Enhancing BPEL scenarios with Dynamic Relevance-Based Exception Handling

Kareliotis Christos
Phd Candidate,
Department of Informatics and Telecommunications,
University of Athens
ckar@di.uoa.gr

Dr. Vassilakis Costas,
Assistant Professor,
Department of Computer Science and Technology,
University of Peloponnese,
costas@uop.gr

Dr. Georgiadis Panayiotis,
Associate Professor,
Department of Informatics and Telecommunicaions,
University of Athens,
georgiad@di.uoa.gr

Abstract

Web services have become the key technology in business processes management. Business processes can be self-contained or be composed from sub-processes; the latter category is typically specified using the Web Services Business Process Execution Language (WS-BPEL) and executed by a Web Services Orchestrator (WSO). During the execution however of such a composite service, a number of faults stemming from the distributed nature of the SOA architecture, e.g. network or server failures may occur. WS-BPEL includes provisions for exception handling, which can be exploited for detecting such failures; once detected, a failure can be resolved by invoking alternate web service implementations that perform the same business task as the failed one. However, the inclusion of such provisions is a tedious assignment for the business process designer, while additional effort would be required to maintain the BPEL scenarios in cases that some alternate WS implementations cease to exist or new ones are introduced. In our research we are developing a framework for automating handling of that kind of exceptions. The proposed solution employs a pre-processor that enhances BPEL scenarios with code that detects failures, discovers alternate WS implementations and invokes them, fully thus resolving the exception. Alternate WS implementation discovery is based on service relevance, which takes into account both functional and qualitative properties of web services.

Keywords: WS-Orchestration, Exception Handling, BPEL enhancement, Dynamic Service Discovery, Semantic Web, SWS, Service Relevance, Qualitative Attributes.

1. Introduction

Web services are unanimously supported by major software vendors of middleware technology [1]. The main objective of web service technology and related research [2] is to provide the means for enterprises to do business with each other and provide joint services to their customers under specified Quality of Service (QoS) levels. BPM addresses how organizations can identify, model, develop, deploy, and manage their business process, including processes that involve IT systems and human interaction. An important aspect of business processes is the definition of a binding agreement or contract between the two suppliers and customers, specifying QoS items such as deadlines, quality of products, and cost of services.

Business processes can be composed from sub-processes that act as atomic processes in the execution scenario. Composite services include two or more distinct services and are frequently specified using the Web Services Business Process Execution Language (WS-BPEL) and executed by a Web Services Orchestration (WSO) platform. If multiple, possibly cross-organization, business processes need to collaborate, their interaction can be modeled using the Web Services Choreography Description Language (WS-CDL).

Due to their distributed, heterogeneous and highly volatile nature, services-based systems are inherently vulnerable to exceptions: software, machine or communication link failures may render certain sub-process of composite services unavailable, precluding thus the successful execution of the business process. In addition to these transient failures, certain web services may be permanently withdrawn and/or alternatives to some services may be offered by different providers. In these cases, a replacement component should be identified and substituted for the failed one. The replacement component should have the “same skills” with the failed one i.e. to have same functionality and QoS [3]. Note that the dynamic and volatile nature of the execution environment implies that it is infeasible to list all possible alternatives in the BPEL execution scenario; instead, a dynamic approach should be adopted, where service selection is undertaken by a distinct module, which has access to a registry listing all pertinent functional and qualitative characteristics of available services, as illustrated in Figure 1. Component replacement should respect Service Level Agreement signed between the consumer and the provider; moreover, when a service selection succeeds a hot-swapper takes over in order to replace at run-time the corrupted service with the working one.

