9 WS-DIAMOND: Web Services—DIAgnosability, MONitoring, and Diagnosis

Luca Console, Danilo Ardagna, Liliana Ardissono, Stefano Bocconi, Cinzia Cappiello, Marie Odile Cordier, Philippe Dague, Khalil Drira, Johann Eder, Gerhard Friedrich, Mariagrazia Fugini, Roberto Furnari, Anna Goy, Karim Guennoun, A. Hess, Volodymyr Ivanchenko, Xavier Le Guillou, Marek Lehmann, J. Mangler, Yingmin Li, Tarek Melliti, Stefano Modafferi, Enrico Mussi, Yannick Pencolé, Giovanna Petrone, Barbara Pernici, Claudia Picardi, Xavier Pucel, S. Robin, L. Rozé, Marino Segnan, Amirreza Tahamtan, Annette Ten Tejie, D. Theseider Dupré, Louise Travé-Massuyès, Frank Van Harmelen, Thierry Vidal, and Audine Subias

9.1 Introduction

The goal of this chapter is to present the guidelines and achievements of the WS-DIAMOND Step Project, funded by the EU Commission under the FET-Open Framework, grant IST-516933. It started in September 2005 and will run until February 2008.

The project aims at making a step in the direction of self-healing Web Services. In the framework of the book, WS-DIAMOND deals with service architectures, mechanisms to achieve integration and interoperation among services, and engineering approaches for developing dependable services (Console et al. 2007a). In particular, it addresses two very different issues concerning self-healing capabilities:

1. 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 nonfunctional, such as a loss of service quality. Self-healing can be performed at the level of the single service and, at a more global level, with support to identify critical misbehavior of groups of services and to provide Web Services with reaction mechanism to global-level failures. 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, and "conversationally complex" means that during service provision a Web Service needs to carry out a complex interaction with the consumer application in which several conversational turns are exchanged between them.
2. To devise guidelines and support tools for designing services in such a way that they can be easily diagnosed and recovered during their execution. Moreover, software tools to support this design process will be developed.

According to the principle of service integration in the NESSI road map (see chapter 1), WS-DIAMOND aims at developing a service integration platform that provides tools and methods for configuration and composition of self-healing units able to self-diagnose

and self-repair. Moreover, WS-DIAMOND develops a platform for dynamic reconfiguration where software can be modified without stopping execution, thus meeting the demands for high availability. In fact, as computing facilities are increasing at a very rapid pace, presenting new forms of interaction, such as portable and mobile devices, home and business intelligent appliances, new technology shows a tremendous potential for the development of complex services to support human activities (e.g., work, health care, communities of workers, leisure, etc., possibly in a mobile context), in particular creating networks of cooperating services (Web Services).

According to the Services Research road map (see chapter 1), WS-DIAMOND addresses mainly the service composition and management planes and w.r.t. service characteristics that cut across the two planes; it also tackles nonfunctional service properties and quality of service (QoS).

The availability of reliable self-diagnostic and repairable services will be critical to enable activities to be carried on in dynamically reconfigurable runtime architectures, as required in the Service Foundation Grand Challenges, as well as for infrastructure support for process integration and for QoS-aware service composition. In the same way as today, we cannot work without access to corporate or external knowledge sources, on a broader prospective. Business-to-business and business-to-customer activities will be enriched with advanced capabilities and functionalities, thus enhancing business integration. Complex services will be made available through networks of cooperating services (Web Services). These networks are open, and may dynamically accommodate the addition and removal of new services and the disappearance of other ones. The availability and reliability of services (especially complex ones) will be of paramount importance. Indeed, the reliability and availability of software, together with the possibility of creating self-healing software, is recognized as one of the major challenges for IST research in the next years, according to ISTAG 2004.[1]

WS-DIAMOND studies and develops methodologies and solutions for the creation of self-healing Web Services able to detect anomalous situations, which may manifest as the inability to provide a service or to fulfill QoS requirements, and to recover from these situations (e.g., by rearranging or reconfiguring the network of services). WS-DIAMOND will also provide methodologies for the designer of services, supporting service design and service execution mechanisms that guarantee diagnosability and reparability of runtime failures (e.g., thanks to the availability of sets of observables or of sets of redundancies in the services or alternatives in the execution strategies). These goals are achieved by combining results from different research areas, as will be outlined in section 9.2.

WS-DIAMOND research concerns a number of "grand challenges," as described within the Service Research road map, at all levels: in the service foundations WS-DIAMOND studies dynamic connectivity capabilities, based on service discovery; at the service composition level, QoS-aware service composition is considered; in the service management level,

self-healing services are developed as a main goal of the project; finally, at the service design and development level, design principles for self-healability are defined.

This chapter describes the results achieved in the first phase of the WS-DIAMOND project and is organized as follows. After framing the project in the literature context, section 9.3 introduces an application scenario. Section 9.4 presents the conceptual framework and architecture we developed for the platform supporting self-healing service execution, considering the requirements addressed in the first year of the project (namely, the design and development of a platform supporting self-healing execution of Web Services). Sections 9.5, 9.6, and 9.7 detail the three main components of such a framework: the self-healing layer supporting service execution (section 9.5), the diagnostic algorithm (section 9.6), and the approach to repair planning and execution, with a sample scenario (section 9.7). Section 9.8 concludes the chapter, discussing current and envisioned directions of research in the project.

9.2   Related Work

An area which considered self-healing systems is autonomic computing, proposed in the literature (Kephart and Chess 2003) to create systems capable of self-management, in particular, systems with self-configuration, self-optimization, self-healing, and self-protection properties. In the service area Papazoglou and Georgakopoulos (2003) advocate the need for extending the service-oriented approach in regard to service management. Several approaches have been proposed in the composed services and workflows systems areas to provide adaptive mechanisms, in particular for such process-based systems. In Mosaic (Benatallah et al. 2006) a framework is proposed for modeling, analyzing, and managing service models, focusing on systems design. Meteor-S (Cardoso and Sheth 2003) and other semantic-based approaches (e.g., WSMO) explicitly define the process goal as the basis both for service discovery and for composition. In WSMO, a goal-based framework is proposed to select, integrate, and execute semantic Web Services. No separation between design and runtime phases is proposed, nor is specific support to design adaptivity addressed. However, whereas goal-based approaches open up the possibility of deriving service compositions at runtime, their applicability in open service-based applications is limited by the amount of knowledge available in the service. In Meteor-S, semantically annotated services are selected, focusing on the flexible composition of the process and also on QoS properties, but runtime adaptivity to react to changes and failures is not considered. More recently, autonomic Web processes have been discussed in Verma and Sheth (2005), where a general framework for supporting self-* properties in composed Web Services is advocated. In the workflow area, the work of Hamadi and Benatallah (2004) presents SARN (Self-Adaptive Recovery Net), a Petri Net-based model for specifying exceptional behavior in workflow systems at design time. Following a failure, the standard behavior of the

workflow is suspended and a recovery transition is enabled and fired. Hamadi and Bena-tallah (2004) also specify a set of recovery policies that can be applied in SARN both on single tasks and on workflow regions. The work in Eder et al. (1996) presents WAMO, which widely supports recovery actions in composed services. WAMO enables a workflow designer to easily model complex business processes in a simple and straightforward manner. The control structure is enriched by transactional features.

