TRAP/BPEL: A Framework for Dynamic Adaptation of Composite Services

Technical Report: FIU-SCIS-2006-06-02

Onyeka Ezenwoye and S. Masoud Sadjadi

Autonomic Computing Research Laboratory
School of Computing and Information Sciences
Florida International University
{oezen001,sadjadi}@cs.fiu.edu

Abstract. TRAP/BPEL is a framework that adds autonomic behavior into existing BPEL processes *automatically* and *transparently*. We define an *autonomic* BPEL process as a composite Web service that is capable of responding to changes in its execution environment (*e.g.,* a failure in a partner Web service). Unlike other approaches, TRAP/BPEL does not require any manual modifications to the original code of the BPEL processes and there is no need to extend the BPEL language nor its BPEL engine. Furthermore, TRAP/BPEL promotes the reuse of code in BPEL processes as well as in their corresponding autonomic behavior. In this paper, we describe the details of the TRAP/BPEL framework and use a case study to demonstrate the feasibility and effectiveness of our approach.

Keywords: TRAP/BPEL, generic proxy, self-management, dynamic service discovery.

1 Introduction

Service Oriented Computing (SOC) allows for reusable components to be discovered and dynamically integrated to create new applications. Web services play a vital role in facilitating the realization of the SOC paradigm. With Web services, autonomous, self-contained and remotely accessible components can be integrated to create aggregate services. The characteristics of Web services that make them so suitable for SOC also present big challenges to their reliability. Specifically, these challenges arise from the following: (1) the autonomy of the services in any interaction gives rise to concerns about their continued availability and trust (*i.e.,* those service will actually do what they are expected to do) (2) the best-effort delivery method of the communication channel through which these service interact is known to be unreliable (3) the availability of the many number of new services that are being developed often makes composed services quickly obsolete and leads to the frequent redevelopment.

As an example, nodes on computational Grids are currently being exposed as services to ensure openness. Grid programming environments [23] allow for the creation of applications that integrate Grid services for coordinated problem solving (*e.g.,* for Bioinformatics and Computational Chemistry). For such applications, when a Grid service partner fails, the whole application fails and has to be restarted even though there are other nodes on the Grid that can substitute for the failed service [24]. This problem

is made more sever by the fact that such applications are often long running. This concern is major characteristic of composed services, but it is often not addressed in the specification of composition languages since it is orthogonal. There is therefore, a need to make aggregate services adaptable. Our work focuses on adapting services composed using Business Process Execution Language (BPEL), a common XML-based language for composing aggregate Web services.

Any adaptation framework for composed services must be able to deal with all the issues raised above. It should be possible to adapt existing applications so that they can discover and bind to new services when their original partner services fail or become too slow. Adapted services should be able to deal with unanticipated faults while still providing a reasonable level of service. Also, because of the very dynamic nature of SOC, any adaptation framework needs to enable the *separation of concerns* [7, 19]. That is, it needs to allow for the separate development of the functional from the non-functional requirements of the application by not entangling the code for QoS with the code for business logic.

The focus of our ongoing research is on how to *transparently* adapt existing aggregate service to encapsulate *autonomic* (self-management) behavior [17] and make them more resilient to failure. Specifically, we aim to make an aggregate Web service continue its valid function even after one or more of its constituent Web services have failed. To achieve this, we developed the RobustBPEL framework [12, 13] for automatically adapting BPEL processes, to monitor the invocation of their partner Web services at runtime. To achieve this, events such as faults and timeouts are monitored from within the adapted process. An adapted process is augmented with a *specific* proxy that replaces failed services with predefined or newly discovered alternatives. We say a proxy is specific if its interface is an aggregation of the interfaces of *all* the monitored partner services of the adapted BPEL processes. In managing failed invocations, the fault tolerance and performance of BPEL processes is improved and the behavior adapted transparently. *Transparency* ensures that the adaptation preserves the original behavior of the business process and does not tangle the code that provides autonomic behavior with that of the business logic of the process [22].

While with the use of a specific proxies we are able to adequately encapsulate autonomic attributes and replace failed services, development and maintenance of specific proxies for a vast number of available services is quite cumbersome. In this paper we present a solution to this problem by introducing the use of *generic* proxies rather than specific ones. We show how with the use of generic proxies and some recovery policies we are able to extend the autonomic capabilities of adapted BPEL processes. Also we show the design of the generic proxy, details of the adaptation process.

The rest of this paper is is structured as follows. Section 2 provides a background on our adaptation method. Section 3 motivates the need for generic proxies. Section 4 describes the generic proxy. In section 5 we use a case study to demonstrate the feasibility and usefulness of our approach. Section 6 contains some related work. Finally, some concluding remarks and a discussion on further research directions are provided in Section 7.

2 Background

In this section, we provide some background information for Web services, BPEL, Autonomic Computing, Transparent Shaping, and RobustBPEL. You can safely skip this section if you are familiar with all the above concepts, technologies, and frameworks.

2.1 Web Services & BPEL

A Web service is a software component that can be programmatically accessed over the Internet. The goal of the Web services architecture [6] is to simplify application-to-application integration. The technologies in Web services are specifically designed to address the problems faced by traditional middleware technologies in the flexible integration of heterogeneous applications over the Internet. Its lightweight model has neither the object model nor programming language restrictions imposed by other traditional middleware systems (*e.g.,* DCOM and CORBA) and its messaging protocol ensures that packets are able to traverse Internet firewalls. The interface to a Web service is described in Web services Description Language (WSDL) [20].

