Fault Tolerant Web Service Orchestration by Means of Diagnosis*

Liliana Ardissono, Roberto Furnari, Anna Goy, Giovanna Petrone, and Marino Segnan

Dipartimento di Informatica - Università di Torino
Corso Svizzera 185, 10149 Torino - Italy
{liliana,furnari,goy,giovanna,marino}@di.unito.it

Abstract. Web Service orchestration frameworks support a coarse-grained kind of exception handling because they cannot identify the *causes* of the occurring exceptions as precisely as needed to solve problems at their origin.

This paper presents a framework for Web Service orchestration which employs diagnostic services to support a fine grained identification of the causes of the exceptions and the consequent execution of effective exception handlers. Our framework is particularly suitable for intelligent exception handling in Enterprise Application Integration.

1 Introduction

The importance of Enterprise Application Integration is growing due to the emerging demand for short software development time, and to the fact that several services have to be developed by composing heterogeneous applications owned by different organizations. Workflow management systems have originally been developed to coordinate the execution of tasks within a single organization. Later on, Service Oriented Architectures [16], Web Service description languages (such as WSDL [19]) and Web Service composition languages (such as WS-BPEL [14]) have been introduced in order to abstract from location, communication protocol and deployment environment issues, therefore supporting the integration of distributed, heterogeneous software in open environments.

In current workflow management and Web Service orchestration environments, the development of fault tolerant distributed processes is based on the adoption of exception handling techniques which support a graceful termination, rather than the service continuation, as the recovery actions are associated to the observable effects of the occurred problems, rather than to their *causes*. Indeed, effective recovery strategies might be adopted to let the process progress towards the service completion, or to support the human user in performing ad hoc corrective actions, if the *causes* of the exceptions were recognized.

We thus propose to support *intelligent exception management* in Web Service orchestration environments by introducing:

* This work is supported by the EU (project WS-Diamond, grant IST-516933) and by MIUR (project QuaDRAnTIS).

V. Gruhn and F. Oquendo (Eds.): EWSA 2006, LNCS 4344, pp. 2–16, 2006.
© Springer-Verlag Berlin Heidelberg 2006

- *Diagnostic capabilities* supporting the identification of the causes of the exceptions occurring during the execution of a composite service. The analysis of the exceptions is carried out by *diagnostic Web Services* which explain the possibly incorrect execution of the orchestrated service providers by employing Model-Based Diagnosis techniques [1].
- *A novel methodology to apply exception handlers*, in order to make them sensitive to diagnostic information without modifying the standard exception management mechanisms offered by Web Service orchestration environments.

The rest of this paper is organized as follows: Section 2 outlines exception handling in workflow systems and provides some background on Model-Based diagnosis. Section 3 presents the architecture of our framework. Section 4 describes some related work and Section 5 concludes the paper.

2 Background

In Web Service orchestration, the development of fault tolerant distributed processes relies on the introduction of scopes, exception handlers and compensation handlers. When a scope fails, compensation handlers are executed to undo the completed activities. Moreover, exception handlers are performed to enable forward progress toward the termination of the process; e.g. see [14,11].

Several classifications of exceptions in categories have been made; e.g., see [9], [18] and [13]. These studies report a wide variety of events triggering the exceptions. For instance, [9] introduces different types of exceptions, among which the *expected* and the *unexpected* ones. *Expected exceptions* are associated to anomalies frequently occurring during the execution of activities, and should be managed by means of exception handlers. *Unexpected exceptions* derive from unexpected problems and typically have to be handled via human intervention, at the workflow instance level.

Various approaches have been proposed to handle the exceptions. For instance, [11] presents a technique based on *spheres of atomicity* to handle transactions. Moreover, [13] presents strategies enabling human users to participate in the recovery of a workflow instance during the service execution. Furthermore, [9] proposes a classification of activity types on the basis of factors such as the presence/absence of side-effects in compensation, and the development of a smart execution engine which may ignore the failure of *non vital* actions during the execution of a workflow.

Before concluding this section we would like to briefly introduce Model-Based Diagnosis (MBD), which we adopt in our framework to reason about exceptions. Model-Based Reasoning and, in particular, MBD, have been proposed and used within the Artificial Intelligence community for reasoning on possibly faulty physical and software systems; e.g., see [5,10]. In both cases, the system to be diagnosed is modeled in terms of components, and the goal of *diagnostic reasoning* is to determine a set of components whose incorrect behavior *explains* a given set of observations.

