ENABLING ROBUSTNESS IN EXISTING BPEL PROCESSES

Onyeka Ezenwoye and S. Masoud Sadjadi
School of Computing and Information Sciences
Florida International University
11200 SW 8th St., Miami, FL 33199

Keywords: ECommerce, Web Service Monitoring, Robust BPEL Processes.

Abstract: Web services are increasingly being used to expose applications over the Internet. To promote efficiency and the reuse of software, these Web services are being integrated both within enterprises and across enterprises, creating higher function services. BPEL is a workflow language that can be used to facilitate this integration. Unfortunately, the autonomous nature of Web services leaves BPEL processes susceptible to the failures of their constituent services. In this paper, we present a *systematic* approach to making existing BPEL processes more fault tolerant by monitoring the involved Web services at runtime, and by replacing delinquent Web services. To show the feasibility of our approach, we developed a prototype implementation that generates more robust BPEL processes from existing ones automatically. The use of the prototype is demonstrated using an existing Loan Approval BPEL process.

1 INTRODUCTION

Web services are becoming prevalent in the electronic marketplace and are often used to represent specific business functions (*e.g.*, ticket reservation). In order to create coarse grained business processes that constitute a number of related business functions, a high-level work flow language such as BPEL (Weerawarana and Curbera, 2002) is often used. However, due to the autonomous nature of the Web services, it is difficult for a third party user to ensure and check that a Web service will fulfill its requirements (functional or non-functional) (Robinson, 2003; Menasc, 2002). Consequently, the inherent openness of the interactions among Web services leaves BPEL processes susceptible to the failures of their constituent services. In this paper, we present a *systematic* approach to making existing BPEL processes more robust. By systematic, we mean that the code for robustness is generated and woven into the BPEL process; so the developers of the BPEL processes do not need to be aware of such code.

BPEL provides a coarse grain approach to building complex business processes by integrating separate and simpler Web services. The goal of BPEL is to improve corporate efficiency by promoting application integration and software reuse. Figure 1 depicts an example business process (Sales service) that involves four functional units. In this example, the activity of processing a purchase order from a customer involves the accounting, production, inventory and delivery services.

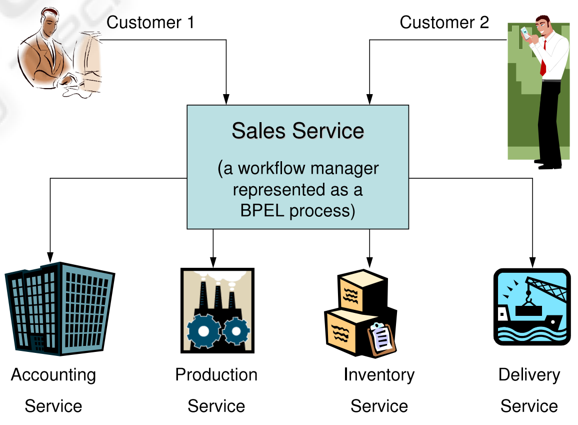

Figure 1: An example Sales Business Process.

To provide some robustness to BPEL processes, we need to monitor the behavior of the partner Web services. The problem of monitoring Web services is however compounded by the fact that traditional monitoring, debugging and testing tools do not suffice because some services may be outside the control of the developer of the BPEL process (Vogels, 2003). One type of monitoring that can be done on such autonomous Web services is to check that they fulfill their service contracts (Robinson, 2003). Existing methods of monitoring BPEL processes either do not fully separate the functional from the non-functional requirements of the process (Baresi et al., 2004) or do not maintain the portability of the BPEL process by extending the BPEL language or the BPEL engine (M. Blow et al., 2004).

In this paper, we show how an existing BPEL process can be made robust by automatically identifying and monitoring desired services and replacing them upon failure. The rest of this paper is structured as follows. Section 2 provides a background on the key technologies used in this paper. Section 3 overviews our approach and provides some detailed description. A case study using our prototype implementation is presented in Section 4. Section 5 contains related work. Finally, a discussion of future work and some concluding remarks are in Section 6.