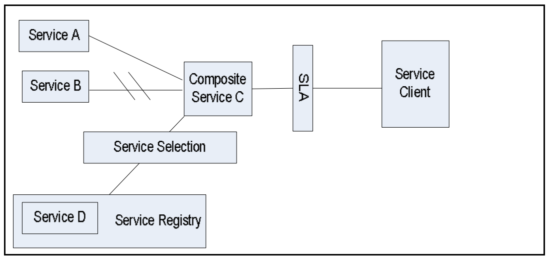

Figure 1. Replacing a failed component with one having "same skills"

The Service Relevance and Replacement Framework (SRRF) has been introduced in [4]; this framework is responsible of resolving execution exceptions occurred by a web service that becomes unavailable. Our approach is based on replacement of this web service as business process flow is performed. In this paper we refine the SRRF architecture, presenting the underpinnings that make such an approach possible to implement and feasible to maintain. As we shall present latter in this paper, no additional burdens are placed on the BPEL designer, since failure-handling code is automatically provided by the SRRF processor, which takes as input already designed BPEL scenarios and injects into them code for failure detection and hot-swapping; an additional SRRF component, the *Alternate WS Locator Module* is responsible for finding services that are functionally equivalent to the failed one, so as to be used in the hot-swapping procedure.
The rest of the paper is organized as follows: section 2 presents related work, while section 3 briefly presents the SOA and BPEL provisions used for our purposes. Sections 4 and 5 present the overall architecture and the specific components of the SRRF framework. Finally, in section 6 conclusions are drawn and future work is outlined.

2. Related Work

In orchestration, which is usually used in private business processes, a central process (which can be another web service) takes control of the involved web services and coordinates the execution of different operations on the web services involved in the operation. The involved web services do not "know" (and do not need to know) that they are involved in a composition process and that they are taking part in a higher-level business process. Only the central coordinator of the orchestration is aware of this goal, so the orchestration is centralized with explicit definitions of operations and the order of invocation of web services (see Figure 2).

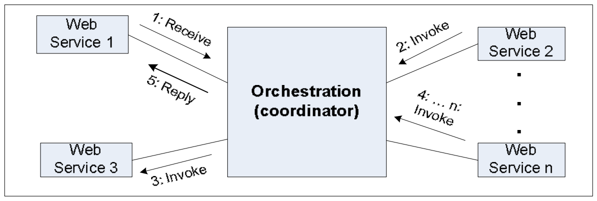

Figure 2. Composition of web services with orchestration

A BPEL process specifies the exact order in which participating web services should be invoked, either sequentially or in parallel. BPEL allows the expression of conditional behaviors; for example, an invocation of a web service can depend on the value of a previous invocation. The designer can also create loops, declare variables, copy and assign values, define fault handlers, and so on. By combining all these constructs, complex business processes can be specified in an algorithmic manner.
A BPEL processor is responsible for executing the BPEL scenario. There are many tools that provide BPEL execution environments and UML-like BPEL design environments such as ActiveBPEL [5], Oracle BPEL Process Manager in Oracle Application Server [6], Eclipse [7] and so forth.
While executing business processes, especially long-running ones, exceptions may occur. BPEL provides constructs to enable handling of exceptions (or *faults*, as termed in BPEL specifications), and our approach exploits exactly these constructs, thus no modification to BPEL itself or BPEL designer and orchestrator tools is needed. In order to better present the interaction between the SRRF and BPEL orchestrator, we first present briefly the fault handling capabilities of BPEL, and afterwards present how we take advantage of them to provide automatic replacement of some failed service with a functionally equivalent one in BPEL