There are several formalizations of MBD that characterize the informal notion of explanation. In *consistency-based diagnosis* [17], which we utilize in our work, a *diagnosis* is an assignment of behavior modes to components that is consistent with observations. For static models this means that the hypotheses predict, for observable variables,

a set of possible values which includes the observed one. A *diagnostic engine* should, in general, explore the space of candidate diagnoses and perform discrimination among alternative candidates, possibly suggesting additional pieces of information to be acquired to this purpose.

3 Intelligent Exception Management Framework

We consider two types of failures: errors in data elaboration, and errors that either alter or block the execution flow. Our exception handling approach relies on:

- A *smart failure identification* aimed at identifying the cause of an observed exception, at the level of the composite service (global diagnosis). The goal is to identify the Web Service responsible for the occurred problem, the faulty activities and the other Web Services that may have been involved in the failure.
- *Diagnostic information aware exception handlers*, executed by the orchestrated Web Services. The global diagnosis is used to identify the recovery strategy to be applied by the service providers.

Consistently with Model-Based Diagnosis, the analysis of the exceptions is carried out in a component-based way[1] by introducing local diagnostic services that analyze the internal behavior of the orchestrated Web Services, and by employing a global diagnostic engine to combine the local diagnoses.

Each local diagnoser analyzes the behavior of the corresponding orchestrated Web Service WS_i by utilizing a *diagnostic model* M_i which describes the control and data flow of WS_i by specifying the possible correct and incorrect behavior. The *diagnostic model* of a Web Service is a declarative description of its business logic and it specifies, as precisely as possible, the activities carried out by the Web Service, the messages it exchanges, information about dependencies between parameters of the executed activities and the fault messages it may generate [1].

Web Services have rather diverse nature: some may execute articulated workflows; others may partially hide their internal business logic. Moreover, they may be implemented in different process languages, such as WS-BPEL [7], or the OPERA [11], XPDL [20] and BPML [3] workflow description languages. We introduced the diagnostic model as a separate representation of the business logic of a Web Service for two main purposes:

- The first one is to make local diagnosers independent of the implementation language adopted within the orchestrated Web Services.
- The second one concerns the possibility of extending the description of the business logic of a Web Service with information useful for diagnostic reasoning, without modifying the core of the Web Service; e.g., *alarm conditions* may be specified to enable the efficient isolation of the faulty activities during the Web Service execution.

[1] Notice that the orchestrated Web Services interact only by message passing. Thus, they do not share any data items which might influence each other's behavior as a side-effect.

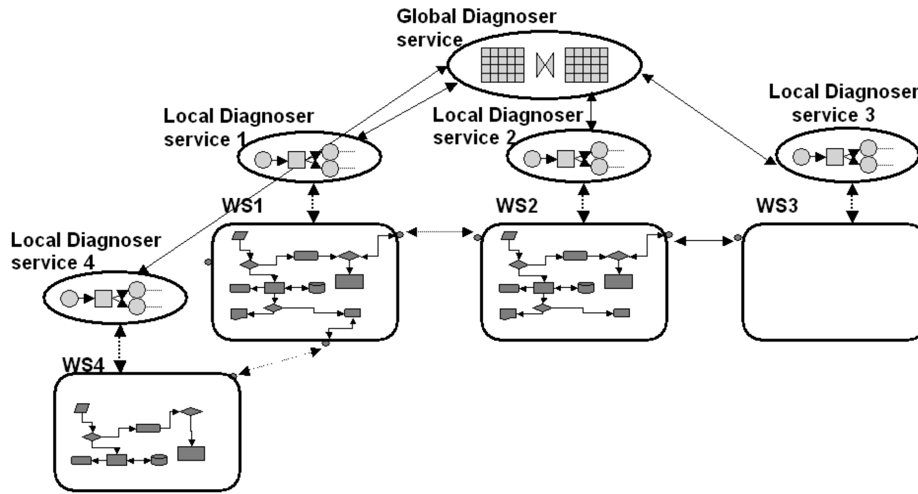

Fig. 1. Architecture of the Proposed Framework

3.1 Architecture

Figure 1 shows the proposed architecture in a composite service based on the orchestration of four Web Services. The orchestrated Web Services are depicted as rectangles with rounded corners, while the Diagnoser Web Services are represented as ovals. The dotted double arrows between Web Services and Local Diagnosers represent the asynchronous messages exchanged during both the identification of failures and the selection of the exception handlers to be executed.

