WS-DIAMOND
Web Services – DIAgnosability, MONitoring and Diagnosis

The Ws-DIAMOND Team*

Abstract

Self-healing software is one of the challenges for IST research. The WS-DIAMOND project aims at making a step in this direction by developing a framework for self-healing Web Services. In particular, the project aims at:

- Defining an framework for self-healing service execution of conversationally complex Web Services, where monitoring, detection and diagnosis of anomalous situations, due to functional or non-functional errors, are carried on and repair/ reconfiguration is performed, thus guaranteeing reliability and availability of Web Services;
- Defining a methodology and tools for service design that guarantee effective and efficient diagnosability/ repairability during execution.

The research builds upon results in different areas such as model-based diagnosis, semantic Web Services, cooperative information systems and Web Service composition. It goes beyond a number of current projects in the area of Service Oriented Computing, which do not consider the monitoring, diagnosing and repairing of Web Services. This paper describes the achievements in the first phase of the project.

Introduction

Ws-DIAMOND is a Strep Project funded by the EU commission under the FET-Open framework. It started in September 2005 and will run until 2008. The project aims at making a step in the direction of Self Healing Web Services. The project, in particular, addresses two different issues concerning Self Healing capabilities:

- **The first goal of WS-Diamond** is to develop a framework for **self-healing** Web Services. A self-healing Web Service is able to **monitor** itself, to **diagnose** the causes of a failure and to **recover** from the failure, where a failure can be either functional, such as the inability to provide a given service, or non-functional, such as a loss of service quality. The focus of WS-Diamond is on composite and conversationally complex Web Services, where *composite* means that the Web Service relies on the integration of various other services, while *conversationally complex* means that during service provision a Web Service needs to carry out a complex interaction with the consumer application.
- **The second goal of WS-Diamond** is to devise guidelines (and tools) for designing services in such a way that they can be easily diagnosed and recovered during their execution.

During the first phase the project concentrated on the first issue and this will be the focus of this paper.

Why is self-healing critical for Web Services? Computing facilities are increasing at a very rapid pace, presenting new forms of interaction, such as in portable and mobile devices, home and business intelligent appliances. This led to the development of complex services to support human activities), in particular creating networks of co-operating services (Web Services). Various complex and intelligent services are becoming available, to support most of our activities, possibly creating a new social environment for interacting and co-operating with other people. The same availability of these services will be critical for allowing us to carry on such activities, in the same way as today we cannot work without having access to corporate or external knowledge sources. The availability and reliability of services will be of paramount importance. Indeed the reliability and availability of software and the possibility of creating self-healing software are recognized as one of the major challenges for IST research in the next years (ISTAG 2004).

Current standards for Web Service design and execution do not include the support for self-healing execution. Simple approaches to perform recovery after failures have been developed but none of them includes approaches to detect the actual causes of the problem (fault diagnosis) and does not support recovery based on fault localization.

WS-DIAMOND proposes the adoption of Model-based diagnosis to tackle this problem. Indeed MBD recently extended its focus from "traditional domains" (artefacts) to new ones, including in particular software systems, communication networks, distributed systems. Particularly significant with respect to this project is the application to

---

(*) WS-DIAMOND (IST-516933) partners: Dipartimento di Informatica – Università di Torino (coordinator), Vrije Universiteit Amsterdam, Diparimento di Elettronica e Informatica Politecnico di Milano, Universitaet Klagenfurt, Université Paris Sud, LAAS Toulouse, IRISA Université Rennes 1, Universitaet Vienna.

WS-DIAMONDers: L. Console, D. Ardagna, L. Ardissono, S. Bocconi, C. Cappiello, M.O. Cordier, P. Dague, K. Drira, J. Eder, G. Friedrich, M.G. Fugini, R. Furnari, A Goy, K. Guennoun, A. Hess, V. , Ivanchenko, X. Le Guillou, M. Lehmann, J. Mangler, Yingmin Li, T. Melliti, S. Modafferi, E. Mussi, Y. Pencole, G. Petrone, B. Pernici, C. Picardi, X. Pucel, S. Robin, L. Rozé, M. Segnan, A. Tahamtan, A Ten Tejie, D. Theseider Dupré, L. Trave Massuyes, F. Van Harmelen, T. Vidal.