2 BACKGROUND

The use of the term Web service is often overloaded and confusing. In its broadest definition, a Web service is a software component that is accessible over the Internet via some middleware protocols. A more precise definition however is, any software component that utilizes WSDL (explained next) as the primary means to describe its interface and can be accessed over the Internet with the use of a messaging protocol such as SOAP (Gudgin et al., 2003). Web services aim to promote the use of distributed applications by harmonizing the heterogeneity between interacting systems. Unlike previous attempts, Web services have been successful in facilitating application-to-application and business-to-business integration *via the Internet* (Vinoski, 2003). This section presents in brief the key technologies that are necessary for the understanding of the material in the following sections.

WSDL. Web Services Description Language (WSDL) (Chinnici et al., 2004) is an XML-based standard for describing a Web service. A WSDL definition is divided into two parts: abstract and concrete. The *abstract* part describes the service interface, the operations it performs and the messages involved; the *concrete* part describes the location of the service and how to access and bind to a Web service. An abstract definition can be mapped to multiple concrete implementations. Both abstract and concrete WSDL definitions are independent of the service implementation. In other words, the description of service end points, the operations and messages, the message formats and the network protocols used in the communication are independent of how the services are implemented (*e.g.,* what programming languages have been used or whether the services are using other services to perform their tasks).

BPEL. Business Process Execution Language (BPEL) (T. Andrews et al., 2003) is a high-level workflow language for defining more coarse-grained Web services by integrating a number of existing Web services. BPEL provides many constructs for the management of the process including loops, conditional branching, fault handling and event handling (such as timeouts) and it allows for partner service to be accessed asynchronously, sequentially or in parallel. BPEL process can catch faults from a service invocation. A caught fault is handled by specified catch activity which could be a reply containing an error message. Faults are caught with `catch` or `catchAll` handlers that defined inside `faultHandlers` tags. The `catch` elements specify custom fault-handling activities that execute on a given fault name or fault variable While the `catchAll` elements specify fault-handling activities that execute when a fault is not caught by a `catch` fault handler. The `scope` is a container that provides a context for a subset of activities. It can contain fault and event handler for activities nested within it. The `scope` allows the activities enclosed within it to be managed as logical unit.

3 OVERVIEW OF APPROACH

Our approach to making an existing BPEL process robust involves monitoring the invocation of partner services from within the BPEL process. Events such as timeouts and faults are monitored and upon the occurrence of such events, a specific proxy Web service is invoked to find and replace the faulty services.

3.1 Modifying BPEL Processes

By modifying existing BPEL processes, either by a developer or by an automatic generator, our approach aims to achieve a separation of concerns (McKinley et al., 2004). That is, enabling the separate development of the process's functional requirements (the business logic) from the non-functional requirements (fault tolerance, in our case). This allows the initial developer of the BPEL process, which could be a business analyst, to determine the functional requirements of the process and compose it as such. Adding non-functional requirements involves identifying points in the process at which monitoring is required and inserting appropriate code to do so. The monitoring code we insert is in the form of standard

BPEL constructs to ensure the portability of the modified process.

BPEL provides constructs for the management of the faults and events by using the `faultHandlers` and `eventHandlers` clauses, respectively (see section 2). A fault can be a programmatic error generated by the Web service partners of the BPEL process or unexpected errors (e.g., service unavailability) from the Web service infrastructure. A timeout is an "event" in BPEL that is defined by the `onAlarm` clause. We use a timeout to limit the amount of time that the process can wait for a reply from an invoked Web service.

We modify the existing BPEL process by identifying points in the process at which external Web services are invoked and then wrapping each of those invocations with a BPEL `scope` that contains the desired fault and event handlers. The following XML code is an example of a service invocation in BPEL.

```
1. <invoke name="invokeApprover"
2.   partnerLink="approver"
3.   portType="loanApprovalPT"
4.   operation="approve"
5.   inputVariable="request"
6.   outputVariable="approvalInfo">
7. </invoke>
```

Figure 2: An unmonitored invocation.

