On the Enhancement of BPEL Engines for Self-Healing Composite Web Services

Sattanathan Subramanian\textsuperscript{1}, Philippe Thiran\textsuperscript{2}, Nanjungud C. Narendra\textsuperscript{3}, Ghita Kouadri Mostefaoui\textsuperscript{4}, and Zakaria Maamar\textsuperscript{5}

\textsuperscript{1}INRIA Saclay-Île-de-France, Orsay, France, sattanathan.subramanian@inria.fr
\textsuperscript{2}Precise, University of Namur, Namur, Belgium, pthiran@fundp.ac.be
\textsuperscript{3}IBM India Research Lab, Bangalore, India, narendra@in.ibm.com
\textsuperscript{4}Comlab, Oxford University, Oxford, UK, ghitak@gmail.com
\textsuperscript{5}College of Information Technology, Zayed University, Dubai, UAE, zakaria.maamar@zu.ac.ae

Abstract

The dynamic nature of the Internet poses various challenges to the successful execution of composite Web services. Failures are samples of these challenges. It needs to be addressed for the smooth progress of Web service composition. Unfortunately, the de facto standard for modeling composition namely BPEL is not equipped with mechanisms that let Web services "heal" themselves in case of failures. In addition, current BPEL engines lack appropriate facilities that permit satisfying self-healing requirements like handling in a transparent way functional failures of component Web services during runtime. This paper presents an approach for enhancing BPEL engines with such facilities. This enhancement happens through the following steps: identify possible categories of failures, develop solutions to automatically recover from these failures, and suggest architectural extensions to BPEL engines (with focus on ActiveBPEL) to support these solutions. We also present a proof-of-concept prototype that illustrates our ideas.

Keywords. BPEL, Composition, Self-healing, Web service.

1. Introduction

Modern B2B applications requirements stress the need to compose Web services, which results in developing composite Web services. Simply put, composition is about making independent Web services interact with one another according to a specific business logic [7]. Different specification languages of composition exist including the Business Process Execution Language (BPEL) [1] and the Web Service Choreography Interface (WSCI) [4]. Although BPEL is the de facto standard for Web services composition, current engines that implement BPEL specifications suffer from a major limitation. Indeed these engines do not allow changes to take place in the middle of a composition specification execution-stream. This makes designers envisage alternatives to address all possible failures, which quickly turns out to be cumbersome and quite impossible [15]. In this paper, we show how a BPEL engine such as ActiveBPEL \textsuperscript{1} can be enhanced with facilities to be qualified in this paper by self-healing. Such facilities permit overseeing Web services execution, identifying corrective strategies in case of failures, and implementing these strategies.

In a dynamic environment such as the Internet, applications built around software components for example Web services can be subject to unexpected failures. By unexpected, we mean failures for which recovery strategies were not planned at design time, even if the risk of failure occurrence was identified and probably minimized. Nowadays, developers are pressured and put on the front line of satisfying the promise of Web services’ providers to deliver a new generation of B2B applications. This pressure makes developers release applications without complete testing, relying on the next versions to fix all the reported failures. In general, failures can be unexpected for developers but not for software systems. A software system can handle all kinds of failures by just throwing exceptions that disrupt its normal functioning. In [9], Chan et al. mention that failures can be detected by software, since most failures are capable of producing some error message or undesired result. Unexpected failures could have a severe impact on businesses by making them suspend operation for a couple of days and sometimes weeks. In [12], He reports on the importance of failure recovery in Web services applications by describing a real incident that affected eBay. The company lost $5 million due to a 22-hour server-outage in April 2002. In this paper we identify the necessary steps to take in order to enhance Web services with self-healing capabilities, so they could become “immune” to unexpected failures. Failures mean here results of events, such as resource unavai...
ity or overload, that prevent Web services from successful completion. In addition, self-healing should make Web services maintain operation and recover to normal levels of operation after failures. Our approach is to predict potential unexpected failures, assess their impact, take corrective actions, and resume operation. Because Web services’ strengths revolve around composition, we discuss how to anchor self-healing mechanisms to a BPEL engine, namely ActiveBPEL. Current BPEL recovery mechanisms are not suitable for developing self-healing Web services. For instance, rollback, execution of alternative patterns, or substitution are not offered. More details on and limitations in BPEL are discussed in [1] and [15], respectively. Our contribution versus other works [10, 11, 15] is to make the self-healing functionality an intrinsic part of a BPEL engine.