Other proposals tackle single aspects of adaptation. In business process optimization approaches, the process specification is provided and the best set of services is selected at runtime by solving an optimization problem (Zeng et al. 2004). In a similar way, in Grid systems, applications are modeled as high-level scientific workflows where resources are selected at runtime in order to minimize, for example, workflow execution time or cost. Even if runtime reoptimization is performed and provides a basic adaptation mechanism, user context changes and self-healing are not addressed.

In the SeCSe project (SeCSe Team 2007), self-healing is addressed in terms of QoS optimization and reconfiguration. Dynamic binding is supported through dynamic service selection and through adapters, based on optimization and reoptimization of QoS (Di Penta et al. 2006). Reconfiguration, as described in SCENE (Colombo et al. 2006), is based on the definition of rules for activating repair actions such as alternative services, rebinding, and process termination.

Self-healing processes in the Dynamo approach (Baresi and Guinea 2007) combine monitoring and reactions associated with pre- and post-conditions for activities.

The approach proposed in WS-DIAMOND is based on the infrastructure provided by PAWS (Processes with Adaptive Web Services) (Ardagna et al. 2007), a framework and tool kit for designing adaptable services, defining QoS agreements, and optimizing service selection, substitution, and adaptation at runtime. In WS-DIAMOND, such an approach has been coupled with the ability to automatically derive a repair plan based on repair actions provided by the PAWS self-healing interface. The strategy for deriving repair plans has its basis on the WAMO (Eder and Liebhart 1996) approach, where workflow transactions are supported by repair plans which have the goal of minimizing the impact of repair on the executed activities. To support such an approach, the definition of compensation operations, in addition to retry and substitution of service operations, as well as execution sequences for such actions, have been defined in WS-DIAMOND.

QoS has been the topic of several researches efforts crossing distinct communities, in particular the Web and Web Service community (Keller and Heiko 2003; Ran 2003) and the networking and internetworking communities. The highly fluctuating radio channel conditions, jointly with the heavy resource request deriving from multimedia applications, force thinking in terms of adaptive QoS or soft QoS, the opposite of the traditional hard QoS or static QoS idea.

In Marchetti et al. (2003) the representation of quality aspects in Web Services is discussed in the context of multichannel information systems. Their proposal includes the

concept of service provider communities as a basis for a homogeneous definition of quality, and a classification of quality parameters, based on negotiable/nonnegotiable parameters and on provider/user perspectives. Zeng et al. (2004) provides a thorough analysis of quality evaluation in the context of adaptive composed Web Services, focusing on price, duration, reputation, success rate, and availability as quality parameters.

An important need which is emerging for highly distributed processes is execution monitoring. Mechanisms for automatically augmenting processes with monitoring functionalities have been proposed in Spanoudakis et al. (2005), considering that traditional monitoring tools cannot be applied to a heterogeneous environment such as the Internet.

Last but not least, a methodological approach is needed to design all aspects of such systems, focusing on exception handling and compensation mechanisms (Alonso and Pautasso 2003).

In the MAIS project, adaptive Web-based process execution has been developed based on flexible services, considering service similarity and QoS, and runtime service substitution mechanisms (Pernici 2006). Extended Petri nets have been used as a modeling technique and a basis for building analysis tools in process modeling (van der Aalst et al. 2003). However, little attention is paid to conforming to patterns of interactions between organizations and to providing inherent flexibility and fault tolerance in process execution.

Basic recovery is being proposed in the literature with retry and substitution operations. In Web-based process evolution, recovery has been proposed in Erradi et al. (2006), based on adaptive service composition and service substitution. Research work in Grid services is based on retry and substitution operations, but a more comprehensive approach to service repair is advocated in Candea et al. (2006), considering also the context of execution.

However, the above-mentioned approaches focus only on a direct repair of failed services, based on monitoring of failures; the link between failures and the causes of failures (faults) is not considered as a basis for repair strategies.

Current standards for Web Service markup describe operational behavior and syntactic interfaces of services, but they do not describe the semantics of the operations performed by the Web Services. Such a description is required to enable diagnosable and self-healing Web Services.

Research in Meteor-S has proposed a template-based process representation to enhance flexibility of processes through a dynamic selection of component services, based on service semantic similarity and QoS properties (Patil et al. 2004). The Semantic Web community is working toward markup languages for this Web Service semantics. The most prominent initiatives in this area are OWL-S (Martin et al. 2005), developed by a joint US/EU consortium, and the EU-based WSMO initiative (Web Service Modeling Ontology), developed by the EU-funded DIP consortium (Fensel and Bussler 2002).

These efforts, joined with proposals about discovery and dynamic composition (Spanoudakis et al. 2005; Colombo et al. 2006), aim at supporting retrieval and automatic composition of services (for example, the EU-funded consortium on Semantically Enabled Web

Services (SWWS; swws.semanticweb.org). However, aspects of diagnosability have been largely ignored until now by these Semantic Web initiatives. In order to enable diagnosable and self-healing Web Services, we will build upon and extend the currently proposed markup languages. In particular, the current semantic service markup languages focus on functional properties of the service (the input/output behavior), but are limited with respect to nonfunctional aspects. To enable diagnosability, the existing markup languages have to be extended to deal with such nonfunctional aspects as QoS descriptions, monitoring information, and repair options in case of failure.

In the research on automated monitoring and diagnosis, through the 1990s the model-based approach emerged as extremely interesting and led to several studies, methodologies, solutions, and applications (Hamscher et al. 1992; AI Magazine 2003). The approach focused mainly on the diagnosis of artifacts (moving from electronic circuits to more complex systems, such as subsystems in automotive or aerospace domains).

A current trend of research in diagnosis is the analysis of complex systems, taking into account the dynamic and distributed nature of the systems to be diagnosed. Indeed, focus has moved progressively from static to dynamic and then time-evolving systems, where parameters or structure itself (reconfigurable systems) can vary over time, and from global to hierarchical and then distributed systems, where components can communicate with each other.

Moreover, the focus has moved from "traditional" application areas to new ones such as economic systems, software, communication networks, and distributed systems. Particularly significant with respect to this project is the application to software diagnosis in which the same basic technologies have been successfully applied to debug programs (Mateis et al. 2000; Wotowa et al. 2002) and component-based software (Grosclaude 2004; Peischl et al. 2006). The same approach defined for debugging and diagnosing software has also been applied to Web Services (Mayer and Stumptner 2006). The focus of that work is different from our approach in that it aims at debugging (and diagnosing) problems in the composition of services (orchestration), rather than diagnosing and repairing problems arising during service execution.

9.3   An Application Scenario

The application scenario used in the project is an e-commerce application, concerned with a FoodShop company that sells food products on the Web.