processes.
The BPEL orchestrator handles exceptions occurred in the BPEL process runtime. As we shall see latter in this paper the logical faults in the business process execution are resolved through code explicitly provided by the BPEL designer for this purpose. Exceptions that occurred due to network, server or other system-related problems (*system faults*) are handled either by the failover and retry features of BPEL or in an execution environment-dependent fashion –e.g. while working with Oracle Process Manager, by sending an exception message in a JMS Dead Letter Queue. In our approach, we extend the simple "failover and retry" system fault resolution mechanisms of BPEL by introducing the dynamic discovery of "functionally equivalent" web services and using hot-swapping to substitute them for the failed one. The proposed approach is based on enhancing the BPEL scenario with code which intercepts system faults and invokes the Alternate WS Locator Module web service (an SRRF component).
In the past few years, the issue of exception resolution in composite web services has drawn the researchers' attention. A noteworthy approach to exception handling is the one undertaken by METEOR-S project [8], [9] in cooperation with WSMX (Web Services Execution Environment) [10]. WSMX contains the discovery component, which undertakes the role of locating the services that fulfill a specific user request. This task is based on the WSMO conceptual framework for discovery [11]. WSMO includes a Selection component that applies different techniques ranging from simple "always the first" to multi-criteria selection of variants (e.g., web services non-functional properties as reliability, security, etc.) and interactions with the service requestor. Both in the METEOR-S and other approaches, functional and non-functional properties are represented using shared ontologies, typically expressed using DAML+OIL and the latter OWL-S. Such annotations enable the semantically-based discovery of relevant web services and can contribute towards the goal of locating services with "same skills" [3] in order to replace a failed service in the process flow.
The main difference of our research with the one referenced above is that selection of replacements to services that have failed within an execution plan is made dynamically, instead of using pre-determined exception resolution scenarios. Replacement service selection is based on both functional equivalence (performed through semantic matching) and qualitative replaceability (considering non-functional attributes). Furthermore, qualitative replaceability criteria may be defined by the composite service invoker, to more accurately specify which replacement service is the most suitable one in the context of the current execution.

3. SOA provisions for fault handling

3.1 Logical versus System Faults

Business processes specified in BPEL will interact with partner processes through operation invocations on web services. Web services usually communicate over internet connections that are not highly reliable. Web services can also raise faults due to logical and execution errors. Therefore, BPEL business processes need to handle faults appropriately and may also need to signal faults themselves.
There are two kinds of faults that may occur in a BPEL process: *logical* and *system*. The first category includes those faults deliberately raised by constituent services to indicate that some form of special handling is required. For example, an *InsufficientCredit* exception thrown by some *CreditCardPayment* service indicates that payment through the credit card is impossible because the credit limit has been exceeded; the BPEL scenario designer may catch this fault type and either end the scenario or attempt to use alternative payment methods, such as direct withdrawal from a savings account or cash payment, if applicable. In such cases, there is no reason to try other implementations of the *CreditCardPayment* service with the same inputs, since they are bound to fail as well (the failure reason is independent of the specific invocation).
The second category, namely *system faults*, includes faults not directly raised by constituent services but rather detected by the execution environment. Examples of such faults are the inability to communicate with the hosting server (server down or network partitioning), system-generated responses indicating that the service is not offered at the specific address, parameter number or type mismatches (service has been altered) and timeouts in receiving replies. If a system fault occurs while executing a BPEL scenario, it is possible to remedy the situation by invoking some alternate implementation, since the fact that the particular invocation failed does not imply that other implementations will fail as well (the failure reason is directly bound to the particular invocation).
To make this distinction more clear and illustrate the SOA provisions for performing fault classification, consider the example of the *Book Purchase* composite web service that consists of a *Book Rating* Service and a *Credit Approval* Service. The first one ranks the offerings for the book that the client wants to purchase, with the cheapest one coming first. The second one, checks whether or not the client's credit card has sufficient credit for paying the book price.

In a WSDL description, logical faults can be specified through the *fault* constructs, which may in turn include any message elements further describing the error. Listing 1 presents an excerpt of the *CreditApprovalService*'s WSDL, in which the exception deliberately raised by the service's business logic (*InsufficientCredit*) is defined (the pertinent WSDL sections are included in boxes for reader convenience). For more information on these WSDL constructs, the interested reader is referred to [12].
As we shall show later in this paper, logical faults can be encountered by the BPEL process scenario. BPEL catches the fault and then sends a callback message to the client, since the services that comprise the composite web service have the appropriate code that handles the logical errors specified in the WSDL of *Credit Card Approval* service (Listing 1).

3.2 Fault handling in BPEL

The BPEL specification provides fault handling capabilities via the *faultHandler* construct. BPEL programmers are able to deal with different faults in catch-and-handle fashion. Sometimes faults, especially ones raised while executing an *invoke* activity, occur due to network instability or configuration changes that have not been reflected in the BPEL scenario. It would be tedious for the programmer to handle these kind of faults at the level of each and every invoke activity. The BPEL specification provides the following features to assist developer in dealing with these errors:

- *failover*: Allows multiple service implementations to be configured for a given *partnerLink*. If a retryable runtime fault (discussed in the following section) occurs, the BPEL orchestrator will try other implementations.