Section 2 presents a running example. Section 3 classifies failures related to composite Web services. Solutions to tackle these failures are listed in Section 4. Moreover, this section discusses the integration of self-healing mechanisms into ActiveBPEL. Prior to concluding in Section 7, proof-of-concept and related work aspects are reported in Sections 5 and 6, respectively.

2. Travel-composite-service scenario

Using the Travel Composite Service (TCS) scenario, we identify the necessary components to develop self-healing Web services. In this scenario, a client wishes to arrange a trip from India to Belgium using Flight/Taxi/Hotel WSs for flight/taxi/room booking and cancellation, and Bank WS for paying these services. The way these Web services are identified does not fall into this paper’s scope.

Flight WS implements two operations: ReserveFlight and CancelFlight. ReserveFlight accepts and returns a complex type object called travelPlan and flightDetails, respectively. CancelFlight accepts and returns a complex type object called flightDetails and cancelInfo, respectively. Taxi WS implements two operations: ReserveTaxi and CancelTaxi. ReserveTaxi accepts and returns a complex type object called travelPlan and taxiDetails, respectively. CancelTaxi operation accepts and returns a complex type object called taxiDetails and cancelInfo, respectively. Hotel WS implements two operations: ReserveRoom and CancelRoom. ReserveRoom accepts and returns a complex type object called travelPlan and roomDetails, respectively. CancelRoom accepts and returns a complex type object called roomDetails and cancelInfo, respectively. Finally, Bank WS implements two operations: ViewBalance and MoneyTransfer. ViewBalance accepts two complex type objects called accountInfo and loginInfo, and returns a complex type object called balanceInfo. MoneyTransfer accepts two complex type objects called accountInfo and loginInfo. MoneyTransfer returns a complex type object called transferConfirmation.

3. Failures hindering Web services

Using TCS scenario, we discuss the main failures that could hinder Web services execution. A first description of these failures is provided in works like [9] and [12] but from the perspective of an individual Web service. We use this classification to categorize failures from the perspective of a composite Web service comprising several component Web services. This would help in suggesting appropriate self-healing actions.

- **Fa₁**: Functional failure happens due to a software bug in a Web service that results in unsuccessful completion of the invoked operations in this Web service. This includes the faults such as incorrect order of messages, misbehaving execution flow, which are listed in [9]. In TCS, Flight WS’s ReserveFlight operation receives a message call along with travelPlan details. TCS expects flightDetails back from Flight WS, but due to a software bug, null-value (or empty) is returned.

- **Fa₂**: Operational failure is related to Web service unavailability due to hardware or software failure. This includes time-out fault, which is listed in [9]. In TCS, Flight WS’s ReserveFlight operation gets invoked but there is no response and exceeding the waiting time.

- **Fa₃**: Semantic failure occurs due to lack of understanding over the exchanged data among Web services. This includes faults such as misunderstood behavior (i.e. service requester that requests a service from a service provider, expecting a service different from the provided one) and incorrect service, which are listed in [9]. Since this type of semantic failure is outside this paper’s scope, readers are referred to [17].

- **Fa₄**: Privacy failure occurs due to access privileges violation for example on data. In TCS, Bank WS’s MoneyTransfer operation requires date-of-birth on top of account number and customer name, while other banking-related Web services require account number and customer name only. Privacy has to be taken care during design time as violations could arise. Fixing this type of failure requires some additional or updated details from the client during the run time.

- **Fa₅**: Security failure happens when a Web service fails to supply the necessary credentials (e.g., certificate) to a peer. In TCS, Bank WS’s MoneyTransfer operation requires loginInfo with username/password. According to a recent security policy in Bank WS, password must be changed monthly. Unfortunately, TCS supplies an old password, which does not meet this policy. Fixing this type of failure requires some additional or updated details from the client during the run time.
In addition to the aforementioned categories of failures, we identified two others. The first failure focuses on data mismatches between Web services. This type of failure is different from semantic failure since the focus is on data structure and not semantics. The second failure focuses on the non-functional aspects of Web service.

