Enhancing Web Services with Diagnostic Capabilities

Liliana Ardissono, Luca Console, Anna Goy, Giovanna Petrone, Claudia Picardi, Marino Segnan
Dipartimento di Informatica, Università di Torino
{liliana,lconsole,goy,giovanna,picardi,marino}@di.unito.it

Daniele Theseider Dupré
Dipartimento di Informatica, Università del Piemonte Orientale
dtd@mfn.unipmn.it

Abstract

Fault management in Web Services composed by individual services from multiple suppliers currently relies on a local analysis, that does not span across individual services, thus limiting the effectiveness of recovery strategies. We propose to address this limitation of current standards for Web Service composition by employing Model-Based Diagnosis to enhance fault analysis. We propose to add Diagnostic Web Services to the set of Web Services providing the overall service, acting as supervisors of their execution, by identifying anomalies and explaining them in terms of faults to be repaired. This approach poses the basis for the development of specialized recovery and compensation techniques aimed at addressing different problems, which could not be otherwise discriminated.

1. Introduction

Service Oriented Architectures [13] and standard languages for the publication and invocation of Web Services, such as WSDL [18], enable the exploitation of heterogeneous software by abstracting from the features of the deployment environment of applications. On top of these basic communication languages, standard Web Service composition languages, such as WS-BPEL [10], are being defined to support the development of complex applications based on the orchestration of simpler ones. Moreover, in the Semantic Web community (see, e.g., [7, 12]), languages and frameworks are being defined to support a rich specification of services and intelligent service cooperation (e.g., see [11]).

The growing worldwide acceptance of these standards is an excellent start for a realistic integration of services in the Web, as well as in Enterprise Application Integration, which represent two mainstreams of software development in the next future [1]. However, several issues have to be addressed in order to enable the effective integration of non trivial applications. In fact, rather straightforward solutions are currently adopted to support the reliability of services. The ability to detect and isolate faults during service execution would be very desirable in order to apply effective recovery actions, especially in case of complex services composed of simpler ones whose implementation is not publicly available.

In this paper we propose a framework for adding diagnostic capabilities to Web Services, using a model-based perspective [5]. The ultimate goal is to design **self-healing** services which guarantee autonomous diagnostic and recovery capabilities. When defining a complex service, composed of simpler ones, we propose to add to each service S a local diagnoser which relates hypotheses about incorrect outputs of S to a misbehavior of S itself, or to incorrect inputs from other services. A global diagnoser service is then associated with the complex service. The global diagnoser coordinates the local diagnosers, exchanging messages with them, and it can in turn compute diagnoses at the level of the global service. The structure of the resulting diagnostic service is then analogous to the structure of the original service.

For generality purposes, we aim at developing diagnostic services that work as Web Services themselves, and can be exploited within a specific complex service without requiring changes to its internal implementation. To this purpose, although we consider complex services based on the cooperation of multiple suppliers, we do not make assumptions on how the cooperation is orchestrated. In other words, the global diagnoser service does not need to know in advance how the various service suppliers interact; moreover, it does not rely on any information about the internal structure of the sub-services.

In the paper we discuss a protocol for a global diagnoser service, and we characterize the operations that local diagnosers must support in order to comply with such a protocol. The goal is the identification of the faulty service, not debugging the service itself. In addition, the local diagnoser may identify a part of the service which is claimed to be responsible for the fault. An early fault detection and a fault identification which is as precise as possible are a necessary precondition for a better recovery from faults.

We choose to adopt an approach based on the introduction of a global diagnoser service because this enables to recursively partition Web Services into aggregations of subservices, hiding the details of the aggregation to higher-level services. This is in accordance with the privacy principles which allow to design services at enterprise level (based on intra-company services) and then use such services in extranets (with other enterprises) and public internets. The global diagnoser service only needs to share a protocol with local diagnosers.

The rest of the paper is organized as follows. Section 2 sets the context of Web Service diagnosis; section 3 summarizes the main concepts of Model-Based Diagnosis and section 4 introduces the approach we propose for an architecture for Web Service diagnosis. Section 5 describes how to model services for diagnosis while section 6 introduces the protocol for the global diagnoser service, and characterizes local diagnosers. Finally, section 7 overviews existing research and future work on the topic.

2. Context

Legacy software has to be considered as a “black box” because the complete specification of its behavior is rarely provided. Moreover, when embedding legacy software in complex systems, errors may occur because software is used in ways, or run on inputs, not foreseen by its developer. The same considerations can be extended to Web Service composition, where remote services are integrated without knowing their implementation, but only relying on their public interfaces and on a high-level description of their behavior.

In Web Service composition, WS-BPEL introduces fault handlers to specify the activities to be performed when the execution of a Web Service fails. Fault handlers are associated in *ad hoc* ways to fault types and are suitable to efficiently manage local problems in the execution of activities, such as the invocation of an unknown operation. However, fault handlers do not offer actions aimed at understanding the causes of an occurred problem. This may be a limitation, especially in complex services, composed of several Web Services, where problems might be caused by the interaction between service suppliers and the absence of specialized diagnostic capabilities usually imposes the execution of coarse grained repair actions.