Applications that provide specific business functions (*e.g.*, price quotation) are increasingly being exposed as Web services. These services then become reusable components that can be the building blocks for more complex aggregate services (business processes). To facilitate the creation of business processes, a high-level workflow language, such as Business Process Execution Language (BPEL) [11,25], is used. BPEL is an XML-based workflow language that weaves together *basic* and *structured* activities to create the logic of a business process. A *basic* activity is a primitive BPEL activity that performs an atomic action, while a *structured* activity is derived from a combination of several activities. For example, the `invoke` activity is a basic activity that performs an operation on a partner Web service. Structured activities specify the order in which combined activities execute. The XML grammer that defines a BPEL process is interpreted and executed by a virtual machine called a BPEL engine. Although the BPEL specification provides constructs for fault handling and event handling (such as timeout), such language constructs are not sufficient to make a BPEL process self-manageable. The management of non-functional issues is assumed to be a function of the BPEL engine [4].

2.2 Autonomic Computing & Transparent Shaping

Autonomic computing [17] promises to solve the management problem by embedding the management of complex systems inside the systems themselves, freeing the users from potentially overwhelming details. The ultimate goal of autonomic computing is to create *self-managing* systems that are able to function with very little direct human intervention. A Web service is said to be autonomic if it encapsulates some autonomic attributes [14]. Autonomic attributes include (1) *Self-Configuration*: for the automatic configuration of components (2) *Self-Optimization*: for automatic monitoring and control (3) *Self-Healing*: for automatic discovery, and management of faults (4) *Self-Protection*: for automatic identification and protection from attacks or failure. The

focus of our ongoing research is to encapsulate self-management behavior in composed services.

As BPEL language constructs and its programming model are not sufficient to encapsulate self-management behavior inside the processes, we use Transparent Shaping [22] to augment BPEL processes with self-management behavior. *Transparent Shaping* is a new programming model that provides dynamic adaptation in applications. Its goal is to adapt *existing* applications in order to better respond to changes in their non-functional requirements or execution environment [22]. In transparent shaping, an application is augmented with *hooks* that intercept and redirect interaction to *adaptive code*. An adapted application is said to be *adapt-ready*. The adaptation is transparent because it preserves the original functional behavior and does not tangle the code that provides the new behavior (adaptive code) with the application code. By adapting *existing* applications, transparent shaping aims to achieve a separation of concerns [7, 19]. That is, enabling the separate development of the functional requirements from the non-functional requirements of an application.

2.3 RobustBPEL

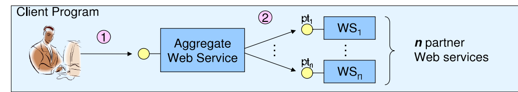
RobustBPEL [12, 13] is a framework that we developed as part of the transparent shaping programming model. Using RobustBPEL, we can automatically generate an adapt-ready version of an existing BPEL process. We note that in our previous study, we focused on adding self-healing and to some extent self-optimization behavior to existing BPEL processes. Specifically, our goal was to make an aggregate Web service continue its valid function after one or more of its constituent Web services have failed or is determined to be slow.

An adapt-ready process generated by Robust-BPEL is capable of monitoring the invocation of its Web service partners and will tolerate their failure. An adapt-ready process is augmented with invocations to a proxy service through which autonomic behavior is provided. We have developed two versions of RobustBPEL. In RobustBPEL-1 [12] a *static* proxy is used, whereas in RobustBPEL-2 [13] uses a *dynamic* proxy. Both the static and dynamic proxies are specifically generated for the BPEL processes they augment.

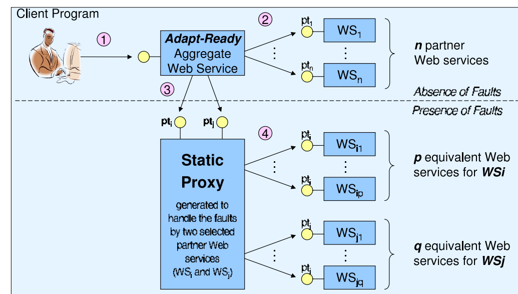
To understand how the static and dynamic proxies work, in Figure 1 we have provided three architectural diagrams showing the differences between the sequence of interactions among the components in a typical aggregate Web service and its corresponding generated adapt-ready versions. In a typical aggregate Web service (Figure 1(a)), first a request is sent by the client program, then the aggregate Web service interacts with its partner Web services (*i.e.,* WS_1 to WS_n) and responds to the client. If one of the partner services fails, then the whole process is subject to failure. To avoid such situations, adapt-ready process monitors the behavior of it partners and tries to tolerate their failure by forwarding the failed request to its proxy, which in its turn will find an equivalent service to substitute the failed one[1].

---

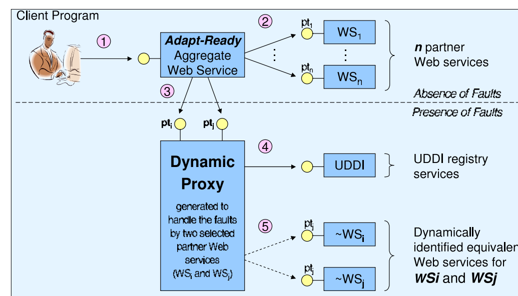
[1] At this point in our research, we make the assumption that two services are *equivalent*, if they implement the same port type. A *port type* is similar to an interface in the Java programming language. So, when two Web services implement the same port type, only their internal imple-



(a) Architecture of a typical aggregate Web service.