\textbf{Fa}_1: Data-match failure happens due to an upgrade or downgrade (or version changes) in a Web service as part of its regular adaptations. It is related to the input/output parameters (like data type, data values, etc.) of Web service operation. This type of failure includes faults such as parameter incompatibility, and response fault, which are listed in [9]. In addition, this type of failure includes cardinality discrepancies between Web services. Cardinality mean the number of elements contained in messages received from and sent to Web services. Usually, Web services return a set of responses following users’ requests, which are denoted as elements. Examples of element sets include search Web services (GoogleSearch\(^2\)) and e-commerce Web services (Amazon\(^3\)). Cardinality discrepancies manifest themselves when Web services have different requirements concerning the minimal and maximal number of elements that they can receive as input or return as output.

For example, in TCS, Flight WS’s ReserveFlight operation receives a message call along with travelPlan details. TCS expects flightDetails back from Flight WS. Due to a recent upgrade in Flight WS, newFlightDetails is submitted back, which results in data-mismatch failure. Here readers should note that this is not an operational failure because the Web service does reply back, but the data types it uses mismatches with the data types of the recipient Web service.

\textbf{Fa}_2: Non-functional failure happens following the violations of the service level agreements between Web services and between users and Web services [9]. Non-functional properties are captured at runtime and then compared with the announced properties. In TCS, Travel Composite WS agrees with Taxi WS to provide details at a fixed rate. However, due to an overload situation that required using extra resources, Travel Composite WS requests for a different rate, which results in a non-functional failure. Fixing this type of failure requires some additional or updated details (like agreement failure, QoS preference) from the client during the request-time/run time.

The different types of failures can also be categorized as per their impact, i.e., whether impacting a component Web service or the entire composite Web service. Component impact of type component means that the failure of a Web service does not affect peers. For instance, a Web service that suddenly goes down cannot put the blame on other Web services if there is no data/execution dependency. The composite impact is in contradiction with the component impact. A Web service fails because of the failure of another Web service upon which it depends. Functional (Fa1) and Operational (Fa2) failures affect both component and composite Web services, whereas the other failures affect only the composite Web service.

4. Self-healing Web service composition

Naccache et al. report that “a self-healing system must be able to recover from the failure of underlying components and services; the system must be able to detect and isolate the failed component, fix or replace the component, and finally, reintroduce the repaired or replaced component without any apparent application disruption” [16]. In this section, we detail our proposal for extending a BPEL engine with self-healing capabilities. We begin first by listing potential solutions to failures identified in Section 3.

4.1. Potential solutions to failures

Fig. 1 illustrates the categories of solutions that we suggest to address the failures which we listed in Section 3. The rationale behind our solution categories is to choose the solution that would cause the least disruption to the existing Web service composition, based on the severity of the failure. Fig. 1 shows the severity of impact of these solutions on the progress of a composition specification as well.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{figure1.png}
\caption{Proposed solutions to failures}
\end{figure}

\textbf{So}_1: Retrial is suitable for operational failures (Fa2) and aims at reinvoking a Web service a certain number of times. This solution severity is rated minor as it does not impact other Web services. Retrial can introduce the additional burden to the customer (example, double spending of payment) or Web service (example, duplicating the results) or both when there is a separate protocol (like payment protocol) for receiving the requests before reaching the intended Web service. So the retrial should be avoided when there is a separate protocol usage for remembering the service requests in the component Web service side. The composite Web service should have the knowledge about such protocol usage in component Web service, since this information is not explicitly published anywhere like WSDL.

\textbf{So}_2: Data mediation is suitable for data-match failures (Fa6) and aims at providing data in an appropriate form.

\(^2\)On reception of a user’s query, GoogleSearch Web service returns a set of elements that contain detailed information on the links provided.

\(^3\)Amazon Web service provides facilities for searching products and returns a set describing the selected products.
to recipient Web services. This solution severity is rated minor; it does not impact other Web services.