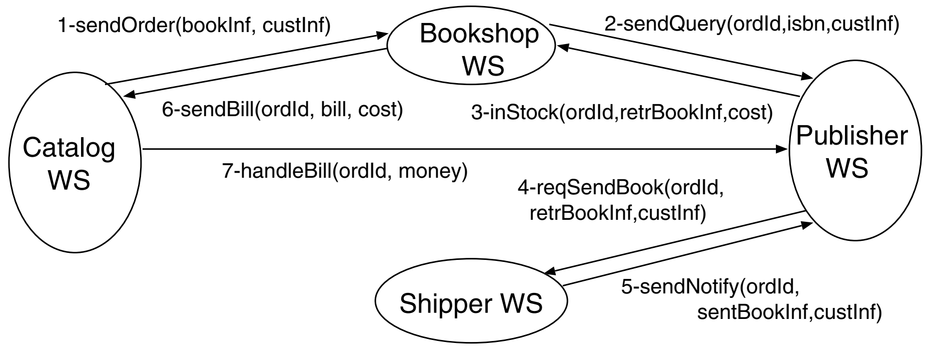
We claim that, in order to enhance the capabilities to reason about faults, and the consequent flexibility in taking recovery and compensation actions, the basic techniques introduced in process languages such as WS-BPEL should be complemented with a deeper fault analysis. We propose to distinguish local errors, which can be treated by means of *ad hoc* fault handlers, from global failures of the overall service, which require advanced reasoning techniques. Diagnosis techniques can be applied to the analysis of global failures; starting from the observation of a problem in the execution of a Web Service (e.g., the Web Service sends a fault message to its own consumer), one or more possible causes could be identified to apply a suitable recovery strategy.

We show our viewpoint on an example adapted from [17]. A bookshop offers a Web-based catalog whose user interface is implemented as a Web Service (Catalog WS) interacting with the main backoffice Web Service of the bookshop (Bookshop WS). When a customer selects a book, the Web Services exchange the following messages (see Fig. 1):

- The Catalog WS sends an order of a book to the Bookshop WS (message 1).
- The Bookshop WS retrieves the ISBN number of the book. Then it sends a request to the Publisher WS to deliver the book to the customer (message 2).
- The Publisher WS retrieves book details from the ISBN and notifies the Bookshop WS that the book is available (3). Then, the Publisher WS asks the Shipper WS to deliver the book to the customer and gets back the delivery acknowledgment (messages 4 and 5). The physical delivery of the book is not shown in the figure.
- The Bookshop WS sends the bill to the Catalog WS (6).
- Finally, the customer pays through the Catalog WS that notifies the Publisher WS (7).

Now, suppose the customer receives the wrong book. The problem might be caused by errors occurring during the execution of different Web Services and the identification of the faulty one (which is not obvious, unless suitable diagnostic reasoning is employed) is the key element for the recovery of the overall service. Specifically, different recovery actions could be performed during the service execution, depending on the source of the error.

For instance, if the (human) shipper picked the wrong book from the store (regardless of the book details), the Publisher WS should request the shipper to deliver the correct book. Instead, if the Bookshop WS made the mistake (e.g., by computing the wrong ISBN number), the problem might not be solved at all (in which case the customer should be refunded) or another service might need to be



1-sendOrder(bookInf, custInf)
Bookshop WS
2-sendQuery(ordId,isbn,custInf)
6-sendBill(ordId, bill, cost)
3-inStock(ordId,retrBookInf,cost)
Catalog WS
Publisher WS
7-handleBill(ordId, money)
4-reqSendBook(ordId, retrBookInf,custInf)
Shipper WS
5-sendNotify(ordId, sentBookInf,custInf)

Figure 1. Collaboration diagram for a book sales scenario.

contacted to retrieve the correct ISBN number and the Publisher WS should be contacted again as done in message 2.

3. Model-Based Diagnosis

Model-Based Reasoning (MBR) and, in particular, Model-Based Diagnosis (MBD) [5], have originally been proposed and used within the Artificial Intelligence community for reasoning on possibly faulty physical systems, especially in technical domains (from electronic circuits to cars and spacecrafts[3]), but they have also been applied in other domains, such as software diagnosis; see, e.g., [6].

Most MBD approaches rely on a component-oriented model of the system to be diagnosed. As noticed in [4], the emergence of component-oriented software development is therefore a good reason for evaluating the adoption of an MBR approach for diagnosing faults in component-based software systems. Before showing how MBR techniques and tools are used within the framework we propose in this paper for dealing with faults in Web Services, we sketch the relevant ideas of component-based MBD. As regards models:

- The system to be diagnosed is modeled as a set of components (e.g., for physical systems: AND gates, but also hydraulic pipes or electric resistors) and interconnections between components. In several domains, models of component types can be given and reused for multiple component instances.
- The behavior of each component is modeled as a relation on component variables. Such a model is provided for the correct and/or faulty behavior of the component; in technical domains, in particular, the behavior under alternative known *fault modes* can be provided.
- Variables typically range on discrete, finite domains, which in case of physical systems may also correspond to qualitative abstractions of continuous domains.
- Component variables include *interface variables*, that are used to define component interconnections in the system, by equating interface variables of different components; e.g., an output variable of a component with an input variable of another component. Therefore a model for the overall system, as a relation on all component variables, is at least implicitly given.
- The model of component behavior can be a static, atemporal model that relates values that different variables take at the same time, but can also relate values at different times; a way to do this is constraining changes of *state variables*, thus providing a dynamic model; see [2] for a general discussion on temporal diagnosis.

