RobustBPEL-2: Transparent Autonomization in Aggregate Web Services Using Dynamic Proxies

Technical Report: FIU-SCIS-2006-06-01

Onyeka Ezenwoye and S. Masoud Sadjadi

Autonomic Computing Research Laboratory
School of Computing and Information Sciences
Florida International University
11200 SW 8th St., Miami, FL 33199
{oezen001,sadjadi}@cs.fiu.edu,
WWW home page: http://www.cs.fiu.edu/acrl

Abstract. Web services paradigm is allowing applications to electronically interact with one another over the Internet. BPEL facilitates this interaction by providing a platform with which Web services can be integrated.
Using RobustBPEL-1, we demonstrated how an aggregate Web service, defined as a BPEL process, can be instrumented automatically to monitor its partner Web services at runtime and replace failed services via a generated proxy. While in the previous work the proxy is *statically* bound to a limited number of alternative Web services, in this paper we extended the RobustBPEL-1 toolkit to generate a proxy that *dynamically* discovers and binds to existing services. Further, we present details of the generation process and the architecture of dynamically adaptable BPEL processes and their corresponding dynamic proxies. Finally, we use two case studies to demonstrate how generated dynamic proxies are used to support self-healing and self-optimization (specifically, to improve the fault-tolerance and performance) in instrumented BPEL processes.

Keywords: Web service monitoring, BPEL processes, dynamic proxies, self-healing, self-optimization, dynamic service discovery.

1 Introduction

Web services are facilitating the uptake of Service-Oriented Architecture (SOA) [10], allowing business organizations to electronically interact with one another over the Internet. In this architecture, reusable, self-contained and remotely accessible application components, which are exposed as Web services, can be integrated to create more course-grained aggregate services (*e.g.,* a flight reservation service). For this, high-level workflow languages such as BPEL [29, 33] can be used to define aggregate services (*a.k.a.,* business processes) that constitute a number of related services (*a.k.a.,* business functions) [23]. Unfortunately, these types of business processes are known to be very fragile. According to [25], about 80 percent of the total amount of time used in developing business processes is spent in exception management.

The integration of multiple services, which are potentially developed and maintained on heterogeneous environments, introduces new levels of complexity in management. The management of aggregate services goes beyond the administration boundaries of individual services, involving different policies unknown to the aggregate services. Also, services interacting with these aggregate services are often geographically scattered and communicate via the Internet, which is unreliable and prone to failure. Given the unreliability of such communication channels, the unbounded communication delays, and the autonomy of the interacting services, it is difficult for developers of business processes to anticipate and account for all the dynamics of such interactions. In addition, the high-availability nature of some business processes requires them to work in the face of failure of their constituent parts [9, 12]. It is then important to make aggregate services more resilient to the failure of their partner services.

Autonomic computing [19] promises to solve the management problem by embedding the management of complex systems inside the systems themselves, freeing the users from potentially overwhelming details. A Web service is said to be autonomic if it encapsulates some autonomic attributes [16]. Autonomic attributes include self-configuration, self-optimization, self-healing, and self-protection [19]. The focus of our ongoing research is to encapsulate *self-healing* and *self-optimizing* behavior in business processes in order to make them more resilient to the potential failures of their partner services. Specifically, we aim to make an aggregate Web service continue its valid function after one or more of its constituent Web services have failed.

We recently introduced RobustBPEL-1 [14], a software toolkit that provides a *systematic* approach to making existing aggregate Web services more tolerant to the failure of their constituent Web services. Using RobustBPEL-1, we demonstrated how an aggregate Web service, defined as a BPEL process, can be instrumented automatically to monitor its partner Web services at runtime to check if these services actually fulfill their service contracts. To achieve this, events such as faults and timeouts are monitored from within the adapted process. We showed how our adapted process is augmented with a *static* proxy that replaces failed services with predefined alternatives.

While in the previous work the proxy is *statically* bound to a limited number of alternative Web services, in this paper we extend the RobustBPEL-1 toolkit to generate a proxy that *dynamically* discovers and binds to existing services. The recent proliferation of Web services has convinced us that more appropriate services may become available after the composition and deployment of the BPEL process and its corresponding static proxy. So, it makes sense that upon failure or delay of any of the partner Web services of the BPEL process, an equivalent service can be discovered dynamically (at run-time) to serve as a substitute for the service. In doing this, we improve the fault tolerance and performance of BPEL processes by transparently adapting their behavior. By *transparent* we mean that the adaptation preserves the original behavior of the business process and does not tangle the code that provides self-healing and self-optimization behavior with that of the business process [28]. This transparency is achieved by using a *dynamic* proxy that encapsulates the autonomic behavior (adaptive code).

The rest of this paper is is structured as follows. Section 2 provides a background on some related technologies and gives a brief introduction to the RobustBPEL-1 toolkit. Section 3 describes the dynamic proxy and how it is generated. In section 4 we use two examples as case studies to demonstrate the feasibility and usefulness of our approach. Section 5 contains some related work. Finally, some concluding remarks and a discussion on further research directions are provided in Section 6.

2 Background

In this section, we provide some background information for Web services, BPEL, Transparent Shaping and RobustBPEL-1. You can safely skip this section if you are familiar with all the above technologies, protocols and toolkits.

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 [10] 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 the functionality provided by a Web service is described in Web services Description Language (WSDL) [26]. To make a call on these functions remotely, a messaging protocol such as SOAP [21] can be used.

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). Currently search engines like Google, Yahoo! and MSN are being exposed as Web services and provide functions that range from simple queries, to generation of maps and driving directions. A business process that can be derived from the aggregation of such services would be one that, for instance, generates driving directions. As illustrated by Figure 1, the process could work by integrating two service; (1) a service that retrieves the addresses of nearby businesses and (2) a service that gets the driving directions to a given address. This business process can then be used from the on-board computer of a car to generate driving directions to the nearest gas station, hotel, etc.

To facilitate the creation of business processes, a high-level workflow language, such as Business Process Execution Language (BPEL) [32], is often used. BPEL provides many constructs for the management of a process including loops, conditional branching, fault handling and event handling (such as timeout). To make a BPEL process fault tolerant, BPEL fault handling activities (*e.g.*, `catch` and `catchAll` constructs) can be used. We aim to separate the task of making a BPEL process more robust from the task of composing the business logic of the process [14].

Web Services Protocol Stack. As illustrated in Figure 2, the key to achieving interoperability in Web services is a layered architecture. The dependency relation between each two adjacent layers is top-down. While the process layer (BPEL) takes care of

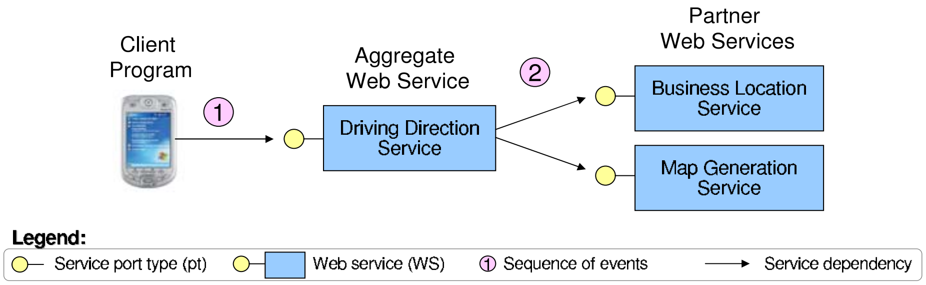


Fig. 1. A Business Process that integrates remote components to create a new application that gets driving directions.

the composition of Web services, the service layer (WSDL) provides a standard for describing the service interfaces. At the messaging layer (typically SOAP), the operations defined in the service layer are realized as two related output and input messages, which serialize the operation and its parameters. At the bottom of the stack is the transport layer (typically HTTP) that facilitates the physical interaction between the Web Services. Although Web services are independent of transport protocols, HTTP is the most commonly used protocol for Web service interaction. Except for the transport layer, all the protocols in the other layers are typically based on XML.