- A Local Diagnoser service LD_i is associated to each orchestrated Web Service WS_i, in order to generate diagnostic hypotheses explaining the occurred exceptions from the local point of view.
  - To this purpose, LD_i utilizes the *diagnostic model* of WS_i. Moreover, LD_i interacts with WS_i by invoking a set of WSDL operations defined for diagnosis; Figure 2 shows the additional WSDL operations (described later on) that must be offered by an orchestrated Web Service in order to interact with its own Local Diagnoser.
  - The local hypotheses generated by LD_i specify various types of information, such as the correctness status of the input and output parameters of the operations performed by WS_i and the references to other orchestrated Web Services $\{WS_1, ..., WS_k\}$ which might be involved in the failure of the composite service. Errors may propagate from one Web Service to the other via message passing; thus, $\{WS_1, ..., WS_k\}$ is the set of orchestrated Web Services that have sent and/or received messages from WS_i.
- Global reasoning about the composite service is performed by a Global Diagnoser service which interacts with the Local Diagnosers of the orchestrated Web

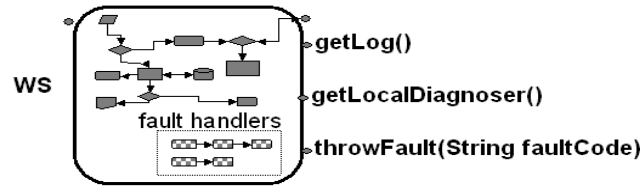

Fig. 2. Web Service Extended for Advanced Exception Management

Services. The Global Diagnoser combines the local hypotheses generated by Local Diagnosers into one or more global diagnostic hypotheses about the causes of the occurred exceptions.

Local and Global Diagnosers are themselves Web Services communicating via WSDL messages. This component-based approach has various advantages. For instance:

- Local Diagnosers may be associated to Web Services in a stable way; therefore, the diagnostic model of a Web Service WS_i has to be defined only once, at Local Diagnoser LD_i configuration time (although, as described later on, the management of diagnosis aware exception handlers requires the introduction of supplementary information, which is specific for the composite service invoking WS_i).
- Privacy preferences possibly imposed by the organizations owning the orchestrated Web Services may be satisfied. In fact, the diagnostic model of a Web Service WS_i can only be inspected by its Local Diagnoser LD_i, which does not directly interact with the Local Diagnosers of the other orchestrated Web Services. Moreover, the information provided by LD_i during the interaction with the Global Diagnoser concerns the presence of errors in the data received from, or propagated to other Web Services, omitting internal details of the Web Service implementation.
- Due to the clear separation between Local and Global Diagnosers, the latter may be developed as general services supporting diagnosis in different composite services; in fact, a Global Diagnoser only makes assumptions on the communication protocol to be adopted in the interaction with the Local Diagnosers.

3.2 Smart Failure Identification

Diagnosis is needed when exceptions occur, but can be avoided when the composite service progresses smoothly. Thus, we propose to trigger diagnosis immediately after a problem is detected.

Specifically, when an exception occurs in a Web Service WS_i, its Local Diagnoser LD_i is invoked to determine the causes. To this purpose, LD_i exploits the diagnostic model M_i and may need to analyze the messages exchanged by WS_i with the other peers (e.g., to check the parameters of the operations on which WS_i was invoked). LD_i retrieves this information by requesting from WS_i the log file of its activities, i.e., by invoking the `LogFile getLog()` WSDL operation; see Figure 2. After the generation of the local hypotheses, LD_i invokes the Global Diagnoser (`activate(Collection`

`hypotheses)` WSDL operation) to inform it about the local hypotheses it made on the causes of the exception. The Global Diagnoser, triggered by the incoming message, starts an interaction with other Local Diagnosers to identify the problem. The double arrows between Diagnosers in Figure 1 represent the synchronous messages exchanged in this phase.

The interaction between Global Diagnoser and Local Diagnosers is managed as a loop where the Global Diagnoser invokes different Local Diagnosers to make them generate their own local hypotheses, which are incrementally combined into a set of global diagnostic hypotheses about the occurred problem. Below, we summarize the actions performed in the loop.

- By analyzing the local hypotheses it has received, the Global Diagnoser identifies a list of Local Diagnosers to be invoked in order to make them generate their local hypotheses.[2] It then invokes the `Collection extend(Collection hypotheses)` WSDL operation on each Local Diagnoser in the list. This operation, given a set of hypotheses, returns a revised set of hypotheses to the caller.
- The Global Diagnoser combines the local hypotheses and generates a set $\mathcal{H}$ of global hypotheses about the causes of the occurred exceptions, consistent with the local hypotheses; in Figure 1 the local hypotheses have been depicted as tables.