The company has an online shop (that does not have a physical counterpart) and several warehouses ($WH_1, \ldots, WH_n$) located in different areas that are responsible for stocking nonperishable 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 the case of perishable items that cannot be stocked, or of out-of-stock items, the FoodShop Company must interact with one or more suppliers ($SUP_1, \ldots, SUP_m$).

In the following we describe the business process from the customer order to the parcel delivery, which is 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 carried out by humans, such as the preparation of the order 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 the shipping note. On the other side, the person receiving the physical supply clicks on a ''received'' button on her PC, entering the data shown on the shipping note.

9.3.1 The FoodShop Business Process

Figure 9.1 depicts a high-level view of the business process, using a simplified UML-like representation. When a customer places an order, the Shop service selects the Warehouse that is closest to the customer's address, and that will thus take part in process execution.

Ordered items are split into two categories: perishable (they cannot be stocked, so the warehouse will have to order them directly) and nonperishable (the warehouse should have them in stock). Perishable items are handled directly by the Shop, exploiting the services of a Supplier, whereas nonperishable items are handled by the Warehouse; all of them are eventually collected by the Warehouse in order to prepare the parcel for the customer.

The Shop checks whether the ordered items are available, either in the Warehouse or from the Supplier. If they are, they are temporarily reserved in order to avoid conflicts between orders.

Once the Shop receives all the information on item availability, it can decide whether to give up on the order (to simplify, 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 proceeds, the Shop computes the total cost (items plus shipping) with the aid of the Warehouse, which 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, who 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 the cases of perishable and 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.

Figure 9.1
An abstract view of the FoodShop business process

9.4 WS-DIAMOND Architecture

In the first year of the project, we concentrated on the design and development of a platform supporting the self-healing execution of Web Services. This means that we concentrated on the runtime problems, and design issues were faced in the second phase. A second very general consideration is that we are focusing on diagnosing problems that occur at runtime and we are not considering the issue of debugging a service (we assume that code has been debugged).

This led us to the definition of the following:

1. The types of faults that can occur and that we want to diagnose:
a. Functional faults, and specifically semantic data errors (such as wrong data exchanges, wrong data in databases, wrong inputs from the users, and so on).
b. QoS faults.
In this chapter, we will focus on the first ones.

2. The types of observations/tests that can be available to the diagnostic process:

a. Alarms raised by services during their execution.

b. Data possibly exchanged by services.

c. Data internal to a service (we will return later to privacy issues).

3. The types of repair/recovery actions that can be performed, such as compensating for or redoing activities.

We also decided to concentrate on orchestrated services, although some of the proposed solutions are also valid for choreographed services.

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

· We proposed a Semantic Web Service definition language which includes features needed to support the diagnostic process (e.g., observable 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, as described in section 5.

· We characterized diagnosis and repair for Web Services. In particular, we defined a catalog of faults (Fugini and Mussi 2006) and possible observations, and proposed an architecture for the surveillance platform (diagnostic service; see above). The core of the platform is a diagnostic problem solver, and thus we proposed algorithms for performing the diagnostic process, focusing attention on functional faults. In particular, the diagnostic architecture is decentralized; a diagnoser is associated with each individual service. A supervisor is 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. The correctness of the algorithms has been proved formally.

· We defined repair as a planning problem, where the goal is to build the plan of the recovery actions to be performed to achieve recovery from errors. The actions are those supported by the execution environment and involve backtracking the execution of some services, compensating for some of the actions that were performed, redoing activities, or replacing faulty activities (or services) with other activities (services).
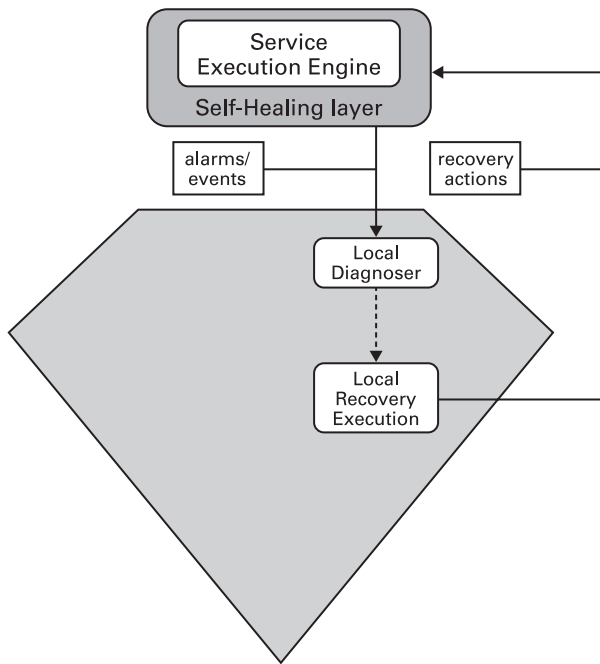
9.4.1 DIAMONDS: Diagnostic Services

A diamond (in charge of the enhanced service execution and monitoring, diagnosis, and recovery planning and execution) is associated with each service and with the orchestrator (figure 9.2).
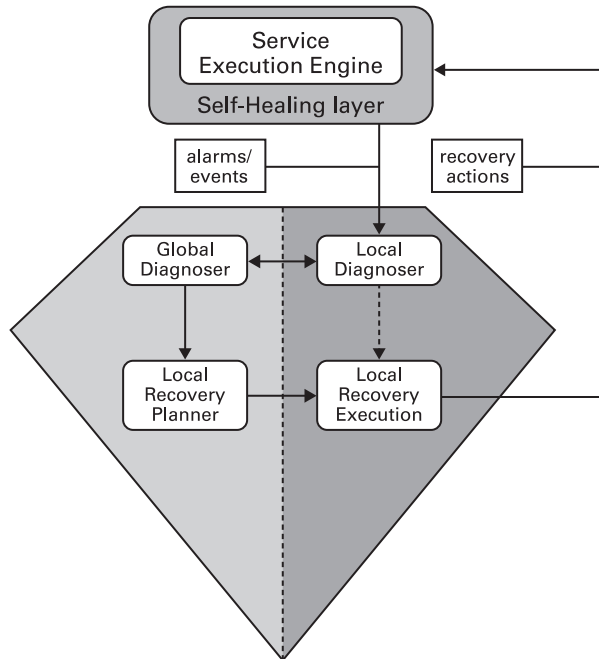
Web Service's DIAMONDS The DIAMOND associated with a basic service is depicted in figure 9.3.
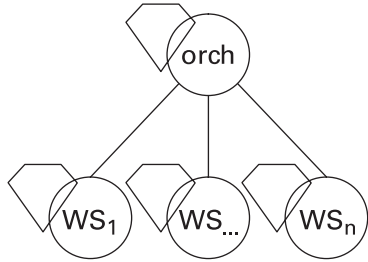
Figure 9.2
Services and DIAMONDS

Figure 9.3
Web Service's DIAMOND

Figure 9.4
Orchestrator's DIAMOND

The self-healing layer is the set of extensions to the service execution engine designed in the project, which enables monitoring, diagnosis, and repair.