(b) Architecture of RobustBPEL-1 [12] using a *static* proxy.



(c) Architecture of RobustBPEL-2 [13] using a *dynamic* proxy.

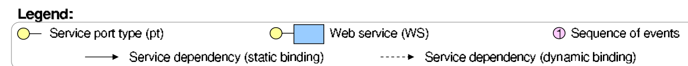


Fig. 1. Sequence of interactions among the components in three different settings.

As monitoring all the partner Web services might not be necessary, the developer can select only a subset of Web service partners to be monitored. For example, in Figure 1(b) WS_i and WS_j have been selected for monitoring. The adapt-ready process monitors these two partner Web services and in the presence of faults it will forward the corresponding request to the *static proxy*. The static proxy is generated specifically for this adapt-ready process and provides the same port types as those of the monitored Web services (*i.e.,* pt_i and pt_j). Thus, the operations and input/output variables of the proxy are the same as those of the monitored invocations. When more than one service is monitored within a BPEL process, the interface for the specific proxy is an

mentations may vary, their interfaces remain the same. In other words, applications with the same functional requirement are equivalent, regardless of implementation.

aggregation of all the interfaces of the monitored Web services. The static proxy in its turn forwards the request to an *equivalent* Web service. Information about equivalent services is "hardwired" into the code of this proxy at the time it was generated. This means that the number of choices for equivalent services are limited to those known at the time the static proxy was generated.

Given the rapid uptake of the SOC, we expect the emergence of numerous services that are functionally equivalent and thus can be substituted [15]. Therefore, in RobustBPEL-2 we replaced the static proxies with *dynamic* proxies that can find equivalent services at run time. As illustrated in Figure 1(c), the job of a dynamic proxy is to discover and bind equivalent Web services that can substitute for monitored services. Similar to a static proxy, the interface for the generated dynamic proxy is exactly the same as that of the monitored Web service.

As illustrated in Figure 1(c), when the dynamic proxy is invoked upon failure of a monitored service, the proxy makes queries against a registry service to find equivalent services. During the lifetime of the business process, more services that can be used as substitutes could be published with the registry. The registry technology used in the RobustBPEL fremework is the Universal Description, Discovery and Integration protocol (UDDI) [1], which is a specification for the publication and discovery of Web services. UDDI specifies a set of data structures, messages and API for creating and maintaining information about Web services in distributed registries.

3 Why TRAP/BPEL and Generic Proxies?

Although the RobustBPEL framework is able to provide some self-healing and self-optimizing behavior, it is limited in the level of adaptive behavior it can provide. Recall that the static and dynamic proxies only intercepts the normal operation of a BPEL process if a fault occurs upon the invocation of a partner Web service (Figures 1(b) and 1(c)). Therefore, RobustBPEL will only exhibit adaptive behavior upon the occurrence of a fault. Even after a service is determined to be faulty, the default invocation to that service will still have to be made before the adaptive code can intervene. So RobustBPEL is not able to proactively *self-protect* by preventing the invocation of a faulty service. Also *self-optimization* can be extended by providing a choice for service invocation. If some other service is determined to provide better QoS than the default service in the composition, it should be possible to switch to the better service (even if temporarily). The adapt-ready process from RobustBPEL is also not dynamically reconfigurable. Recall that during the adaptation process, a timeout is inserted as part of the monitoring code around selected service invocations. In order to change the value associated with this timeout event, another adapt-ready process will have to be generated. RobustBPEL therefore does not support *self-configuration*.

In addition to the above limitation, we note that each proxy service generated by RobustBPEL is *specific* to one BPEL process and cannot be reused for any other processes. Therefore, it is not possible to provide a common autonomic behavior to a set of services. A proxy will have to be generated for every process. This lack of reuse and the maintenance overhead that comes from having many proxies runs counter to the

promise of autonomic computing and SOC. In the rest of this paper, we show how TRAP/BPEL addresses the above limitations by using a *generic* proxy.

4 Generic Proxies

For the TRAP/BPEL framework we have developed a *generic* proxy that has a standard interface and works for all partner services of one or more adapt-ready BPEL processes. A recovery policy is used in the proxy to dictate the adaptive behavior for each monitored service. Apart from the used of a generic, a different approach is used in the process of making BPEL processes adapt-ready. In this section, we shows the high-level architecture of the generic proxy and the generation process.

4.1 High-Level Architecture

Figure 2 illustrates the architectural diagram of TRAP/BPEL at run time. As can be seen from the figure, several adapt-ready BPEL processes can be assigned to one generic proxy, which augments the BPEL processes with self-management behavior. Similar to the dynamic proxy (Figure 1(c)), the generic proxy uses a look-up mechanism to query a registry service at runtime for services that can be used to replace failed services. But unlike the specific proxies (Figures 1(b) and 1(c)), the generic proxy has a standard interface which bears no relation to the interfaces of the monitored services. The generic proxy in Figure 2 has as interface pt_g that is able the accept requests for the any monitored Web services (*e.g.,* WS_{11} and WS_{kn} partner Web services) with different port types.