The Global Diagnoser exits the loop when it cannot invoke any further Local Diagnosers. This happens either because all of them have contributed to the diagnosis, or because the remaining ones cannot provide any discriminating information, nor can they broaden the search for the causes of the exception. This means that, although several messages might be exchanged by the Local and Global Diagnosers, the diagnostic process always terminates.

The set $\mathcal{H}$ obtained by the Global Diagnoser at the end of the loop represents the global result of diagnosis about the exception occurred in the composite service. $\mathcal{H}$ can be seen as the solution of a Constraint Satisfaction Problem [8] where constraints express the relation between failures and service behavior. See [1] for details about the adopted diagnostic algorithm.

3.3 Diagnostic Information Aware Exception Handling

Local and Global Exceptions. The causes of exceptions reported in the result $\mathcal{H}$ of the global diagnosis can be employed at the level of the composite service for the selection of specific exception and compensation handlers, which might substantially differ from those that would be adopted by the orchestrated Web Services on the sole basis of the locally raised exceptions. The cardinality of $\mathcal{H}$ has to be considered:

[2] The `activate(Collection hypotheses)` message specifies the references of the orchestrated Web Services possibly involved in the failure. By invoking such Web Services, the Global Diagnoser retrieves the references of the Local Diagnosers to be contacted in order to acquire further information. The reference to the Local Diagnoser of a Web Service is obtained by invoking the `LocalDiagnoser getLocalDiagnoser()` WSDL operation; see Figure 2.

(a) If $\mathcal{H}$ is a singleton, the Global Diagnoser could find a single cause (h) explaining the occurred exceptions. The Local Diagnosers might need to inhibit the default exception handling to take h into account; in order to make the Web Services execute the appropriate handlers, Local Diagnosers should make Web Services throw *global exceptions* activating the handlers needed to repair h, instead of continuing the execution of the handlers associated to their local exceptions. Of course, if the default exception handlers are suitable to repair h, they should continue their regular execution; this corresponds to the case where the global exception coincides with the local one that was raised in the Web Service.

(b) If $\mathcal{H}$ includes more than one global diagnostic hypothesis, it should be reduced to a singleton (e.g., via human intervention). If this is not possible, default fault management behavior should be adopted. We leave this aspect apart as it concerns the diagnostic algorithm, which is out of the scope of this paper.

The execution of exception handlers depending on diagnostic information requires that, when the composite service is set up, the possible diagnostic hypotheses are mapped to global exceptions to be handled in the overall service. Specifically, for a complex service CS, the Local Diagnoser LD_i of a Web Service WS_i must store a set of mappings $map(h_x, e_{ix}, CS)$ between each possible global diagnostic hypothesis h_x and the corresponding exception e_{ix} to be raised in WS_i. These mappings complement WS_i's diagnostic model, as far as service CS is concerned.

Defining Diagnostic Information Aware Exception Handlers. We now describe how default exception handling can be overridden in order to take diagnostic information into account. If the composite service is designed by specifying the orchestration of service providers in a process language such as WS-BPEL, a different fault handler may be associated to each type of exception (*WS-BPEL fault*). Thus, we propose to modify each orchestrated Web Service WS_i as follows; see Figure 2 as a reference:

1. *Make the fault handlers of the Web Service aware of the diagnostic information.* To this purpose, each fault handler f has to be modified so that it invokes the Local Diagnoser LD_i by means of a synchronous `String getHypothesis(String localFaultCode)` message and waits for the result before executing any further actions.[3] The result is generated after the interaction between Local and Global Diagnosers and it is a String value corresponding to the global exception to be raised, depending on $\mathcal{H}$. The result received by the local fault handler f after having invoked the Local Diagnoser is characterized as follows:
   a) If f is the appropriate handler for the current case (i.e., the global exception coincides with the local one), the result is the value held by *localFaultCode*, which means that f can continue its own regular execution.[4]

[3] There is a chance that the Local Diagnoser fails itself and does not send the response message. In order to handle this case, the fault handlers might be extended by introducing a time out mechanism which enables them to resume execution after a certain amount of time. We leave this aspect to our future work.