The above code instructs the BPEL engine (a virtual machine that interprets and executes BPEL processes) to invoke a partner Web service. The actual Web service partner is defined by the `partnerLink` (line 2). Lines 3 and 4 identify the interface (`portType`) of the partner and what method (`operation`) the invocation wishes to call. The input and expected output messages are specified in lines 5 and 6. This bare invocation can be identified and wrapped with the monitoring code. The process of identifying an invocation to monitor includes noting the exact interface (operation, input and output variables) of the invocation. This is important for defining the proxy service as we will explain shortly.

The XML shown in Figure 3 is a version of the above invocation after our monitoring code has been wrapped around it. The unmonitored invocation (Figure 2) is first wrapped in a `scope` container which contains fault and event handlers (lines 2-11 and 12-21 respectively in Figure 3). The `scope` allows the activities enclosed to be managed as one logical unit. A `catchAll` fault handler is added (lines 3-10) to the `faultHandlers` to handle any faults generated as a result of the invocation of the partner Web service. The fault-handling activity defined is the invocation of the proxy Web service (lines 4-9). Thus, when a fault is generated by the partner service invocation, this fault is caught by the `catchAll` and the proxy service is invoked to substitute for the unavailable or failed service.

```
1. <scope>
2.   <faultHandlers>
3.     <catchAll>
4.       <invoke name="InvokeProxy"
5.       partnerLink="proxy"
6.       portType="proxyPT"
7.       operation="approve"
8.       inputVariable="request"
9.       outputVariable="approvalInfo"/>
10.    </catchAll>
11.  </faultHandlers>
12.  <eventHandlers>
13.    <onAlarm for="'PT15S'">
14.      <invoke name="InvokeProxy"
15.      partnerLink="proxy"
16.      portType="proxyPT"
17.      operation="approve"
18.      inputVariable="request"
19.      outputVariable="approvalInfo"/>
20.    </onAlarm>
21.  </eventHandlers>
22.  <invoke name="invokeApprover"
23.   partnerLink="approver"
24.   portType="loanApprovalPT"
25.   operation="approve"
26.   inputVariable="request"
27.   outputVariable="approvalInfo">
28.  </invoke>
29.</scope>
```

Figure 3: A monitored invocation.

A similar construct is used for the event handler. An `onAlarm` event handler (lines 13-20) is used to specify a timeout. That is, a duration within which the partner service invocation must complete. If the partner service fails to reply within the stipulated time, the proxy service is invoked (lines 14-19) as a substitute. In our case, we have arbitrarily specified a timeout duration of 15 seconds (line 13), but this can vary depending on the type application or personal preference. Note that the interface for the proxy Web service is exactly the same as that of the monitored Web service. This is because we intend to generate a proxy with an exact interface as that of the monitored service. Thus, the operation, input and output variables of the proxy are the same as that of the monitored invocation.

As stated in section 2, the BPEL specification allows for the definition of two types of fault handlers (`catch` and `catchAll`). Our decision to use a `catchAll` is based on the fact that, we expect the initial developer of the process would have defined explicit fault-handlers (using `catch`) for predefined exceptions (T. Andrews et al., 2003). Recall that the `catch` handler is used to handle specific predefined programmatic exceptions from the partner services. The `catchAll` would then handle any unforeseen faults that might arise from the Web service infrastructure, for instance, if the invoked service is unavailable. If it so happens that the specific `catch` fault-handlers are missing, the `catchAll` would handle any fault that is thrown, which may or may not be desirable. Finally, if there exists a `catchAll` in the original BPEL, then the wrapping code will be added at the beginning, which again may not be desirable. This shortcoming is due to the fact that the BPEL specification still does not have a proper runtime exception specification for unavailability of service partners.

3.2 Defining the Proxy Web Services

We have previously referred to the proxy Web service as a "specific proxy". By *specific*, we mean that the proxy for a monitored service has to have the same interface as the monitored service. Infact, the specific proxy only exists after the service to be monitored has been identified. The abstract portion of WSDL description for the proxy Web service is similar to that of the monitored service, only the concrete portions are different.