The generic proxy can provide self-management behavior either common to all adapt-ready BPEL processes or specific to each monitored invocation using some high-level policies. At this point of our research, these high-level policies are specified in a configuration file that is loaded at startup time into the generic proxy. We plan to allow runtime modification to these high-level policies in the future versions of TRAP/BPEL. Figure 3 shows an example policy file where each unique monitored invocation can have a policy specified under a `<service>` element. The `<InvokeName>` element (line 4) has a value that uniquely identifies a monitored invocation in an adapt-ready BPEL process. The generic proxy checks all intercepted invocations and tries to match these invocations with the specified policies. If it finds a policy for that invocation, the proxy behaves accordingly, otherwise it follows its default behavior. If a policy exists, the generic proxy may take one of the following actions according to the policy: (1) invoke the service being recommended in the policy (line 6); (2) find and invoke another service to substitute for the monitored service (3) retry the invocation of the monitored service in the event of its failure (line 10). The policy also specifies the time interval between retries (line 12).

The default behavior of the proxy is to consult the registry to find a service that implements the same port type as the monitored invocation, this service is then invoked as a substitute.

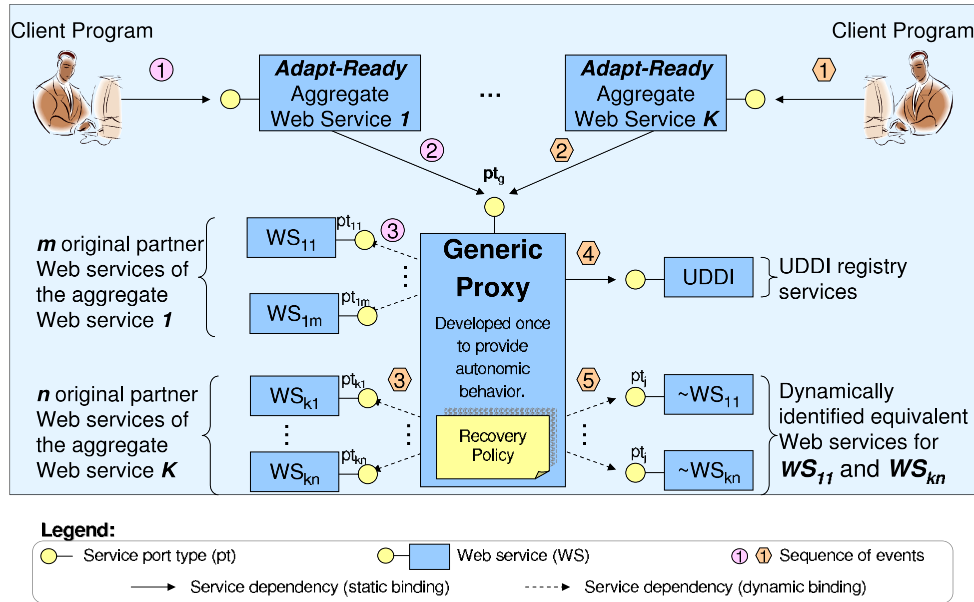


Fig. 2. Architectural diagram showing the sequence of interactions among the components in TRAP/BPEL during runtime.

4.2 Incorporating Generic Hooks

Following the Transparent Shaping programming model [22], we first need to incorporate some generic hooks at *sensitive joinpoints* in the original BPEL process. These joinpoints are certain points in the execution path of the program at which adaptive code can be introduced at run time. Key to identifying joinpoints is knowing where in the BPEL process *sensing* and *actuating* are required and inserting appropriate code (hooks) to do so. Because a BPEL process is an aggregation of services, the most appropriate place to insert interception hooks is at the *interaction joinpoints* (*i.e.*, the `invoke` instructions) [21]. All the modifications to the BPEL process are in the form of standard BPEL constructs to ensure the portability of the modified process.

For the RobustBPEL framework, we adapted the existing BPEL processes by wrapping each selected invocation with a BPEL `scope` that contains fault and event handlers. For the TRAP/BPEL framework, however, we took a different approach. As listed in Figure 4, rather than wrapping selected invocations with monitoring code, an invocation that is selected for monitoring is replaced with an invocation to the generic proxy. This approach allows more flexibility as to what kind of adaptive behavior to provide. We note that as the port type of the proxy is generic, all the contents of the original method invocation are serlialized in the input variable. Therefore, the process of replacing the target invocation with the proxy invocation involves identifying all the messages that are needed to create the input message for the proxy. Also, the sequence of activities needed to deserialize the output message from the proxy need to be created. This is done in a way that does not affect the original execution sequence of the BPEL process.

```
1.  <Policy>
2.    <Service>
3.      <!--a unique name for monitored invocation-->
4.      <InvokeName value="WS-Invoke"/>
5.      <!--WSDL for a default Web service for the monitored invocation-->
6.      <WsdlUrl preferred="true" value="http://.../WS-Description.wsdl"/>
7.      <!--timeout value for the monitored invocation-->
8.      <Timeout seconds="2"/>
9.      <!--the number of times to retry the invocation upon failure-->
10.     <MaxRetry value="2"/>
11.     <!--time to wait between retries-->
12.     <RetryInterval seconds="5"/>
13.   </Service>
14.   <Service>
15.     ...
16.   </Service>
17. </Policy>
```

Fig. 3. A portion of a policy file for the generic proxy.

```
1. <invoke name="InvokeWS11"
2.   partnerLink="..."
3.   portType="pt11"
4.   operation="operation1"
5.   inputVariable="..."
6.   outputVariable="...">
7. </invoke>
```

```
1. <invoke name="ProxyInvokeWS11"
2.   partnerLink="..."
3.   portType="pxns:proxyPT"
4.   operation="genericInvocation"
5.   inputVariable="..."
6.   outputVariable="...">
7. </invoke>
```

Fig. 4. Left: an invocation in the original BPEL. Right: an invocation in the adapt-ready BPEL.