- *retry*: The BPEL orchestrator retries the invocation, using a user-specified retry interval and retry count.

Listing 2 shows how alternate implementations of a web service can be specified, while illustrates the BPEL definitions needed in order to perform the retry error handling features. We need to add within the *properties* element of Book Rating service all the services for the same functional requirements, which in this case is the booking service. If the first service fails, the alternate services declared are tried. It suffices for the BPEL designer to specify the address of the service description (WSDL). Note however that whether alternate implementation specifications are exploited as well as the specific way in which fallback is performed is dependent on the particular implementation of the BPEL orchestrator.

```
<definitions name="CreditApprovalService"
targetNamespace="http://myservices.com"
  Xmlns="http://schemas.xmlsoap.org/wsdl/"
xmlns:tns=http://myservices.com"

Xmlns:plnk="http://schemas.xmlsoap.org/ws/2003/05/partne
r-link/"
  xmlns:soap=http://schemas.xmlsoap.org/ws/wsdl/soap/>
    <types>
        <schema attributeFormDefault="qualified"
elementFormDefault="qualified"
        targetNamespace="http://myservices.com"
        xmlns="http://www.w3.org/2001/XMLSchema">
        <element name="ssn" type="int" />
        <element name="approval" type="string" />
        <element name="error" type="string" />
        </schema>
    </types>
    <message
name="CreditApprovalServiceResponseMessage">
        <part name="approval" element="tns:approval" />
    </message>
    <message name="CreditApprovalServiceFaultMessage">
        <part name="approval" element="tns:error" />
    </message>
    <message
name="CreditApprovalServiceRequestMessage">
        <part name="approval" element="tns:ssn" />
    </message>
    <portType name="CreditApprovalService">
        <operation name="process">
          <input
message="tns:CreditApprovalServiceRequestMessage" />
            <output message=

    "tns:CreditApprovalServiceResponseMessage" />
            <fault name="InsufficientCredit"
message="tns:CreditApprovalServiceFaultMessage" />
        </operation>
    </portType>
    <binding name="CreditApprovalServiceBinding"
type="tns:CreditApprovalService">
    <service name="CreditApprovalService">
    <plnk:partnerLinkType name="CreditApprovalService">
        <plnk:role name="CreditApprovalServiceProvider">
            <plnk:portType
name="tns:CreditApprovalService" />
        </plnk:role>
    </plnk:partnerLinkType>
</definitions>
```

Listing 1. WSDL error type messages

```
<properties id="RatingService">
  <property name="wsdlLocation">
   http://localhost:8080/axis/services/RatingService1?wsdl
   http://localhost:8080/axis/services/RatingService2?wsdl
  </property>
</properties>
```

Listing 2. Providing alternate implementations for a web service

There are, however, other runtime faults that the above two mechanisms cannot handle, for example, if a remote service has upgraded and the interface has changed. This kind of fault is called ”binding fault” or “system fault” and the usual strategies adopted by

BPEL tools for dealing with a binding fault is either to delegate its handling to a human administrator via the built-in Task Manager service, or to place the exception in a dead letter queue via a JMS service [13]. Moreover, it is necessary for the BPELprocess designer to continuously maintain the BPEL scenarios, keeping the alternate service specifications up-to-date.whenever new such services are introduced or existing ones are withdrawn.

```
<properties id="RatingService">
  <property name="wsdlLocation">
http://localhost:8080/axis/servicesRatingService?wsdl</prope
rty>
  <property name="location">
http://localhost:2222/services/axis/RatingService
</property>
  <property name="retryCount">2</property>
  <property name="retryInterval">60</property>
</properties>
```

Listing 3. BPEL specification for automatic retry

Finally, as shown in Listing 3, execution faults not handled by BPEL with failover or retry, are not dynamically resolved. The Oracle BPEL Process Manager deals with them by sending a message to JMS Dead Letter queue.