Contact address: Luca Console, Dipartimento di Informatica, Università di Torino, Corso Svizzera 185, I-10149 Torino (Italy). Email: luca.console@di.unito.it

WS-DIAMOND is funded by the European Commission, IST, FET-OPEN framework, under grant IST-516933.

software diagnosis in which the same basic technologies have been successfully applied to debug programs (Mateis *et al.* 00; Mayer *et al.* 02) and. component-based software (Grosclaude 04; Peischl et al. 06). The same approach has been applied to the case of Web Services (Mayer et al., 06). The focus of those works is different from our work since they aim at debugging) problems in the composition of Services (orchestration), rather than diagnosing (and repairing) problems arising during service execution.

The paper describes the results of the first phase of WS-DIAMOND. We first introduce an application that will be used as an example in the paper. Then we overview the results we achieved, discussing the assumptions we made in this first phase; we introduce the conceptual framework and architecture we developed for the platform supporting self-healing service execution. In the following sections we discuss the approach we adopted to diagnosis and repair planning. We conclude discussing directions of research in the project.

An application scenario

The main application scenario that we selected is an e-commerce scenario, concerned with a company (called FoodShop) that sells food products on the Web. The company has an online shop (that does not have a physical counterpart) and several warehouses (WH_1, …, WH_n) located in different areas that are responsible for stocking unperishable goods and physically delivering items to customers, depending on the area each customer lives in. Customers interact with the FoodShop Company in order to place their orders, pay the bills and receive their goods. In case of perishable items, that cannot be stocked, or in case of out-of-stock items, the FoodShop Company must interact with one or more suppliers (SUP_1, …, SUP_m).

In the following we describe the business process that brings from the customer order to the parcel delivery, which is of course executed through the cooperation of several services. In particular, in each business process instance we have one instance of the **Shop** service, one instance of a **Warehouse** service, and one or more instances of **Supplier** services. It is important to point out that the business process includes activities that are actually carried out by humans, such as the preparation of the supply package or the physical delivery to the customer. However, we will assume that these activities have an electronic counterpart (a so called wrapper) in the Web Services, whose goal is to track the process execution. For example, when a **Supplier** physically sends supplies to a **Warehouse**, we assume that the person responsible for assembling the supply clicks on a "sent" button on her PC that saves down the shipping note. On the other hand, the person receiving the physical supply clicks on a "received" button on her PC entering in the data shown on the shipping note. Thus we do not make any distinction between electronic and non-electronic activities.

The FoodShop business process

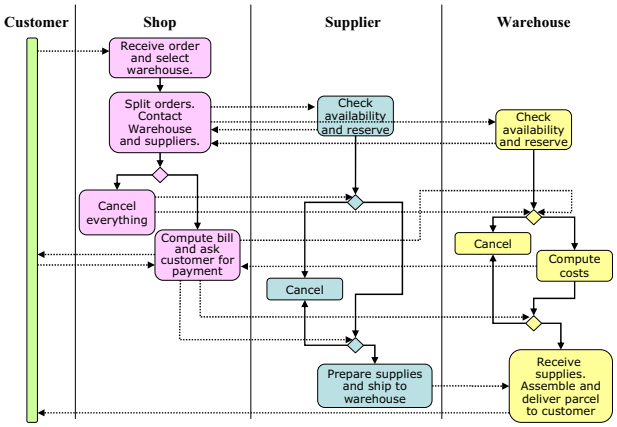

Figure 1: Abstract view of the FoodShop business process.

