Enhancing Web Service Composition by Means of Diagnosis *

Liliana Ardissono[1], Stefano Bocconi[2], Luca Console[1], Roberto Furnari[1], Anna Goy[1], Giovanna Petrone[1], Claudia Picardi[1], Marino Segnan[1], and Daniele Theseider Dupré[3]

[1] Dipartimento di Informatica, Università di Torino, Italy
[2] Department of Artificial Intelligence, Vrije Universiteit Amsterdam, The Netherlands
[3] Dipartimento di Informatica, Università del Piemonte Orientale, Italy

Abstract. This paper proposes a framework based on Service Oriented Architecture which integrates diagnostic services in the architecture of a composite service in order to improve the recovery from the occurring exceptions. Our framework supports a fine-grained selection of exception handlers suitable to repair the problem without modifying the basic mechanism offered by standard orchestration engines such as the WS-BPEL compliant ones.

1 Introduction

Service Oriented Architecture (SOA) [1] is a logical way of designing a software system to provide services to either end-user applications or to other services distributed in a network, via published and discoverable interfaces. Standard languages for the publication and invocation of Web Services, such as WSDL [2], have been developed. On top of the basic communication languages, many Web Service composition languages, such as WS-BPEL [3], are being defined to support the development of complex applications based on the orchestration of simpler ones; see [4] for a survey.

The design of a service based on the invocation of a set of Web Services challenges the exception management. In fact, in a composite service, the occurrence of exceptions within an invoked Web Service may involve other participants, which are indirectly affected by the execution problems.

In order to enable a composite service to recover from the occurrence of an exception, the source of the problem has to be identified from the global point of view. Only in this way the most suitable recovery actions can be applied within all the involved Web Services. We can say that the recovery actions have to be performed within the *context* of the complex service, in contrast to a completely autonomous exception handling behavior adopted by individual Web Services.

In current Web Service-based SOA middleware, the development of fault tolerant distributed processes is based on the application of repair techniques on the basis of the

* This work was funded by projects WS-Diamond (IST-516933) and QuaDRAnTIS (MiUR).

observed exceptions. Although that approach can be utilized to gracefully terminate the execution, it does not support an effective recovery and continuation of the service.

In order to address such issues, we propose a framework for the management of composite services which exploits diagnostic techniques to reason about the causes of the exceptions occurring during the service execution; the diagnostic information is utilized in our framework to select the recovery behavior to be applied in the context of the complex service.[4]

- The analysis of the exceptions concerns the execution of the composite service and is carried out by a set of embedded diagnostic services. As the individual Web Services synchronize and cooperate by relying only on message passing, this global analysis is achieved by combining the results of separate analyses of their internal behavior.
- The recovery is based on the selection of exception handlers that provide the most appropriate recovery with respect to the cause of the problem. Our framework supports the activation of the suitably chosen exception handlers (complementing the default exception handlers) without modifying the standard exception management mechanisms offered by EAI and Web Service composition.

With respect to the work described in [5], the present paper improves the diagnosis techniques utilized to steer the selection of the exception handlers to be applied at recovery time. The rest of this paper is organized as follows: Section 2 describes a simple example, aimed at providing a scenario in which our framework could be applied. Section 3 describes the diagnostic approach adopted in our framework and the framework architecture. Section 4 describes the diagnostic model describing the Web Services behavior, and how it can be built. Section 5 describes the adopted exception handling technique. Sections 6 and 7 present the related work and conclude the paper, respectively.

2 Motivating example

Let's imagine an on-line foodshop, involving three Web Services: a *Shop WS* that receives orders from a customer (through Web-based product catalog); a *Warehouse WS* that stocks and provides imperishable goods, prepares the parcel and delivers it to the customer; a *Supplier WS* that provides perishable goods (fresh food). Figure 1 shows a subset of the messages exchanged by the WSs participating to the composite service.