4. SRRF Architecture

The overall architecture of the Service Relevance and Replacement Framework is illustrated in Listing 3. The client installation is complemented with an additional module, namely the SRRF preprocessor. The SRRF preprocessor accepts as input a BPEL scenario (typically created by an expert using a BPEL editor) describing a business process; The preprocessor produces as output an SRRF-aware BPEL scenario, which includes logic for detecting execution faults (e.g. server unavailabilities or network problems) and resolving them by locating and invoking service implementations with “same capabilities” as the ones that failed. The SRRF-aware BPEL scenario produced by the preprocessor can then be submitted to any WS-BPEL orchestrator (Oracle BPEL Process Manager, [7, 14], ActiveBPEL [5]).

During the execution of the SRRF-aware BPEL scenario, it is possible that the invocation of some web service fails, due to a system fault. In this case, the code inserted by the SRRF preprocessor will trap the fault and invoke the *Alternate WS Locator* module, which is an integral part of SRRF. The *Alternate WS Locator* module can be invoked as a regular web service accepting as input a specification of the web service that failed and possibly a *replacement policy* (explained later in the paper). The result of the *Alternate WS Locator* module is a list of web services that have the “same capabilities” as the service that failed. This result is returned to the SRRF-aware BPEL scenario, which arranges for invoking the alternate service implementations designated therein, until some of them succeeds or the list is exhausted; in the latter case, the BPEL scenario will fail since no further remedial actions can be taken.

Internally, the Alternate Service Locator Module comprises of four components, namely the *Request Interceptor*, the *Dynamic Discovery Module*, the *Task Relevance Module* and the *Task Priority Module*. Further information about SRRF can be found in [5] The Task Priority Module sorts the list according to the policy specification (e.g. *cheapest service first*). The result of this step is returned to the SRRF-aware BPEL scenario, which may then proceed to the invocation of the services designated in the reply in order to fully resolve the original execution exception.

The SRRF architecture has been formulated to ensure viability in its implementation, guarantee privacy in the communication of the orchestrator with the web service implementations and not raise any issues with security enforcement mechanisms. In particular:

- the Alternate WS Locator Module is a completely distinct module, which can either be installed and maintained by the organization running the BPEL orchestrator, or be offered by some third-party. Small enterprises are not expected to develop and maintain task ontologies since this is inherently a resource-consuming operation, and will probably thus resort to using publicly accessible alternate WS Locators, provided either for free (e.g. built and maintained with community contribution with each WS provider registering own services) or offered on a fee-basis in the form of a value-added service. This is analogous to the operation of telephone directories and yellow pages services.
- when the Alternate WS Locator needs to be invoked, only the name/WSDL location of the failed WS is sent to the locator service; thus, parameters passed to individual web services as well as results are never disclosed outside the organization running the orchestrator. Note that parameters and results may include authentication credentials, authorization information, or sensitive and confidential data, thus non-disclosure of these elements is of particular importance.
- finally, hot-swapping arrangements are included in the SRRF-aware BPEL scenario and performed by the BPEL orchestrator. The alternative of delegating hot-swapping to some entity outside the organization -besides disclosure of the