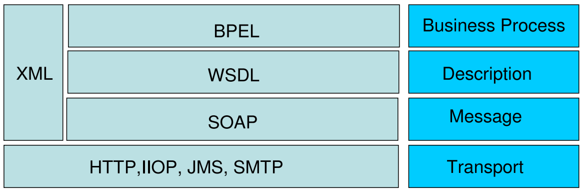


Fig. 2. The Web services stack.

2.2 Transparent Shaping & RobustBPEL-1

Transparent Shaping is a new programming model that provides dynamic adaptation in existing applications. The goal is to respond to the dynamic changes in their non-functional requirements (*e.g.*, changes request by end users) and/or environments (*e.g.*, changes in the executing environment) [28]. In transparent shaping, an application is augmented with *hooks* that intercept and redirect interaction to *adaptive code*. 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 [13, 24]. That is, enabling the separate development of the functional requirements (the business logic) from the non-functional requirements of an application.[1]

RobustBPEL-1 [14] is a toolkit that we developed previously as part of the transparent shaping programming model. Using RobustBPEL-1, we can automatically generate an *adapt-ready* version of an existing BPEL process that can adapt its non-functional behavior in response to the dynamic changes in its environment or its non-functional requirements. We note that in our previous study, we only focused on adding self-healing (fault-tolerant) behavior to existing BPEL processes. In other words, an adapt-ready process generated by Robust-BPEL-1 is capable of monitoring the operation of its Web service partners and will tolerate their failure – which might arise from software/hardware crashes, partial network outages and busy services.

To better understand how the adapt-ready version behaves, in Figure 3 we have provided 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 version. In a typical aggregate Web service (Figure 3(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.

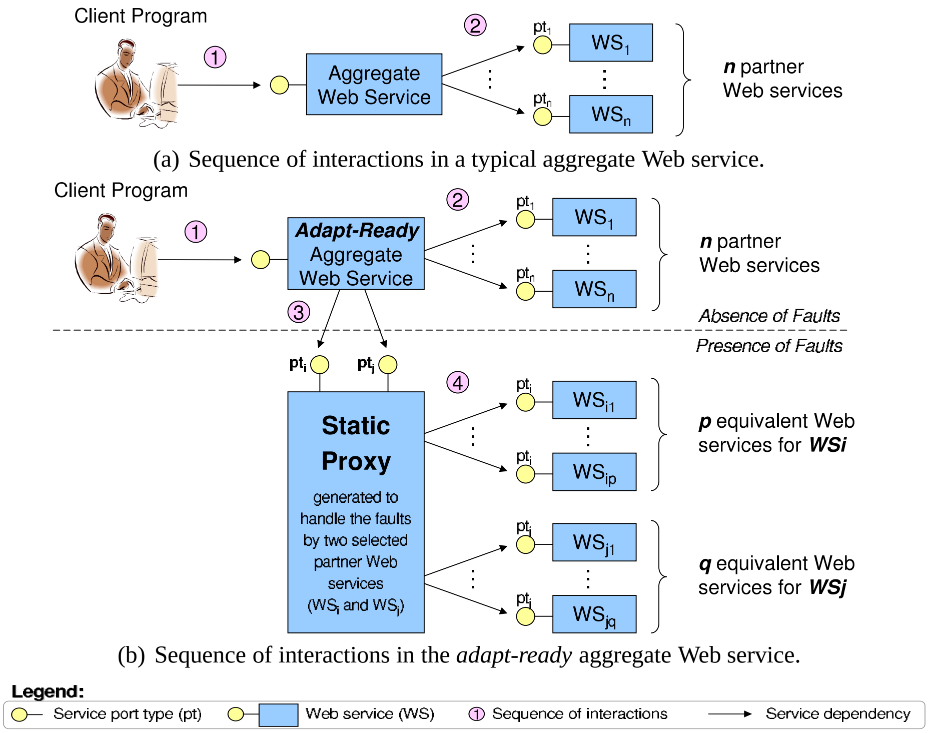
As monitoring all the partner Web services might not be necessary, the developer can select only a subset of Web service partners to be monitored. For example, in Figure 3(b) WS_i and WS_j have been selected for monitoring. The adapt-ready process monitors these two partner Web services and in the presence 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). The static proxy in its turn forwards the request to an *equivalent* Web service, which is "hardwired" into the code of this proxy at the time it was generated.[2] This means that the number of choices for equivalent services are limited to those known at the time the static proxy was generated.

2.3 Why Dynamic Proxies?

Given the rapid uptake of the service oriented programming model, we expect the emergence of numerous services that are functionally equivalent with one another and thus can be substituted [17]. For instance, in our driving-direction example (Figure 1), if the

[1] *Functional requirements* describe the interaction between the system and its actors (*e.g.,* end users and other external systems) independent of its implementation while *non-functional requirements* are those aspects of the system that are not directly related to the functional requirements (*e.g.,* QoS, security, scalability, performance and fault-tolerance).

[2] At this point in our research, we make the assumption that two services are *equivalent*, if they implement the same porttype (and thus business logic). A porttype is similar to an interface in the Java programming language. So, when two Web services implement the same porttype, only their internal implementations may vary, their interfaces remain the same. In other words, applications with the same functional requirement are equivalent, regardless of implementation.



(a) Sequence of interactions in a typical aggregate Web service.

(b) Sequence of interactions in the *adapt-ready* aggregate Web service.

Fig. 3. Architectural diagrams showing the difference between the sequence of interactions among the components in a typical aggregate Web service and its corresponding generated adapt-ready version.

default map generation service provided by Google fails, it should be possible to substitute this service with that of MSN, Yahoo! or Mapquest. Also, in Grid programming environments [30] where scientific applications (*e.g,* for Bioinformatics and Computational Chemistry) are run on computational Grids, a failed (or slow) node can be replaced by another node on the Grid [31].

In this paper, we extend RobustBPEL-1 in two directions: (1) by replacing static proxies with *dynamic* proxies that can find equivalent services at run time (described in Section 3); and (2) by adding self-optimizing behavior in existing BPEL processes, which is demonstrated using the case studies in Section 4.

3 Dynamic Proxies

In our approach, a *dynamic proxy* is a Web service that corresponds to a specific adapt-ready BPEL process and its job is to discover and bind *equivalent* Web services that can substitute for the services being monitored in the adapt-ready process. In this section, we first provide an architecture that shows the high-level functionality of a dynamic proxy and its interactions with other services in the architecture. Next, we explain how the adapt-ready BPEL process is instrumented so that it can monitor its partner Web services and interoperate with the dynamic proxy. Then, we explain how dynamic proxy interacts with a registry service to get the required information about the substitute services. Finally, we show a high-level view of the RobustBPEL-2 Generator that automatically generates the adapt-ready BPEL and dynamic proxy for an existing BPEL process.