- *Shop WS.* The Shop WS receives from a customer an order (*o-id*) consisting of a list of items, and splits the list into a set of perishable items and a set of imperishable ones. Then the Shop WS asks the Supplier WS to check the availability of perishable items (*p-items*) and to reserve them (msg 1). The same is done (*i-items*) with the Warehouse WS for imperishable goods (msg 2). The Shop WS gets an answer from both the Warehouse WS (msg 4) and the Supplier WS (msg 3) about

[4] In this paper, we use the term *exception* to denote the manifestation of a problem and the term *failure* to denote the cause of the problem. Diagnosis is thus the process that allows identifying failures from the exceptions they have raised.

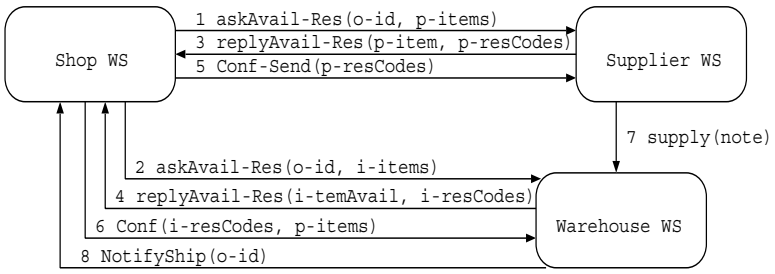

Fig. 1. Portion of a sample sales scenario.

items availability[5] (*i-itemAvail* and *p-itemAvail*) and reservation (*i-resCodes* and *p-resCodes*). If the items are available, the Shop WS asks the customer to confirm or cancel the order. If the customer confirms the order (the other case is not shown in Figure 1), the Shop WS asks the Supplier WS (msg 5) to provide the Warehouse WS with the goods (*p-resCodes*), and it confirms the order (*i-resCodes*) to the Warehouse WS (msg 6). The Shop WS also forwards the list of perishable items (*p-items*) to the Warehouse WS.

– *Supplier WS*. The Supplier WS receives, from the Shop WS, a request to check for the availability of a list of products and to reserve them (msg 1). It answers by providing information about availability and a notification of the reservation (msg 3). In the meantime it waits for a message from the Shop WS about order confirmation (msg 5) or cancelation. In the former case, it supplies the goods to the Warehouse WS, together with a shipping note (*note*, msg 7); in the latter case, it cancels the reservation.
– *Warehouse WS*. The Warehouse WS receives, from the Shop WS, a request to check for the availability of a list of products and to reserve them (msg 2). It provides the requested information and a notification of the reservation (msg 4). From the Shop WS it also receives the data about the part of the order concerning the perishable goods (msg 6). Then it receives the products from the Supplier WS, together with their description in the shipping note (msg 7). Finally, the Warehouse WS assembles the goods, ships the parcel to the customer and notifies the Shop WS that the order (*o-id*) has been processed (msg 8).

The warehouse activity of assembling the parcel is an important check point for error detection. For instance, a mismatch between one of the goods received from the supplier and its description provided by the shop can be observed. This mismatch can have three different possible causes: (a) when the Shop WS reserved the goods from the supplier, it sent the wrong product code; (b) when the Supplier WS received the reservation request, it wrote down the wrong code (e.g., in its internal order database); (c) when the Supplier WS sent the goods to the warehouse, it put the wrong item in the parcel. Thanks to some additional tests, the diagnostic reasoning mechanism can distinguish these three situations and, on the basis of the identified cause, a suitable recovery strategy can be applied.

[5] For simplicity, we suppose that the items are always available.

3 Diagnostic Approach and Architecture

For solving the problem of diagnosing failures of composite Web Services we adopt the approach described in [6]. We refer the reader to that paper for the details on the algorithms and their properties; here we will briefly sketch the main features and explain how this diagnostic framework can be applied to Web Services.