To better understand the need for serlialization and deserialization, we provide a section of the WSDL of the generic proxy Web service in Figure 5. As can be seen from its description, the interface for the proxy has two operations: `genericInvocation` and `extract` (lines 22-25 and 26-29 respectively). The input message for the proxy `genericInvocation` operation (lines 1-6) has four parts: (1) `invokename`, which is used to identify the monitored service; (2) `porttype`, which identifies the port type of the monitored invocation (this variable is the unique key used to query the UDDI registry for services that implement the same interface); (3) `operation`, which identifies the exact operation of the port type being called; and (4) `variables`, which contains the serialized input message for the monitored service. When the proxy `genericInvocation` operation called with the `genericInputMessage`, the proxy identifies which service is being monitored and the necessary details about its invocation. The proxy can then take one of several actions as specified in the policy file.

At runtime the input message for each monitored service is serialized and used as part of the input message for the proxy `genericInvocation` operation. The proxy invokes any equivalent service with that same input message. Service invocation from within the proxy is done with the Web Service Invocation Framework (WSIF) [8]. A reply from the substitute service is serialized into the `genericOutputMessage` (lines 8-10) and sent back to the adapted BPEL process from the proxy. We need to serialize the input and output messages (for the monitored invocations) because the proxy needs the have standard interface through which messages for any service can be sent.

```
1.  <message name="genericInputMessage">
2.    <part name="invokename" type="xsd:string"/>
3.    <part name="porttype" type="xsd:string"/>
4.    <part name="operation" type="xsd:string"/>
5.    <part name="variables" type="xsd:string"/>
6.  </message>
7.
8.  <message name="genericOutputMessage">
9.    <part name="reply" type="xsd:string"/>
10. </message>
11.
12. <message name="extractInputMessage">
13.   <part name="values" type="xsd:string"/>
14.   <part name="param" type="xsd:string"/>
15. </message>
16.
17. <message name="extractReply">
18.   <part name="value" type="xsd:string"/>
19. </message>
20.
21. <portType name="proxyPT">
22.   <operation name="genericInvocation">
23.     <input message="tns:genericInputMessage"/>
24.     <output message="tns:genericOutputMessage"/>
25.   </operation>
26.   <operation name="extract">
27.     <intput message="tns:extractInputMessage"/>
28.     <output message="tns:extractReply"/>
29.   </operation>
30. </portType>
```

Fig. 5. A section of the WSDL description of the interface of the generic proxy.

When the `genericOutputMessage` arrives at the adapted BPEL process, it will need to be deserialized for further processing within the BPEL process, as part of the original execution. Since BPEL is not a general-purpose programming language, it lacks the necessary constructs that would be needed to deserialized the generic output message (`genericOutputMessage`) of the generic proxy. One solution to this problem, without extending the BPEL language, is to use a partner Web service to perform more complicated data manipulation [11]. To this end, we have decided to use the generic proxy to deserialize the `genericOutputMessage`.

Since a WSDL message can comprise of one or more parts [5], the deserialzation would have to extract the value for each message part. Therefore, after the adapted BPEL process receives the `genericOutputMessage`, it then make a call to the generic proxy's `extract` operation for each part of the reply message. The input message (`extractInputMessage`, lines 12-14) for the proxy's `extract` operation has two parts, `values` and `param`. The `values` part contains the serialized `genericOutputMessage` and the `param` parts specifies which parameter value to extract from the `genericOutputMessage`. The proxy then sends this value back to the BPEL process.

4.3 The Generation Process

The RobustBPEL framework requires the generation of both the adapt-ready version of a given BPEL process and its associated *specific* proxy. However, as TRAP/BPEL

uses *generic* proxies, the TRAP/BPEL Generator only needs to generate the adapt-ready BPEL processes. The only input to this generator is a configuration file. First, the `Parser` reads the information needed for generating adapt-ready BPEL process and sends them to the adapt-ready BPEL compiler. Next, the generator uses the information provided by the parser and retrieves the required files from the local disk and starts its compilation process. The primary information the generator are the original BPEL file, the list of which invocation to monitor and the WSDL files for all the partner Web services. The WSDL files are needed so that the generator can get details about the specification of the messages that are exchange by the monitored services. This information is used for the serialization and deserialization of messages.

5 Case Studies

In this section, we use a case study to demonstrate the feasibility of our approach. For the case study, we start by describing the applications, then we present the configuration and results of the case study.

5.1 The Google-Amazon Process

The Google-Amazon business process integrates the Google Web service for spelling suggestions with the Amazon E-Commerce Web service for querying its store catalog. The business process takes as input a phrase (keywords) which is sent to the Google spell-checker for corrections. If any word in the input phrase is misspelled, the Google spell-checker sends back as reply the phrase with the misspelled words corrected (the phrase is unchanged if the spellings are correct). The reply from the Google service is used to create keyword search of the Amazon bookstore via the Amazon Web service.

From this original Google-Amazon process, we used the generator to generate the adapt-ready process. For this adaptation we have selected to have the generator only adapt the invocation of the Google spell-checker. We then found another publicly available Spell-checker Web service from Cydne to act a substitute for the Google service. The differences between the Cydne service and that of Google are in the signature of their operations. First of all, the operation names are different. Second, the Google service takes as input two strings: (1) a license key and (2) a phrase. The Cydne service also takes as input a license key and a phrase but in the reverse order. Also, rather than returning a string in which misspelled words have been corrected, the Cydne service returns a data type that contains the original input string, the misspelled words and an unranked list of suggested words for each misspelled word.