The diagnosis and repair include the following events:

1. Alarms and events generated by the service go to the diamond (to local diagnoser).
2. Local diagnoser privately owns the model of the service and is in charge of explaining alarms (events) by either
a. Explaining them with internal faults.
b. Blaming other services (from which inputs have been received) as the cause of the problem.
3. The local recovery execution module receives recovery actions to be performed from the global recovery planner (see below). Repair actions can also be selected by the local diagnoser.
4. Recovery actions are passed to the self-healing layer.

Orchestrator's DIAMONDS   The diamond associated with the orchestrator is made up of two parts (see figure 9.4).
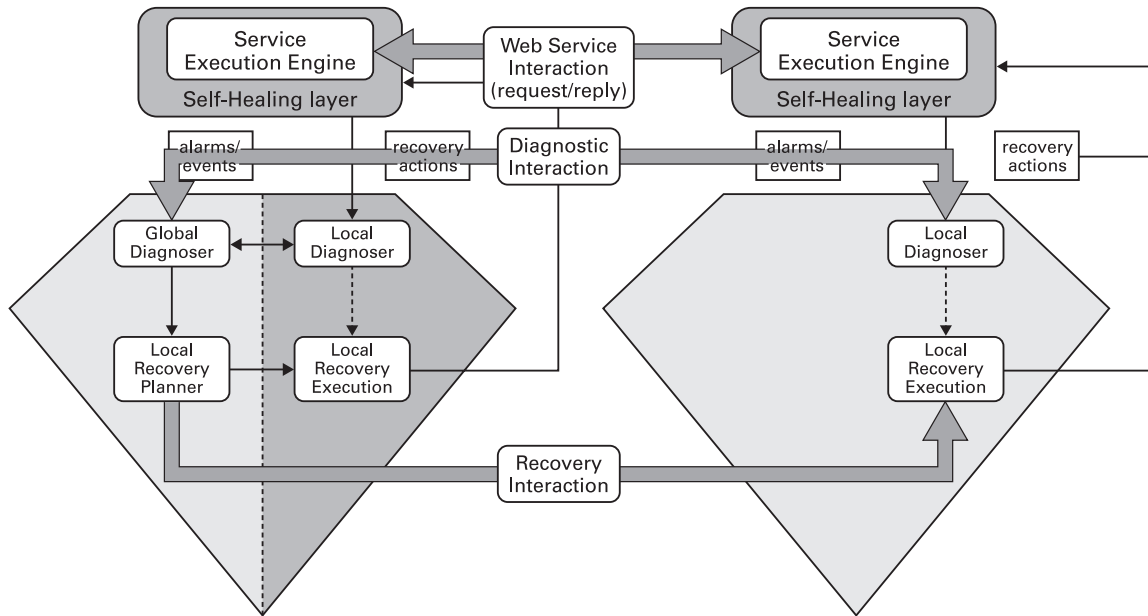
Figure 9.5
Overall Architecture

- Global diagnoser and global recovery planner modules (left part).
- Local diagnoser and local recovery execution modules (right part).

The latter is relevant because an orchestrated service may in turn be a subservice of a higher-level orchestrated 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.

9.4.2   Overall Architecture

Figure 9.5 depicts the overall architecture with interaction between a service (and its diamond) and the orchestrator (and its diamond). In particular, it shows the following:

- The two-way interaction between the local diagnoser(s) and the global diagnoser, to compute a global diagnosis in a centralized way.
- The sequential interaction between the global diagnoser and the recovery planner.
- The one-way interaction from the recovery planner to the local recovery execution module(s).

9.5   Self-Healing Layer

In this layer of WS-DIAMOND, anomalous situations may become evident at application runtime as the inability to provide a service or to fulfill a contracted QoS. To recover from these situations, various actions can be undertaken, depending on where the fault has occurred (in the network, in a server, in the application flow, etc.) and on the type of fault (blocking, interapplication, intraapplication, owing to missing data or faulty actions performed by human actors, and so on).

The methodologies provided by WS-DIAMOND for the design of self-healing services supporting service design and service execution mechanisms are embedded in this layer. These guarantee the diagnosis and repair of runtime failures through a set of observables, of exception handlers, or of sets of redundancies in the services or alternatives in the application execution strategies.

This layer handles the correct execution of complex services. The process is described using process description languages, in particular WS-BPEL (Business Process Execution Language). Process representation in the self-healing environment allows monitoring of Web Service choreography/orchestration and of its conversational behavior, as well as of data and temporal dependencies among process activities. The activity performed in this layer is the execution of mechanisms for the evaluation of actual and potential violations of process functions and quality, and for the concretization of self-healing composed services, through exploitation of diagnosis and actuation of repair actions.

The service execution engine uses, executes, and coordinates recovery actions. Recovery acts both on single Web Services and on processes composed of several Web Services. A list of repair actions is available for the service execution engine; such list can be extended when global repair strategies involve several processes. Repair actions are based on the selection of alternative services, on compensation actions, on ad-hoc exceptions handling actions, and on renegotiation of quality parameters.

The service engine is in charge of selecting the most suitable service able to provide a requested service. Services and processes have a management interface, as explained below. Compared with the existing Web Service-enabled application servers, a WS-DIAMOND server provides an environment in which to run adaptive Web Services on the basis of a QoS-driven approach. QoS is managed as a contract between the Web Service provider and the Web Service requester. The requester may specify quality requirements at Web Service invocation time, or these requirements may be implicitly specified as annotations of the services. If the WS-DIAMOND platform realizes that the QoS of a Web Service is decreasing to an unacceptable level, then sample strategies that can be adopted are channel switching, to provide the Web Service on a channel with better QoS characteristics, or Web Service substitution, selecting an alternative Web Service for the user. Alternatively, concurrent Web Services can be started to obtain the "first

responding'' best result. The substitution of alternative Web Services of course depends on an appropriate choice of a service with similar functionality. Such similarity computation is based on a semantic-based analysis of the involved Web Service and is supported by a Web Service ontology. Since substituted and substituting Web Services may have different signatures, a wrapper is used to conciliate such differences. The wrapper is created on-the-fly according to information contained in the mapping document, a configuration file that lists the set of transformations to apply on exchanged messages in order to conciliate their different schemas. The mapping document is predefined by designers at design time, with the support of the WS-DIAMOND design tools.

Monitoring is provided to capture process-related, potentially faulty behaviors and to trigger recovery actions. The focus is on aspects which are not treated in the diagnostic modules, such as proactively monitoring time constraints that can anticipate possible future time violations; monitoring errors owing to architectural problems or to QoS violations; and monitoring conversations and analyzing their behavior with respect to their expected behavior.

Different measuring and monitoring functionalities may be associated with the execution environment. Some are directly related to the execution of a service, and provide metering services to observe the service behavior, either externally or through its management interface. Other monitoring functionalities are provided, with alarms generated by monitoring process execution and exchanged messages.