The context of reference is that of Model-Based Diagnosis [7], where the diagnostic process is based on a model of the system to diagnose, that provides a description of the expected behavior of the system itself. When some behavior is observed that contradicts the model, then a computation is started to find its causes. Different formalizations of the notions of explanation, cause and even diagnosis exist, depending mainly on the nature of the model, the information it contains, and the language in which it is provided. For the case of Web Services, we found convenient the use of *constraint-based models*, where the behavior of the system (in this case a Web Service) is described as a set of constraints (or, in other words, relations) among finite-domain variables, some of which represent the presence or absence of failures (called *mode* variables). Given some observations, that provide the actual values for some variables, the constraint system can be solved by finding tuples of variable values that meet all constraints. The values of mode variables in the solutions correspond to the possible diagnoses.

The algorithm in [6] provides a *decentralized* solution to the problem of diagnosis, in order to tackle the case where there is no single, monolithic model of the system to diagnose, but the system is rather described by a set of distributed models together with the relations that link them. It assumes that each component of the complex system has a Local Diagnoser, that owns the model of the component and does not share it with the other diagnosers. The Local Diagnoser performs local reasoning in order to provide local hypotheses on the observed behavior. A local hypothesis can blame the misbehavior on any internal part of the component itself, or on incorrect information received from other system components.

A Global Diagnoser is then associated with the complex system. It has no knowledge of the internal details of the components, but it knows the links between them. By exploiting such links, and the information received from Local Diagnosers, the Global Diagnoser checks the compatibility of local hypotheses, merging and/or discarding them, and possibly making inquiries on Local Diagnosers of its choice, in order to reach what can be then called a global diagnosis.

This approach is particularly suited to the case of composite Web Services, where each basic Web Service involved in the composition can be analyzed separately by a Local Diagnoser, while the Global Diagnoser is associated with the composite service, merging the local views of what has happened into a global diagnosis and explaining the occurred exception at the level of the composite service. The global diagnosis may be used to identify the most suitable recovery strategy to be applied by the service providers. In the following we describe the architecture of our proposed framework, focusing first on failure identification and then on recovery.

In order to support the identification of problems occurring during a composite Web Service execution, we propose to:

- Associate a Local Diagnoser service LD_i to each individual Web Service WS_i, in order to generate diagnostic hypotheses explaining the occurred exceptions from

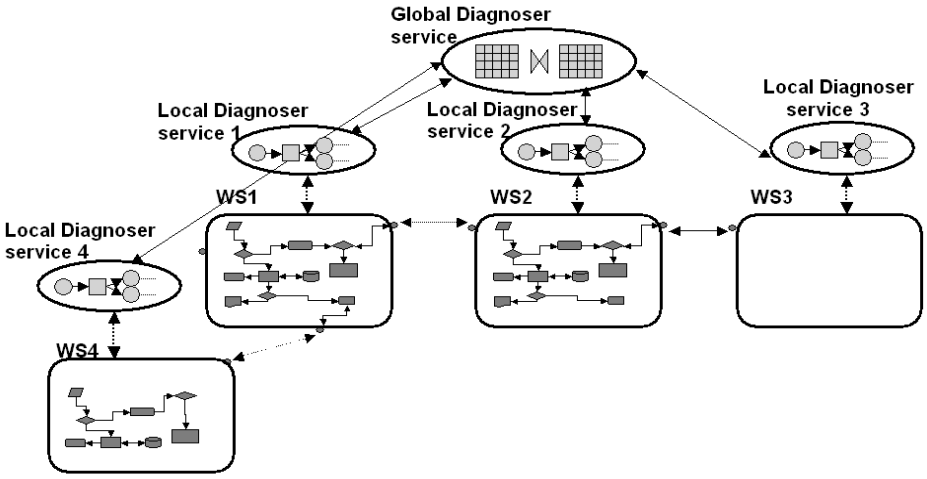

Fig. 2. Architecture for advanced fault the individual in composite Web Services.

the local point of view. 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 Web Services $\{WS_1, ..., WS_k\}$ which might be involved with the failure of the composite service. $\{WS_1, ..., WS_k\}$ is a subset of the set of composed Web Services that have sent and/or received messages from WS_i.