The job of the proxy Web service is to discover and bind *equivalent* Web services that can substitute for the monitored services, upon the failure of those services. By *equivalent*, we mean services that implement the same port type (and business logic) as that of the monitored service. A port type is similar to an interface in the Java programming language. So, when two Web services implement the same port type, only their internal implementations may vary, their interfaces (the abstract portions of their WSDL descriptions) remain the same. Although the Web service paradigm is still in its infancy, the idea of having equivalent services is not far fetched. The business logic of applications such as search engines, flight reservation services, driving directions or even a network of printers, are all the same. Thus applications with the same functional requirement are equivalent. It is expected that in the near future, standard organizations will specify standard service interface definitions for different business domains that providers can choose from, implement and deploy (Kreger, 2001; Januszewski, 2002).

Equivalent services can be discovered at design time (static discovery) (Kreger, 2001). When static discovery is used, services that can substitute for the monitored service are noted and tightly associated with the code for the proxy.

One concern about using a specific proxy is that we need to have several specific proxies, if the BPEL process has several partner Web services that need to be monitored. However, this problem can be easily overcome by defining one proxy with an interface that is an aggregation of all the interfaces of the monitored Web services.

3.3 Generating the Robust BPEL

The task of manually modifying a BPEL process to insert monitoring code and then producing the corresponding proxy service, can be quite cumbersome. This task becomes even more daunting when there are multiple service invocations that need to be monitored. A systematic approach to achieving these goals can however be devised. This systematic approach makes it possible to come up with a generator that can automatically generate the more robust version of a given BPEL process and its associated proxy service.

Such a generator, as illustrated in Figure 4, would need as input three sets of documents: (1) the original BPEL process, (2) the WSDL descriptions of all the Web service partners of the BPEL process that need to be monitored, and (3) the WSDL descriptions of all the substitutes for the monitored services.

The generator would add all the necessary monitoring code to the BPEL process. It needs the abstract descriptions of the partner services from their WSDL files in order to be able to define the interface for the proxy service. The WSDL files for the substitutes are needed so that they can be statically associated with the developed proxy as described in section 3.2. The WSDL files for the substitutes may not actually need to be fed into the generator. They could be enumerated on a list and the list is read by the proxy. The point here is that the substitutes and their bindings are known at design time. The output of the generator is the monitored BPEL and a proxy Web service that utilizes static discovery.

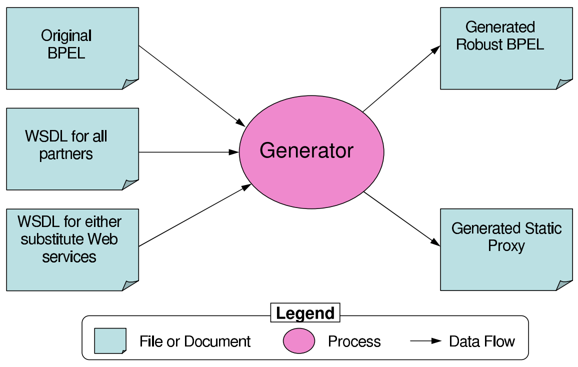

Figure 4: The generation process.

4 CASE STUDY

To show the feasibility of our approach presented in the previous section, we developed a prototype implementation of the generator in Java. Although the previous section provides an overview of the approach, some detail about the adaptation and generation process has been omitted for the purpose of clarity. In this section, we use a case study to demonstrate in more detail the issues we encountered and the decisions we made. For our case study, we use an existing Loan Approval process that is commonly used as a sample BPEL process.

4.1 The Loan Approval Process

The Loan Approval BPEL process is a Web service (`LoanApproval`) that is an aggregation of two other Web services. The two Web services involved in the process are a risk assessor service (`LoanAssessor`) and a loan approver (`LoanApprover`). The Loan Approval process implements a business process that uses its two partner services (`LoanAssessor` and `LoanApprover`) to decide whether a given individual qualifies for a given loan amount. Please note that in this case study we assume that we have access to the Loan Approval BPEL process, but the two other Web service partners are assumed to be developed, deployed and managed by third parties (outside our control). In this case study, we try to use our prototype implementation of the generator to generate a more robust version of the Loan Approval BPEL process without manually modifying the source code of the original BPEL process and without the need to modify the two Web service partners.