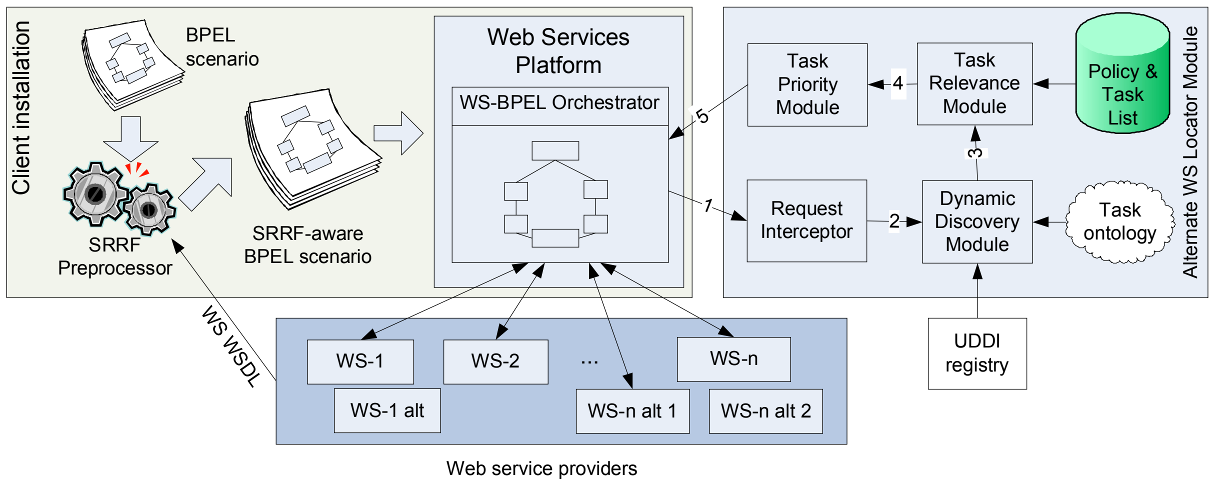

Figure 4. Architecture of the Service Relevance and Replacement Framework

parameters- might lead to failed invocations, due to security arrangements: for instance, a WS provider may employ IP-based authentication for WS invocations, thus requests not directly arriving from accredited partners may be rejected.

5. The SRRF modules

5.1 The SRRF preprocessor

The SRRF pre-processor analyzes its input BPEL scenario to identify invocations of web services and arranges for complementing each such invocation with code that (a) intercepts system faults (b) invokes as necessary the Alternate WS Locator to determine possible substitutes for the failed service and (c) invokes the substitute services until one of them succeeds.

The first action thus taken by the SRRF pre-processor is the syntactic analysis of the BPEL scenario to locate WS invocations. Error-handling activities provided by the scenario designer for the invocation are identified as well, since these typically include elaborate strategies for addressing runtime exceptions, proficiently crafted by the BPEL designer. Thus any exception resolution effort should first try the methods designated by the BPEL designer and then resort to any fallback strategies that will be supplemented by the SRRF.

For each such invocation, the invoked service's WSDL is retrieved and analyzed to locate *fault* declarations (c.f. section 3.1). The names of the faults declared therein correspond to *logical faults* and should thus be excluded from handling through alternate implementation invocation.

Having collected the necessary information, the SRRF pre-processor may proceed in the creation of the SRRF-aware BPEL scenario. First, the pre-processor adds the appropriate declaration of *partnerLinkType* in BPEL for the new Alternate WS Service Locator binding within *partnerLinks* construct (Listing 4). Thus the Alternate WS Service Locator is now known to the BPEL script and can be invoked by the fault handlers (discussed afterwards).

```
<plnk:partnerLinkType name="WSLocator">
  <role name="AlternateService" />
  <portType name="AltSRV" />
</plnk:partnerLinkType>
```

Listing 4. Declaring the Alternate WS locator service

Then, the preprocessor uses a *scope* construct to provide the appropriate fault handlers for each service invocation within the BPEL scenario. Scopes are employed to enable the definition of different fault handlers for different activities (or sets of activities gathered under a common structured activity such as <sequence> or <flow>). Additionally scopes may include local variable declarations, local correlation sets, compensation handlers, and event handlers, capabilities that are useful for the formulation of the pre-processor's output. For more information on scopes, the interested reader is referred to [15]. Fault handler definition for the inserted scopes must be performed in a way guaranteeing that:

- existing fault handlers provided by the scenario designer are preserved and take precedence over fallback operations.
- logical faults raised by the invoked web service are not retried but rather propagated to the invoking client.
- when system faults occur, the equivalent services are located and invoked until one of them succeeds

or the list is exhausted.

In order to achieve these goals, for each service invocation an arrangement with two nested scope constructs is formulated, as illustrated in listing 5. The inner scope contains the invocation of the web service together with the BPEL designer-provided fault handlers. The outer scope includes fault handlers generated by the preprocessor, which are created as follows:

1. for each logical fault name declared in the WSDL file, a separate *catch* construct is generated. The code in the catch construct invokes a fault callback in the invoking client (if one is provided) and rethrows the fault, to terminate the BPEL scenario.
2. following all logical fault-specific handlers, a *catchAll* handler is inserted, which is normally entered in the event of a system fault. The code in this handler invokes the alternate WS locator module to retrieve the web services which can be substituted for the failed one. Once the list is retrieved, the fault handler attempts to invoke the first alternate service specified therein. If the attempt is successful or a logical fault is raised, fault handling is assumed to have concluded; otherwise, the next alternate service in the list is tried.