– Perform global reasoning about the composite service by means of a Global Diagnoser service (henceforth, Global Diagnoser) which interacts with the Local Diagnosers of the individual Web 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 and can run on remote servers, if a tight coupling with the Web Services to be diagnosed is not possible. Figure 2 shows the proposed architecture in a composite service based on the composition of four Web Services. In the figure, the composed 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 messages exchanged during both the identification of failures and the selection of the exception handlers to be applied. Moreover, the double arrows between Local Diagnosers and Global Diagnoser represent the messages exchanged to produce the global diagnostic hypotheses.

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 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.

Each Local Diagnoser relies on a diagnostic model M_i of the Web Service in its care to generate the local diagnostic hypotheses about the occurred exceptions. The model describes the control and data flow of WS_i as precisely as possible. It should be noticed that the Web Services may have a rather diverse nature: some may execute articulated workflows; others may expose their internal business logic in a more or less detailed way, e.g., because they rely on legacy software. Moreover, they may be implemented in different process languages; e.g., the BPEL [8] Web Service composition language, or a workflow description language such as BPML [9]. The diagnostic model abstracts from these details; as described in Section 4, this model can be derived from the service implementation expressed in UML 2.0 [10] or from other descriptions of the executed workflow.

4 Modeling Web Services for Diagnosis

As we said above, in our approach a diagnostic model is essentially a *relation* among finite-domain variables that expresses the intended behavior of the modelled system at a proper level of detail. It should also describe how the correct behavior of the system depends on its components working properly.

In this case, the modelled system is a (composite) Web Service. We assume that each Local Diagnoser owns a separate model of the composed Web Service in its care. Moreover, some additional information available at run-time allows the Global Diagnoser to relate hypotheses coming from different Local Diagnosers.

The features of a Web Service behavior we want to include in its diagnostic model are the following:

- Which are the actions the Web Service could perform during its execution. Actions have a central role in diagnosis, since we assume that errors arise from the failed execution of some action (i.e., an action that given a correct input does not produce a correct output, or does not produce an output at all).
- What is the dependency between the inputs and the outputs of an action, which outputs are affected by which inputs, and which outputs may result incorrect due to a failure in the action execution.
- What is the relation between the execution of the different actions. This corresponds to the control flow of the process and it includes precedence relations, conditional behavior, and so on. It also includes the relation between the outputs of an action and the inputs of another one, which allows to state how the failure of an action can have an impact on the following ones.

Since the connection between Web Services in a composition is given by an exchange of messages, it can be expressed as an additional relation between two actions, in this case belonging to two different Web Services: the output of a *send* action becomes the input of a *receive* action.

All this information (actions, data dependencies, control flow, messages exchange) can be obtained from the workflow of the business process associated to each Web Service enriched with data dependencies.

We will now briefly describe how this information is represented in order to be suitable for diagnosis; it is important to point out that given an enriched workflow representation the derivation can be performed automatically. For lack of space, it will not be possible to describe in detail the derivation process; we will therefore limit ourselves to the general principles behind it.

In order to model the above mentioned features of a Web Service with a finite-domain relation, we need to introduce four variable types:

- *Mode* variables represent the status of an action, stating whether it has been executed properly or not. Their domain is $\{ok, ab\}$. Discovering which values of such variables are consistent with the observed behavior is the goal of diagnosis.
- *Activation* variables represent the process flow at a different points in the execution, mainly before and after the execution of an action. Their domain is $\{yes, no\}$. Their intuitive meaning is "the process flow has/has not (yet) reached this point".
- *Deviation* variables represent the *correctness* of the process flow a different points in the execution, and their domain is $\{ok, ab\}$. Their intuitive meaning is "the status of the process flow at this point (whether it is active or not) is / is not the expected one".
- *Data* variables represent the correctness of the data processed by actions and flowing from one action to the next. Also their domain is $\{ok, ab\}$.