The resulting overall model of the system is therefore able to predict, or at least constrain, the effect of the incorrect behavior of a component also on variables that are not directly related to the component. This is especially important where there is, or there has to be, limited observability on the system, either because there has to be a fixed set of viable observation points (e.g., a fixed set of sensors), or because we need to minimize at run time the amount of information to be considered for discriminating among alternative diagnoses.

Diagnostic reasoning should identify diagnoses, as **assignments of behavior modes** to components, for a given set of observations (values for observable variables). 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. Discrimination should only be performed towards a given diagnostic goal, e.g., selecting an appropriate *repair* action.

There are several formalizations of MBD; see [5]. In *consistency-based diagnosis* [15], which will be used in this paper, a **diagnosis** is an **assignment of behavior modes** to components that is **consistent with observations**. For static models this means that the candidate predicts, for observable variables, a set of possible values which includes the observed one.

Under worst-case complexity analysis, the problem of finding all diagnoses, or even a single diagnosis, is intractable. In practice, the actual occurrence of combinatorial explosion heavily depends on the model and on the availability of discriminating observations; it is definitely more likely to occur for reasoning on dynamic models. It can be avoided or mitigated in several ways. For example, search could be limited to, or at least start with, single faults, i.e., candidates that assign the correct behavior to all but one component. Several optimizations are also possible to avoid redoing the same inferences for different mode assignments or different sets of observations. For instance, the model can be precompiled, or the results of reasoning on the model can be compiled, e.g., into a decision tree.

4. Architecture

In order to enhance fault management in complex services with the ability of reasoning on global failures of the overall service, we propose to:

- Associate with each basic service a **local diagnoser**, owning a description of how the service is supposed to work (see section 5); the role of local diagnosers is to provide the global diagnoser with the information needed for identifying causes of a global failure.

- Provide a **global diagnoser** which is not tied to any specific service, but is able to invoke local diagnosers and relate the information they provide, in order to reach a diagnosis for the overall complex service. In case the supply chain has several levels, several global diagnosers may form a hierarchy, where a higher level global diagnoser sees the lower level ones as local diagnosers.

Each local diagnoser interacts with its own Web Service and with the global diagnoser. The global diagnoser interacts only with local diagnosers. More precisely, the interaction follows this pattern:

- During service execution, each local diagnoser should monitor the activities carried out by its Web Service, logging the messages it exchanges with the other peers. The diagnoser exploits an internal "observer" component collecting the messages and locally saving them for later inspection. Notice that when a Web Service composes a set of sub-suppliers, the local diagnoser role must be filled by the global diagnoser of the sub-network of cooperating services. On the other hand, an atomic Web Service can have a basic local diagnoser, that does not need to exploit other lower-level diagnosers in order to do its job.

  Local diagnosers need to exploit a model of the Web Service in their care, describing the activities carried out by the Web Service, the messages it exchanges, the information about dependencies between parameters and alarm messages, as detailed in section 5.

- When a local diagnoser receives an alarm message (denoting a problem in the execution of the service it monitors), it starts reasoning about the problem to identify its possible causes, which may be internal to the Web Service or external (erroneous inputs from other services). The diagnoser can do this by analyzing the messages it previously logged.

- The local diagnoser informs the global diagnoser about the alarm it received and the hypotheses it made on the causes of the error. The global diagnoser starts invoking other local diagnosers (following a diagnostic reasoning pattern, detailed in section 6.2) and relating the different answers, in order to reach one or more global candidate diagnoses that are consistent with reasoning performed by local diagnosers.

- The global diagnosis can be employed at the level of the complex service for the selection of very specific recovery and compensation strategies. If the complex service is designed by specifying the choreography of the invoked services in a process language such as WS-BPEL, a different handler may be associated to each type of problem (WS-BPEL fault). In this perspective, the global diagnoser, with its higher capability of identifying faults, may be decisive in allowing the workflow engine of the complex Web Service to activate the most suitable recovery actions.

From the communication point of view, the inclusion of local and global diagnosers in the architecture of a complex Web Service is relatively seamless because diagnosers can be implemented as Web Services (local/global diagnoser WS) interacting with the other peers via WSDL messages. Specifically:

- Local diagnosers must offer a WSDL operation (`logMessage(String wsdlMsg)`) for the reception of the messages to be logged.

- Each Web Service must send copies of the inbound and outbound messages to its local diagnoser. To this purpose, each Web Service must be equipped with a "logging service" proxy which intercepts WSDL messages and sends a copy of each message to Local Diagnoser WS through the "logMessage" port.

- The global diagnoser must offer a WSDL operati (`activate(Collection hypotheses)`) to be us by local diagnosers to trigger the global diagnos process (explained in Sect. 6).

- Local diagnosers must offer a WSDL operati (`extend(Collection hypotheses)`) to be used the global diagnoser in order to invoke them (also d scribed in Sect. 6).