Note that alternate service invocations within the *catchAll* handler are again protected using a nested *scope* construct with the necessary fault handlers (this is not shown in Listing 5 for brevity reasons). This is required since otherwise a fault (logical or system) occurring in the invocation of an alternate service would cause the BPEL scenario to fail.

5.2 The Alternate WS locator module

When a system fault occurs, the handlers generated by the SRRF preprocessor invoke the alternate WS locator module to retrieve a list of services which can be substituted for the failed one. For a service to be considered as possible substitute, it must be found to have the *same skills* as the failed one. The *same skills* relation between two services is determined by comparing task *attributes*, which define both *functional* and *qualitative* aspects of each service. For example, we consider a task that receives as input a date of birth, a nationality specification and a social security number, and produces a birth certificate as output data. Inputs and outputs are functional attributes of the task. Note here that descriptions of inputs and outputs go beyond the specifications employed in a WSDL specification, since the latter are machine-oriented types, whereas the former include higher-level semantics.

```
<scope name="OuterScope">

  <scope name="InnerScope">
    <!-- service invocation -->
    <invoke partnerLink="..." />
    <!-- fault handlers written by the BPEL designer -->
  </scope>

<faultHandlers>
  <!-- fault handler generated by the pre-processor>
  <!-- for each LogicalFaultName listed in the service WSDL --
>
  <catch faultName="LogicalFaultName">
    <!-- Notify the invoking client -->
    <assign>
      <copy>
        <from expression="string('LogicalFaultName)">
        <to variable="Fault" part="error">
      </copy>
    </assign>
    <invoke partnerLink="Client"
              portType="ClientCallbackPT"
              operation="ClientCallbackFault"
              inputVariable="Fault" />
    <!-- Rethrow the fault to terminate the scenario -->
    <throw faultName="LogicalFaultName"/>
  </catch>
  <!-- System faults enter the following handler -->
  <catchAll>
    <!-- Invoke alternate WS locator module to determine
alternate services that may be invoked -->
    <sequence>
      <assign>
        <copy>
          <from expression=
            "string('WSDLLocation of Service,
Other useful info - BPEL location...')" />
          <to variable="failedService" part="info" />
        </copy>
      </assign>
      <invoke partnerLink="WSLocator"
        portType="AltSRV"
        operation="AlternateService"
        inputVariable="failedService" />
    <!-- for each possible substitute identified, invoke the
service, and if the invocation is successful or a logical fault is
raised, the effort is concluded. If a system fault occurs, the
next possible substitute in the list is tried -->
    </sequence>
  </catchAll>
</faultHandlers>
</scope>
```

Listing 5. Using nested scope elements for exception handling

To formally model these semantics, domain ontologies or domain taxonomies can be employed; for example, in the e-government domain, the ontology presented in [16]. Adopting high-level semantics is indispensable, since if machine-oriented types are employed, comparison of functional attributes will be imprecise. For instance, a task modelling an application for a green card might accept as input an application date, a nationality specification and a social security number and produce a green card certificate as output. At machine-type level, the birth certificate and the green card task are indistinguishable since both the input types (date, string, number) and the output type (byte array) are identical; at a higher level of semantics though, it can be easily determined that the tasks are not functionally equivalent.

Task response time, availability, reliability, cost, encryption, reputation and authentication are the qualitative attributes of tasks, complementing the functional attributes. Domain ontologies-taxonomies (for high-level type semantics) along with the task attributes constitute the task ontology which is used in the process of selecting *same skilled* tasks.