As illustrated in Figure 5, the Loan Approval BPEL process receives as input a loan request (`Receive-CustomerRequest`). The loan request message comprises two variables: the name of the customer and the loan amount (not shown in the figure). If the loan amount is less than $10,000, then the risk assessor Web service is invoked (`InvokeRiskAssessor`), otherwise the loan approver Web service is invoked (`InvokeLoanApprover`). The risk assessor and the loan approver services take as input the loan request message (not shown in the figure). After the risk assessor is invoked, the BPEL process expects to receive as reply a risk assessment message. This risk assessment message is a string with a value of either "high" or "low". When the risk assessment is "low", this means the loan is approved and the loan approval process sends an approval message (with "yes" value, `AssignYestoAccept`) to the customer and terminates (`ReplyCustomerRequest`). If the risk assessment message is "high", the loan approver service is invoked (`InvokeLoanApprover`). The loan approver service returns a loan approval message (either "yes" or "no"), which is then sent as reply to the customer (`AcceptMessageToCustomer`). Both the risk assessor service and the loan approver service can also return a predefined fault message to the BPEL process. When any of these services reply with a fault message, the BPEL process sends an error message to the user and terminates (not shown in the figure).

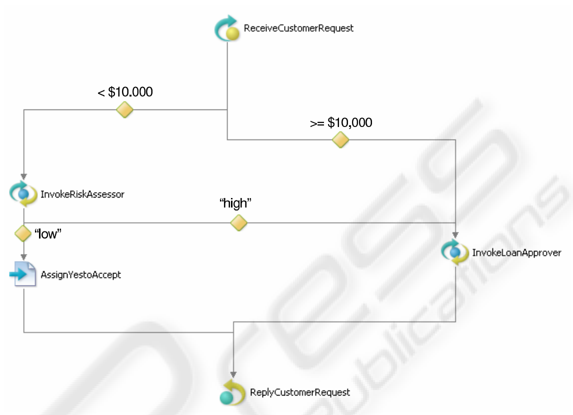

Figure 5: The work flow of the loan approval BPEL Process.

Both the risk assessor and the loan approver Web services were implemented previously by ActiveWebflow in Java (`http://www.active-endpoints.com/`). For the purpose of our case study, we deployed the Web services locally on an Apache Tomcat server with Apache AXIS (`http://www.apache.org/`) as the SOAP engine. The BPEL process was hosted on an Active-Bpel engine.

To ensure the clarity of the instrumentation process, it is important to briefly introduce the key WSDL files involved.

loandefinitions.wsdl: This file contains definition of the common messages between the risk assessor and the loan approver Web services. These messages are the `creditInformationMessage` and the `loanRequestErrorMessage`. The `loanRequest-ErrorMessage` defines the input message of the services. It is a 3-tuple that consists of the first and last name of the customer and the loan amount. The `loanRequestErrorMessage` defines the programmatic fault message that can be returned by both services. This file ensures consistency in the specification of the services. In this case, the services share the definition of two messages.

loanapprover.wsdl: This file describes the loan approver Web service. It describes the location of the service, the methods it exposes and the three messages involved. The “loandefinitions.wsdl” file is imported so as to include the two messages specified therein. A third message, the loan approval message, is explicitly defined inside this file.

loanassessor.wsdl: This file contains the description for the loan assessor service. It is much similar to the “loanapprover.wsdl”, the only different being that, for this interface, the name of the operation and output message are different.

4.2 Monitoring the Process

As previously described (Figure 4), our static generator uses the original BPEL process code, the WSDL file of the partners and their substitutes to generate a more robust BPEL process and its proxy Web service. The generator we are developing tries to achieve this in two steps: generate the modified BPEL process and generate the proxy Web service. These steps are not necessarily ordered as both can be achieved independently. In the following sections, we describe how we have done this.