The proposed approach is modular and supports a sea less introduction of advanced fault reasoning in the ma agement of complex Web Services. The key point is th specialized reasoning techniques can be exploited by loc diagnosers without imposing the same techniques on any the involved Web Services. Although we require that W Services notify local diagnosers about (normal and fau messages they receive from or send to other services, this feature can be added to the invoked services without changing their internal structure. Moreover, if one of the involved services does not have a local diagnoser, or the model of the service exploited by the local diagnoser is very rough, the global diagnoser can still perform its job but the results may be less precise (e.g., it may not be possible to rule out the non-diagnosed service as the cause for the error).

5. Models for the Diagnosis of Web Services

We assume that each Web Service is modeled as a set of inter-related activities which show how the outputs of the service depend on its inputs. The model of the simplest Web Services consists of a single activity; the model of a complex one specifies a partially ordered set of activities which includes internal operations carried out by the service and invocations of other suppliers (if any).

The model of a Web Service enables diagnostic reasoning to correlate input and output parameters and to know whether an activity carries out some computation that may fail, producing as a consequence an erroneous output. In particular, we are interested in distinguishing three cases:

1. when an output parameter coincides with an input parameter, we say that the activity *forwards* (FW for short) the input to the output;

2. when an output parameter is created during an activity, we say the activity is the *source* (SRC) for it;

3. when an output parameter is computed by the activity from one or more inputs, we say that it is the result of an *elaboration* (EL).

We represent this with a block diagram. Figure 2 shows, as an example, the block diagram for an internal activity of the Bookshop WS performed upon reception of message 1.

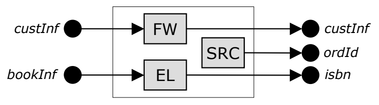


Figure 2. Dependencies for an activity.

The diagram states that the value of *custInf* is forwarded by this activity, while the value of *ordId* is provided by it, and the value of *isbn* is computed starting from *bookInf*.

Intuitively, an activity can introduce errors in parameters it computes (EL blocks) or it is source of (SRC blocks), while in the case of forwarded (FW blocks) parameters, it simply propagates errors that are someone else's responsibility.[1]

Symptom information is provided by the presence of alarms, which triggers the diagnostic process; by the absence of other alarms; or by additional test conditions on logged messages introduced for discrimination.

The goal of diagnosis is to find activities that can be responsible for the alarm, performing discrimination to the purpose of selecting the appropriate recovery action. In defining the search space for diagnoses, there are some issues to take into account. First, EL and SRC blocks play a critical role, as the computation of the values could be faulty. Second, FW blocks can be ignored under the assumption that they do not modify the correctness of parameters. Third, alarm points are very important to assess the correctness of portions of service execution and focus the search on limited areas of the computation.

The model for the Web Services is derived from the activity representation described earlier, as follows.[2] Each activity corresponds to a component with interface variables. For each input (resp., output) variable v of an activity a, a variable $a.v_{in}$ (resp., $a.v_{out}$) is introduced in the model, with a binary domain:

- The *ok* value for $a.v_{in}$ represents the fact that in a given execution of the service, v has the expected value;

- The *ab* (abnormal) value means that v has a different value from the expected one.

This distinction does not mean that, for all variables mentioned in the activity representation, it must be possible to know whether they have the expected value or not; even for models of physical systems, not all variables in the model are observable.

Each activity has an *ok* mode and a *fail* mode. For each activity a, the behavior mode is represented by an additional variable $a.m$.

[1]We assume that the transmission of messages is not error prone, and thus the relation between input and output parameters, in this case, can be seen as an identity relation.

[2]We limit the discussion to the case of acyclic dependencies.

$a.m$	$a.in_1$	$a.in_2$	$a.out$
ok	*ok*	*ok*	*ok*
ok	*ab*	*ok*	*
ok	*ok*	*ab*	*
ok	*ab*	*ab*	*
fail	*	*	*

Table 1. Model for an EL block with 2 inputs, ok mode.

A basic model for EL dependencies of components (and SRC dependencies, which can be seen as ELs with no inputs) states that *(i)* in the *ok* mode, if one of the inputs is incorrect, then the output may be incorrect, while if the inputs are all correct then also the output is correct; *(ii)* in the *fail* mode, the output *may* be incorrect regardless of the inputs' status, i.e., all combinations are possible.

Table 1 shows the model for the case of an EL dependency with two inputs ("$*$" means "any value"), depending on the behavior mode of its activity a. This model is a general one and includes the following possibilities:

- In the *ok* mode, an incorrect input gives rise to a correct output, i.e., the error does not propagate through an activity. This might be possible depending on the specific elaboration abstracted in the EL dependency.[3]
- Multiple incorrect inputs to an *ok* activity, or an incorrect input in the *fail* mode, may give rise to the correct output, a phenomenon known as *fault masking*.

Such general models are necessary to provide complete coverage of fault., but specific knowledge about some activity may allow to restrict them, e.g. by stating that if the activity is in *fail* mode then the output *must* be incorrect. Part of this generality is not needed to diagnose a single execution of the Web Service, but could be useful to put together diagnostic information from several runs, where a faulty behavior manifests itself only in some run.

Alarms are modeled as additional activity variables, tied to the rest of the activity model by an explicitly stated relation. As an example, table 2 shows the model for an alarm point *al* inside activity a, that compares two of the activity's variables (x and y) and signals an error if they are different. The model for additional test conditions is analogous.

$a.x$	$a.y$	$a.al$
ok	*ok*	*ok*
ab	*ok*	*ab*
ok	*ab*	*ab*
ab	*ab*	*

Table 2. Model for an alarm or test.