3.1 High-Level Architecture

Figure 4 illustrates the architectural diagram of an application using an adapt-ready BPEL process augmented with its corresponding dynamic proxy. This figures shows the steps of interactions among the components of a typical adapt-ready BPEL process. Let us remind you that when static discovery is used, services that can substitute for the monitored services are noted and tightly associated with the code for the static proxy (see the architectural diagram for static proxies in Figure 3). For the dynamic proxy, however, a look-up mechanism is utilized to query a registry service at runtime for services that can be used to replace failed services.

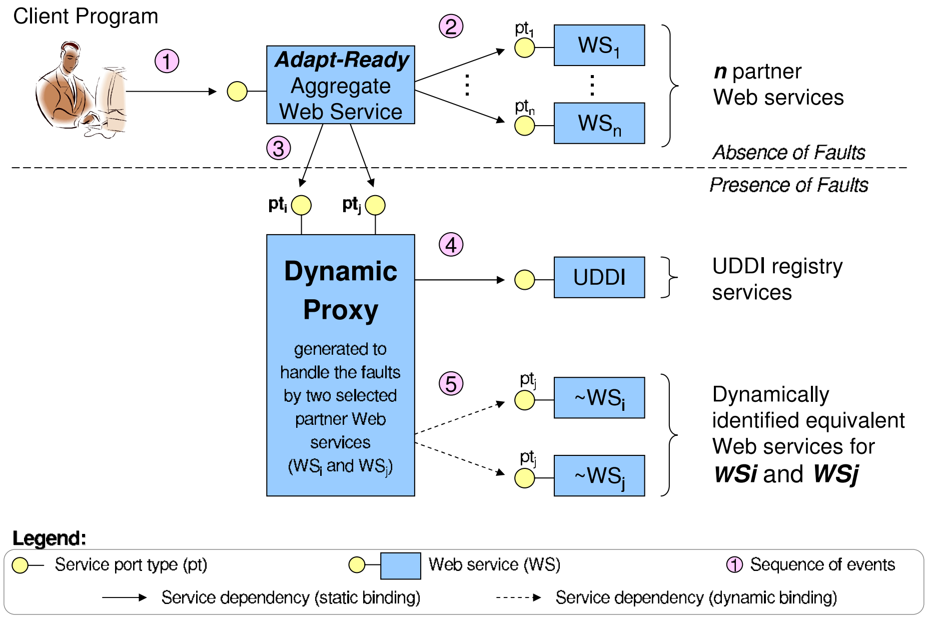

Client Program
1
2
3
4
5
Adapt-Ready Aggregate Web Service
pt_1
WS_1
pt_n
WS_n
n partner Web services
Absence of Faults
Presence of Faults
pt_i
pt_j
Dynamic Proxy
generated to handle the faults by two selected partner Web services (WS_i and WS_j)
UDDI
UDDI registry services
pt_i
$\sim WS_i$
pt_j
$\sim WS_j$
Dynamically identified equivalent Web services for WSi and WSj
Legend:
Service port type (pt)
Web service (WS)
Sequence of events
Service dependency (static binding)
Service dependency (dynamic binding)

Fig. 4. Architectural diagram showing the sequence of interactions among the components in an adapt-ready BPEL process augmented with its corresponding *dynamic* proxy.

Similar to a static proxy, the interface for the generated dynamic proxy is exactly the same as that of the monitored Web service. Thus, the operations and input/output variables of the proxy are the same as that of the monitored invocation. When more than one service is monitored within a BPEL process, the interface for the specific proxy is an aggregation of all the interfaces of the monitored Web services. For example, the dynamic proxy in Figure 4 has pt_i and pt_j, which are the port types of the two monitored Web services (namely, WS_i and WS_j). At runtime, if a monitored service fails (or an invocation timeout occurs), the input message for that service is used as input message for the proxy. The proxy invokes the equivalent service with that same input message. A reply from the substitute service is sent back to the adapted BPEL process via the proxy.

Although the adapt-ready BPEL process remains a functional Web service and the proxy is an autonomic Web service (encapsulates autonomic attributes), functional Web services can behave in an autonomic manner by using autonomic Web services [16]. In our architecture, by replacing failed and delayed services with their equivalents (substitutes), the proxy Web service provides self-healing and self-optimization behavior to the BPEL process, thereby making the BPEL process autonomic. Case studies in Section 4 demonstrate the autonomic behavior of some adapt-ready BPEL processes.

3.2 Incorporating Generic Hooks inside the Adapt-Ready BPEL Processes

Following the Transparent Shaping programming model [28], 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) [27]. The monitoring code we insert is in the form of standard BPEL constructs to ensure the portability of the modified process.

We adapt the existing BPEL process by identifying points in the process at which external Web services are invoked (interaction joinpoints) and then wrapping each of those invocations with a BPEL `scope` that contains the desired fault and event handlers. A fault can be a programmatic error generated by a Web service partner of the BPEL process or unexpected errors (*e.g.*, service unavailability) from the Web service infrastructure. The following snippet BPEL code (Figure 5) is an example of a service invocation in BPEL. Lines 3 and 4 identify the interface (`portType`) of the partner and what method (`operation`) the invocation wishes to call.

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

Fig. 5. An unmonitored invocation.