Figure 1 depicts a high-level view of the business process (some of the details described below are not explicitly shown for the sake of readability). When a customer places an order, the **Shop** service first selects the **Warehouse** that is closest to the customer's address, and that will thus take part in process execution. Ordered items are then split into two categories: perishable (they cannot be stocked, so the warehouse will have order them directly) and unperishable (the warehouse should have them in stock). Perishable items are handled directly by the **Shop**, while unperishable items are handled by the **Warehouse**, although all of them are eventually collected by the **Warehouse** in order to prepare the parcel for the customer. At this point the **Shop** checks whether the ordered items are available, either in the **Warehouse** or from the **Suppliers** (we have not considered items exchanges among different warehouses, in order not to make the example too complicated). If they are, they are temporarily reserved in order to avoid conflicts between several orders. Once the **Shop** receives all the answers on item availability, it can decide whether to give up with the order (again, in order to keep things simple, this happens whenever there is at least one unavailable item) or to proceed. In the former case, all item reservations are canceled and the business process ends. If the order goes on, the **Shop** computes the total cost (items plus shipping) with the aid of the **Warehouse** that provides the shipping costs depending on its distance from the customer location and the size of the order. Then it sends the bill to the customer, that can decide whether to pay or not. If the customer does not pay, all item reservations are canceled and, again, the business process terminates. If the customer pays, then all item reservations are confirmed and all the **Suppliers** (in case of perishable or out-of-stock items) are asked to send their goods to the **Warehouse**. The **Warehouse** will then assemble a package and send it to the customer.

Ws-DIAMOND architecture

Since the overall goal of achieving self healing behaviour is very ambitions, we started by defining in a precise way the requirements we wanted to address in the project and specifically in the first phase.

First of all, we concentrated on the first goal of the project, that is the design and development of a platform for supporting the self healing execution of Web Services. This means that we are concentrating on the problems that occur at run time, while the design issues will be faced in a second phase of the project. A second very general consideration is that we are not considering the issue of debugging a service; in other words we assume that code has been debugged. This led us to defining:

- The types of faults that can occur and we want to diagnose, that is:
  - functional faults and specifically semantic data errors (such as wrong data exchanges, wring data in database, wrong inputs from user, ..);
  - quality of service faults (e.g., delays, quality of data).

  In this paper we will focus on the former.
- The types of observations/tests that can be available to the diagnostic process:
  - alarms raised by services during their executions;
  - data exchanged by services;
  - internal data to a service (we will return later on privacy issues).
- The types of repair/recovery actions that can be performed, such as compensating, re-doing, replacing activities.

In this first phase of the project, moreover we decided to concentrate on orchestrated services (i.e., the case where there is an service orchestrating sub-services, see (Peltz, 2003), even if some of the solutions we are proposing already take into account the case of choreographed services (Peltz, 2003).

The main achievements in the first phase of the project are the following:

- We proposed a Semantic Web Service definition language which includes features that are needed o support the diagnostic process (e.g., observability of some parameters); moreover we started to analyze how these semantic annotations can be learned from service execution logs.
- We extended Web Service execution environments to include features which are useful to support the diagnostic/fault recovery process. In particular, an architecture supporting self-healing service execution has been defined. The architecture provides support for associating a diagnostic service with each service (see below), for gathering observations about service execution (e.g., data exchanged between services) and provides a set of recovery and repair actions. The architecture also includes a monitoring service aimed at identifying Quality of Service and communication problems, and a repair module aimed at supporting the execution of recovery plans on the basis of the diagnostic information.
- We characterized diagnosis and repair for Web Services. In particular we defined a catalogue of faults and possible observations and we proposed an architecture for the surveillance platform (diagnostic service, see below). The core of the platform is a diagnostic problem solver and thus we proposed algorithms for performing the diagnostic process, focusing the attention on functional faults. In particular, the diagnostic architecture is a decentralized one where a diagnoser is associated with each individual service; a supervisor is then associated to the orchestrator service or to the process owner. We defined a communication protocol between local diagnosers and the supervisor, assuming that no knowledge about the internal mechanisms of a Web Service is disclosed by its local diagnoser. A global diagnosis can be computed by the supervisor after exchanging information with local diagnosers (invoking only those that are relevant to solve the specific problem under analysis). The correctness of the algorithms has been proved formally.
- Repair has been then characterized as a planning problem, where the goal is to build the plan of the recovery actions to be performed in order to recovery from errors. The actions are those supported by the execution environment discussed above and involve backtracking the execution of some services, compensating some of the actions that were performed, re-doing activities or replacing faulty activities (or services) with other activities (services).

This led to a set of architectural choices that will be introduced in the next sections.