[3]For instance, knowing that a specific EL instance corresponds to an injective function would eliminate this possibility.

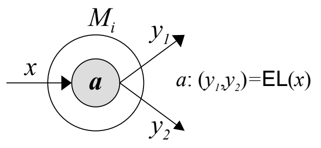


Figure 3. A simple model M_i

Again, this is a general model that takes into account the possibility that $a.x$ and $a.y$ could be both incorrect, but consistently with respect to each other (for example because they are both derived from the same incorrect source).

6. The Diagnostic Protocol

We first give an informal description of the interaction between local diagnosers and the global diagnoser service ***D*** (Sect. 6.1). Then we formalize a protocol for ***D*** (Sect. 6.2). As to local diagnosers, we characterize their operations, without providing specific algorithms (Sect. 6.3).

6.1. Interaction Among Diagnosers

The global diagnoser ***D*** does not initially have any information on the individual Web Services. Its main job is to put together information coming from local diagnosers and to select which local diagnosers can provide useful information.

When an alarm is raised in a Web Service W_i, the local diagnoser A_i receives it. A_i must explain it, and provide ***D*** with its hypotheses, invoking the `activate` operation. Each explanation may ascribe the malfunction to failed internal activities and/or abnormal inputs. It may also be endowed with predictions of additional output values, which can be exploited by ***D*** in order to validate or reject the hypothesis. For example, let us consider the simple service whose model is depicted in Fig. 3. Suppose that the alarm states that $a.y_2 = ab$: then the two possible explanations are that either $a.m =$ *fail* (and then the correctness of $a.x$ is irrelevant) or $a.x =$ *ab* (and then the behavior mode of a is irrelevant). None of the two hypotheses, however, enables us to make any prediction over $a.y_1$, which may be either *ok* or *ab* in both cases.

When ***D*** receives a local explanation from a local diagnoser A_i, it can proceed as follows:[4]

- If a Web Service W_j has been blamed of incorrect outputs, then ***D*** can ask its local diagnoser A_j to explain them. A_j can either reject the blame, explain it with an internal failure or blame it on another service that may have sent the wrong input.

[4]We assume that each interaction among Web Services is identified by a *conversation id* which is mentioned in each information exchange between local diagnosers and ***D***, in order to identify a diagnostic session.

- If a fault hypothesis by A_i has provided additional predictions on output values sent to a Web Service W_k, then ***D*** can ask A_k to validate the hypothesis by checking whether the predicted symptoms have occurred, or by making further predictions.

Hypotheses are maintained and processed by diagnosers as *partial assignments* to interface variables and behavior modes of the involved local models. Unassigned variables represent parts of the overall model that have not yet been explored, and possibly do not need to be explored, thus limiting invocations to local diagnosers. For example the two hypotheses mentioned above for the model in Fig. 3 may be represented by the following two assignments:

$a.m$	$a.x$	$a.y_1$	$a.y_2$
fail	*	*	*ab*
*	*ab*	*	*ab*

The presence, in both cases, of a $*$ for $a.y_1$ means that it is not possible to validate or reject these hypotheses by asking the service that receives that value in input.

The global diagnoser sends hypotheses to local diagnoser for explanation and/or validation by invoking the `extend` operation. As we will see in detail in Sect. 6.3, local diagnosers explain blames and validate symptoms by providing extensions to partial assignments that assign values to relevant unassigned variables.

6.2. A Protocol for the Global Diagnoser *D*

While performing diagnosis, ***D*** keeps track of the progress by means of a list H of current partial assignments. Values are only assigned by local diagnosers, thus ***D*** becomes aware of the existence of a variable x only when a local diagnoser assigns a value to it. We will denote with $\alpha(x)$ the value of variable x in assignment α. We will write $\alpha(x) = *$ to denote that α does not assign any value to x.

For each assignment $\alpha \in H$ and for every interface variable x such that $\alpha(x) \neq *$ we assume that the identities of the sender $\mathsf{SND}(x)$ and the receiver $\mathsf{RCV}(x)$ of the messages where x is specified are known to ***D***: one is the web service W_i whose local diagnoser A_i first assigned a value to x, the identity of the other is provided by A_i itself, that retrieves it from logged messages. Notice that the receiver and sender of a message only need to be known at run-time. Moreover, ***D*** associates with each $\alpha \in H$ a list $\mathbf{L}_\alpha$ of local diagnosers that should extend α.

Given a partial assignment $\alpha \in H$ we denote by $\alpha(M_i)$ its restriction to interface variables and behavior modes of M_i, and by $\alpha(\overline{M_i})$ its restriction to interface variables and behavior modes *not* in M_i.

Local `extend` operations work on partial assignments restricted to the local model they are invoked on. `extend` will be precisely characterized in the following section; for now it suffices to know that, for each $\alpha(M_i)$ it receives in input, it returns a set of extensions $\textbf{Ext}(\alpha(M_i))$ which relate values assigned in $\alpha(M_i)$ to values of other interface variables of M_i or to behavior modes of activities in M_i; if the set of extensions is empty the assignment is considered to be *rejected*, because (as we will see in the next section) this means that the assignment is inconsistent with M_i and/or observations performed by its local diagnoser. The diagnostic process is started by a local diagnoser which is awakened by an alarm, and calls `extend` on itself to explain it. The result is provided to ***D*** as the initial value for H. ***D*** then executes a loop with the following steps.