So.: Substitution is suitable for all categories of failures mainly functional and non-functional failures (F₁₁, F₇) and aims at replacing a failed Web service with a functionally equivalent Web service [14, 19]. This solution severity is rated average; other Web services (running in sequential or parallel to the failed Web service) can be affected when it comes for example on the agreed set of data to exchange.

So.: Process reorganization is suitable for all categories of failures (F₈, ₇) that cannot be handled with the three previous solutions. This solution severity is rated major; a complete or partial reorganization of the composition specification is required. For instance, the failed Web service is replaced with a group of Web services. For this, the process reorganization needs a complete knowledge about the failed business process.

4.2. BPEL engine enhancement

As part of the process of enhancing BPEL engines from a self-healing point-of-view, we started by developing a self-healing policy called sh-policy (Listing 1). Sh-policy gathers the following necessary information at run-time: (i) conditions that BPEL activities have to satisfy (Listing 1, Lines 02..10), (ii) BPEL activities to monitor (Listing 1, Lines 11), (iii) unexpected failures that affect BPEL activities (Listing 1, Lines 12..16), and (iv) solutions to these failures (Listing 1, Lines 17..34). Sh-policy has a 4-part structure: plan part details the BPEL activities to track during performance, monitor part details the unexpected failures and their root-causes relation to the BPEL activities, and recover part details the solutions to recover from the unexpected failures affecting the BPEL activities (as proposed in Section 4.1). Sh-policy’s plan and monitor parts are identified during BPEL process compilation, while sh-policy’s diagnose and recover parts are identified during BPEL process execution.

Figure 2. Self-healing operations

Listing 1. Sh-policy structure

01: <sh-Policy>
02: <Plan>
03:  <Activity name="">
04:  <pre><dataType /><range from="" to="" />
05:  <post><dataType /><range from="" to="" /></post>
06:  <Timeout />
07:  <NFPropertiesPreference><QoS /><SLA /></NFPropertiesPreference>
08:  </Activity>
09:  </Plan>
11:  <Monitor><Activity name="" /></Monitor>
12:  <Diagnose>
13:  <UnexpectedFailure name="" />
14:  <RootCause /><Solution />
15:  </UnexpectedFailure>
16:  </Diagnose>
17:  <Recover>
18:  <UnexpectedFailure name="" />
19:  <Retry enable="true/false" /><Timeout /><MaxTry />
20:  </Retry>
21:  <DataMediation enable="true/false" /><Utility name=""><Accessibility path=""></Utility>
22:  </DataMediation>
24:  <Substitution enable="true/false" />
25:  <Activity name="" />
26:  <NewWebService WSDL-url="" />
27:  <Activity />
28:  <Registry WSDL-url="" />
29:  </Substitution>
30:  <ProcessReorganization enable="true/false" />
31:  ! Alternative BPEL process --->
32:  </ProcessReorganization>
33:  </UnexpectedFailure>
34:  </Recover>
35:  </sh-Policy>

In compliance with the 4 parts of sh-policy, we associated them with modules namely planning, monitoring, diagnosis, and recovery. These modules along with the sequence of self-healing operations are described in Fig. 2. Planning module identifies the pre- and post-conditions (e.g., data type, range) and preferences over the non-functional properties per BPEL activity. This requires parsing a BPEL process and recording these conditions in the plan part of sh-policy. In addition, the planning module identifies the BPEL activities to monitor at runtime and records these details in the monitor part of sh-policy. This identification is based on the BPEL activities that have “direct” impact on a Web service output. For example, in TCS, Reserve a Taxi Web service, is to be monitored. This operation has the direct impact with the customer request. But this is not the same case with the other sub-activities like SearchForTaxi, SearchForBestFare of Taxi Web service. It is because these operations have less impact on the composition process. This way, a fine-grained monitoring is achieved and can be further customized to provide the best tradeoff between the precision of Web service monitoring and data traffic volume.