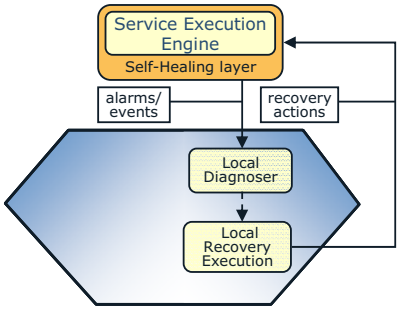

Figure 2: Web Service's DIAMOND

DIAMONDS and Self Healing layer

A DIAMOND is associated with each service and with the orchestrator and is in charge of the enhanced service execution and monitoring, diagnosis and recovery planning and execution. The DIAMOND associated with a basic service is depicted in Figure 2.

The self-healing layer is the set of extensions to the Service Execution Engine that has been designed in the project and that enables monitoring, diagnosis and repair (see below).

The DIAMOND includes diagnosis and repair:

- Alarms and events generated by the service go to the DIAMOND (to Local Diagnoser)
- The Local Diagnoser owns privately the model of the service and is in charge of explaining alarms (events) by either
  - Explaining them with internal faults
  - Blaming other services (from which inputs have been received) as the cause of the problem

  See below for the interaction between Local and Global Diagnosers.
- The Local Recovery Execution module receives recovery actions to be performed from the Global Recovery Planner (see below). It is also admitted that repair actions are selected by the Local Recovery Execution.
- Recovery actions are passed to the Self-healing layer

The DIAMOND associated with the orchestrator is made of two parts (see Figure 3):

- A Global Diagnoser and Recovery Planner (left part)
- A Local Diagnoser and Local Recovery Execution module (right part,)

The latter is meaningful as an orchestrated service may in turn be a sub-service of a higher-level composed service.

The Global Diagnoser interacts with Local Diagnosers to compute a global diagnosis; it does not have access to local models. The diagnosis is computed in a decentralized way. The Recovery Planner operates sequentially after a global diagnosis has been computed. It generates a plan for recovery and passes it to Local Recovery Execution modules.

Self-healing layer

The platform for service execution supports the following features:

- associating a diagnostic service with each service;
- gathering observations about service execution (e.g., data exchanged between services), which are input to diagnosis;
- providing a monitoring service aimed at identifying problems during execution, including Quality of Service and communication problems;
- providing a set of recovery and repair actions (that can be exploited by the repair planner);
- providing a repair module aimed at supporting the execution of recovery plans.

Such a layer can thus be seen as a set of new functionalities on top of a service execution platform (engine)

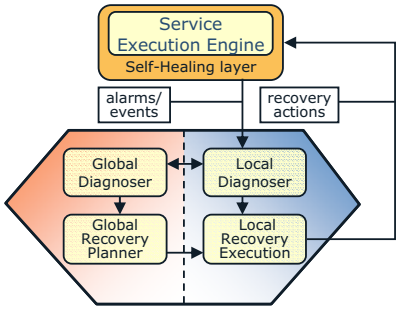

Figure 3: Orchestrator's DIAMOND

Diagnostic approach

The approach we adopted for diagnosis is a model-based one and is based on a decentralized paradigm:

We associate with each basic service a local diagnoser, owning a description of how the service is supposed to work (a model); the role of local diagnosers is to provide the global diagnoser with the information needed in order to identify the causes of a global failure.

We provide a global diagnoser which is able to invoke local diagnosers and relate the pieces of 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.

We choose to adopt such an approach because this enables to recursively partition Web Services into aggregations of sub-services, hiding the details of the aggregation to higher-level services. This is in accordance with the privacy principle which allows designing 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 know the interfaces of local services and share a protocol with local diagnosers. This will mean, in particular, that the model of a service is private to its local diagnoser and needs not be made visible to other local diagnosers or to the global one.

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 the pattern described in the following:

During service execution, each local diagnoser receives information from the self healing layer (monitoring 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.

When a local diagnoser receives an alarm message (denoting a problem in the execution of a service), it starts reasoning about the problem in order to identify its causes, which may be internal to the Web Service or external (erroneous inputs from other services). The diagnoser can do this by exploiting the logged messages. 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 and relating the different answers, in order to reach one or more global candidate diagnoses that are consistent with reasoning performed by local diagnosers.

According to the approach discussed above, each local diagnoser needs to have a model of the service in its care. 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 a complex service, in particular, 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. Thus fault localization is performed at the level of the activities of a service.