To overcome these differences, we developed a simple wrapper Web service for the Cydne spell-checker. This wrapper Web service harmonizes the difference between the interfaces of the two spell-checkers. In order to be able to select the best word from the list of suggestions from the Cydne spell-checker, we have incorporated into the wrapper an open source Java spell-checker API from IBM, called Jazzy. Each misspelled word identified by the Cydne spell-checker is fed into Jazzy with the list of suggestions as the dictionary. The highest ranked suggestion from Jazzy is chosen as the return data for the wrapper Web service. Because we have this wrapper service, there is no need to register the Cydne service in the UDDI registry, rather, it is the WSDL for the wrapper service that is mapped to the registry.

5.2 The Loan Approval Process

The Loan-Approval process is a commonly used sample BPEL process. The Loan-Approval BPEL process is an aggregate Web service composed of two other Web services: a low-risk assessor service (LowAssessor) and a high-risk assessor service (High-Assessor). The Loan-Approval process implements a business process that uses its two partner services to decide whether a given individual qualifies for a given loan amount. Both the business process and the risk assessor (simulated) service are deployed locally. The Loan-Approval BPEL process receives as input a loan request. The loan request message comprises two variables: the name of the customer and the loan amount. If the loan amount is less than $10,000, then the risk assessor Web service is invoked, otherwise the loan approver Web service is invoked. The risk assesement services take as input the loan request message. After the LowAssessor 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", the loan is approved and the Loan-Approval process sends a positive message to the customer. If the risk assessment message is "high", the HighAssessor service is invoked. The HighAssessor service returns an approval message ("yes" or "no"), which is then sent as reply to the customer.

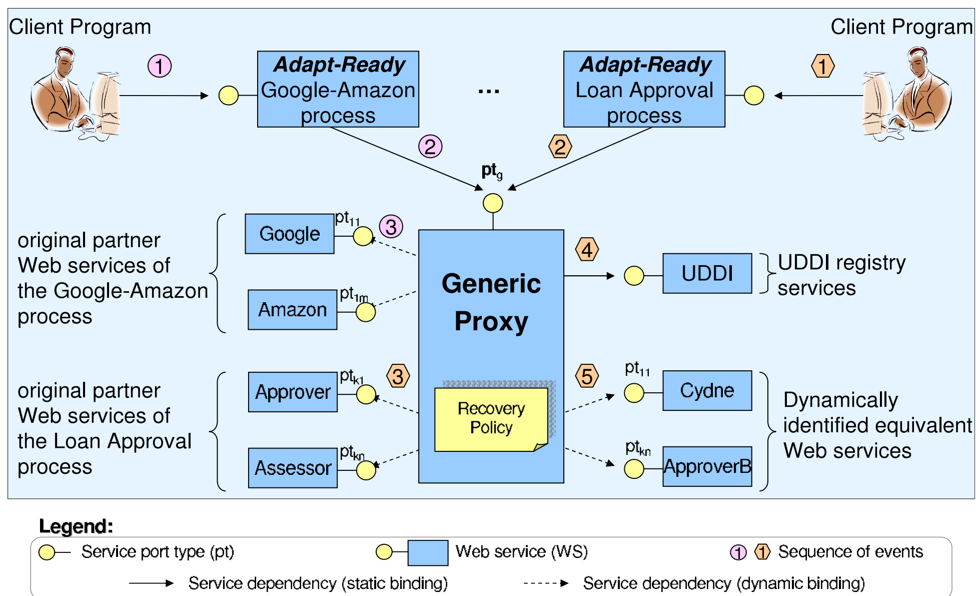


Fig. 6. The interaction between the applications and the proxy.

5.3 Configuration and Result

As illustrated in Figure 6, client requests are made to the BPEL process (labeled 1), which results in the invocations to the generic proxy services (labeled 2). When the call arrives at the proxy, it uses the policy file (Figure 7) to decide what action to take. Since the interaction between the proxy and both applications is the same, we will only describe that of the adapt-ready Google-Amazon process. In this case the policy is to use the default Google spell-checker. After some successful executions, we made the endpoint of the Google service WSDL to point to a phantom location. There upon failure, the generic proxy would query the UDDI registry to find and invoke the wrapper Web service for the Cydne spell-checker as a substitute (labeled 4 and 5). This shows self-healing bahavior. The result of the invocation of the substitute is sent back to the adapt-ready Google-Amazon process and then used as input to query the Amazon store service. For example, we used “Computer Algorthms” as input keyword to the process, Google corrected it to “Computer Algorithms”, and Amazon found this book: “Bruce Schneier, Applied Cryptography: Protocols, Algorithms, and Source Code in C, Second Edition”.[2] Self-Optimization is achieved through the timeout specified in the policy for this invocation, upon expiration of this timeout and retries are exhausted, a substitute is invoked. The use of a caching mechanism helps improve self-optimization. After the invocation to the Google service failed, its WSDL is purged from the cache and replaced with that of the Cydne wrapper service which found in the registry. This also limits the number of queries made to the registry. The cache is cleared at intervals. Further configuration and optimization can be achieved through runtime modification of the policy file. This can be done by either a human or machine agent. We plan to add runtime policy adjustment in the future.

```
<!-- Policy file for the generic proxy -->
<policy>
  <service>
    <InvokeName value="GoogleInvoke"/>
    <WsdlUrl preferred="true" value="http://.../GoogleSearch.wsdl"/>
    <Timeout seconds="2"/>
    <MaxRetry value="2"/>
    <RetryInterval seconds="2"/>
  </service>
  <service>
    <InvokeName value="InvokeHighAssessor"/>
    <WsdlUrl preferred="true" value="http://.../HighAssessor.wsdl"/>
    <Timeout seconds="1"/>
    <MaxRetry value="2"/>
    <RetryInterval seconds="1"/>
  </service>
</policy>
```