Monitoring is performed at different levels, from the infrastructure, where suitable mechanisms and probes are provided, to the Web Service and the process levels, where other mechanisms are provided, such as time-outs.

Monitoring also deals with time management of Web Services, to cope with situations when, for example, several messages are to be exchanged and delivered within the communication patterns in a timely manner, in order to meet internal and external deadlines. In fact, composite self-healing Web Services often include several service providers and/or consumers, and span different companies to support the business processes. Time management is included in the core concept of Web Service functionality at the provider's side. Besides these considerations, it is necessary that the flow of work and information be controlled in a timely manner by temporal restrictions, such as bounded execution durations and absolute deadlines often associated with the activities and subprocesses. However, arbitrary time restrictions and unexpected delays can lead to time violations, increasing both the execution time and the cost of the process because they require some kind of exception handling.

Time management for Web Services is performed using predictive time management, proactive time management, and reactive time management. At design time, the process designer enriches the process with temporal information and time-related properties (execution, duration, earliest allowed start, latest allowed end, implicit and explicit temporal constraints between tasks, and so on) (Eder et al. 2006).

Repair actions are located at various levels of the self-healing architecture and may be involved in a whole plan (Modafferi and Conforti 2006). Repair can act at the instance level (e.g., redo an operation) and at the class level, modifying the service class characteristics (e.g., change the QoS profile or modify the process structure for the service and not only for a specific instance).

Repair actions are performed on Web Services through a management interface which is part of the self-healing layer. Through the management interface, particular instances of Web Services, involved in a particular instance of a process, can be repaired from a failed state. Repair actions are the following:

▪ Infrastructural actions: These allow reconfiguring the set of interacting Web Services through reallocation of services and resources at the infrastructural level.
▪ Service-level actions: These allow retrying service invocation, redoing operations with different parameters, and compensating for previous operations performed on the service; a substitution operation can also be performed at the service level.
▪ Flow-level actions: These act at the process level, changing the execution flow (skip, re-execute from) or modifying the process variables, or acting on parts of the process with process-level compensation, such as in fault handlers in BPEL (Modafferi and Conforti 2006).

An example of execution of repair actions in the self-healing layer is shown in figure 9.6.

If the Warehouse service invoked in the Foodshop has a permanent fault, and this fault is identified (e.g., during the calculation of costs), the process administrator is notified through the management interface (Modafferi et al. 2006).
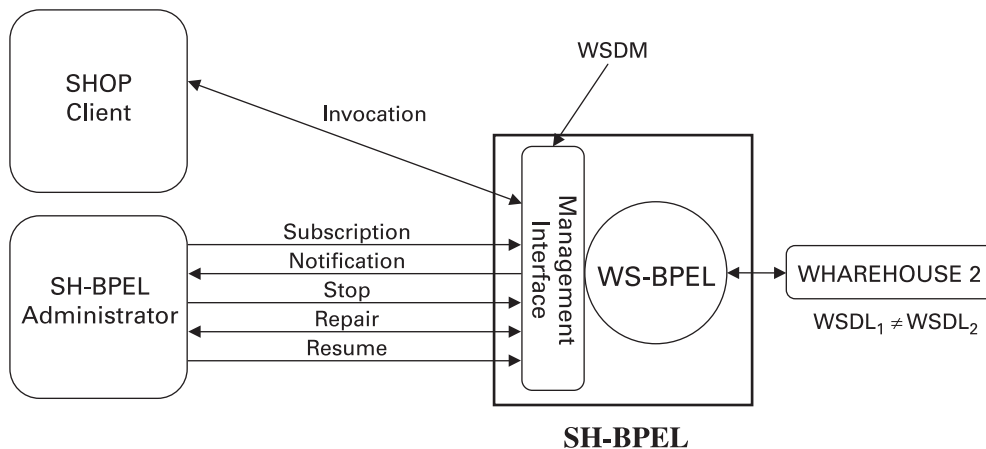
Figure 9.6
Self-healing process execution

The process manager chooses the substitution of the Warehouse as a repair action. Substitution is performed at runtime and services with compatible, even if not identical, interfaces may be invoked, through an adaptation mechanism provided in the self-healing interface.

This also implies that previous actions involving the Warehouse have to be redone (check availability of goods and calculation of costs). Then normal execution can be resumed.

9.6   Diagnostic Algorithm

The adopted diagnosis approach is model-based diagnosis (MBD). MBD (Hamscher et al. 1992) was originally 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 spacecraft (Console and Dressler 1999), but it has also been applied in other domains.

Most MBD approaches rely on a component-oriented model of the system to be diagnosed:

· The system to be diagnosed is modeled as a set of components (e.g., for physical systems, hydraulic pipes or electric resistors) and interconnections between components.
· 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 the case of physical systems may also correspond to qualitative abstractions of continuous domains.
· Component variables include interface variables, 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, temporal model that relates values that different variables take at the same time, but it can also relate values at different times; a way to do this is constraining changes of state variables, thus providing a dynamic model.

The resulting overall model of the system is therefore able to predict, or at least constrain, the effect of the correct and incorrect behavior of a component, as well as of variables that are not directly related to the component.

Diagnostic reasoning should identify diagnoses as assignments of behavior modes to components that explain a given set of observations (values for observable variables). A diagnostic engine should, in general, explore the space of candidate diagnoses and discriminate among alternative candidates, possibly suggesting additional pieces of information to

be acquired for this purpose. Discrimination should be performed only toward a given diagnostic goal (e.g., selecting an appropriate repair action).

There are several formalizations of MBD (Hamscher et al. 1992). Consistency-based diagnosis (Reiter 1987) is used in our approach: 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.

In WS-DIAMOND, diagnosis is decentralized. This means the following:

▪ 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 for identifying the causes of a global failure.
▪ We provide a global diagnoser which is able to invoke local diagnosers and relate the information they provide, in order to reach a diagnosis for the overall complex service. If 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.

This approach enables recursive partition of Web Services into aggregations of subservices, hiding the details of the aggregation from higher-level services. This is in accordance with the privacy principle which allows designing services at the enterprise level (based on intracompany services) and then using 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 need 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 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 that collects the messages and locally saves them for later inspection. When a Web Service composes a set of subsuppliers, the local diagnoser role must be filled by the global diagnoser of the subnetwork 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.
▪ 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 possible 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 illustrated approach, each local diagnoser needs a model of the service it is in charge of. We assume that each Web Service is modeled as a set of interrelated activities which show how the outputs of the service depend on its inputs. The simplest model 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 an erroneous output as a consequence.

Symptom information is provided by the presence of alarms, which trigger 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 for the purpose of selecting the appropriate recovery action.

When an alarm is raised in a Web Service W_i, the local diagnoser A_i receives it and must give an explanation. 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 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 can proceed as follows:

· If a Web Service W_j has been blamed for incorrect outputs, then the global diagnoser can ask its local diagnoser A_j to explain them. A_j can reject the blame, or explain it with an internal failure, or blame it on another service that may have sent the wrong input.
· If a fault hypothesis by A_i 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 few local diagnosers as possible, excluding those which would not contribute to the computation of an overall diagnosis (explanation).