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 then to find activities that can be responsible for the alarm, performing discrimination to the purpose of selecting the appropriate recovery action. When an alarm is raised in a Web Service W_i, its local diagnoser A_i receives it. A_i must explain it, Each explanation may ascribe the malfunction to failed internal activities and/or abnormal inputs. It may also make predictions of additional output values, which can be exploited by the global diagnoser in order to validate or reject the hypothesis. When the global diagnoser receives a local explanation from a local diagnoser A_i, it proceeds as follows:

- If a Web Service W_j has been blamed of incorrect outputs, then the global diagnoser 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.
- If a fault hypothesis by Ai has provided additional predictions on output values sent to a Web Service W_k, then the global diagnoser 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.

The global diagnoser sends hypotheses to local diagnosers for explanation and/or validation. Local diagnosers explain blames and validate symptoms by providing extensions to partial assignments that assign values to relevant unassigned variables. In particular the global diagnoser exploits a strategy for invoking as less local diagnosers as possible, excluding those which would not contribute to the computation of an overall diagnosis.

Details about the strategies adopted by the global diagnoser, about the communication protocol between global and local diagnosers and about local diagnosers can be found in [Console et al., 2007], where also properties about the correctness of the adopted algorithms are proved.

Example on the FoodShop business process

Let us anlyse how the approach work on the FoodShop example presented above. The faults we consider within the FoodShop business process are (according to the focus of the current project work,) those that:

- affect the correctness of data exchanged between services or between services and customer
- can be detected by looking only at one instance of the business process.

Accordingly, the diagnosis we carry out focuses on finding, within the faulty instance, the last point in execution where data can be safely considered correct. Repair focuses then on planning and executing an alternative portion of the business process that tries to achieve the same goals as the first one without incurring in the same problem. Achieving this my entail substituting a partner (e.g. turning to a different supplier), undoing and redoing some actions (e.g. sending back supplies and receiving correct one), carrying out some compensation activity (e.g. restoring the previous item quantities in the Warehouse database if the wrong quantities were decreased due to a misplaced order), or carrying out some activity-specific repair action (e.g. realigning the Shop database to the Supplier one if it contains obsolete item codes).

Some examples of the faults that we consider within the FoodShop business process are:

- The Shop incorrectly matches ordered items to their codes; since most of the order process works on codes rather than on names this results in the Warehouse not being able to assemble the customer parcel.
- The Shop selects the wrong Warehouse; this can have as effect a higher delivery cost for the customer.
- The Shop does not contact all suppliers, either when reserving (then some items will be missing in the Warehouse when assembling the parcel) or when cancelling or confirming the reservation (these will raise a timeout in the Supplier).
- The Warehouse or the Supplier make a mistake in answering on the availability of the items, so that a non-

available item is declared available; this results in the Warehouse not being able to assemble the parcel.

- The Supplier makes a mistake in saving the supply order, or in physically preparing the supply package; again, the consequence is that the Warehouse will not be able to assemble the customer parcel.

It is easy to see from this list that many faults actually have the same symptoms (namely, that the Warehouse cannot assemble the customer parcel).

We will now see a diagnosis example that shows how some of these faults can be diagnosed by automated reasoning on a distributed model of the business process that includes data dependencies.

Let us assume that the Warehouse service in the example raises an exception because, when assembling the parcel for the customer, the supplies do not correspond to the order list. The Warehouse local diagnoser, from information on data dependencies, can automatically derive that the reason can either be that the supplies sent by one Supplier were wrong, or that the item named in the order list from the Shop were the incorrect ones.

The Global diagnoser, upon receiving this information from the Warehouse local diagnoser, asks to the Supplier local diagnoser to evaluate the hypothesis that it sent the wrong items. Based on the Supplier model, this local diagnoser computes that this can be happened for three reasons: a) when sending the package with the Supplies to the warehouse the wrong items were inserted in it; b) when receiving the order, the wrong codes were written down; c) the initial item codes were wrong. It also computes that in case a) the codes on the shipping note sent to the Warehouse should be correct and should not match the supply contents, while in case b) and c) the codes on the shipping note are also incorrect, and the shipping note should match the contents of the supply.