Given the data dependencies associated with it, it is quite straightforward to represent each individual action as a relation stating that:

- The correctness of each output is a function of the correctness of the inputs it depends on, *and* of the *ok* or *ab* mode of the action itself.
- The activation of the process flow, as well as its correctness, *after* the activity execution, are a function of its activation *before* the activity execution *and* of the activity mode, since a possible failure for an activity is to get blocked and prevent the process flow from continuing.

For example, suppose that an action a computes an output from two different inputs. Then we have two data variables, let us say $a.x_1$ and $a.x_2$, representing the correctness of the inputs, and a data variable $a.y$ representing the correctness of the output. Then we have a mode variable $a.m$ representing the status of the action itself, two activation variables $a.in$, $a.out$ representing the activation status before and after the action execution, and two deviation variables $a.D\_in$, $a.D\_out$ representing the correctness of such activation.

The relations for this action would then be the following:

- If $a.x_1 = ok$ and $a.x_2 = ok$ and $a.m = ok$ then $a.y = ok$. This states that the only case in which we can be sure that the output is correct is when the inputs and the action itself are correct as well.
- If $a.m = ok$ then $a.in = a.out$. Moreover, it is not possible that $a.in = no$ and $a.out = yes$. This states that when the action is working properly then the flow in output is active if and only if the flow in input is. Even if the action is failed, however, it is not possible that the flow in output activates by itself, without the action being executed.

– If $a.m = \textit{ok}$ then $a.D_{in} = a.D_{out}$. Moreover, if $a.\textit{in} \neq a.\textit{out}$ then $a.D_{in} \neq a.D_{out}$. This states that when the action is not failing then the correctness of the output flow depends solely on the correctness on the input flow. In addition, if the activation status of the input flow differs from that of the output flow, then one of the two is correct while the other is not.

The overall process flow of the Web Services is mainly translated in a relation by:

– Equating data variables when they represent the same data at the same time in the execution process. For example, if $a.x$ represents the output of action a, and $b.y$ the input of action b, and we know that a produces its output towards b, we can simply add the relation $x = y$.
– Expressing relations between the activation status of the different actions, in order to describe the process flow. For example, if we know that two actions a and b are mutually exclusive, and that variables $a.\textit{in}, b.\textit{in}$ represent respectively the input activation of a and b, we can write $(a.\textit{in} = \textit{no}$ or $b.\textit{in} = \textit{no})$ meaning that the two variables cannot have both value *yes* at the same time.

Finally, the connections between different Web Services are seen as equalities between variables. This information pertains only the Global Diagnoser; therefore, when the Global Diagnoser receives from a Local Diagnoser L_1, associated with a Web Service W_1, information concerning some inputs coming from Web Service W_2, the names of the variables dealing with those inputs are translated into the names of the variables associated with the corresponding outputs in W_2. For example, suppose that a Local Diagnosis from W_1 makes the hypothesis that its input x is incorrect (*ab*). L_1 communicates to the Global Diagnoser the name of the message containing this piece of data (e.g., *sendX*), and the sender of the message, that is W_2. Then the Global Diagnoser informs the Local Diagnoser L_2 of W_2 that it should start an investigation on the possibility that the output W_2 sent out to W_1 with message *sendX* may be incorrect. L_2 knows that the variable associated with that output is called y and can translate the information it received in its own model.

Local Diagnosers perform local diagnosis by putting in relation the model they have of a Web Service with the observations available at runtime. For such observations to be useful, they must be in relation with the model, that is, they must state something on the model variables. In other words, observations are interesting if they assess either the activation status of the process flow at a given point (e.g. “it has been observed that action a was executed”), its correctness (e.g. “it has been observed that action a was *unexpectedly* executed”) or the correctness of a piece of data (e.g. “it has been observed that output x has not a correct value”). The last two types of observation usually come through *exceptions* that the Web Service raises. In this case, the model must contain information that allows to relate the fact that an exception was raised to the values of the variables concerned by it. In most cases, an exception does not point directly to the incorrectness of a piece of data, but it rather signals a mismatch between two or more different sources, so it is possible that relations modelling exceptions have the form “If e was raised then either $a.x = \textit{ab}$ or $b.y = \textit{ab}$”. Relations modeling exceptions are thus a fundamental part of a Web Service model.