Details on 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. 2007b), where properties of the correctness of the adopted algorithms are also proved.

9.7 Planning and Execution of Repair

The WS-DIAMOND part devoted to repair actions aimed at recovering the failed services at runtime is now presented. We have seen how the diagnosis specifies the execution of distributed conversations among local and global diagnosis services (Console et al. 2007b). We assume that the Shop is the process and that each service has its own local database, where faulty data can exist.

In general, within an organization, a process P based on Web Services invokes both operations of internal services, located within the organization boundaries, and operations of external services, located outside the organization boundaries. Both kinds of services are invoked by sending messages and receiving response messages synchronously or asynchronously. In this chapter, we consider that process services are "WS-Diamond enabled" (i.e., they are endowed with self-healing capabilities).

We define $P_j = \langle IS\_OP_j, ES\_OP_j \rangle$, where IS_OP_j is the set of the operations of the internal services and ES_OP_j is the set of operations of the external services. For each invoked service operation S_OP, its input, output, and fault messages are defined, that is, $S\_OP_i = \langle IM_i, OM_i, FM_i \rangle$. Each of these messages is composed of data parts (i.e., $M_k = \{D\}$). External services can in turn be complex processes and invoke other services. Failures (i.e., the observed symptoms of faults) can occur during the execution of the process, and manifest as fault messages for which a fault handler has not been defined. Failures occur during the execution of actions in the process, where actions are either the execution of internal service operations or invocation of external service operations.

Repair is based on repair plans (generated online or prepared offline for a given process) which are executed if a failure occurs in the process and a fault has been diagnosed. Faults are diagnosed in a distributed way, indicating which service originated the fault and faulty messages, in particular the erroneous message(s) deriving from the faulty execution in the faulty service. Hence, a fault is identified by a service-message pair $F = \langle S,M \rangle$, where S is the faulty service and M is the erroneous message originating subsequent failures. In particular, from the diagnosis we get the faulty operation and (if diagnosis allows that), in addition, the faulty output of the operation (e.g., the message which is faulty). For each failure-fault pair, a plan contains the repair actions needed to resume the correct execution

of a process. The plan is sequential, and can contain alternative paths which define alternative repair action sequences, whose execution depends on conditions on the variables evaluated during repair plan execution. The plan is generated on the basis of the analysis of which of the actions following the erroneous messages have been affected during their execution.

Repair actions are the following:

- Retry an operation
- Redo an operation: re-execute with different message data
- Compensate for the effects of an operation: invoking an operation which is defined as a compensation for a given operation in a given state
- Substitute a service.

Faults can be either permanent or temporary. If a fault is permanent, invoking the same operation of the service again will result in a failure again. If a fault is temporary, reinvocation of the operation originating the erroneous message in the $F = \langle S,M \rangle$ pair may result in a correct message. *Retry* addresses temporary faults of services, and *redo* addresses temporary faults on data (e.g., a wrong item code), whereas *compensate* can be a complex sequence of actions aimed at rolling back to a safe process state. Finally, *substitute* addresses permanent faults (e.g., a service does not answer within a given deadline). In a repair plan, other actions allow changing data values and evaluating conditions, in addition to normal service operation execution.

9.7.1   Sample Failures and Faults

In figure 9.7, the Web Services of the FoodShop interact to fulfill an order from a customer. For the sake of simplicity, we consider that the order is a list of requested items and does not exist explicitly. We assume that a failure (that is, the symptom of an error) occurs in the ForwardOrder step of the Shop service workflow. The diagnosis step detects which of the three possible faults (F1, F2, F3 in figure 9.7) has originated the failure, and then a repair action or plan is executed.

In the example, we consider the following:

1. Item_description = "lasagna"
2. Shipping_note = item_code+item_description
3. Warehouse crosschecks the item description, the shipping note, and the package before sending the package upon receipt of the request to perform ForwardOrder.

We assume that faults occur one at a time, and that *retry* and *substitute* actions are always successful.

During the diagnosis, the possible faults F are as follows:

Possible faults $F = \langle S,M \rangle$
$F1 = \langle SHOP, Item\_code \rangle$

Figure 9.7
UML-like representation of the application scenario: interacting Web Services, symptoms and fault

F2 = ⟨SUPPLIER, reservation⟩

F3 = ⟨WAREHOUSE, ItemList⟩—the WAREHOUSE fills a package with wrong goods.

In F3 the customer has to confirm the correctness of the package contents by checking the list of items.

Let us examine the case of a fault in the SUPPLIER service (F2). This service has a wrong correspondence between the name of the item and its code. It thus sends a wrong shipping note to the warehouse (and possibly a wrong response to the shop), and the warehouse (provided the goods exist) prepares the package. When the warehouse receives the item description from the shop with the ForwardOrder operation, it verifies that the ordered goods (item_description in figure 9.7), the shipping note, and the package are not consistent, and then it raises failure Fai1. However, exactly the same failure can occur if the SHOP sent the wrong code to the supplier and if the warehouse prepared the wrong package.

To decide which action(s) need to be performed during repair, diagnosis can be beneficial to select the right strategy. In fact, even with the same failure, in the three cases above, three different repair plans should be executed. For instance, if the error were in SHOP, the shop can send the correct data to the supplier and compensate for affected actions (see, for instance, plan P1 below), whereas if SUPPLIER has a permanent fault in providing a given item, the most convenient action to get the right package would be to substitute the supplier, if the supplier cannot be repaired.

9.8   Execution and Repair Actions in Plans

Table 9.1 summarizes some possible situations of faults in the various Web Services of the example. In addition, a fault type characterization is given. To test the approach, a fault injection tool has been developed that acts on data faults and on service delays (Fugini et al. 2007).

Three different plans can be generated, depending on the fault and on the infected Web Services. In table 9.2, the possible repair plans are reported. The repair strategies that can be adopted are reported in table 9.3. The plan generation is built on top of the DLV (Eiter et al. 2004) reasoning framework. DLV offers disjunctive Datalog and the handling of incomplete knowledge by negation as failure. Consequently, by exploiting DLV, we are able to model rich background theories. Such background theories are necessary in order to asses the quality of objects (faulty or correct) produced by a fault workflow instance and by subsequent repair actions. Depending on the result of repair actions, the quality of objects is changed (e.g., faulty objects may be turned into correct ones). However, in order to handle conditional branches of the plan, we had to extend the basic time model of the DLV planning system into a forward-branching time model in order to reflect different continuations of the current state of the world.