4.2.1 Generating the BPEL Process

We have successfully derived and implemented an algorithm that, when given any syntactically correct BPEL process, would generate a more robust but equivalent version of the process. The task of transparently adapting any program code is made complicated by the need to preserve its original functionality (McKinley et al., 2004). This, coupled with the fact that BPEL is not a sophisticated programming language and is thus limited in the number of available programming language constructs, makes the task of instrumentation more arduous.

Because BPEL uses `Flow` and `Sequence` containers to specify parallel and sequential activities, individually wrapping the invocations in those containers with a `Scope` does not affect the sequence of their execution. However, when the order of execution between activities is specified with the use of a `Link`, the links have to be removed as part of the `Invoke` to become part of the `Scope`. Recall (from Section 2) that a `Scope` allows the activities within to be managed as one logical unit. So moving the moving the links to the `Scope` preserves the execution sequence of the activities. Figure 7 below is the modified version of the invocation in Figure 6 after the generation process. It can be seen that the target and source links in Figure 6 (lines 7-9) have been moved outside of the invocation in Figure 7 (line 29) to become immediate children of the `Scope` (lines 2-5).

The original Loan Approval BPEL process has a single fault handler for the whole process. This fault

```
1. <invoke inputVariable="request"
2.    name="InvokeLoanApprover"
3.    operation="approve"
4.    outputVariable="approval"
5.    partnerLink="approver"
6.    portType="lns:loanApprovalPT">
7.    <target linkName="receive-to-approve"/>
8.    <target linkName="assess-to-approve"/>
9.    <source linkName="approver-to-reply"/>
10.</invoke>
```

Figure 6: The invocation of the Loan Approver Service.

handler sends a fault message to the customer in the event of a fault. Our algorithm leaves this fault handler intact. But our policy of making sure that each target invocation has a specific fault and event handler means that our monitoring code is closest to the invocation. A limitation of this approach is that by using a `catchAll` fault handler, all faults generated by the invocation will first be caught by this handler even though there may already be explicitly defined handlers for known programmatic faults. Currently, we are working on a solution to address this problem. In addition, the current prototype wraps *all* the invocations in the BPEL process with monitoring code. It would be easy to come up with an implementation that allows the user to select which invocations to monitor. To avoid redundacy, only the transformation of the invocation for the Loan Approver service is shown in Figure 7.

The loan approver service was made to generate a `loanProcessFault`, which was caught in the `faultHandlers` clause and the proxy service invoked. The loan approver service is disabled to simulate an unavailable service. The generated exception is now successfully caught and the proxy is invoked. The modified process is also able to deal with a timeout event after the loan approver is purposefully made to loop to simulate a slow service. The timeout is caught by the specified `eventHandlers` clause (not shown in the code). Instead of the BPEL failing on these events, the proxy is successfully able to invoke an equivalent loan approver service, which substitutes for the failed loan approver.

4.2.2 Generating the Proxy Web Service

The second part of the generator is to automatically produce a Java implementation of the proxy Web service. As of this writing, this part of the generator is a work-in-progress. We have however, manually composed this proxy service. Currently, we employ a static discovery method (Kreger, 2001) for the discovery of equivalent services. The implementation of the proxy involves the use of a stub interface generator through which the discovered equivalent ser-

```
<scope variableAccessSerializable="no">
  <target linkName="receive-to-approve"/>
  <target linkName="assess-to-approve"/>
  <source linkName="approver-to-reply"/>
  <faultHandlers>
    <catchAll>
      <invoke inputVariable="request"
              name="proxy"
              operation="approve"
              outputVariable="approval"
              partnerLink="proxy"
              portType="pxns:proxyPT"/>
    </catchAll>
  </faultHandlers>
  <eventHandlers>
    <onAlarm>
      <invoke inputVariable="request"
              name="proxy"
              operation="approve"
              outputVariable="approval"
              partnerLink="proxy"
              portType="pxns:proxyPT"/>
    </onAlarm>
  </eventHandlers>
  <invoke inputVariable="request"
          name="InvokeLoanApprover"
          operation="approve"
          outputVariable="approval"
          partnerLink="approver"
          portType="lns:loanApprovalPT"/>
</scope>
```

Figure 7: The monitored loan approver service.

vices can be invoked. A stub interface is a Java interface for WSDL Web services using their port type descriptions. To generate the stub interface, you can use any tool that generates Java interfaces for WSDL services using their port type descriptions, such as WSDL2Java from AXIS or Java WSDP from Sun Microsystems.