At this point, the Global diagnoser invokes the Shop local diagnoser in order to evaluate the hypothesis that the Shop may be the source of the problem. The Shop local diagnoser computes that the source may indeed be in the Shop, and in particular that the Shop might have made an error in matching the ordered items to their codes (possibly because of a database error). The Shop local diagnoser also computes that, however, the item names cannot be wrong, because they were provided by the user.

Eventually, the Shop local diagnoser computes that if it made this error, then the item codes in the order list should be wrong. Given these results, the Supervisor invokes again the Warehouse local diagnoser for analyzing the new information, namely to check the predicted consequences of the different fault hypothesis. The Warehouse thus checks the order list against the shipping notes and the following situations can happen: a) The codes in the order list match those in the shipping note, and the shipping note matches the contents of the supply. This means that the responsible for the problem is the Shop. b) The codes in the order list match those on the shipping note, but the shipping note does not match the contents of the supply. Then the responsible for the problem is the Supplier, and the problem occurred while assembling the supply. c) The codes in the order list do not match those on the shipping note. In this case the responsible for the problem is the Supplier, and the problem occurred while writing down the codes at the initial stages.

Repair planning

A composition of Web Services can be considered as a workflow of activities. An invocation of a web service is an execution of an activity which has some pre-defined input and output objects. We also say that output objects are the affected objects of an activity. The input of the repair process is a set of activities and objects which are considered as faulty. All other activities are considered to be correct. However, some correct activities might have produced wrong outputs (i.e. suspect objects) because of wrong inputs or because their invocation was mistakenly caused by a wrong branching of the workflow. Repairing this composition of faulty activities and suspect objects is the task of repair.

Let us assume that in the FoodShop example described above the “split orders” activity of the Shop is faulty (i.e. the single fault diagnosis submitted to the repair process). As mentioned above the wrong product codes for some product names were determined which was discovered when assembling the parcel. Since this activity is performed by a human, there is hope that this failure is intermittent. Consequently, the repair of this faulty activity is just a REDO. Basically all subsequent activities of the “split orders” (Shop) activity in the Shop, Suppliers, and Warehouse must be redone. However, it is not clear which one. This indeterminism depends on the unknown branching of the choosing blocks and on the possibility of fault masking, i.e. although some input objects are faulty some of the output objects could be correct. E.g. some orders to suppliers could have been correct. This can be determined after the REDO of the “split orders” (Shop) activity and by comparing the newly generated output objects with the old ones. Furthermore repair planning has to take into considerations when a compensation action should be applied. Roughly speaking, compensation actions restore the state of objects to a state which was in effect before then objects were affected by an activity. E.g. compensating the effects of “prepare supplies and ship to the warehouse” (Supplier) implies that goods are returned to the supplier. If a compensation action is applied depends on the plan generation. In our scenario it might be more cost efficient to dump the unnecessary items in the warehouse than to send them back.

Repairing a compositional, conversationally complex Web Service consists of plan generation and execution of (repair) actions until the original intended goals of the web service are achieved. These actions also include DO and REDO of activities of the original workflow. Since we have to deal with indeterminism, the generated repair plan is conditional. Reasoning is based mostly on the analysis of the dependencies between the input and output sets of

activities, their preceding relations and dependencies on goal objects.

We propose the following algorithm (repair procedure):

1. Represent the description of a workflow process in terms and models of the planning system;
2. Generate a repair plan using a planner;
3. If the repair plan does not contain any repair actions - exit;
4. Execute first repair action in the set;
5. Evaluate the new state of the workflow instance;
6. React on the new information received by the workflow;

If old plan can be continued go to Step 3 else go to Step 2.

Note that we are interested in generating optimal and save plans. However, it is also worth to consider unsafe plans (in case safe plans do not exist) if the expected cost of repairing is lower than doing nothing.

There are different planning systems that use such planning languages as A^c, PDDL, K-language and their dialects (Eiter et al 03., Huy Tu et al., 04). In most cases knowledge is separated into two groups. First, background knowledge comprises static information that cannot be changed during planning. In our case this background knowledge includes the set of goal objects, input and output parameters of activities, preceding relations between activities. The second part includes the planning code with fluents, actions, and causal/execution rules. This part must be independent of a description of a specific workflow. It includes a general specification applicable to any workflow that is correctly described in the background knowledge.

