exception handling in the execution environment, while in [5] the authors propose the use of "worklets", a repertoire of self-contained subprocesses and associated selection and exception handling rules to support the modelling, analysis and enactment of business processes. The work in [9] presents the requirements of a Web Service Management framework which also includes the typical functionalities addressed in self-healing systems. The authors analyze and compare multiple alter-

native architectures for the implementation of Web Service Management systems proposing Web service substitution and complex service re-compositions as repair actions.

In addition, an extensive amount of work on flexible recovery in the context of advanced transaction models has been done, e.g., in [11, 18]. They particularly show how some of the concepts used in transaction management can be applied to workflow environments.

The approach presented in [16] is mainly related to adaptivity in Information Systems, but some solutions based on the concept of proxi, under certain conditions, enables the reaction to faults in a way totally hidden to the process engine. Similar features, this time explicitly called recovery actions, are presented in [10].

In [14] the authors consider a set of recovery policies both on task and region of a workflow. They use an extend Petri Net approach to change the normal behavior when an expected but unusual situation or failure occurs. As in our approach the recovery policies are set at design time.

6. CONCLUSIONS AND FUTURE WORK

In this paper we have presented the Architecture of SH-BPEL engine, a Self-healing Process engine based on a standard Ws-BPEL engine. Self Healing is becoming a real exigence for information systems, and sophisticated recovery strategies are necessary for coping with the highly dynamic and complex business environment. Our proposal is to build a plug-in for standard Ws-BPEL engines, avoiding any modification of the exiting ones. Three different level of recovery mechanisms are available: i) the standard ones that are fully defined by designer and are out of paper focus, ii) pre-Processing based Ws-BPEL recovery mechanisms that, according to designer annotation combine standard Ws-BPEL activities for exploiting powerful recovery actions and that can be executed on a standard Ws-BPEL engine, iii) extended recovery mechanisms that are managed outside the standard Ws-BPEL engine by a specific module.

Recovery actions can be performed autonomously storing their logic inside the Sh-BPEL engine or they can be chosen according to more complex policies that are implemented outside of the SH-BPEL engine.

Moreover SH-BPEL engine supports the possibility of exploiting the recovery action at class level, that is for all running and future instances.

The actual implementation of the presented architecture is under development and specific care is being taken for internal modules of Process Manager for ensuring the maximum portability and flexibility both for standard Ws-BPEL engine evolution and for new kind of recovery actions that could be developed. A first prototype which allows executing basic recovery operations based on pre-processing, such as going back to safe points and redo actions has been developed using ActiveBPEL as standard Ws-BPEL process engine.

As future work we are studying an advanced use of SH-BPEL engine in self-healing environments, that is how to manage the interaction among different actors cooperating in a choreographed process and being in "repair mode" at the same time.

7. REFERENCES

[1] http://www.alphaworks.ibm.com/tech/bpws4j, 2002.

[2] http://www.oracle.com/technology/products/ias/bpel/index.html, 2003.

[3] http://www.activebpel.org, 2004.

[4] http://ws.apache.org/muse/index.html, 2005.

[5] M. Adams, A.H.M. ter Hofstede, D. Edmond, and W.M.P. van der Aalst. Facilitating flexibility and dynamic exception handling in workflows through worklets. In *Short Paper Proceedings at (CAiSE)*, volume 161 of *CEUR Workshop Proc.*, Porto, Portugal, 2005.

[6] F. Casati, S. Ceri, B. Pernici, and G. Pozzi. Workflow evolution. *Data Knowl. Eng.*, 24(3):211–238, 1998.

[7] F. Curbera, Y. Goland, J. Klein, F. Leymann, D. Roller, S. Thatte, and S. Weerawarana. *Business Process Execution Language for Web Services, v.1.0*, 2002.

[8] J. Eder and W. Liebhart. Workflow recovery. In *Proc. of IFCIS Int. Conf. on Cooperative Information Systems (CoopIS)*, pages 124 – 134, Brussels, Belgium, 1996. IEEE.

[9] E. Esfandiari and V. Tosic. Towards a web service composition management framework. In *Proc. of Int. Conf. on Web Services*, Orlando FL, USA, 2005.

[10] T. Friese, J.P. Müller, and B. Freisleben. Self-healing execution of business processes based on a peer-to-peer service architecture. In *Proc. of Int. Conf. on Architecture of Computing Systems (ARCS)*, pages 108–123, Innsbruck, Austria, 2005. Springer.

[11] D. Georgakopoulos, M.F. Hornick, and F. Manola. Customizing transaction models and mechanisms in a programmable environment supporting reliable workflow automation. *IEEE Trans. Knowl. Data Eng.*, 8(4):630–649, 1996.

[12] D. Grigori, F. Casati, U. Dayal, and M.C. Shan. Improving business process quality through exception understanding, prediction, and prevention. In *Proc. of Proceedings of Int. Conf. on Very Large Data Bases (VLDB)*, Roma, Italy, 2001.

[13] C. Hagen and G. Alonso. Exception handling in workflow management systems. *IEEE Trans. Software Eng.*, 26(10):943–958, 2000.

[14] R. Hamadi and B. Benatallah. Recovery nets: Towards self-adaptive workflow systems. In *Proc. of Int. Conf. on Web Information Systems Engineering (WISE)*, pages 439–453, Brisbane, Australia, 2004.

[15] S. Modafferi and E. Conforti. Methods for enabling recovery actions in ws-bpel. In *Proc. of Int. Conf. on Cooperative Information Systems (CoopIS)*, Montpellier, France, 2006.

[16] B. Pernici (Ed). *Mobile Information Systems Infrastructure and Design for Adaptivity and Flexibility.* Springer, 2006.

[17] M. Reichert, S. Rinderle, U. Kreher, and P. Dadam. Adaptive process management with ADEPT2. In *Proc. of Int. Conf. on Data Engineering ICDE*, pages 1113–1114, Tokyo, Japan, 2005.

[18] H. Wächter and A. Reuter. The ConTract model. In A.K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, pages 219–263. Morgan Kaufmann, 1992.