[4] If $\mathcal{H}$ includes more than one global diagnostic hypotheses, the default handling behavior can be performed by making Local Diagnosers return the *localFaultCode* values to the fault handlers.

b) If the case has to be treated by means of another fault handler of the same Web Service, the result returned by the Local Diagnoser is set to the code of a different fault event. In that case, f has to throw the new fault and terminate the execution of the current scope. The occurrence of the new fault event in the same Web Service automatically triggers the appropriate fault handler.
c) If the Web Service is not requested to perform any fault handling procedure (i.e., another Web Service has to trigger a handler of its own), the result returned by the Local Diagnoser is a null String, and the active fault handler f has to terminate the execution.

2. *Possibly add new fault handlers.* Most original fault handlers can be employed to manage both local and global exceptions, after having been revised as specified in item 1. However, additional fault handlers might be required to handle new types of exceptions, derived from the global diagnosis.
3. *Offer a WSDL operation, `throwFault(String faultCode)`,* which a Local Diagnoser LD_i may invoke on Web Service WS_i when the execution of the composite service has failed, but no exceptions were raised in WS_i, nonetheless some recovery action implied by the global diagnosis has to be performed. When the Web Service performs the `throwFault(String faultCode)` operation, it throws the *faultCode* fault, which activates the corresponding fault handler.

All the items above, except for the last one, involve local changes to the code of the exception handlers, which may then be performed by any standard workflow or orchestration engine. In contrast, item 3 has to be implemented in different ways, depending on the characteristics of the engine. The problem is that the `throwFault(String faultCode)` might be invoked at any instant of execution of a Web Service, regardless of its business logic. It is therefore necessary that the Web Service catches the invocation of the operation as a high priority event, to be handled by possibly interrupting the regular execution flow.

For experimental purposes, we have developed our methodology for diagnostic information aware exception handling in the JBPM workflow management environment [12] and the following subsection provides some details about the implementation of the `throwFault(String faultCode)` operation. In other environments, e.g., in a BPEL one, the same methodology can be applied by exploiting the features provided by the execution engine (e.g., defining an eventHandler for an `onMessage` event whose activity will throw the related $faultCode$ fault).

Flexible Exception Management in a Workflow Management Environment. An initial prototype of the framework we propose in this paper has been developed on top of jBPM [12], a business process management system based on Petri Net model implemented in Java. jBPM is based on Graph Oriented Programming model, which complements plain imperative programming with a runtime model for executing graphs (workflows). `Actions` are pieces of Java code that implement the business logic and are executed upon events in the process. In our framework the Web Service is composed of a workflow and a client (Workflow Controller) in charge of monitoring the workflow execution. When a fault occurs, the parent hierarchy of the graph node is searched for an appropriate exception handler. When it is found, the actions of the handler are executed.

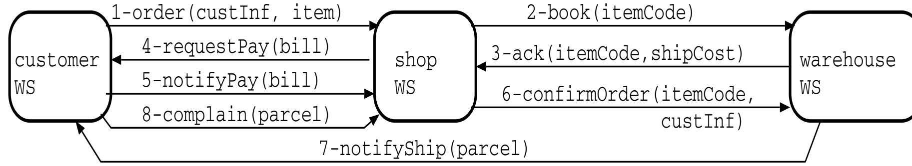

Fig. 3. Portion of a Sample Sales Scenario

Using jBPM makes the implementation of our framework straightforward, in particular referring to item 3 in Section 3.3. When the Web Service receives the `throwFault(String faultCode)` message, the Workflow Controller transfers control to the workflow by generating a special `ExternalFaultEvent`. This event causes the execution of an `Action` that can perform a repair or throw another exception, which activates the corresponding exception handler. In the jBPM case the `Action` can access the current state of the workflow execution by accessing the tree of the active tokens and the associated current graph nodes.

3.4 Example

We sketch the interaction between Diagnoser services and orchestrated Web Services in a sample scenario. Figure 3 shows a portion of the interaction diagram of an e-commerce service based on the orchestration of three Web Services: a *customer WS* enables the user to browse a product catalog and purchase goods. A *shop WS* manages orders, bills and invoicing. A *warehouse WS* manages the stock and delivers the goods to the customer.

In this scenario, the customer places an order for a product by specifying the name of the good (msg 1); the shop reserves the requested item (msg 2) from the warehouse and receives an acknowledgement message (msg 3) where the shipping cost of the parcel is specified; we assume that the shipping cost may vary depending on the address of the customer (included in the *custInf* parameter) and the size of the good that has been purchased.

Given the shipping cost, the shop sends the bill to the *customer WS* (msg 4). The customer pays the bill and notifies the shop (msg 5). At that point, the shop confirms the order (msg 6) and the warehouse sends the package to the customer and notifies the *customer WS* accordingly (msg 7).