The invocation showed in Figure 5 is identified and wrapped with monitoring code. The code in Figure 6 shows what the invocation looks like after the monitoring code is wrapped around it. The unmonitored invocation is first wrapped in a `scope` container which contains fault and event handlers (lines 5-14 and 15-19 respectively in Figure 6). A `catchAll` fault handler is added (lines 6-13) to the `faultHandlers` to handle any faults generated as a result of the invocation of the partner Web service. The fault-handling activity is defined in lines 7-12, which basically forwards the request to the dynamic proxy. 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.   <!-- linking instructions -->
5.   <faultHandlers>
6.     <catchAll>
7.       <invoke name="InvokeProxy"
8.               partnerLink="..."
9.               portType="pti"
10.              operation="operation1"
11.              inputVariable="..."
12.              outputVariable="..."/>
13.    </catchAll>
14.  </faultHandlers>
15.  <eventHandlers>
16.    <onAlarm for="’PT1S’">
17.      <throw faultName="bpws:forcedTermination"/>
18.    </onAlarm>
19.  </eventHandlers>
20.  <invoke name="InvokeWSi"
21.          partnerLink="..."
22.          portType="pti"
23.          operation="operation1"
24.          inputVariable="..."
25.          outputVariable="..."/>
26.</scope>
```

Fig. 6. A monitored invocation.

For the event handler, an `onAlarm` event handler (lines 16-18) is used to specify a timeout. An `onAlarm` clause is used to specify a timeout “event” in BPEL. A timeout can be used, for instance, to limit the amount of time that a process can wait for a reply from an invoked Web service. A `throw` activity is inserted inside the `onAlarm` event handler (line 17) as the action that is carried out upon the timeout. If the partner service fails to reply within the time stipulated in the timeout event, the `throw` activity generates a standard BPEL `forcedTermination` fault. This fault forces the monitored invocation to terminate. The generated `forcedTermination` fault is then caught by the fault handler and the proxy service is invoked (lines 7-12) as a substitute.

We note that the monitoring code in Figure 6 differs from the code generated by RobustBPEL-1 [14]. One major difference is that in RobustBPEL-1, we added a proxy invocation code to handle the events resulting from delay in service, which is now replaced by the `throw` activity (line 17). The reason for this change is that after the expiration of the timeout, the monitored invocation is not terminated. This means that it is possible to receive a reply from the monitored Web service (although after some delay) and another one from the proxy, which causes the BPEL process to become unstable. This led to the need to use the `forcedTermination` fault to ensure that the monitored invocation is destroyed upon the occurence of a timeout event.

3.3 Interaction of Dynamic Proxy with the Registry Service

When the dynamic proxy is invoked upon failure of a monitored service, the proxy makes queries against the registry service to find equivalent services. At runtime, any service provider can publish new equivalent services with the registry, which can potentially substitute a failed service in the future.

The registry technology used in the RobustBPEL-2 toolkit is the Universal Description, Discovery and Integration protocol (UDDI) [7], 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. The registry allows for three categories of information to be published: (1) *white pages* that contain contact information such as the name, address and telephone number of a given business; (2) *yellow pages* that contain information that categorizes businesses based on some existing taxonomies; and (3) *green pages* that contain technical information about the Web services provided by the published businesses (this can include the URL of the service and its WSDL).

There are four key components of the UDDI data model: businessEntity, businessService, bindingTemplate and tModel. The information published in the white, yellow and green pages are captured under businessEntity, businessService and bindingTemplate, respectively. The tModel (or technical model) is used to represent service taxonomies as well as store metadata about a service, such as the location of its WSDL [22].

In order to adequately categorize services in a UDDI registry, certain conventions have to be adhered to. The method of classification we use focuses on registering services based on the information in their WSDL descriptions, in other words, mapping WSDL to UDDI. Figure 7 (derived from [22]) helps to illustrate how this mapping is achieved. Certain information contained in parts of a service's WSDL definition are stored in UDDI tables. For clarity, not all the utilized tables are shown in Figure 7 and some fields of the shown tables have been omitted. Details about the port types and bindings of the WSDL are stored as tModels, the information stored include the names of the port types and bindings, and the location of the WSDL. The protocol, transport and the portType for each binding is also registered.

Information about the WSDL services and their ports are stored under businessService and bindingTemplate, respectively. Data registered includes the URL for each port. At this stage of our work, no selection criteria is used when multiple services are found. The services are just chosen arbitrarily from a list, although some selection policies can be easily incorporated into the proxy to introduce some added quality-of-service.

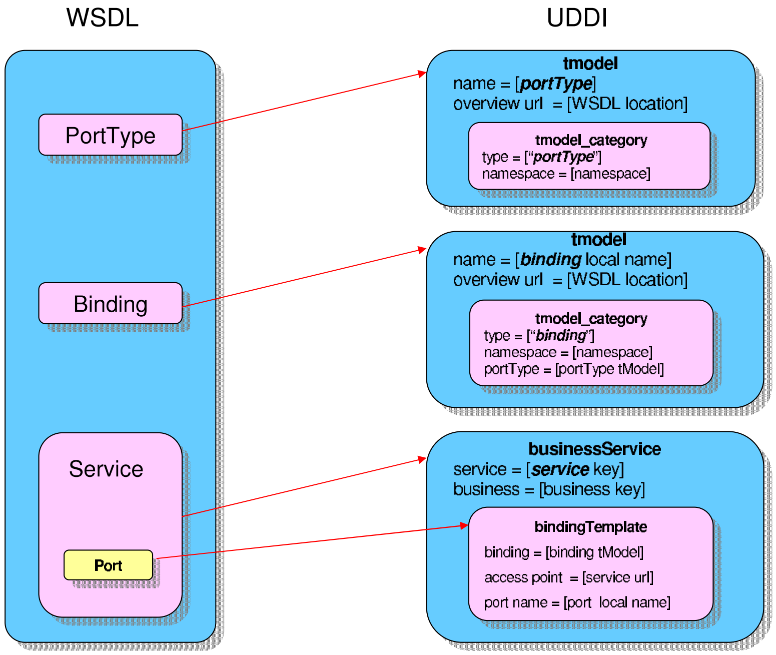


Fig. 7. Mapping WSDL to UDDI.

3.4 The Generation Process in RobustBPEL-2

As part of the RobustBPEL-2 toolkit, we developed the `RobustBPEL-2 Generator` that automatically generates the adapt-ready version of a given BPEL process and its associated dynamic proxy. The input to this generator is a configuration file. Figure 8 shows the contents of a configuration file that has all the required information: lines 2-10 specify the input needed for the generation of the adapt-ready BPEL process, while lines 11-36 specify that for the generation of the proxy. As illustrated in Figure 9, first, the `Parser` separates the information needed for generating adapt-ready BPEL process and for generating the dynamic proxy and sends them to the corresponding compilers. Next, each of these two generators uses the information provided by the parser and retrieves the required files from the local disk and starts its compilation process.

The `Adapt-Ready BPEL Compiler` retrieves the original BPEL process using the location of the original BPEL file, which is stated in line 5. It then uses the `<adapt>` element (lines 7-9) to find out the names of the invocations to be monitored. An `<invoke>` element (line 8) with a "*" as the value of the name attribute declares that all invocations should be monitored. With all this information, the `Adapt-Ready BPEL Generator` is ready to starts its compilation process.

The `Dynamic Proxy Compiler` gets the location of the proxy template is on line 12 and the necessary information about every monitored service type (classified by portType) from the `<substitutes>` tags (lines 19-34). The information needed to generate the proxy code to find and bind to services that implement the portType pt_i