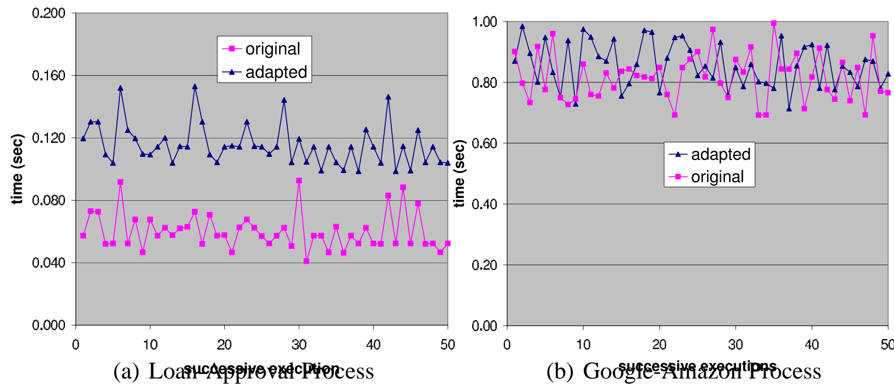
Fig. 7. Policy for the generic proxy.

We now use the rest of this case study to show the comparison of the execution time between the original business processes and their adapt-ready versions. we configured

[2] The Amazon store service actually returned a list of books but we only show the first one.

the client applications to run with different number of threads that sequentially make calls on the processes. As the X-axis of both charts in Figure 8 shows, the number of total threads we used in this experiment is 50. The initial runs were made against the original BPEL process. We noted the completion times (in seconds) for each thread. The result is plotted in the charts in Figure 8 under the *original* curve. For the original Loan-Approval process, the average completion time was approximately 0.06 seconds, while that original Google-Amazon process was 0.82 seconds.

The result of the observed completion times of the adapt-ready verisons is plotted in Figure 8 under the *adapted* curve. For the adapted Loan-Approval process, the average completion time was approximately 0.11 seconds, while that original Google-Amazon process was 0.86 seconds. This experiment shows that there is a slight overhead of approximately 0.05 and 0.04 seconds (for Loan-Approval and Google-Amazon processes, respectively) in completion time incurred by the adapt-ready compared to the original processes. The redirection through the proxy is responsible for this overhead.



(a) Loan-Approval Process (b) Google-Amazon Process

Fig. 8. This charts shows the comparison of the completions times between the original and adapt-ready processes.

6 Related Work

Birman et al. [3] propose extensions to the Web services architecture to support mission-critical applications. They propose the following five extensions; Component Health Monitoring (CHM), Consistent and Reliable Messaging(CRM), Data dissemination (DDS), Monitoring and Distributed Control (MDC) and Event notification (EVN). Similar to ours, this work aims to improve the reliability of Web services, but it proposes extensions to the Web services architecture.

Baresi's approach [2] 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

misbehaviour, 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; however, it requires modifying the original BPEL processes *manually* and 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.

Charfi et al [4] use an aspect-based container to provide middleware support for BPEL. The two inputs to the framework are the BPEL process and a deployment descriptor. The descriptor specifies the non-functional requirements (*e.g.*, security, persistence and transactions). The process container is the runtime environment for the BPEL process. All interactions go through the container which plugs in support for non-functional requirements. Aspects can be generated using the deployment descriptor to specify the pointcuts. Aspects specify what and how SOAP messages can be modified to add, for instance, security information to the header. This framework is different form our because it requires a purpose built BPEL engine. Also, the adaptation is done at a much lower level (the messaging layer).

AdaptiveBPEL [10] is much like Charfi [4], with the only major differences being that AdaptiveBPEL proposes to augment an *existing* BPEL engine with aspect weaving capabilities to address QoS concerns and adapt processes logic. In addition, adaptation is driven by a policy negotiated at runtime between the interacting endpoints.

Erradi et al. [9] provide reliability through a policy driven middleware named Web Services Message Bus (wsBus), which is used to transparently enact recovery actions. The wsBus intercepts the execution of composite services and transparently provides recovery services based on an extensible set of recovery policies (*e.g.*, retry, skip, and use equivalent service). The wsBus provides exception-handling and recovers from failures such as service unavailability and timeout. It also enforces SLA agreements. This approach is modular and separates the business logic of the process from the QoS requirements, however, adaptation is done at a much lower messaging layer.

Finally, BPELJ [18] is an extension to BPEL. The goal of BPELJ is to improve the functionality and fault tolerance of BPEL process. This is accomplished 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 singular or aggregate Web services, they do not dynamically replace the delinquent services once failure or extensive delay has been detected.

7 Conclusion and Future Work

We presented an approach to transparently incorporating self-management behavior into existing BPEL processes. We have introduced the TRAP/BPEL framework and its generic proxy, and demonstrated how a generic proxy can be used to encapsulate autonomic behavior through the use of policies. Using a case study, we demonstrated the autonomic behavior of the generic proxy.