Table 9.1
Faults characterization

Correct execution	F1 (SHOP)	F2 (SUPPLIER)	F3 (WAREHOUSE)
Receive (lasagna)	Receive (lasagna)	Receive (lasagna)	Receive (lasagna)
SplitOrder (lasagna, O4)	SplitOrder (lasagna, O5)	SplitOrder (lasagna, O4)	SplitOrder (lasagna, O4)
Check&Reserve (O4)	Check&Reserve (O5)	Check&Reserve (O4)	Check&Reserve (O4)
Prepare-package (O4, lasagna)	Prepare-package (O5, spaghetti)	Prepare-package (O4, spaghetti)	Prepare-package (O4, lasagna)
Package: lasagna	Package: spaghetti	Package: spaghetti	Package: spaghetti
ForwardOrder: lasagna *OK*	ForwardOrder: lasagna *Failure Fail*	ForwardOrder: lasagna *Failure Fail*	ForwardOrder: lasagna *Failure Fail*
Fault type	Permanent (error in SHOP DB)	Permanent (error in SUPPLIER DB)	Temporary (error WAREHOUSE in filling package; when detected by diagnosis, is corrected by WAREHOUSE)

Table 9.2
Three possible repair plans

P1	P2	P3
Repair after SplitOrder	Repair after SplitOrder	Retry ForwardOrder(lasagna)
Change value: item-code=O4	Compensate(Check&Reserve)	
Compensate(Check&Reserve)	**Substitute** SUPPLIER	
Redo Check&Reserve(O4)	Invoke Check&Reserve(O4)	
IF reservation=OK	IF reservation=OK	
Retry ForwardOrder(lasagna, *OK*)	Retry ForwardOrder (lasagna, *OK*)	
Resume after ForwardOrder	Resume after ForwardOrder	
OTHERWISE	OTHERWISE	
Compensate(AskCustomer)	Compensate(AskCustomer)	
Compensate(ForwardOrder)	Compensate(ForwardOrder)	
Invoke Cancel	Invoke Cancel	

Table 9.3
Repair strategies

Failure	Fault	Plan
Fail	F1: SHOP, item code	P1
Fail	F2: SUPPLIER, reservation	P2
Fail	F3: WAREHOUSE, ItemList	P3

The execution of repair plans is based on the PAWS adaptive process execution framework (Ardagna et al. 2007). PAWS provides a set of design support tools to select candidate services for execution and substitution from an enhanced UDDI registry (URBE); to configure service mediators to enable dynamic binding and runtime substitution when service interfaces are different; to design compensate operations and a process management interface (SH-BPEL) to control process execution (start repair and resume process execution), the repair operations (retry, substitute, compensate, change values of variables), and their effects on the process execution state.

9.9 Discussion and Future Work

In this chapter, we have presented the approach of the WS-DIAMOND project to self-healing Web Services. The approach consists of a set of methodologies, design-time tools, and runtime tools to design and develop a platform for observing symptoms in complex composed applications, for diagnosis of the occurred faults, and for selection and execution of repair plans.

The chapter has presented the main results achieved in the first phase of WS-DIAMOND, where we concentrated on some specific problems, making assumptions in order to constrain the problem.

Such assumptions are being progressively relaxed or removed in the final part of the project; this approach is allowing us to manage the complexity of the problem and of the task.

A first major assumption 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 enabled, that is, that they have an associated diagnostic service. Such an assumption is being removed, and we are considering the case where some services are black boxes.

Another major assumption in the first phase is that we are concentrating on functional errors, whereas in the second phase we also consider problems related to the QoS. We will also extend the range of functional faults we are considering. This means, in particular, that the self-healing layer is being modified to include modules for monitoring QoS parameters.

Still regarding faults, we are currently concentrating on "instance-level diagnosis," that is, diagnosis and recovery (repair) of problems arising during a single execution of a service. We are currently moving to analyze issues related to multiple execution of the same service, which is especially important for QoS faults.

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. However, 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 diagnosis, which is a very important one in diagnostic problem solving, will be a topic for future investigations outside the project.

In addition, we assumed that the model of service activities which is needed by its local diagnoser is handmade. However, the dependencies that are needed can be derived from the service description, and we are currently investigating how the model can be produced in a partially automated way. The approach to diagnosis presented in the chapter is the one that in our view is best suited for the problem of explaining the causes of errors. We are currently investigating an alternative approach which is more integrated with monitoring service execution and that could be useful to detect situations that may lead to error during monitoring.

Finally, other issues that are considered in WS-DIAMOND regard security and trust in the selection of substitute services. A first approach is presented in Fugini and Pernici (2007).

Acknowledgment

Note

1. http://cordis.europa.eu/ist/istag-reports.htm.

References

AI Magazine. 2003. Special issue on model-based diagnosis (Winter 2003).

Alonso, G., and C. Pautasso. 2003. Visual composition of Web Services. In *Proceedings of the 2003 IEEE Symposia on Human Centric Computing Languages and Environments*, pp. 92–99.

Ardagna, D., M. Comuzzi, E. Mussi, B. Pernici, and P. Plebani. 2007. PAWS: A framework for processes with adaptive Web Services. *IEEE Software* (November/December), pp. 39–46.

Baresi, L., and S. Guinea. 2007. Dynamo and self-healing BPEL compositions. In *Proceedings of ICSE Companion 2007*, pp. 69–70.

Benatallah, B., F. Casati, F. Toumani, J. Ponge, and H. R. M. Nezhad. 2006. Service Mosaic: A model-driven framework for Web Services life-cycle management. *IEEE Internet Computing* 10(4): 55–63.

Candea, G., E. Kiciman, S. Kawamoto, and A. Fox. 2006. Autonomous recovery in componentized Internet applications. *Cluster Computing Journal* 9(1): 175–190.

Cardoso, J., and P. Sheth. 2003. Semantic e-workflow composition. *Journal of Intelligent Information Systems* 21(3): 191–225.

Console, L., L. Ardissono, S. Bocconi, C. Cappiello, L. Console, M. O. Cordier, J. Eder, G. Friedrich, M. G. Fugini, R. Furnari, A. Goy, K. Guennoun, V. Ivanchenko, X. Le Guillou, S. Modafferi, E. Mussi, Y. Pencole, G. Petrone, B. Pernici, C. Picardi, X. Pucel, F. Ramoni, M. Segnan, A. Subias, D. Theseider Dupré, L. Travé Massuyès, and T. Vidal. 2007a. WS-DIAMOND: An approach to Web Services—DIAgnosability, MONitoring and Diagnosis. In *Proceedings of the E-Challenges Conference*, pp. 105–112.

Console, L., and O. Dressler. 1999. Model-based diagnosis in the real world: Lessons learned and challenges remaining. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence*, pp. 1393–1400.

Console, L., C. Picardi, and D. Theseider Dupré. 2007b. A framework for decentralized qualitative model-based diagnosis. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence*, pp. 286–291.

Colombo, M., E. Di Nitto, and M. Mauri. 2006. SCENE: A service composition execution environment supporting dynamic changes disciplined through rules. In *Proceedings of the 4th International Conference on Service Oriented Computing*, pp. 191–202.