As stated in the overview (Sections 3.2 and 3.3), the interface for the proxy service can be an aggregation of the interfaces of all the monitored services. This is the approach we have taken so as to have a single proxy for all monitored services. In order to achieve this, the WSDL of the proxy was composed with a new port type definition called `proxyPT`. This port type contains the operations and messages defined in the interfaces of the Loan Assessor and Loan Approver services. It contains both the `check` and `approve` operations of the loan assessor and loan approver services, respectively. The WSDL2Java tool from AXIS was used to generate a Java interface from the proxy WSDL. This Java interface was then implemented with code that would bind to the equivalent services when the corresponding operation is invoked. For the equivalent services, we created replicas of the Loan Approver and Loan Assessor services.

When a monitored service fails, the input message for that service is used as input message for the proxy. The proxy will then invoke the equivalent service with the same input message. A reply from the substitute service is then sent back to the loan approval BPEL process via the proxy. More detailed information about this case study can be found in (Ezenwoye and Sadjadi, 2005).

5 RELATED WORK

There has been a lot of work done both in web service monitoring (Robinson, 2003; Birman et al., 2004; G. Canfora et al., 2005) and in adding fault tolerant to existing systems (L. Moser et al., 1999; Natarajan et al., 2000), but they are not specifically addressing fault tolerance in BPEL processes. . So, we only focus on the most related ones.

Baresi's approach (Baresi et al., 2004) to monitoring involves the use of annotations that are stated as comments in the source BPEL program and then translated to generate a target monitored BPEL program. In addition to monitoring functional requirements, timeouts and runtime errors are also monitored. Whenever any of the monitored conditions indicates misbehavior, suitable exception handling code in the generated BPEL program handles them. This approach is much similar to ours in that monitoring code is added after the standard BPEL process has been produced. This approach achieves the desired separation of concern. This approach however requires modifying the original BPEL processes *manually*. The annotated code is scattered all over the original code. The manual modification of BPEL code is not only difficult and error prone, but also hinders maintainability. In our approach, there is no need for annotation and manual modification of the original BPEL processes.

BPELJ (M. Blow et al., 2004) is an extension to BPEL. The goal of BPELJ is to improve the functionality and fault tolerance of BPEL process. This is accomplishes by embedding snippets of Java code in the BPEL process. This however requires a special BPEL engine, thereby limiting its portability of BPELJ processes. The works mentioned above, although are able to provide some means of monitoring for BPEL processes, they not dynamically replace the delinquent services once failure has been established.

6 CONCLUSIONS

We have described and implemented a way of instrumenting BPEL processes so as to make them more fault tolerant. By adding code for monitoring the individual services that are part of the BPEL composition, failed services are replaced via a purposely generated proxy web service. We described how to automatically generate the more robust BPEL process and the corresponding *static* proxy.

In our future work, we plan to address the following issues. First, in our current work, we do not use *dynamic* service discovery. But because services are continuously published with the broker, there are bound to be appropriate services that become available after the composition of the BPEL process and the static proxy. So, it makes sense that upon failure of any of the constituent Web services of the BPEL process, an equivalent service can be dynamically discovered (at run-time) to serve as a substitute for the failed service. When dynamic discovery is used, care must also be taken not to re-discover the failed service. Second, substituting service implementations at runtime may lead to failures on the client-side (Denaro et al., 2005). Thus it is important to be able to detect and resolve potential integration problems from discovered equivalent services. This could include making sure that they actually fulfill their functional requirements. Third, we realized that the task of improving fault tolerance for multiple service collaborations is made even more complex if the collaborating services are *stateful*. Some techniques (Dialani et al., 2002) for dealing with such stateful services can be employed.