5 Exception handling

In order to support the global management of exception and compensation handlers, the standard exception handlers defined within the Web Services participating to the composite service have to be modified. In fact, a local exception handler has to interact with the reference Local Diagnoser in order to retrieve the diagnostic hypotheses (Δ) explaining the occurred exception; then, it has to handle a *context-dependent exception* concerning the composite application.[6] The context-dependent exception management is based on two main elements. First, within a composite service CS, diagnostic hypotheses must be mapped to context-dependent exceptions which are, indeed, local exceptions of some of the involved Web Services. The Local Diagnoser LD_i of a Web Service WS_i has to store a set of mappings $map(h_x, eh_x, CS)$ between each possible global diagnostic hypothesis h_x (belonging to Δ) and the corresponding exception eh_x. These mappings complement WS_i's diagnostic model in CS. Second, the default exception handling must be overridden to take the diagnostic information into account. For this purpose, a default exception handler has to throw the context-dependent exception which describes the problem at the level of the composite application.

If the workflows implemented by the individual Web Services are specified in a process language, e.g., WS-BPEL, a specific exception handler, usually called *fault handler*, may be associated to each type of exception (*WS-BPEL fault*). Thus, we propose to modify each individual Web Service WS_i as follows:

1. Each exception handler f has to be modified so that it invokes the Local Diagnoser LD_i and waits for the result before executing any further actions. The result is generated after the interaction between Local and Global Diagnosers and it consists of the context-dependent exception (eh_x) to be raised. Such value depends on the global diagnosis Δ and is selected on the basis of the previously mentioned mappings. The result received by the local exception handler f, after having invoked the Local Diagnoser, is characterized as follows:
   - If f is the appropriate handler for the current case (i.e., the context-dependent exception coincides with the local one), the result is the value held by *localExcCode*, which means that f can continue its own regular execution.
   - If the case has to be treated by means of another exception handler of the same Web Service, the result returned by the Local Diagnoser is set to the code of a different exception. In that case, f has to throw the new exception and terminate the execution of the current scope. The occurrence of the new exception in the same Web Service automatically triggers the appropriate exception handler.
   - If the Web Service is not requested to perform any exception handling procedure (i.e., another Web Service has to trigger a handler of its own), the result returned by the Local Diagnoser is null, and the active exception handler f has to terminate the execution.
2. Although most original exception handlers can be employed to manage both local and context-dependent exceptions, new exception handlers might have to be developed in order to handle new types of exceptions.

[6] The global diagnosis Δ can include more than one hypothesis h, mapping on different context-dependent exceptions. In many cases, due to the criticality of the operations involved, the selection of the handler to be triggered can involve a human intervention.

3. A WSDL operation `throwException(String excCode)`, should be provided, in order to be invoked by LD_i when the global diagnosis requires a recovery action on WS_i although WS_i itself did not raise any exception. When the Web Service performs the `throwException(String excCode)` operation, it throws the *excCode* exception, which activates the corresponding exception handler.