If the customer receives a parcel including a good different from the ordered one, she can complain about the delivery problem via the *customer WS* user interface and her complaint makes the Web Service send a `complain(parcel)` message to the *shop WS* (msg 8). The occurrence of this message denotes that there was a delivery problem; therefore, we assume that it is interpreted as an exceptional case by the *shop WS*, i.e., a case in which the execution of the corresponding WSDL operation throws an exception within the Web Service execution instance.

Suppose that that the delivery of a wrong parcel may be due to the following alternative causes:

- h_1: the *shop WS* provided a wrong item code for the requested product.
- h_2: the wrong parcel is picked within the warehouse.

When an exception occurs, the repair strategy to be adopted within the overall service may differ depending on which is the faulty Web Service, i.e., on the global diagnostic hypothesis ($\mathcal{H} = \{h_1\}$ or $\mathcal{H} = \{h_2\}$) and on the corresponding global exceptions to be raised in the Web Services by their own Local Diagnosers, according to the service specific mappings held by the Local Diagnosers. In our example, we assume the following mappings:

- The Local Diagnoser of the *shop WS* maps h_1 to the $wrongItemFault$ exception and h_2 to a null exception: *map(*h_1*,* $wrongItemFault$*)*, *map(*h_2*,* $null$*)*.
- The Local Diagnoser of the *warehouse WS* maps h_1 to a null exception and h_2 to the $wrongParcelFault$ exception: *map(*h_1*,* $null$*)*, *map(*h_2*,* $wrongParcelFault$*)*.

Specifically we assume that the possible failures are handled as follows:

(a) If the *shop WS* provided a wrong item code for the requested product ($\mathcal{H} = \{h_1\}$, $wrongItemFault$ exception), the order to the warehouse and possibly the shipping cost are incorrect. Therefore, the *shop WS* should perform recovery actions that include internal activities and invocations of the other Web Services.

    In detail, the *shop WS* should execute an exception handler which prescribes to: correct the item code, reserve the correct item from the warehouse and recompute the bill; if the new bill differs from the previous one, send it to the customer asking for the difference (or refund her for the extra money). The last steps of the handler include requesting the warehouse to take the wrong parcel from the customer, deliver the new parcel and refund the warehouse for the extra delivery costs.

(b) If the *shop WS* reserved the correct item, but the wrong parcel was picked for the delivery within the warehouse ($\mathcal{H} = \{h_2\}$, $wrongParcelFault$ exception), the first part of the composite service was correctly executed and the bill was correct as well. The problem should be repaired by the *warehouse WS*, which should perform an exception handler prescribing to take the wrong parcel from the customer, deliver the new one, and cover the extra delivery costs.

In our framework, the occurrence of a `complain(parcel)` message is handled as follows (see Figures 4 and 5 for a description of the diagnosis aware fault handlers):[5]

1. Upon receiving the `complain(parcel)` message, the *shop WS* throws an internal *wrongItemFault* event to be caught by the *wrongItemFaultHandler* exception handler.

   The *wrongItemFaultHandler* invokes the Local Diagnoser of the *shop WS* (message `getHypothesis(wrongItemFault)`) to receive the global exception.
2. The Local Diagnoser of the *shop WS* retrieves the global diagnostic hypothesis $\mathcal{H}$ by interacting with the Global Diagnoser.

[5] The fault handlers are described in a java-like syntax, as the description in the process language would be too complex.

```
wrongItemFaultHanldler {

// get exception from local diagnoser
  String globException = sendReceive(shopWS, locDiagnoser,
                           getHypothesis("wrongItemFault"));
  if (globException == null)
    terminate;            // no recovery action needed

  else if (globException != "wrongItemFault")
    throw globException;  // recovery performed by other handler

  else {              // execute original exception handler code
           // order the correct item
    String newItemCode = correctCode(item);
    send(shopWS, warehouseWS, book(newItemCode));
    double shipCost = receive(shopWS,warehouseWS,ack(shipCost));
    double newBill = computeBill(newItemCode);
           // refund or additional payment
    if (newBill>bill) {
      double extraMoney = newBill - bill;
      send(shopWS, customerWS, requestPay(extraMoney));
      receive(shopWS, customerWS, notifyPay(extraMoney));
      }
    else
      send(shopWS, customerWS, refund(bill-newBill));
    send(shopWS, warehouseWS,confirmOrder(newItemCode,custInf));
    send(shopWS, warehouseWS, takeBack(itemCode, custInf));
    refund(warehouseWS, shipCost);
    }
} // end wrongItemFaultHandler
```