Step 1: select the next request to a local diagnoser A_i. ***D*** finds a local diagnoser A_i that belongs to $\mathbf{L}_\alpha$ for some $\alpha \in H$; if there is none, exits the loop. From the point of view of correctness, how the choice is performed is ininfluent. In Sect. 7 we will discuss policies.

Step 2: invoke `extend` on A_i. If A_i has never been invoked before in this diagnostic process, then the input to `extend` is $\{\alpha(M_i) \mid \alpha \in H\}$ (that is, the restrictions to M_i of the whole set H). Otherwise the input is the set of assignments $\{\alpha(M_i) \mid \alpha \in H \text{ and } A_i \in \mathbf{L}_\alpha\}$ (that is, the restrictions to M_i of those assignments that have changed from the last invocation).

Step 3: update H and the $\mathbf{L}_\alpha$ lists. ***D*** receives the output of `extend` from A_i. For each $\alpha(M_i)$ in input, `extend` has returned a set $\textbf{Ext}(\alpha(M_i))$ of extensions. Then α is *replaced* in H by the set of assignments

$$\{\beta \mid \beta = \alpha(\overline{M_i}) \cup \gamma \text{ and } \gamma \in \textbf{Ext}(\alpha(M_i))\}.$$

This implies that rejected assignments, having no extensions, are removed from H. For each assignment $\beta = \alpha(\overline{M_i}) \cup \gamma$ added in this way $\mathbf{L}_\beta$ is built as follows:

- For each $j \neq i$, if $A_j \in \mathbf{L}_\alpha$ then $A_j \in \mathbf{L}_\beta$;
- If there is an interface variable x such that $\mathsf{RCV}(x) = A_i$, $\alpha(x) = *$ and $\beta(x) = ab$ then $\mathsf{SND}(x) \in \mathbf{L}_\beta$. Intuitively, if A_i has blamed W_j for an abnormal value on its inputs, then A_j is asked to give an explanation.
- If there is an interface variable y such that $\mathsf{SND}(y) = A_i$, $\alpha(y) = *$ and $\beta(y) \neq *$ then $\mathsf{RCV}(y) \in \mathbf{L}_\beta$. Intuitively, if A_i has predicted a symptom for an output sent to W_j, then A_j is asked to validate it.

Notice that the diagnostic process terminates: new requests for `extend` are generated only if assignments are properly extended, but assignments cannot be extended indefinitely.

At the end of the diagnostic process we can extract minimal consistency- based diagnoses from H as follows. We associate a diagnosis $\Delta(\alpha)$ to every $\alpha \in H$:

$$\Delta(\alpha) = \{x \mid x \text{ is an internal activity and } \alpha(x) = \textit{failed}\}$$

6.3. A Characterization of Local Diagnosers

As described in the previous sections, the input to `extend` is a set of partial assignments of *ok/ab* values to interface variables in M_i and of *ok/fail* modes to internal activities. A local diagnoser A_i regards α as an assignment to *all of its variables* and behavior modes, although internal variables are all unassigned.

The output of `extend` is a set of extensions **Ext**(α) for every assignment α received in input. Given an extended assignment β computed internally, `extend` only returns its restriction *pub*(β) to public variables, which, as explained before, we assume to be interface and behavior mode variables.

Notice that information on the behaviour mode of local activities can be better regarded as private; in this case local diagnosers may not want to share it with ***D***. From the description of ***D***'s protocol it appears that ***D*** only needs this information to send it back to A_i in order to identify a diagnostic hypothesis. Thus ***D*** can receive a *coded* version of assignments that hides behaviour modes, but that the proper local diagnoser is able to decode.

Each local diagnoser should extend partial assignments so that unassigned variables are only those that do not provide relevant information with respect to the current diagnostic process. The notion of *admissibility* of an assignment captures this idea: an assignment is *admissible* in a given model if it does not allow to infer anything more than the model alone on unassigned variables.

Definition 1 *Let us denote by* $\mathsf{DOM}(\alpha)$ *the set of all variables* x *in a given model such that* $\alpha(x) \neq *$, *and by* $\overline{\mathsf{DOM}}(\alpha)$ *the set of unassigned variables. We say that an assignment* α *is* admissible *in* M_i *if* (i) *it is consistent with* M_i *and* (ii) *the restriction of* $M_i \cup \alpha$ *to variables in* $\overline{\mathsf{DOM}}(\alpha)$ *is equivalent to the restriction of* M_i *alone to* $\overline{\mathsf{DOM}}(\alpha)$*:* $(M_i \cup \alpha) \mid_{\overline{\mathsf{DOM}}(\alpha)} \equiv M_i \mid_{\overline{\mathsf{DOM}}(\alpha)}$.

Requirement *(i)* (consistency) is actually implied by requirement *(ii)* for all but total assignments, for which $\overline{\mathsf{DOM}}(\alpha)$ is empty.