Monitoring module tracks the BPEL activities listed in the monitor part of sh-policy and ensures that these BPEL activities’ pre- and post-conditions and non-functional properties are satisfied. Plus, the monitoring module catches the unexpected failure(s) during the execution of BPEL activities and records the failure details in the diagnose part of sh-policy. The recorded details on a failure are: name of the encountered failure, name of the BPEL activity related to the failure, received root cause for the failure, and the expected type of solution for the failure.
Diagnosis module examines the unexpected failure that affected a BPEL activity. This failure is stored in the diagnose part of sh-policy. For this purpose, a database of possible failures is used. The failure database contains information about the failures affecting a BPEL engine at run-time. It is generated from the exception list of a BPEL engine. In general, the exception list is maintained as one or more files, and it is shared by the programs which are in execution for throwing exceptions in the case of failures. Table 1 shows some engine-specific exceptions of ActiveBPEL engine and its direct solutions mapping as a failure database. With the help of this database, the diagnosis module suggests solutions to the recover part of sh-policy. In case the database does not contain any solution, the BPEL process is stopped so that a manual intervention to fix the failure happens. This allows updating the database with new details as well.

Table 1. Some ActiveBPEL exceptions

<table>
<thead>
<tr>
<th>Failure category</th>
<th>Exception name</th>
<th>Reasons of exception</th>
<th>Proposed solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data mismatch</td>
<td>ActUnknownReplyRecei</td>
<td>Exception that is used to indicate for a given durable reply type was not found</td>
<td>Data-mediator, Substitution, Process-reorganization</td>
</tr>
<tr>
<td>Operational</td>
<td>ActFunction.allExcept</td>
<td>This exception is thrown when an exception is found during execution of a function</td>
<td>Substitution, Process-reorganization</td>
</tr>
<tr>
<td>Security</td>
<td>ActSecurityException</td>
<td>Exception related to security</td>
<td>Retrial, Substitution, Process-reorganization</td>
</tr>
<tr>
<td>Functional</td>
<td>ActMissingReplyRecei</td>
<td>Implements an exception that reports a missing reply receiver</td>
<td>Retrial, Substitution, Process-reorganization</td>
</tr>
</tbody>
</table>

Recovery module applies the solutions, which are available in the recover part of sh-policy, and updates the plan part of sh-policy. This update is necessary when the BPEL process is restructured due to lack of replacement to the failed BPEL activity. In some situations like compensating a Web service operation [8] or Web service substitution, it needs to consult with the client to find out their preferences such as cost of compensation, choice, etc.

4.3. The modified ActiveBPEL engine

Fig. 3 depicts the way we modify the architecture of ActiveBPEL engine so that self-healing mechanisms can be integrated. Our primary contribution is the SelfHeal-BPEL engine, which interacts with the ActiveBPEL’s management modules namely process, activity, and queues and alarms. SelfHeal-BPEL engine is a cumulative of the planning, monitoring, diagnosis, and recovery modules as shown in Fig. 2. A new BPEL process gets created when one of its start activities is triggered by either an incoming message or by pick activity alarm. When an incoming message contains failure, the SelfHeal-BPEL engine finds the existing process that matches the received input and suspends the running process. Later, the SelfHeal-BPEL engine provides a recovery solution to the failure and continues the execution of the suspended BPEL process. To diagnose unexpected failures, the SelfHeal-BPEL engine binds to the failure database.

5. Proof-of-concept

We developed a prototype to validate the proposed SelfHeal-BPEL engine using JDK 1.5 as a development language for SelfHeal-BPEL engine, XML and its parsers, Standard Widget Toolkit for graphical user interface, Apache Tomcat 5.5 as a back-end server, and...
Eclipse 3.2 as an integrated development environment. TCS is used for demonstration purposes. After receiving a request, TCS generates the appropriate BPEL process including details on the component Web services. Later, the planning module creates the plan part of sh-policy. To this end, it screens the BPEL activities in ReserveFlight, ReserveTaxi, ReserveRoom, and MoneyTransfer operations. Afterwards, the planning module identifies the activities that need to be monitored during execution. In this case, it considers all the BPEL activities. This information is later submitted to the monitor part of sh-policy for storage.