Fig. 4. Pseudocode of *wrongItemFaultHandler* (*shop WS*), Modified to be Sensitive to Diagnosis Information

- If $\mathcal{H} = \{h_1\}$, the Local Diagnoser of the *shop WS* returns the *wrongItem-Fault* value as the result of the `getHypothesis` message. As *wrongItem-FaultHandler* is the appropriate fault handler to be executed, the Web Service continues its execution, which involves sending another order to the *warehouse WS* and covering the extra costs. No fault events are handled in the *warehouse WS*.
- Otherwise, if $\mathcal{H} = \{h_2\}$, the Local Diagnoser of the *shop WS* returns a null value and the *wrongItemFaultHandler* terminates the execution. Moreover, the Local Diagnoser of the *warehouse WS* invokes the `throwFault (wrongParcelFault)` message on the *warehouse WS*, which throws the *wrongParcelFault*. The occurrence of such fault event starts the appropriate fault handler (take wrong parcel from customer, deliver the correct one, notify the *shop WS* about the shipping and cover extra delivery costs).

```
wrongParcelFaultHanldler {

// get exception from local diagnoser
  String globException = sendReceive(warehouseWS, locDiagnoser,
                              getHypothesis("wrongParcelFault"));
  if (globException == null)
    terminate;          // no recovery action needed



  else if (globException != "wrongParcelFault")
    throw globException;// recovery performed by other handler


  else {             // execute original exception handler code
   // re-execute the operations for the correct parcel
    String newParcel = correctParcel(itemCode);
    double shipCost = computeShipCost(item, custInf);
    takeBack(wrongParcel, custInf);
    deliver(parcel, custInf);
    send(warehouseWS, shopWS, notifyShip(newParcel));
    coverShipCost(shipCost);
    }
} // end wrongParcelFaultHandler
```

Fig. 5. Pseudocode of *wrongParcelFaultHandler* (*warehouse WS*), Modified to be Sensitive to Diagnosis Information

4 Related Work

Various proposals for the management of transactional behavior in centralized Web Service orchestration are being proposed in order to support the development of reliable composite Web Services; e.g., see WS-Transaction [6] and OASIS BTP [15]. In decentralized orchestration there is the additional problem that the entire state of the original composite Web Service is distributed across different nodes. To address this issue, Chafle and colleagues propose a framework where Local Monitoring Agents check the state of the orchestrated Web Services and interact with a Status Monitor that maintains a view on the progress of the composite service [4]. Although there is a direct correspondence with our Local and Global Diagnoser services, the authors apply traditional error detection and are therefore subject to the limitations we discussed. Moreover, the Status Monitor holds complete information about fault and compensation handlers of the orchestrated services; thus, it is only suitable for closed environments where the orchestrated services are allowed to expose complete information about themselves.

Biswas and Vidyasankar propose a finer-grained approach to fault management in [2], where *spheres of visibility, control and compensation* are introduced to establish different levels of visibility among nested Web Services and composite ones in hierarchical

Web Service composition. Although the described approach sheds light in how the most convenient recovery strategies can be selected, the paper does not clarify how the exception handlers to be executed are coordinated at the various levels of the hierarchy.

5 Discussion

We have described a framework enhancing the failure management capabilities in Web Service orchestration by employing diagnostic reasoning techniques and diagnosis aware exception handlers. Our framework is based on the introduction of a set of Local Diagnosers aimed at explaining incorrect behavior from the local viewpoint of the orchestrated Web Services and on the presence of a Global Diagnoser which, given the local diagnostic hypotheses, generates one or more global hypotheses explaining the occurred exceptions from the global point of view.

The introduction of Local and Global Diagnosers enhances the exception management capabilities of the composite service by steering the execution of exception handlers within the orchestrated Web Services on the basis of a global perspective on the occurred problem. However, it introduces at least two main kinds of overhead, concerning the composite service set up and execution time, respectively:

- At set up time, the administrator of the composite service and those of the orchestrated Web Services have to do some configuration work in order to define service specific settings and possibly to define service specific exception handlers. This aspect obviously restricts the applicability of our framework to Enterprise Application Integration, where explicit agreements between the administrator of the composite service and those of the orchestrated service providers may be defined. It should be however noticed that EAI currently represents the most important application for Web Services, if compared with Web Service invocation in open environments.
- At run time, the interaction between diagnosers, and the time needed to find a global diagnostic hypothesis, may delay the execution of the composite service. However, the diagnostic process is only activated when one or more exceptions occur (thus, in situations where the service execution is already challenged). Moreover, the interaction between the diagnosers and the orchestrated Web Services is limited to the retrieval of the log files and the final notification of the global exception to be handled; therefore, the individual Web Services are not affected by the possibly complex interaction concerning diagnosis.