Further Information. A number of related papers, technical reports, and a download of the software developed for this paper can be found at the following URL: `http://www.cs.fiu.edu/~sadjadi/`.

ACKNOWLEDGEMENTS

The authors are very thankful to Eduardo Monteiro who worked on the implementation of the generator. This work was supported in part by IBM SUR grant.

REFERENCES

Baresi, L., Ghezzi, C., and Guinea, S. (2004). Smart monitors for composed services. In *ICSOC '04: Proceedings of the 2nd international conference on Service oriented computing*, pages 193–202. ACM Press.

Birman, K. P., van Renesse, R., and Vogels, W. (2004). Adding high availability and autonomic behavior to web services. In *Proceedings of the 26th International Conference on Software Engineering (ICSE 2004)*, pages 17–26, Edinburgh, United Kingdom. IEEE Computer Society.

Chinnici, R., Gudgin, M., Moreau, J.-J., Schlimmer, J., and Weerawarana, S. (2004). *Web Services Description Language (WSDL) Version 2.0*. W3C, 2.0 edition.

Denaro, G., Pezze, M., and Tosi, D. (2005). Adaptive integration of third party web services. In *in Proceeding DEAS 2005*, St. Louis, Missouri, USA.

Dialani, V., Miles, S., Moreau, L., Roure, D. D., and Luck, M. (2002). Transparent fault tolerance for web services based architectures. In *Eighth International Europar Conference (EURO-PAR'02)*, Lecture Notes in Computer Science, Padeborn, Germany. Springer-Verlag.

Ezenwoye, O. and Sadjadi, S. M. (2005). Enabling robustness in existing bpel processes. Technical Report FIU-SCIS-2005-08, School of Computing and Information Sciences, Florida International University, Miami, Florida.

G. Canfora et al. (2005). The c-cube framework: Developing autonomic applications through web services. In *Proceedings of DEAS'05*, Missouri, USA.

Gudgin, M., Hadley, M., Mendelsohn, N., Moreau, J.-J., and Nielsen, H. F. (2003). *SOAP Version 1.2*. W3C, 1.2 edition.

Januszewski, K. (2002). Using uddi at run time, part ii. *UDDI Technical Article*.

Kreger, H. (2001). *Web Services Conceptual Architecture (WSCA 1.0)*. IBM Software Group.

L. Moser et al. (1999). The Eternal system: An architecture for enterprise applications. In *Proceedings of the Third International Enterprise Distributed Object Computing Conference (EDOC'99)*.

M. Blow et al. (2004). *BPELJ: BPEL for Java, A Joint White Paper by BEA and IBM*.

McKinley, P. K., Sadjadi, S. M., Kasten, E. P., and Cheng, B. H. C. (2004). Composing adaptive software. *IEEE Computer*, pages 56–64.

Menasc, D. A. (2002). Qos issues in web services. *IEEE Internet Computing*, 6(6):72–75.

Natarajan, B., Gokhale, A. S., Yajnik, S., and Schmidt, D. C. (2000). DOORS: Towards high-performance fault tolerant CORBA. In *International Symposium on Distributed Objects and Applications*, pages 39–48.

Robinson, W. N. (2003). Monitoring web service requirements. In *Proceedings of the 11th IEEE International Conference on Requirements Engineering (RE 2003)*, pages 65–74. IEEE Computer Society.

T. Andrews et al. (2003). *Business Process Execution Language for Web Services version 1.1*. BEA Systems, International Business Machines Corporation, Microsoft Corporation, SAP AG, and Siebel Systems., 1.1 edition.

Vinoski, S. (2003). Integration with Web services. *IEEE Internet Computing*.

Vogels, W. (2003). Web services are not distributed objects. *IEEE Internet Computing*.

Weerawarana, S. and Curbera, F. (2002). Business process with bpel4ws: Understanding. *Online article*.