Depending on the problem there may be many repair plans with different structures. As we see from the algorithm, the most important information that can be obtained from the repair plan is which repair action must be applied first. The rest of the repair plan can be changed or removed, so, actually, only the decision which of the repair actions must be applied first is important.

The evaluation of the new state is needed to decide which changes to the repair plan must be applied. This evaluation must answer the following questions after executing a repair action:

- Was the applied repair action successfully or not?
- What are the new values of the objects in the workflow state?
- What was the exit status of the activity after repair (is there any exit code that shows a fault again)?
- Is there any new information about activities which are long-running transactions?

The analysis after action execution must also consider the results produced and compare them with the states of the objects of the original faulty workflow. Since we do not know the internal behaviour of an activity, its effects may not depend on all its input data. Even if all input data was faulty, it may happen that the effects are correct.

Such analysis can reduce the amount of suspect objects and lead to the need for re-planning since more efficient plans are possible. Note that compared to the original workflow the order of executing activities may change because some activities already produced correct results or because we can easily imagine cases where computing the correct values for choosing blocks is necessary and preferred in order to find out which activities are important to produce correct goal objects. Also, we have to consider new data, obtained from long-time running activities. Completed long-running activity that used suspect objects must be compensated, if needed.

The proposed solution is intended to resolve situations in the fully orchestrated workflows. Moving towards choreography, it is proposed to use the same algorithm for repair plan generation for each workflow in the composition but add additional communication protocols for information exchange.

As depicted in Figure 3 the Global Recovery Planner interacts with the Local Recovery Execution of all services. The task of the Global Recovery planner is to mediate between the Local Recovery Execution services in order to optimize the overall repair activities. Repair plans for each of the local workflows (e.g. Shop, Supplier, and Warehouse) can be generated in a separated session of repair planning. The reasoning process in each of these sessions is performed locally using the same techniques and algorithms. The only difference to the orchestration case is that there must be a communication protocol between sessions. As soon as activities in different workflows share objects as well as invoke and wait for results of each other, this information must be shared also between sessions of repair planning. Note that we not assume that sessions know about the workflow structures of each other. The exchanged information includes:

- the objects which are shared between workflows;
- the states of these input and output objects regarding the correctness;
- the activities which are activated in the partner workflow;
- the existence of local goal objects of the partner workflow and the inputs needed by the partner workflow to achieve these local goals.

By exploiting this information we can propagate information about goals, their needed input objects, and the states of objects from one workflow to another.

In our example, the repair plans for Shop, Supplier and Warehouse workflows will be generated in separated sessions. The Warehouse informs the Suppliers about a local goal, i.e. “deliver the correct parcel to the customer” which depends on correct inputs from the suppliers, e.g. correct items. The correctness of these items depends on the correctness of the input provided by the “split orders” (Shop) activity. Therefore, the Shop workflow has to REDO this “split orders” (Shop) activity. Consequently, goal dependencies are propagated through the composite workflow. Conversely, the communication process has to inform the separated repair planning sessions about the correctness of objects.

Discussion and Future work

The sections above presented the results achieved in the first phase of the WS-DIAMOND project. In this phase we concentrated on some specific problems, making assumptions in order to constrain the problem. Such assumptions are being progressively relaxed or removed in the second part of the project; this approach is allowing us to manage the complexity of the problem and of the task. In the following we analyse the assumptions we made, the way we plan to remove or relax them and the way this will impact the architecture presented in the previous sections.

A first major assumption we made is that we are considering orchestrated services. We are currently extending the approach to deal with choreographed services. This actually does not impact the overall diagnostic architecture (which is not influenced by this distinction except that the global diagnoser must be associated with the owner of the complex process rather than with the orchestrator). On the other hand repair and the self healing layer are being modified.

In the current approach we are also assuming that all services are WS-DIAMOND enables, that is that they have an associated diagnostic service. Such an assumption is being removed and we are considering also the case where some services are black boxes.

Another assumption in the first phase is that we are concentrating on functional errors, while in the second phase we are considering also problems related to the Quality of Service. We will also extend the range of functional faults we are considering. This means, in particular, that the self-healing layer discussed in the previous section is being modified to include also modules for monitoring quality of service parameters.