```
1. <generator>
2.   <bpel>
3.     <processName name="processName"/>
4.     <targetNamespace name="http://..."/>
5.     <inputFile name="originalBPEL.bpel"/>
6.     <outputFile name="adaptReadyBPEL.bpel"/>
7.     <adapt>
8.      <invoke name="*" timeout="’PT1S’"/>
9.     </adapt>
10.  </bpel>
11.  <proxy type="dynamic">
12.    <templateFile name="DynamicProxy-Template.java"/>
13.    <outputFile name="DynamicProxy.java"/>
14.    <wsdlFiles>
15.      <wsdl name="wsins" url="WSi.wsdl"/>
16.      <wsdl name="wsjns" url="WSj.wsdl"/>
18.    </wsdlFiles>
19.    <substitutes>
20.      <service name="WSiService"
21.               portType="pti"
22.               wsdl="lns:wsins">
23.        <operations>
24.          <operation name="operation1"
25.                     port="Port"/>
26.        </operations>
27.        <query businessKey="BK1"
28.               serviceName="WSiService"/>
29.        <stub name="stub.wsi"/>
30.      </service>
31.      <service name="WSjService"
32.        ...
33.      </service>
34.    </substitutes>
35.    <package name="processName.proxy.dynamic"/>
36.  </proxy>
37.</generator>
```

Fig. 8. Configuration file for the generator

is listed within the `<service>` elements of lines 20-30 and the operations of every monitored invocation of the portType are listed within the `operations` element. Next, the The `Dynamic Proxy Generator` finds out about the location of the binding stubs for services of the portType are in line 29 (this stub package is associated with the proxy as a Java import statement). It then gets the information needed to query the registry for services that implement the portType from the `<query>` tag (lines 27-28). Finally, it gets the the package for the proxy class from Line 35 (the broker binding stubs are part of this package) and compiles the dynamic proxy.

4 Case Studies

In this section, we use two case studies to demonstrate the self-healing and self-optimization behavior of the generated BPEL processes and their respective dynamic proxies. To better demonstrate the applicability of our approach to any BPEL process, we tried to use existing BPEL processes that are not originated by us. For each case study, we start by describing the application, then we present the configuration of the experiment environment. Finally, we show the results of the experiment.

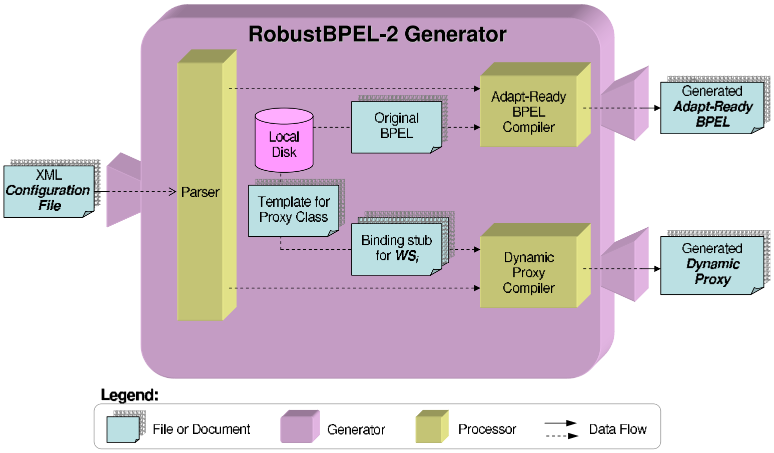
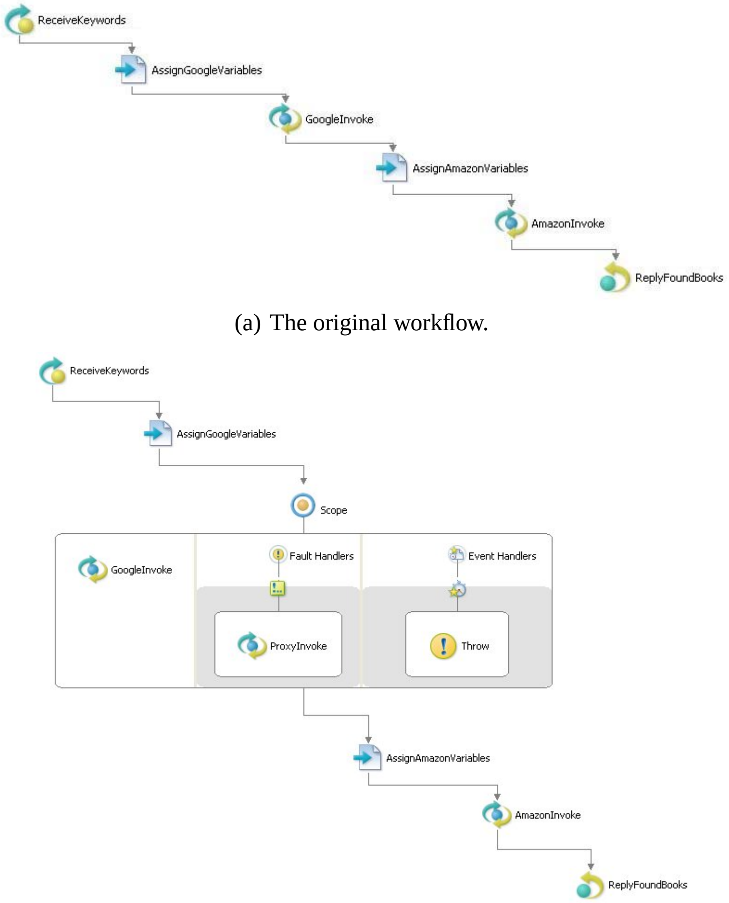


Fig. 9. Inputs and outputs of the dynamic proxy generator.

4.1 The Google-Amazon Process

The Google-Amazon business process is an idea we got from Proteus [6]. The Proteus Runtime Integrator (RTI) aggregates Web services by using relational algebra operators. An example application that was created using the Proteus RTI is one that integrates the Google Web service for spelling suggestions [4] with the Amazon E-Commerce Web service [2] for querying its store catalog. In the Proteus application, the name of an author is sent to the Google spell-checker, if the name of the author is misspelled, the Google spell-checker sends back as reply the correct spelling for the name (the name is unchanged if the spelling is correct). The reply from the Google service (author's name) is used to compose a query for the Amazon Web service (bookstore), to find books by the given author. Our Google-Amazon process differs from the Proteus application in that, our process takes as input a set of keywords rather than an author's name. These keywords are sent to the Google spell-checker for corrections, the reply from the spell-checker is then used to make a keyword search in the Amazon bookstore. Figure 10(a), is a screenshot of the workflow for the Google-Amazon process.

From this original Google-Amazon process, we used the generator to generate the adapt-ready process (Figure 10(b)) and its associated dynamic proxy. For this adaptation we have selected to have the generator only wrap the invocation of the Google spell-checker with the monitoring code, thus the proxy service will only have the same interface as that of the Google service. We then found another publicly available Spell-checker Web service from Cydne [3] 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 (keywords, in our case). The Cydne service also takes as input a license key and a phrase but in the reverse order. Finally, rather than returning a string in which misspelled words have been corrected, the



(a) The original workflow.

(b) The adapt-ready workflow.

Fig. 10. The workflow diagram of the Google-Amazon BPEL Process.

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 (Jazzy [5]) Jazzy takes as input a dictionary and a misspelled word, it then returns a ranked list of suggestions from the dictionary (returns nothing if no match is found). 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. Since the wrapper Web service is specifically for the Cydne spell-checker, it contains the binding information for that service (including the license key). 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.

The Experiment Configuration and Result. We use a total of four machines to conduct our experiments: PC-1, PC-2, PC-3 and PC-4. All the machines have Windows XP as their operating system and running on each (except PC-1) is an Apache Tomcat Web server that includes an Apache AXIS SOAP engine. A BPEL client application written in Java is deployed on PC-1. An Active-Bpel engine [1] is deployed on PC-2. This BPEL engine is deployed as a Web application under the Tomcat Web server and is the platform on which the adapt-ready Google-Amazon BPEL process is hosted. The dynamic proxy is hosted on PC-3, while the UDDI registry and the wrapper Web service are deployed on PC-4. For UDDI registry, we have chosen JUDDI.