Di Penta, M., R. Esposito, M. L. Villani, R. Codato, M. Colombo, and E. Di Nitto. 2006. WS Binder: A framework to enable dynamic binding of composite Web Services. In *Proceedings of the International Workshop on Service Oriented Software Engineering, Workshop at ICSE 2006*, pp. 1036–1037.

Eder, J., M. Lehmann, and A. Tahamtan. 2006. Choreographies as federations of choreographies and orchestrations. In *Advances in Conceptual Modeling—Theory and Practice.* Berlin: Springer. LNCS 4231, pp. 183–192.

Eder, J., and W. Liebhart. 1996. Workflow recovery. In *Proceedings of the Conference on Cooperative Information Systems IFCIS (CoopIS)*, pp. 124–134.

Eiter, T., W. Faber, N. Leone, G. Pfeifer, and A. Polleres. 2004. A logic programming approach to knowledge-state planning: Semantics and complexity. *ACM Transactions on Computational Logic* 5(2): 206–263.

Erradi, A., P. Maheshwari, and V. Tosic. 2006. Policy-driven middleware for self-adaptation of Web Services compositions. In *Middleware 2006.* Berlin: Springer. LNCS 4290, pp. 62–80.

ESSI WSMO Working Group. Web Service Modeling Ontology. http://www.wsmo.org.

Fensel, D., and C. Bussler. 2002. The Web Service modeling framework WSMF. *Electronic Commerce Research and Applications* 1(2): 113–137.

Fugini, M. G., and E. Mussi. 2006. Recovery of faulty Web applications through service discovery. In *Proceedings of the 1st SMR-VLDB Workshop, Matchmaking and Approximate Semantic-based Retrieval: Issues and Perspectives*, pp. 67–80.

Fugini, M. G., and B. Pernici. 2007. A security framework for self-healing services. In *Proceedings of the 5th UMICS Workshop at CAiSE 2007.*

Fugini, M. G., B. Pernici, and F. Ramoni. 2007. Quality analysis of composed services through fault injection. In *CBP 2007 International Workshop on Collaborative Business Processes, Workshop at I5th International Conference, BPM 2007.* LNCS 4714.

Grosclaude, I. 2004. Model-based monitoring of component-based software systems. In *Proceedings of the 15th International Workshop on Principles of Diagnosis* (DX '04), pp. 155–160.

Hamadi, R., and B. Benatallah. 2004. Recovery nets: Towards self-adaptive workflow systems. In *Web Information Systems Engineering (WISE)*. Berlin: Springer. LNCS 3306, pp. 439–453.

Hamscher, W., J. de Kleer, and L. Console. 1992. *Readings in Model-Based Diagnosis.* Morgan Kaufmann. <!Where ws this published?!> DO NOT KNOW

Keller, A., and L. Heiko. 2003. The WSLA Framework: Specifying and monitoring service level agreements for Web Services. *Journal of Network and Systems Management* 11(1): 57–81.

Kephart, J. O., and D. M. Chess. 2003. The vision of autonomic computing. *IEEE Computer* 36(1): 41–50.

Marchetti, C., B. Pernici, and P. Plebani. 2003. A quality model for e-service based multi-channel adaptive information systems. In *4th International Conference on Web Information Systems Engineering, Workshops*, pp. 165–172.

Martin, D., M. Paolucci, S. McIlraith, M. Burstein, D. McDermott, D. McGuinness, B. Parsia, T. Payne, M. Sabou, M. Solanki, N. Srinivasan, and K. Sycara. 2005. Bringing semantics to Web Services: The OWL-S approach. In *Semantic Web Services and Web Process Composition.* Berlin: Springer. LNCS 3387, pp. 26–42.

Mateis, C., M. Stumptner, and F. Wotawa. 2000. Modeling Java programs for diagnosis. In *Proceedings of the 14th European Conference on Artificial Intelligence*, pp. 171–175.

Mayer, W., and M. Stumptner. 2006. Debugging failures in Web Services coordination. In *Proceedings of the 17th International Workshop on Principles of Diagnosis* (DX '06), pp. 171–178.

Modafferi, S., and E. Conforti. 2006. Methods for enabling recovery actions in Ws-BPEL. In *On the Move to Meaningful Internet Systems: Proceedings of the International Conferences CoopIS, DOA, GADA, and ODBASE.* Berlin: Springer. LNCS 4275, pp. 219–236.

Modafferi, S., E. Mussi, and B. Pernici. 2006. SH-BPEL: A self-healing plug-in for Ws-BPEL engines. In *Proceedings of the 1st Workshop on Middleware for Service Oriented Computing.* New York: ACM Press. ACM Conference Proceedings 184, pp. 48–53.

Papazoglou, M. P., and D. Georgakopoulos. 2003. Service-oriented computing. *Communications of the ACM* 46(10): 24–28.

Patil, A., S. A. Oundhakar, A. P. Sheth, and K. Verma. 2004. Meteor-S Web Service annotation framework. In *Proceedings of the 13th International Conference on the World Wide Web*, pp. 553–562. New York: ACM Press.

Peischl, B., J. Weber, and F. Wotawa. 2006. Runtime fault detection and localization in component-oriented software systems. In *Proceedings of the 17th International Workshop on Principles of Diagnosis* (DX '06), pp. 195–203.

Pernici, B., ed. 2006. *Mobile Information Systems—Infrastructure and design for adaptivity and flexibility*, Berlin: Springer.

Ran, S. 2003. A model for Web Services discovery with QoS. *SIGecom Exchanges* 4(1): 1–10.

Reiter, R. 1987. A theory of diagnosis from first principles. *Artificial Intelligence* 32(1): 57–95.

SeCSE Team. 2007. Designing and deploying service-centric systems: The SeCSE way. In *Service Oriented Computing: A Look at the Inside* (SOC@Inside '07).

Spanoudakis, G., and K. Mahbub. 2006. Nonintrusive monitoring of service based systems. *International Journal of Cooperative Information Systems* 15(3): 325–358.

Spanoudakis, G., A. Zisman, and A. Kozlenkov. 2005. A service discovery framework for service centric systems. In *Proceedings of the IEEE International Conference on Services Computing*, vol. 1, pp. 251–259.

van der Aalst, W. M. P., ter Hofstede, A. H. M., and Weske, M. 2003. Business process management: A survey. In *Business Process Management: International Conference* (BPM 2003). Berlin: Springer. LNCS 2678, pp. 1–12.

Verma, K., and Sheth, A. P. 2005. Autonomic Web processes. In *Proceedings of the 3rd International Conference on Service Oriented Computing*, pp. 1–11.

Wotowa, F., Stumptner, M., and Mayer, W. 2002. Model-based debugging or how to diagnose programs automatically. In *Proceedings of IEA/AIE 2002.* LNAI, pp. 746–757.

Zeng, L., Benatallah, B., Ngu, A. H. H., Dumas, M., Kalagnamam, J., and Chang, H. 2004. QoS-aware middleware for Web Services composition. *IEEE Transactions on Software Engineering* 30(5): 311–327.