Our future work concerns two main aspects: first of all, as our approach currently supports the recovery from occurred exceptions, we want to extend it in order to support the early detection of failures. To this purpose, we are analyzing the possibility of monitoring the message flow of a composite service (i.e., the progress in the execution of a conversation graph describing the message exchanges which may occur among the orchestrated Web Services) and to trigger diagnostic reasoning as soon as a deviation from the expected behavior is detected.

Second, our current contribution is related to a rather specific aspect in the general topic of fault tolerant computing. However, our work is part of the WS-DIAMOND European Project which also deals with redundancy and Web Service replacement aspects, to be taken into account in order to recover from faults in complex systems. For more information about such project, see [21].

References

1. L. Ardissono, L. Console, A. Goy, G. Petrone, C. Picardi, M. Segnan, and D. Theseider Dupré. Enhancing Web Services with diagnostic capabilities. In *Proc. of European Conference on Web Services (ECOWS-05)*, pages 182–191, Växjö, Sweden, 2005.
2. D. Biswas and K. Vidyasankar. Spheres of visibility. In *Proc. of European Conference on Web Services (ECOWS-05)*, pages 2–13, Växjö, Sweden, 2005.
3. BPMI Business Process Management Initiative. Business Process Management Language. http://www.bpmi.org, 2005.
4. G. Chafle, S. Chandra, V. Mann, and M.G. Nanda. Decentralized orchestration of composite Web Services. In *Proc. of 13th Int. World Wide Web Conference (WWW'2004)*, pages 134–143, New York, 2004.
5. L. Console and O. Dressler. Model-based diagnosis in the real world: lessons learned and challenges remaining. In *Proc. 16th IJCAI*, pages 1393–1400, 1999.
6. W. Cox, F. Cabrera, G. Copeland, T. Freund, J. Klein, T. Storey, and S. Thatte. Web Services Transaction (WS-Transaction). http://dev2dev.bea.com/pub/a/2004/01/ws-transaction.html, 2005.
7. F. Curbera, Y. Goland, J. Klein, F. Leymann, D. Roller, S. Thatte, and S. Weerawarana. Business Process Execution Language for Web Services, version 1.0. http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/, 2002.
8. R. Dechter. *Constraint Processing*. Elsevier, 2003.
9. J. Eder and W. Liebhart. The workflow activity model WAMO. In *Proc. 3rd Int. Conf. on Cooperative Information Systems*, Vienna, 1995.
10. G. Friedrich, M. Stumptner, and F. Wotawa. Model-based diagnosis of hardware designs. *Artificial Intelligence*, 111(1-2):3–39, 1999.
11. C. Hagen and G. Alonso. Exception handling in workflow management systems. *IEEE Transactions on Software Engineering*, 26(10):943–958, 2000.
12. J. Koenig. JBoss jBPM white paper. http://www.jboss.com/pdf/jbpm_whitepaper.pdf, 2004.
13. H. Mourao and P. Antunes. Exception handling through a workflow. In R. Robert Meersman and Z. Tari, editors, *On the move to meaningful internet systems 2004*, pages 37–54. Springer Verlag, Heidelberg, 2004.
14. OASIS. OASIS Web Services Business Process Execution Language. http://www.oasis-open.org/committees/documents.php?wg_abbrev=wsbpel, 2005.
15. OASIS TC. OASIS Business Transaction Protocol. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=business-transaction, 2005.
16. M.P. Papazoglou and D. Georgakopoulos, editors. *Service-Oriented Computing*, volume 46. Communications of the ACM, 2003.
17. R. Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57–96, 1987.
18. S.W. Sadiq. On capturing exceptions in workflow process models. In *Int. Conf. on Business Information Systems*, Poznam, Poland, 2000.

19. W3C. Web Services Definition Language. http://www.w3.org/TR/wsdl, 2002.
20. Workflow Management Coalition. XML process definition language (XPDL). http://www.wfmc.org/standards/XPDL.htm, 2005.
21. WS-DIAMOND. WS-DIAMOND Web Services DIagnosability MOnitoring and Diagnosis. http://wsdiamond.di.unito.it/, 2005.