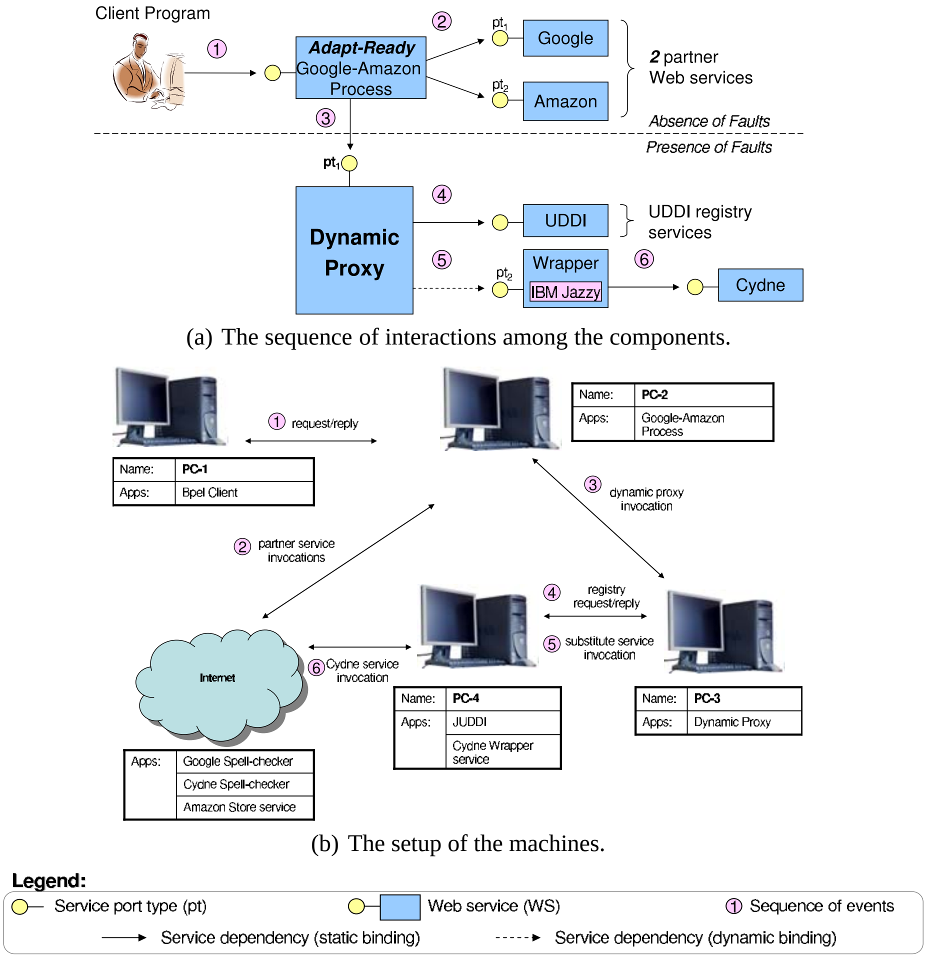
As illustrated in Figure 11(b), client requests are made to the BPEL process on PC-2 (labeled 1), which results in the invocations to the Google Web services (labeled 2). To simulate the unavailability of the Google service, we changed the URL of the service from within the Google-Amazon process, to point to a non-existent address. Thus upon the imminent failure of the invocation for the Google service, the adapt-ready BPEL process invokes the dynamic proxy (labeled 3). The dynamic proxy first queries the JUDDI registry on PC-4 for substitute services (labeled 4). As a result of the query, it finds the wrapper Web service for the Cydne spell-checker. The proxy then binds to the wrapper service, which in turn binds to the Cydne spell-checker with the input keywords (labeled 5 and 6, respectively). The result of this invocation 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 (or the wrapper) corrected it to "Computer Algorithms", and Amazon found this book: "Bruce Schneier, Applied Cryptography: Protocols, Algorithms, and Source Code in C, Second Edition".[3]

Figures 12(a) and 12(b) are screenshots showing the execution graphs of the Google-Amazon process when the process executes normally and with a fault. The execution graphs were generated by the BPEL engine. Although we currently have one alternate spell-checker, more spell-checker services can be added to the registry as they become available. Currently, the selection method for the proxy is random, when more than one substitute service is discovered in the registry. Some QoS selection criteria could be used instead, this is part of our planned future work.

4.2 The Loan Approval Process

The loan-approval process is a commonly used sample BPEL process [1]. The loan-approval BPEL process is an aggregate Web service (`LoanApproval`) composed of two other Web services: a risk assessor service (`LoanAssessor`) and a loan approver service (`LoanApprover`). The loan-approval process implements a business process that uses its two partner services (`LoanAssessor` and `LoanApprover`) to decide whether a

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



(a) The sequence of interactions among the components.

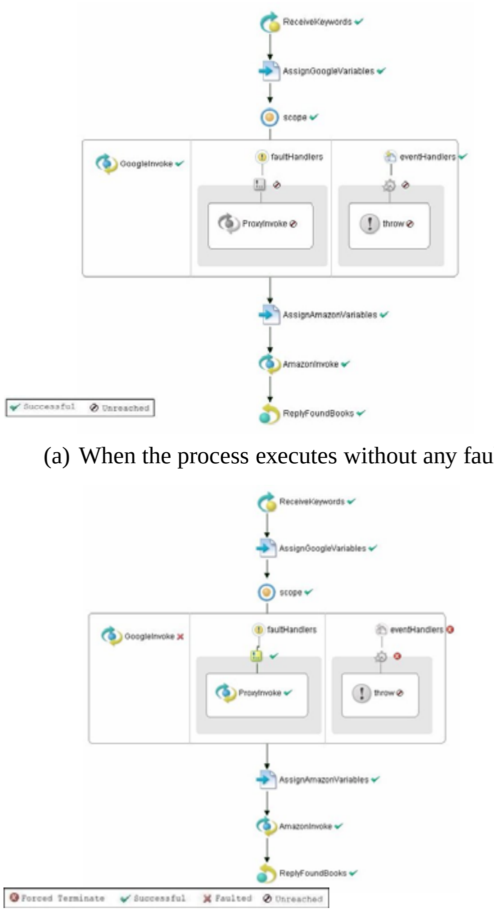
(b) The setup of the machines.

Legend:
Service port type (pt)
Web service (WS)
1 Sequence of events
Service dependency (static binding)
Service dependency (dynamic binding)

Fig. 11. The Google-Amazon case study.

given individual qualifies for a given loan amount. In this case study, we assume that we have access to the loan-approval BPEL process, but the two other Web service partners are developed, deployed and managed by third parties and they are outside our control.

As illustrated in Figure 13(a), the loan-approval BPEL process receives as input a loan request (`ReceiveCustomerRequest`). 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.



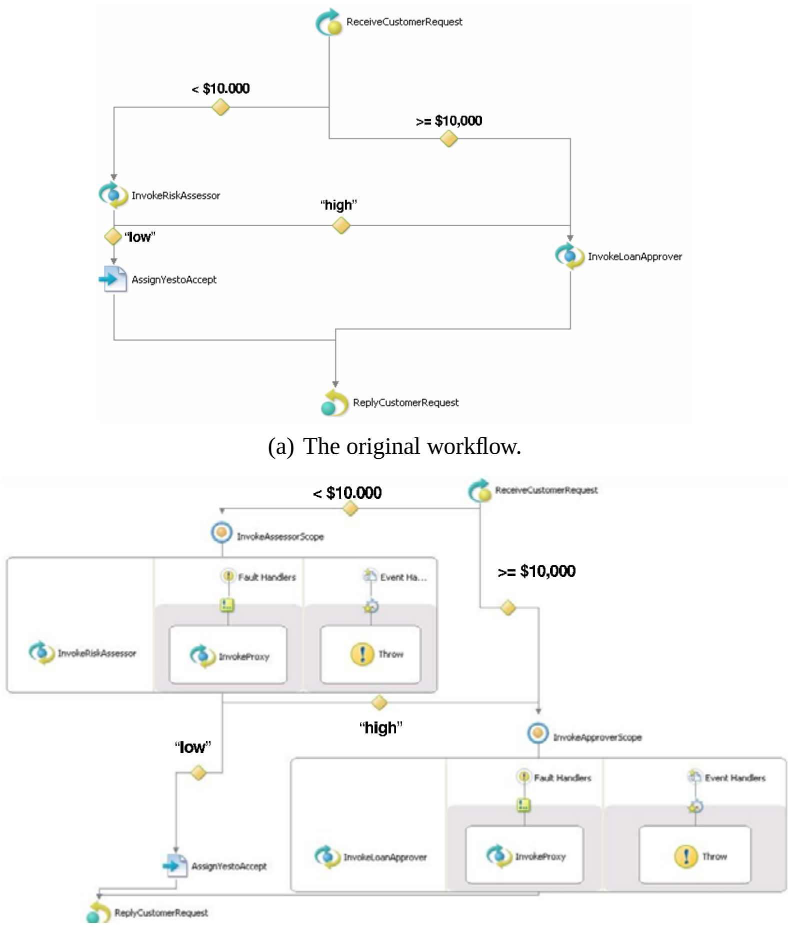
(a) When the process executes without any faults.

(b) When the Google service is unavailable.

Fig. 12. The execution graph of the Google-Amazon BPEL process.

This risk assessment message is a string with a value of either “high” or “low”. When the risk assessment is “low”, it means that 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).

We note that both the risk assessor and the loan approver Web services were implemented previously by ActiveWebflow in Java [1].



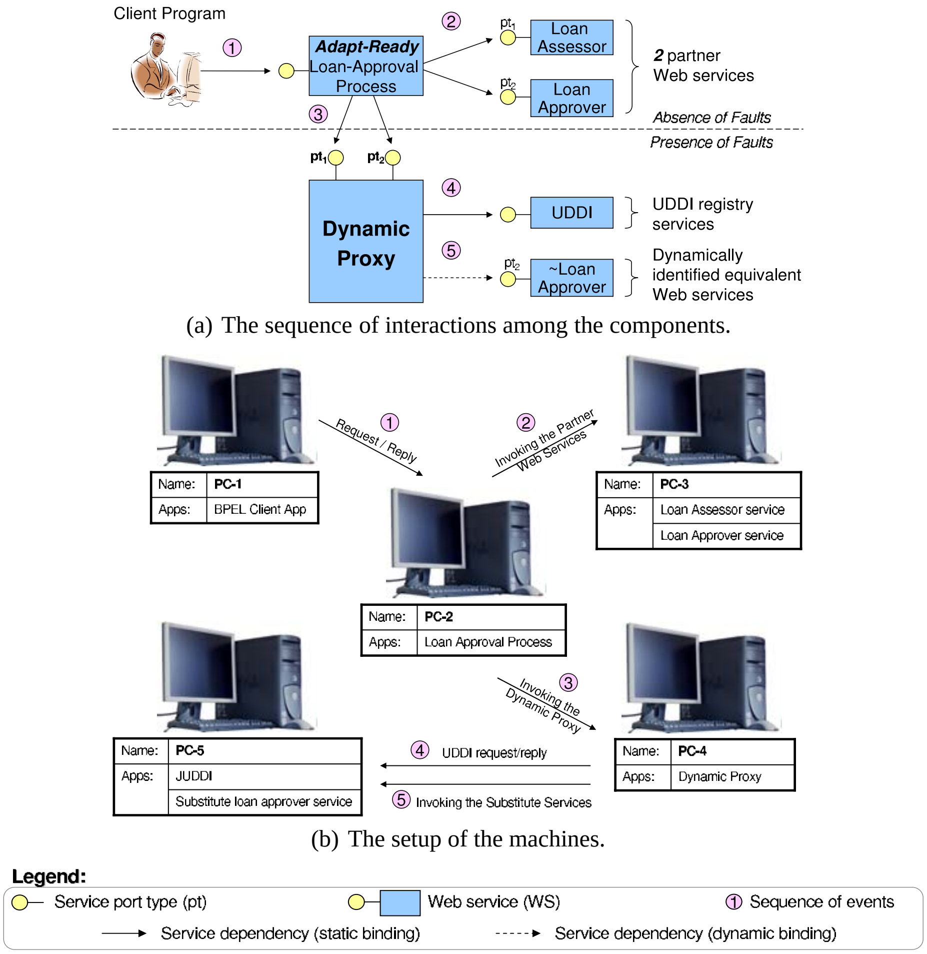
(a) The original workflow.

(b) The adapt-ready workflow.

Fig. 13. The workflow diagram of the loan-approval BPEL Process.

The Experiment Configuration. We have used a total of five machines to conduct our experiments: PC-1, PC-2, PC-3, PC-4 and PC-5. All the machines have Windows XP as their operating system and running on each is an Apache Tomcat Web server. Also deployed on each Web server is an Apache AXIS SOAP engine. A multi-threaded BPEL client application written in Java is deployed on PC-1. An Active-Bpel engine [1] is deployed on PC-2. This BPEL engine is deployed as a Web application under the Tomcat Web server and is the platform on which the loan-approval BPEL process was hosted. The loan-assessor and loan-approver Web services are deployed on PC-3. The dynamic proxy is hosted on PC-4, while the UDDI registry and alternate loan-approver Web services are all deployed on PC-5. For UDDI registry, we chose JUDDI. JUDDI is an open source Java implementation of the Universal Description, Discovery, and Integration (UDDI) specification for Web services from the Apache Software Foundation.

As illustrated in Figure 14(b), client requests are made to the BPEL process on PC-2 (labeled 1), which results in the invocations to the partner Web services on PC-3 (labeled 2). Upon failure of these partner services or an invocation timeout, the adapt-ready BPEL process invokes the dynamic proxy (labeled 3). The dynamic proxy first queries the JUDDI registry on PC-5 for substitute services (labeled 4). The result of the query is used to bind the substitute service on PC-5 and forward the requests to this service (labeled 5).



(a) The sequence of interactions among the components.

(b) The setup of the machines.