In our future work, we plan to address the following issues. First, we plan to provide a GUI for developing high-level policies and enabling a developer to modify the

policy at runtime. Second, we realized that the task of improving fault tolerance and performance for multiple service collaborations is made even more complex if the collaborating services are *stateful*. We plan to apply this approach to grid applications that integrate WSRF-based stateful services [16]. Finally, we plan to study the existing ranking systems and adapt one in the TRAP/BPEL framework.

References

1. B. Atkinson et al. *UDDI Version 3.0.1*. OASIS, 2003.
2. L. Baresi, C. Ghezzi, and S. Guinea. Smart monitors for composed services. In *ICSOC '04: Proceedings of the 2nd international conference on Service oriented computing*, pages 193–202. ACM Press, 2004.
3. K. P. Birman, R. van Renesse, and W. Vogels. 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, May 2004. IEEE Computer Society.
4. A. Charfi and M. Mezini. An aspect based process container for BPEL. In *Proceedings of The First Workshop on Aspect-Oriented Middleware Developement*, Genoble, France, November 2005.
5. E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. *Web Services Description Language (WSDL) 1.1*. W3C, 1.1 edition, March 2001. Available at URL: `http://www.w3c.org/TR/wsdl`.
6. D. Booth et al. . *Web Services Architecture*. W3C, 2004.
7. E. W. Dijkstra. Structured programming. *Software Engineering Techniques, edited by Buxton and Randell (available from NATO, Brussels)*, pages 84–87, 1970.
8. M. J. Duftler, N. K. Mukhi, A. Slominski, and S. Weerawarana. *Web Services Invocation Framework (WSIF)*. IBM T.J. Watson Research Center, August 2001. Available at URL: `http://ws.apache.org/wsif/`.
9. A. Erradi and P. Maheshwari. wsBus: QoS-aware middleware for relaible web services interaction. In *IEEE International Conference on e-Technology, e-Commerce and e-Service*, Hong Kong, China, 2005.
10. A. Erradi, P. Maheshwari, and S. Padmanabhuni. Towards a policy driven framework for adaptive web services composition. In *Proceedings of International Conference on Next Generation Web Services Practices*, 2005.
11. O. Ezenwoye and S. M. Sadjadi. Composing aggregate web services in BPEL. In *Proceedings of The 44th ACM Southeast Conference*, Melbourne, Florida, March 2006.
12. O. Ezenwoye and S. M. Sadjadi. Enabling robustness in existing BPEL processes. In *Proceedings of the 8th International Conference on Enterprise Information Systems (ICEIS-06)*, May 2006.
13. O. Ezenwoye and S. M. Sadjadi. Robustbpel-2: Transparent autonomization in aggregate web services using dynamic proxies. Technical Report FIU-SCIS-2006-06-01, School of Computing and Information Sciences, Florida International University, 11200 SW 8th St., Miami, FL 33199, June 2006.
14. S. Gurguis and A. Zeid. Towards autonomic web services: Achieving self-healing using web services. In *Proceedings of DEAS'05*, Missouri, USA, May 2005.
15. J. Hau, W. Lee, and S. Newhouse. The ICENI semantic service adaptation framework. In *In UK e-Science All Hands Meeting*, Nottingham, UK, September 2003.

16. M. Humphrey, G. Wasson, K. Jackson, J. Boverhof, M. Rodriguez, J. Bester, J. Gawor, S. Lang, I. Foster, S. Meder, S. Pickles, , and M. McKeown. State and events for Web services: A comparison of five WS-Resource framework and WS-Notification implementations. In *Proceedings of The 4th IEEE International Symposium on High Performance Distributed Computing*, North Carolina, USA, July 2005.
17. J. O. Kephart and D. M. Chess. The vision of autonomic computing. *IEEE Computer*, 36(1):41–50, 2003.
18. M. Blow et al. *BPELJ: BPEL for Java, A Joint White Paper by BEA and IBM*. BEA and IBM, March 2004.
19. P. K. McKinley, S. M. Sadjadi, E. P. Kasten, and B. H. C. Cheng. Composing adaptive software. *IEEE Computer*, pages 56–64, July 2004.
20. R. Chinnici et al. *Web Services Description Language (WSDL) Version 2.0*. W3C, 2.0 edition, March 2004.
21. S. M. Sadjadi and P. K. McKinley. Using transparent shaping and web services to support self-management of composite systems. In *Proceedings of the International Conference on Autonomic Computing (ICAC'05)*, pages 88–95, Seattle, Washington, June 2005.
22. S. M. Sadjadi, P. K. McKinley, and B. H. Cheng. Transparent shaping of existing software to support pervasive and autonomic computing. In *Proceedings of the first Workshop on the Design and Evolution of Autonomic Application Software 2005 (DEAS'05), in conjunction with ICSE 2005*, St. Louis, Missouri, May 2005.
23. R. Sirvent, J. M. Perez, R. M. Badia, and J. Labarta. GRID superscalar: a programming paradigm for grid applications. In *Workshop on Grid Applications Programming*, Edinburgh, Scotland, July 2004.
24. A. Slominski. On using BPEL extensibility to implement OGSI and WSRF grid workflows. In *GGF10 Workshop on Workflow in Grid Systems*, Berlin, Germany, March 2004.
25. T. Andrews et al. *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, May 2003.