As an example, let us consider again the simple model M_i in Fig. 3, where we assume that activity a is modeled with a single EL block, whose model we provided in Sect. 5. Let us consider the following partial assignments:

	$a.m$	$a.x$	$a.y_1$	$a.y_2$
α_1	$*$	$*$	$*$	*ab*
α_2	*ok*	*ok*	$*$	*ab*
α_3	*fail*	$*$	$*$	*ab*

Assignment α_1 is consistent with M_i but it is not admissible: in fact, M_i is consistent with $a.m$, $a.x$ and $a.y_1$ being all *ok*, while $M_i \cup \alpha$ it is not, since when both $a.m$ and $a.x$ are *ok* also $a.y_1$ and $a.y_2$ must be ok. For the same reason, assignment α_2 is both inconsistent ad unadmissible wrt M_i. On the contrary, α_3 is admissible: in fact, it is consistent with all values of $a.x$ and $a.y_1$.

Given an input set S of partial assignments, for each $\alpha \in S$, `extend` computes a (possibly empty) set of extensions **Ext**(α), defined as follows:

Definition 2 *Let* A_i *be a local diagnoser with model* M_i*, and let* α *be a partial assignment received by* A_i *as input to an* `extend` *operation. Let moreover* ω *denote the assignment corresponding to internal observations (if any). The set* **Ext**(α) *computed by* `extend` *is the set of assignments:*

$$\{pub(\gamma) \mid \gamma \text{ is a minimal admissible extension of } \alpha \cup \omega\}$$

In the above example, if we compute the extensions of assignment α_1, we have **Ext**$(\alpha_1) = \{\gamma_1, \gamma_2\}$ where:

	$a.m$	$a.x$	$a.y_1$	$a.y_2$
γ_1	*fail*	$*$	$*$	*ab*
γ_2	$*$	*ab*	$*$	*ab*

In this case, all possible extensions of γ_1 and γ_2 are admissible in the model. However, this is not true in general: an admissible assignment may have extensions that are inconsistent in the model. For example, the empty assignment is always admissible in any model, but obviously some of its extensions will not.

Notice that `extend` performs both a *consistency-based* explanation and a *consistency-based prediction*. Given an input assignment α, an observations assignment ω and a minimal admissible extension γ of $\alpha \cup \omega$, we have that:

- newly-assigned values in γ to input variables or behavior modes can be seen as *explanations* of observations or output values assigned in α;
- newly-assigned values in γ to output variables can be seen as additional symptoms *predicted* by the above mentioned explanations.

Minimal admissible extensions capture the idea that a local diagnoser A_i should relate abnormality of one of the interface variables of W_i to other abnormalities (internal faults of W_i, or abnormalities of other interface variables), compromising as little as possible on the internal structure and model of W_i.

7. Conclusions

In this paper we proposed a partially distributed *model-based* approach to diagnosis of complex Web Services. Web Services are modeled in a *component-oriented* fashion, in the style of model-based diagnosis [5]; internal service activities correspond to *components*, in the sense of smallest diagnosable units. For individual activities we adopted *grey box* models: we do not model the internal behavior of an activity, but only the correlation between inputs and outputs. From this information we can infer how the correct/incorrect status of input parameters and of the activity itself affects the correct/incorrect status of output parameters. In this sense our models are close to those in [16], where, however, the focus is not on the diagnosis of composed Web Services, with its specific requirements on distribution of knowledge and reasoning. Their approach is purely distributed; in the context of WSs, we motivated the adoption of a global diagnostic service for the composed service, which allows to reduce the communication flow between services. Moreover, [16] makes some restrictive assumptions on models. Another advantage of our approach is that it makes selected predictions for discriminating candidates, but, by exploiting partial assignments, it avoids investigating those parts of the model that are not involved by blames or predicted symptoms.

A decentralized approach to diagnosis has been proposed in [14]. The application (telecommunication networks) is significantly different from ours, posing a very different problem. In our case, an alarm may be raised in a point that is far away from the failure source. In their case, a failure causes a chain of alarms, the first of which points to the failure source. However, due to the distributed nature of the network, the order in which alarms are received is not the same in which they are raised, thus the problem of finding the failure source.

A similar approach has been proposed for component-based software in [4], where chains of software exceptions are considered instead of alarms. Although the field of application is close to Web Services, the analyzed problem remains different from the one tackled in this paper. Moreover, software components are modeled in black-box fashion, considering only their alarm-raising capability and not the correlations between input and output parameters.

In [9] Web Services are modeled in DAML-S, a Semantic Web ontology with a situation calculus semantics; the model is translated to Petri Nets for simulation and verification. Due to the different goals, their models provide a different abstraction of the Web Services with respect to the models proposed in this paper, with different implications from the computational point of view: for example, our models do not require reasoning on state changes. In principle, simulation/verification and diagnosis of systems (including software systems) could be based on a unified modeling approach.

Before such a goal can be pursued for Web Services, some computational issues related to the diagnosis approach presented in this paper should be further analyzed. First of all, we did not specify which strategy $\boldsymbol{D}$ exploits in order to schedule `extend` invocations on local diagnosers. Such a strategy would strongly depend on whether $\boldsymbol{D}$ knows in advance something about the interaction between the composed WSs. In fact, as we noticed in Sect. 2, the diagnostic framework we define does not make any assumption on how the services coordinate (so that, initially, the global diagnostic service has no information on how the services are composed). Several approaches to coordination have been proposed in the Web Service community; e.g., cooperation based on a workflow orchestrated by a service, or based on intelligent invocation strategies relying on rich Semantic Web descriptions of service specifications. The availability of information about the network of cooperation between services or about semantic specifications of services could be used to focus the diagnostic process, and to define scheduling policies for the invocations to local diagnosers.