Both functional and quantitative attributes of tasks are stored in the *task ontology*, populated and maintained by the organization offering the Alternate WS Locator service. An RDF schema for this ontology can be found in [4]. A necessary condition for two services to be considered as having same skills is that all their functional attributes must be identical. Quantitative attributes, on the other hand, need not be identical: for instance, a service having all attributes equal to those of the failed one but smaller cost can be definitely considered as a replacement candidate. In some cases, even services with higher cost could be considered (e.g. if the transaction should be completed anywise), or some tradeoffs between services could be allowed (e.g. a longer response time could be accepted if the cost were lowered). In overall, for the comparison of quantitative attributes the Alternate WS Locator service should be supplied with a *policy* specifying the rules that should apply to this process. The exact form and contents of such a policy designation is currently being elaborated on.

6. Conclusion and Future Work

In this paper we have presented an approach for resolving exceptions in BPEL scenarios, by locating and invoking web services having the *same skills* as the failed ones. The code for intercepting faults and invoking alternate web services is automatically generated and injected into the BPEL scenario by a preprocessor. Identification of *same skilled* web services is based on both functional and qualitative attributes, where functional attributes are required to be equivalent, while the comparison between quantitative attributes is policy-driven. The proposed approach exploits the exception handling mechanisms of BPEL and can thus be used with any available BPEL orchestrator.

Our future work includes the optimization of the algorithms used for determining "same skilled" services, as well as the optimization of the hot-swapping procedure, since the time needed to replace the failed service and to reconstruct the BPEL process is considerable. Further elaboration on the replacement policy specifications and semantics is also required, while and a thorough evaluation of the performance of the overall system will be also conducted.

7. References

[1] F. Leymann, D. Roller,M.-T. Schmidt, "Web services and business process management", *IBM Systems Journal*, Vol 41, 198 No2, 2002

[2] Eric Newcomer, Greg Lomow, "Understanding SOA with Web Services", Addison-Wesley, Copyright 2005 Pearson Education, Inc.

[3] Dellarocas, C. and M. Klein, "A knowledge-based approach for handling exceptions in business processes", *Information Technology and Management* 2000.

[4] Kareliotis Christos, Vassilakis Costas, Georgiadis Panayiotis, "Towards Dynamic, Relevance-Driven Exception Resolution in Composite Web Services", 4th International Workshop on SOA & Web Services Best Practices, Portland, Oregon, USA at OOPSLA, 2006

[5] http://www.active-endpoints.com/active-bpel-engine-overview.htm

[6] http://www.eclipse.org/bpel/

[7] http://www.oracle.com/technology/bpel/

[8] Kochut, K. J.: METEOR Model version 3. Athens, GA, Large Scale Distributed Information Systems Lab, Department of Computer Science, University of Georgia (1999)

[9] K. Verma, K. Sivashanmugam, A. Sheth, A. Patil, S. Oundhakar, and J. Miller, METEOR-S WSDI: A Scalable Infrastructure of Registries for Semantic Publication and Discovery of Web services. Journal of Information Technology and Management, Special Issue on Universal Global Integration, Vol. 6, No. 1 (2005) 17-39

[10] E. Cimpian, M. Moran, E. Oren , T. Vitvar, , M. Zaremba,: Overview and Scope of WSMX. Technical report, WSMX Working Draft, http://www.wsmo.org/TR/d13/d13.0/v0.2/

[11] C. Feier, D.Roman,, A. Polleres, J Domingue, M. Stollberg, D. Fensel: Towards Intelligent Web Services: Web Service Modeling Ontology, In Proc. of the International Conf on Intelligent Computing (2005)

[12] http://www.w3.org/Submission/wsdl11soap12/#fault-element

[13] Oracle BPEL Knowledge Base - Technical Note #007 Managing BPEL Run-time Exceptions, http://www.oracle.com/technology/products/ias/bpel/htdocs/dev_support.html#notes

[14]http://www.oracle.com/technology/products/ias/bpel/index.html

[15] M. Juric, Business Process Execution Language for Web Services BPEL and BPEL4WS (2nd Edition), Packt Publishing, 2006, ISBN-10: 1904811817

[16] DIP consortium: Data, Information and Process Integration with Semantic Web Services, WP 9: Case Study eGovernment, D9.3, e-Government ontology, with Annex1 eGovernment Ontology (OCML), Annex2 SeamlessUK taxonomy (XML)