Consider our motivating example (henceforth, service $MyShop$) and suppose that there is a mismatch between the description of goods provided by the shop and the parcel prepared by the supplier. The exception ($wrongItemInParcel$) raised in the Warehouse WS can originate from three different causes, each one to be treated by applying different exception handlers within the Web Services. For this purpose, within service $MyShop$ the alternative global hypotheses must be mapped to the corresponding context-dependent exceptions:

- Hypothesis h_1: the supplier put the wrong item in the parcel. In this case, the global hypothesis should be mapped to an exception of the Supplier WS, in order to activate a suitable exception handler aimed at repairing the fault: *map(*h_1*, wrongItemInParcel, MyShop)*.
- Hypothesis h_2: the supplier has received a correct reservation request but it has made a mistake in writing down the list of requested items. Also in this case, the global hypothesis should be mapped to an exception of the Supplier WS: *map(*h_2*, wrongOrderSaving, MyShop)*.
- Hypothesis h_3: the Shop WS made a mistake when sending the supplier the reservation message (`Conf-Send()`). In this case, the global hypothesis should be mapped to an exception of the Shop WS, as it should start the repair activities: *map(*h_3*, wrongReservation, MyShop)*.

Given the mappings, the exception handlers of the Web Services should be modified in order to invoke the Local Diagnosers and receive their responses; see above. Moreover, new exception handlers might be added, if needed. Specifically:

- Hypothesis h_1: the local exception raised in the Warehouse WS should be ignored, by making the activated exception handler terminate execution. Moreover, the $wrongItemInParcel$ exception should be thrown on the Supplier WS in order to make it replace the item with the correct one, by interacting with the Warehouse WS. For this purpose, the Supplier WS needs an exception handler devoted to the management of the $wrongItemInParcel$ exception. The Shop WS is not involved in the management of this exception.
- Hypothesis h_2: the local exception raised in the Warehouse WS should be ignored. Moreover, the $wrongOrderSaving$ exception should be thrown on the Supplier WS in order to make it compensate the previous reservation with the warehouse (`supply()` message). The exception handler is expected to manage the interaction with the Warehouse WS in order to cancel the previous reservation. The Shop WS does not need to handle the exception.
- Hypothesis h_3: the local exception raised in the Warehouse WS should be ignored. Moreover, the $wrongReservation$ exception should be thrown on the Shop WS in order to make it withdraw the previous request (`conf-Send()` message) and

make a new reservation. The other Web Services are not affected by the exception (they are only involved in the interaction handled during the execution of the exception handler).

6 Related work

In this paper we have made the assumption that, during the design of the composite service, the designer has carried out an analysis to prevent the occurrence of deadlocks and other well-formedness problems. For this reason, the stream of work concerning the analysis of distributed processes and distributed object oriented systems is out of the scope of this paper.

Similar to our work, [11] propose to enhance failure identification by embedding Model-Based Diagnosis techniques in Web Service orchestration. However, that work differs from ours since the complex service is modeled as a Discrete Event System. In order to do that, all the activities of a Web Service are modeled explicitly and they do not take into account privacy issues, i.e. the fact that a Web Service may not want to expose its model. Moreover, the proposed method for the definition of the service model is based on the assumption that the service is implemented as a BPEL process.

The models we use are similar to those proposed in [12], where, however, the goal is general multi-agent diagnosis and not diagnosis of composed Web Services, with its specific requirements on distribution of knowledge and reasoning. This leads the authors to make some model assumptions that do not hold in our context. Moreover, that approach is purely distributed, while the context of Web Services in EAI allows to exploit the advantages of a supervised approach, with the adoption of a Global Diagnoser.

Various proposals for the management of transactional behavior in Web Service orchestration are being presented in order to support the development of reliable composite Web Services; e.g., see WS-Transaction [13] and OASIS BTP [14]. Moreover, in some decentralized approaches, e.g., in [15], local and global monitoring agents are employed to control the execution of composite services. However, such works support only basic error detection, without reasoning about exceptions. Moreover, they assume that the composed services disclose their business logic to the global monitoring agents.

Other approaches to exception handling have been proposed. For instance, [16] proposes a mechanism based on *spheres of atomicity* and exception handlers to enable a workflow system to recover from expected failures and compensate activities. Moreover, [17] presents strategies enabling human users to participate in the recovery of a workflow instance during the service execution. Other works provide classifications of exceptions, in order to propose suitable recovery strategies; e.g., see [18] and [17]. In comparison, the novelty of our work is to bind the execution of handlers to the causes of exceptions, by exploiting diagnostic reasoning mechanisms.