As to local diagnosers, we proposed a characterization of their operations (which, like most diagnostic tasks, can be computationally expensive in the worst case) without a specific algorithm. A thorough analysis of scalability issues (also regarding the task of the global diagnoser) is out of the scope of this paper, but for local diagnosis precompilation and approximation techniques can be used to achieve diagnostic results efficiently for at least some classes of models: in particular, using templates and their default models should allow to use precompiled results.

Our goals have some relation with existing work about program specification and verification and for the design and development of reliable and robust Object Oriented software [8] (e.g., Design by Contract™). In fact, the WS models we introduce specify input/output relations between the parameters of the WS operations to be invoked. However, our approach is currently aimed at being integrated with existing standards for the development of complex Web Services, instead of representing a specific design and development programming environment, such as Eiffel. Of course, a long-term approach for the development of self-healing Web Services should benefit from the adoption of design guidelines, and this is in fact among the goals of the WS-Diamond project [19] starting in late 2005.

There are moreover specific differences with respect to the Design by Contract™approach.

- First of all, our approach supports the dynamic integration of Web Services in open environments: the global diagnoser works, in general, without prior knowledge of the choreography according to which the individual Web Services are invoked by the complex service. It should also be noted that, in such an open environment, the invoked Web Services may not respect their own specification, but this can only be discovered at service invocation time. Moreover, additional problems may arise (e.g., transmission errors) with the invocation of remote services, even in case their software is correct.

- Second, the application of diagnostic algorithms supports the identification of non trivial errors, caused by the propagation of wrong inputs which satisfy the input conditions of Web Services and thus are discovered late in the execution of the complex service. Consider, for example, a wrong, but well formed ISBN number in the bookshop example. Moreover, diagnostic reasoning is able to put together information from different observations in order to discriminate alternative causes of a single error. This is especially true in the cases where the choreography of the complex service is known a priori (e.g., in Enterprise Application Integration). In fact, the designer of the complex service can specify test conditions supporting diagnostic reasoning aimed at identifying the best explanations for the wrong behavior.

References

[1] G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services - Concepts, architectures and applications*. Springer, 2004.

[2] V. Brusoni, L. Console, P. Terenziani, and D. Theseider Dupré. A spectrum of definitions for temporal model-based diagnosis. *Artificial Intelligence*, 102(1):39–79, 1998.

[3] L. Console and O. Dressler. Model-based diagnosis in the real world: lessons learned and challenges remaining. In *Proc. 16th IJCAI*, pages 1393–1400, Stockholm, 1999.

[4] I. Grosclaude. Model-based monitoring of component-based software systems. In *Proc. 15th Int. Work. on Principles of Diagnosis*, pages 155–160, 2004.

[5] W. Hamscher, L. Console, and J. de Kleer, editors. *Readings in Model-Based Diagnosis*. Morgan Kaufmann, 1992.

[6] C. Mateis, M. Stumptner, D. Wieland, and F. Wotawa. Model-based debugging of Java programs. In *Proc. 4th Int. Workshop on Automatic Debugging (AADEBUG-00)*, Munich, 2000.

[7] S. McIlraith, T. Son, and H. Zeng. Semantic Web Services. *IEEE Intelligent Systems*, 16(2):46–53, 2001.

[8] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 1997.

[9] S. Narayanan and S. McIlraith. Simulation, verification and automated composition of web services. In *Proc. 11th Int. World Wide Web Conference (WWW-11)*, 2002.

[10] OASIS. OASIS Web Services Business Process Execution Language. http://www.oasis-open.org/committees/documents.php?wg_abbrev=wsbpel, 2005.

[11] OWL Services Coalition. OWL-S: Semantic Markup for Web Services. http://www.daml.org/services/owl-s/1.1B/owl-s/owl-s.html, 2004.

[12] M. Paolucci, K. Sycara, T. Nishimura, and N. Srinivasan. Toward a Semantic Web e-commerce. In *Proc. of 6th Int. Conf. on Business Information Systems (BIS'2003)*, Colorado Springs, Colorado, 2003.

[13] M. Papazoglou and D. Georgakopoulos, editors. *Service-Oriented Computing*, volume 46. Communications of the ACM, 2003.

[14] Y. Pencolé and M. Cordier. A formal framework for the decentralised diagnosis of large scale discrete event systems and its application to telecommunication networks. *Artificial Intelligence*, 164(1-2), 2005.

[15] R. Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57–96, 1987.

[16] N. Roos, A. ten Teije, and C. Witteveen. A protocol for multi-agent diagnosis with spatially distributed knowledge. In *2nd Int. Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS-2003)*, Melbourne, Australia, July 2003.

[17] W. van der Aalst and K. van Hee. *Workflow Management - Models, Methods, and Systems*. The MIT Press, 2002.

[18] W3C. Web Services Definition Language Version 2.0. http://www.w3.org/TR/wsdl20/, 2004.

[19] WS-Diamond. Web Service Diagnosability, Monitoring & Diagnosis. http://wsdiamond.di.unito.it, 2005.