Fig. 14. The Loan Approval case study.

Self-Healing and Self-Optimization. In order to demonstrate the autonomic behavior of the generated BPEL process and its corresponding dynamic proxy, we have programmatically altered the Loan Approver Web service deployed on PC-3 to generate faults and a delay of two seconds after a certain number of successive invocations. The successive invocations to the Loan Approver Web service are the results of requests to the BPEL process made by the client application. These requests are mapped on the X axis of the chart shown in Figure 15. As the plot for the original BPEL shows, for the successive invocations 11 to 20, the Loan Approver Web service generates a fault for those invocations, and for the invocations 31 to 40, the Loan Approver Web service is made to delay for 2 seconds before sending back a reply to the BPEL process. The fault generation is meant to simulate a problematic Web service, a server crash, or a network outage and the delay is meant to simulate an overly loaded Web service or its corresponding host (in this case PC-3). We set the timeout duration for the Loan Approval BPEL process to 1 second.

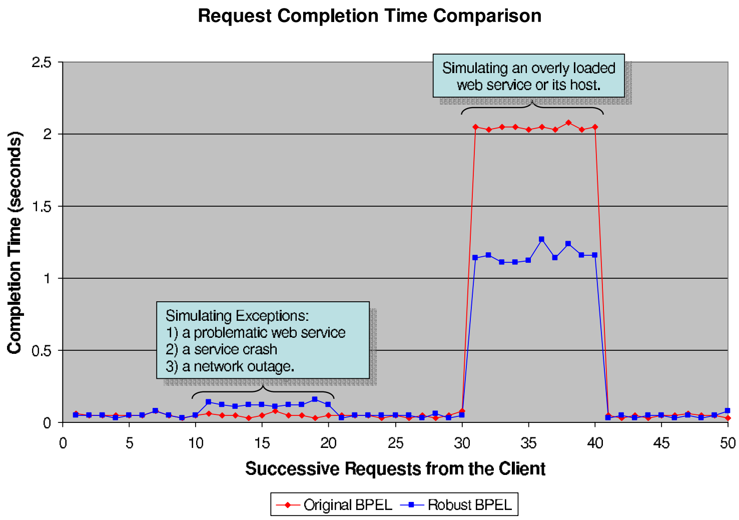


Fig. 15. This chart shows the comparison between the request completion time for the original and the robust BPEL processes.

Figure 15 plots the request completion time for the two sets of experiments (one with the original BPEL process and the other with the generated BPEL process) for the 50 successive requests from the client program to the Loan Approval BPEL process. One plot reflects the behavior of the original BPEL process and the other one reflects the behavior of the generated robust BPEL process and its corresponding dynamic proxy. According to the experiment setup, the first 10 request are completed normally as there are no fault generated or no delay is added on the execution path of these request. As expected, the average completion time for both the original and the robust sets of experiments are almost the same (about 47 milliseconds). This result indicates that in normal operation, the overhead added by the robust BPEL process is negligible.

Right after the completion of the first 10 requests, the Loan Approver Web service starts throwing exceptions for the next 10 requests. Although Figure 15 shows that the completion time for the original BPEL stays as before, but all the requests are returned with exception and the results are of no use. In other words, the original BPEL process fails its clients. The robust BPEL process, however, catches all such exceptions and uses the dynamic proxy, which resides on PC-4, to find an equivalent service. In its turn, the dynamic proxy uses the UDDI server on PC-5 and finds the substituent service also deployed on PC-5. The plot for robust BPEL in Figure 15 shows an increase in the completion time, which is about 127 milliseconds. We believe that for many applications, the extra 80 milliseconds overhead is much more desirable than receiving a faulty response.

For the next 10 requests (21 to 30), the Loan Approver Web service goes back to its normal behavior and responds to the requests without throwing exceptions. As can be seen from Figure 15, the robust BPEL uses the original Web service and in essence optimizes the completion time for these 10 requests. As illustrated in the original BPEL plot, for the next 10 requests (31 to 40), the Loan Approver Web service responds to the requests after 2 seconds of delay. As the time out in the robust BPEL process is set to 1 second, the robust BPEL process withdraws its invocations to the original Loan Approver Web services after 1 second and uses the substitute Web service. In this way, the robust BPEL process completes the request in almost half the time as that of the original BPEL process.

5 Related Work

Dialani et al. [12] provide an approach to enabling fault tolerance in *stateful* Web services by requiring the developer to implement an interface for rollback and checkpoint. In their approach, the service interface of the targeted Web services must be extended to include the methods for fault tolerance. With the use of global and local fault managers, they monitor the Web services. Local fault managers are implemented as libraries that are dynamically bound to the service code. Checkpoints are used locally and a rollback can be initiated locally or globally after a fault is detected. Further, an extension is made to the SOAP communication layer so that messages can be logged and replayed. This work is complementary to ours but it focuses on *stateful* Web services while we specifically focus on *aggregate* Web services with the assumption that the partner Web services are stateless.

Birman et al. [9] 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). CHM represents new services that are used to track the health of individual Web service. CRM involves the use of a group communication interface and an extension to TCP to achieve TCP stream replication over a set of group members. CHM and CRM can be used transparently by Web services routers without the need to modify the client or server applications. MDC deals with monitoring by tracking performance metrics and other state variables and reporting them out. DDS focuses on reliable streaming of data by a service to its clients. EVN is a mechanism for sending urgent one-time event notifications to the client. Similar to ours, this work aims to improve the reliability of Web serives, but it proposes extensions to the Web services architecture.