As regards repair, we worked with a limited set of repair primitives and we are currently extending this set to include further alternatives to be considered during repair planning. On the other hand, in the project we do not expect to remove the general assumption that diagnosis and repair are performed sequentially. The issue of interleaving repair/recovery with (Fridrich, 93) which is a very important one in diagnostic problem solving, will be a topic for future investigations outside the project.

Finally in this phase we assumed that the model of service activities which is needed by its local diagnoser is hand made. However, the dependencies that are needed can be derived from the service description and indeed we are currently investigating how the model can be produced a partially automated way. Moreover we explored how semantic annotations on service properties (e.g., properties of exchanged information) which can be useful as observations to the diagnostic process can be learned from service logs.

Finally, in the second phase of the project we will develop a framework for the analysis of diagnosability and repairability of complex services, starting from previous work and experience in the diagnosticability analysis for artefacts.

References

AI Magazine, Special Issue on Model-based Diagnosis, AAAI, Winter 2003.

ISTAG 2004: Information Society Technologies Advisory Group WG: Grand Challenges in the Evolution of the Information Society, Report (July 2004), ftp://ftp.cordis.lu/pub/ist/docs /istag_draft_report_grand_challenges_wahlster_06_07_04.pdf.

V. Brusoni, L. Console, D. Theseider Duprè, P. Terenziani: A Spectrum of definitions for temporal model-based diagnosis; in *Artificial Intelligence*, Vol 102, no 1, June 1998, pp 39-79.

L. Console, O. Dressler: Model-based diagnosis in the real world: lessons learned and challenges remaining, in *Proc. of the 16th Int. Joint Conf. on Artificial Intelligence. IJCAI 99*, Stockholm, Sweden,1999, pp.1393-1400.

L. Console, C. Picardi, D. Theseider Dupre'. A Framework for Decentralized Qualitative Model-Based Diagnosis. In: *Proc. of the 20th Int. Joint Conf. on Artificial Intelligence. IJCAI-07*. Hyderabad, India, 2007. pp. 286-291. (Preliminary paper in *Proc. 17th Int Work.p on Principles of Diagnosis DX'06*, Penaranda de Duero, Spain, 2006.

T.Eiter, W. Faber, N.Leone: A logic programming approach to knowledge-state planning, II: The DlvK system. *Artificial Intelligence, vol.144,* p.157-211, 2003.

G. Friedrich. Model-based diagnosis and repair. *AI Communications,* Vol. 6(3/4), Sept./Dec. 1993.

I. Grosclaude. Model-based monitoring of component-based software systems, *Proc. 15th Int. Work. on Principles of Diagnosis DX'04*, Carcassonne, France, pp. 155-160, 2004.

W. Hamscher, L. Console, J. de Kleer, *Readings in Model-Based Diagnosis*, Morgan Kaufmann, 1992.

P. Huy Tu, T. Cao Son: Reasoning and Planning with Sensing Actions, Incomplete Information, and Static Casual Laws using Answer Set Programming. *Logic Programming and Nonmonotonic Reasoning,* Springer, pp. 261-274, 2004.

W. Mayer, M. Stumptner, F. Wotawa. Model-Based Debugging or How to Diagnose Programs Automatically, *Proc. IEA/AIE 2002*, Cairns, Australia, Springer LNAI, pp. 746-757, 2002

W. Mayer, M. Stumptner. Debugging Failures in Web Services Coordination, in *Proc. 17th Int. Work. on Principles of Diagnosis DX'06*, Penaranda de Duero, Spain, pp. 171-178, 2006.

B. Peischl, J. Weber, F. Wotava: Runtime fault detection and localization in component-oriented software systems, in *Proc. 17th Int Work. on Principles of Diagnosis DX'06*, Penaranda de Duero, Spain, pp. 195-203, 2006.

Peltz C., Web Service Orchestration and Choreography, *IEEE Computer*, vol. 36, no. 10, 46-52, October 2003.

R. Reiter. A Theory of Diagnosis from First Principles. *Artificial Intelligence*, Vol. 32 (1), 1987, pp. 57-95.

M. Stumptner, F. Wotawa: Modeling Java Programs for Diagnosis, In *Proc. ECAI 2000*, pp. 171-175.