7 Conclusions

The work described in this paper contributes to extend standard middleware for Web Services by adding support for the diagnosis of composite services. Specifically, we

have described a framework enhancing the failure identification capabilities in decentralized Web Service composition by employing diagnostic reasoning techniques for the identification of the causes of the exceptions. These capabilities enable a Web Service workflow engine to execute exception handlers aimed at recovering from the causes of the problem, instead of dealing only with the observed exceptions.

Our proposal is based on the assumption that both the workflow logic and the exception handlers to be executed are well designed, in order to prevent the system from incurring in deadlocks and other similar problems. The design of *well-formed* workflows and compositions has received a lot of attention and is indeed an open issue in the research about distributed processes and Web Services; e.g., see [16, 15]. A next step in our work is then to analyze the design aspects of correct exception and compensation handling in decentralized composition.

References

1. Papazoglou, M., Georgakopoulos, D., eds.: Service-Oriented Computing. Volume 46. Communications of the ACM (2003)
2. W3C: Web Services Definition Language, http://www.w3.org/TR/wsdl (2002)
3. OASIS: OASIS Web Services Business Process Execution Language, http://www.oasis-open.org/committees/documents.php?wg_abbrev=wsbpel (2005)
4. Alonso, G., Casati, F., Kuno, H., Machiraju, V.: Web Services - Concepts, architectures and applications. Springer-Verlag (2004)
5. Ardissono, L., Console, L., Goy, A., Petrone, G., Picardi, C., Segnan, M., Theseider Dupré, D.: Enhancing Web Services with diagnostic capabilities. In: Proc. of European Conference on Web Services (ECOWS-05), Växjö, Sweden (2005) 182–191
6. Console, L., Picardi, C., Dupré, D.T.: A framework for decentralized qualitative model-based diagnosis. In: IJCAI. (2007) 286–291
7. Hamscher, W., Console, L., de Kleer, J., eds.: Readings in Model-Based Diagnosis. Morgan Kaufmann (1992)
8. Curbera, F., Goland, Y., Klein, J., Leymann, F., Roller, D., Thatte, S., Weerawarana, S.: Business Process Execution Language for Web Services, version 1.0, http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/ (2002)
9. Business Process Management Initiative, B.: Business Process Management Language, http://www.bpmi.org (2005)
10. Fowler, M., Scott, K.: UML distilled. ADDISON-WESLEY (2000)
11. Yan, Y., Pencolé, Y., Cordier, M., Grastien, A.: Monitoring Web Service Networks in a Model-based approach. In: Proc. of ECOWS-05, Växjö, Sweden (2005) 192–203
12. Roos, N., ten Teije, A., Witteveen, C.: A protocol for multi-agent diagnosis with spatially distributed knowledge. In: Proc. of AAMAS-2003, Melbourne, Australia (July 2003)
13. Cox, W., Cabrera, F., Copeland, G., Freund, T., Klein, J., Storey, T., Thatte, S.: Web Services Transaction (WS-Transaction), http://dev2dev.bea.com/pub/a/2004/01/ws-transaction.html (2005)
14. OASIS TC: OASIS Business Transaction Protocol, http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=business-transaction (2005)
15. Chafle, G., Chandra, S., Mann, V., Nanda, M.: Decentralized orchestration of composite Web Services. In: Proc. of WWW'2004, New York (2004) 134–143
16. Hagen, C., Alonso, G.: Exception handling in workflow management systems. IEEE Transactions on Software Engineering **26**(10) (2000) 943–958

17. Mourao, H., Antunes, P.: Exception handling through a workflow. In Meersman, R.R., Tari, Z., eds.: On the move to meaningful internet systems 2004. Springer-Verlag, Heidelberg (2004) 37–54
18. Sadiq, S.: On capturing exceptions in workflow process models. In: Int. Conf. on Business Information Systems, Poznam, Poland (2000)