The monitoring module takes now the lead and tracks the activities that are in the monitor part of sh-policy. In this prototype, the user can fail/pass the running Activities. Consider that the user fails ReserveFlight Activity after invocation. After, Flight Web service throws an unexpected failure called "AeFunctionCallexception", it falls into Operational Failure category (Fa2). The failed Flight Web service is shown in Fig. 4. This situation is observed by the monitoring module before this activity reaches to the BPEL engine. The monitoring module updates the unexpected failure details in the diagnose part of sh-policy.

Finally, the recovery module uses the solution that is stored in the recover part of sh-policy. For the "AeFunctionCallexception" case, the recovery module suspends the running BPEL process and then, makes the process component in the ActiveBPEL engine adopt the solutions (i.e. retrial, substitution) which are proposed in this recover part. The process component resumes the BPEL process when the failed activity is fixed.

Figure 4. Execution status of TCS

The diagnosis module provides solutions to the "AeFunctionCallexception" failure by accessing the ActiveBPEL engine specific failure database. The provided solutions are submitted to the recover part of sh-policy, it is shown in the Fig. 4.

6. Related work

In one of the earliest works, Kephart and Chess present the concept of self-healing as a part of autonomic computing [13]. In their vision, self-healing systems will automatically detect, diagnose, and repair localized problems resulting from failures in software and hardware.

In [3], Ardissono et al. employ diagnostic reasoning techniques and diagnosis-aware exception handlers for exception handling. Some of the limitations in Ardissono et al.’s work include (i) an extra Web service is expected to monitor a component Web service, and (ii) a response to the question of “What will happen if local/global diagnosers fail” is required. In [5], Baresi et al. propose that a Web services execution environment should be able to select new Web services and even to reorganize the composition process to find a solution that uses available Web services, if a perfect match does not exist. This solution exploits probes that monitor the execution of the composition and suggest recovery activities to make the system continue its execution. However, this solution does not promote Web services substitution as a potential option. Web service substitution [19] is the only option when a component Web service totally fails to respond. In [2], Antonellis et al. present an approach to evaluate the compatibility among Web Services in order to support Web Service substitution.

In [6], Baresi et al. present an approach to supervise BPEL process by exploiting policies and aspects. The approach uses the Web Service Constraint Language (WS-CoL) and the Web Service Recovery Language (WSRel) to define the pre- and post-conditions at the composite and component levels and the recovery actions, respectively. However, this approach is only concerned with activities and not processes. In addition, it requires additional information from WSDL, which our approach does not require.

In [11], Gurguis and Zeid propose the concept of autonomic Web service. This one provides autonomic capabilities to other Web services so these ones can discover, diagnose, and react to unexpected events. An autonomic Web-service uses logs that a Web service submits. However, delays in providing logs and content sensitivity of logs hinder the functioning of autonomic Web services.

In [10], Ezenwoye and Sadjadi provide a proxy-based solution to BPEL as a framework for dynamic adaptation of composite Web services. This framework defines a generic proxy, which can identify or replace a component Web service. A composite Web service submits calls to the proxy in order to invoke a component Web service instead of directly calling the component Web service itself. This proxy solution increases the availability of Web service as long as the proxy is available to the composite Web service. Finally, in [15], Modafferi et al. propose different recovery mechanisms as BPEL extensions to be managed outside the BPEL engine, and a bayesian classifier is used to drive the
repair strategy selection in [18]. In contrast, our recovery capabilities can be weaved into the BPEL engine itself, in order to provide in-built self-healing capability to the engine.

7. Conclusions

Web services continue to play a major role in bridging businesses. Enhancing them with self-healing will boost their penetration rate in the IT field. On top of others’ research works, our work proposes a solution for extending BPEL engines, e.g., ActiveBPEL, with self-healing mechanisms that go along with recovering from unexpected failures. Our design decisions were driven by minimizing changes in Web-services composition specifications and providing adequate recovery solutions. In this paper, a self-healing policy was suggested in order to hold the following: (i) pre- and post-conditions of BPEL activities, (ii) list the BPEL activities to monitor during BPEL process performance, (iii) diagnose related information to unexpected failures of BPEL activities, and (iv) suggest recovery strategies for these failures. On top of this self-healing policy, we introduced Selfheal-BPEL engine as a new extension to ActiveBPEL architecture for supporting the execution of BPEL processes.

References