Baresi's approach [8] 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. 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.

Finally, BPELJ [20] 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.

6 Conclusion and Future Work

We presented an approach to transparently adapting BPEL processes to tolerate runtime and unexpected faults and to improve the performance of overly loaded Web services. We have introduced the dynamic proxy and demonstrated how it is used to encapsulate autonomic behavior. With the use of case studies, we demonstrated the self-healing and self-optimization behavior of the dynamic proxy.

In our future work, we plan to address the following issues. First, we plan to provide a GUI for the generator so that the configuration file can be generated automatically and the developer does not have to write the configuration file manually, which is a cumbersome and error-prone task. Second, we realize that the performance of the dynamic proxy can be improved further by using a caching mechanism to avoid making repetitive calls to the UDDI registry. Third, substituting service implementations at runtime may lead to failures on the client-side [11]. Thus it is important to be able to detect and resolve potential integration problems from discovered equivalent services. This could include making sure that the substitute services would actually fulfill their functional requirements. Fourth, 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 [18]. Fifth, describing what constitutes an equivalent service can be a contentious issue, there is therefore a need for a formal definition of service equivalence. Sixth, we plan to study the existing ranking systems for

Web services and add this logic to the dynamic proxy. Finally, rather than the current specific proxies, a *generic* proxy that has a standard interface and works for all monitored services would make life much simpler for developers. We are currently working on the architecture for such a proxy.

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://acrl.cis.fiu.edu/`. A technical report of this paper with more detail figures and information is available at [15].

Acknowledgements. This work was supported in part by IBM SUR grant.

References

1. Active Webflow. Available at URL: `http://www.active-endpoints.com/products/`.
2. Amazon E-commerce service. Available at URL: `http://www.amazon.com`.
3. Cydne Spell-checker Web service. Available at URL: `http://www.amazon.com`.
4. Google Web API. Available at URL: `http://www.google.com/apis/`.
5. Java platform's Jazzy spell checker API. Available at URL: `http://www-128.ibm.com/developerworks/java/library/j-jazzy/`.
6. Proteus Runtime Integration. Available at URL: `http://dblab.usc.edu/WebServices/`.
7. B. Atkinson et al. *UDDI Version 3.0.1*. OASIS, 2003.
8. 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.
9. 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.
10. D. Booth et al. . *Web Services Architecture*. W3C, 2004.
11. G. Denaro, M. Pezze, and D. Tosi. Adaptive integration of third party web services. In *in Proceeding DEAS 2005*, St. Louis, Missouri, USA, May 2005.
12. V. Dialani, S. Miles, L. Moreau, D. D. Roure, and M. Luck. Transparent fault tolerance for web services based architectures. In *Eighth International Europar Conference (EURO-PAR'02)*, Lecture Notes in Computer Science, Padeborn, Germany, Aug. 2002. Springer-Verlag.
13. E. W. Dijkstra. Structured programming. *Software Engineering Techniques, edited by Buxton and Randell (available from NATO, Brussels)*, pages 84–87, 1970.
14. 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.
15. 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.

16. S. Gurguis and A. Zeid. Towards autonomic web services: Achieving self-healing using web services. In *Proceedings of DEAS'05*, Missouri, USA, May 2005.
17. 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.
18. 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.
19. J. O. Kephart and D. M. Chess. The vision of autonomic computing. *IEEE Computer*, 36(1):41–50, 2003.
20. M. Blow et al. *BPELJ: BPEL for Java, A Joint White Paper by BEA and IBM*. BEA and IBM, March 2004.
21. M. Gudgin et al. *SOAP Version 1.2*. W3C, 1.2 edition, 2003.
22. A. T. Manes. Registering a Web service in UDDI. Available at URL: `http://webservices.sys-con.com/read/39868.htm`.
23. D. C. Marinescu. *Internet-Based Workflow Management: Towards a Semantic Web*. Wiley-Interscience, 2002.
24. P. K. McKinley, S. M. Sadjadi, E. P. Kasten, and B. H. C. Cheng. Composing adaptive software. *IEEE Computer*, pages 56–64, July 2004.
25. C. Peltz. Web services orchestration a review of emerging technologies, tools and standards. *Technical Paper*, January 2003.
26. R. Chinnici et al. *Web Services Description Language (WSDL) Version 2.0*. W3C, 2.0 edition, March 2004.
27. 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.
28. 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.
29. D. Sherman. Business flows with BPEL4WS. *Online article*, 2005. Available at URL: `http://xml.sys-con.com/read/39780.htm`.
30. 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.
31. 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.
32. 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.
33. S. Weerawarana and F. Curbera. Business process with BPEL4WS: Understanding. *Online article*, 2002.