Recovery of Faulty Web Applications through Service Discovery

M.G. Fugini, E. Mussi

Politecnico di Milano
Dipartimento di Elettronica ed Informazione
Via Ponzio 34/5 - I-20133 Milano
fugini,mussi@elet.polimi.it

Abstract. Failures during Web service execution may depend on a wide variety of causes. In this paper we illustrate how, upon faults of a Web application, the faults can be handled by discovering substitute Web services and re-orchestrating the application. Self-healing capabilities are added to Web service environments. Possible recovery actions at the *Web service* and *Web application* levels are illustrated and discussed with respect to a running example involving coordinated Web services.

1 Introduction

Even if various efforts have been recently done in Service Oriented Computing, the adoption of Web services still remains limited. Especially under the open world assumption, the Service Oriented Architecture, on which Service Computing relies, presents some limitations. In fact, if we consider public available Web service registries, as UDDI, it is easy to find Web services no longer available or Web services having a description which does not correspond to the currently provided Web service, due to new versions or modifications of functionality of the Web services.

As discussed in this paper, even in a more controlled closed world assumption, there are many possible causes of failure. As closed world, we consider the adaptive Web service environment studied in the MAIS (Mobile Adaptive Information Systems) project [13]. In such an environment, adaptivity is provided at both the front-end and the back-end of Web applications. These are dynamically created composing Web services and are able to satisfy both functional and non-functional user requirements. This approach results in a context-aware Web service discovery and selection, which can be performed at runtime, during Web application execution. Under this approach, the management of *failures* poses new research problems related to the identification of causes, of possibly overlapping failure, and to their automatic correction.

The WS-Diamond (Web service DIAgnosability, MONitoring, and Diagnosis) project is aimed at developing methodologies 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 Quality of Service (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 design of services, supporting service design and service execution mechanisms that guarantee diagnosability and reparability of run-time failures (e.g., thanks to the availability of set of observables, of exception handlers, or of sets of redundancies in the services or alternatives in the execution strategies).

The goal of this paper is to present an architecture for self-healing Web services and applications, which contains modules for the detection, diagnosis, and repair of faults. Our goal is to show how faults in Web application can be repaired using repair actions that include switching to a substitute service, by searching it in UDDI registries, based on similarity criteria. In particular, we focus on healing mechanisms based both on service selection and substitution and on addition of new services in composed processes to support self-healing functionalities.

Section 2 introduces self-healing, diagnosis, and quality of service issues in Web services and describes some approaches proposed in the literature. Section 3 presents our architectural scenario of a self-healing Web service environment, and an example. Section 4 introduces an architecture supporting the detection and recovery of possible faults. Section 5 discusses *reactive* and *proactive* healing techniques, based on service execution analysis.

2 Related Work

Recently, self-healing systems have attracted vast attention by the research community [11]. Examples of self-healing architectures are provided in [16]. The work in [16] presents a system architecture to monitor, interpret and analyze system events in order to implement self-healing and self-adaptive systems. The architecture presented in [15] is focused on service level agreement management in a Service Oriented Architecture. The goal is to re-optimize the provisioning infrastructure as a consequence of QoS violations. The work in [9] presents the requirements of a Web Service Management (WSM) framework which also includes the typical functionalities addressed in self-healing systems, analyzing and comparing multiple alternative architectures for the implementation of WSM systems i.e., centralized, federated and peer-to-peer. They propose Web service substitution and complex service re-compositions as repair actions.

In [2, 3, 17], authors propose a model based approach to implement diagnosis functionalities in Web services self-healing environments. The works in [2, 3] model a composite service as a Petri Net and classify Web service invocations. The goal is to correlate input and output parameters and to determine if an activity carries out a computation that may fail and produce erroneous outputs. A service invocation forwards an input parameter, if an output parameter coincides with the value of an input. If the output parameter is created during the service invocation this is classified as a source. Finally, when an output parameter is computed during the service invocation from one or more inputs the service invocation is an *elaboration.* A service invocation can introduce errors

in parameters it computes (elaborations) or it produces (sources), while for forwarded parameters it possibly propagates errors generated by other invocations. Authors propose a centralized diagnoser which identifies faults from "alarms" identified by comparing the value of state variables at checkpoints introduced by the composite Web service designer.

A different approach is used in [10], where authors describe an extended Petri net model for specifying exceptional behavior in workflow systems. The core of this approach are the recovery policies, a combination of generic constructs and primitive operations that can be defined for both single tasks and sets of tasks (i.e., recovery regions). At design time, process designers model exceptions using the generic constructs, while at runtime the workflow system uses the primitive operations to handle the occurred exceptions.

With respect to the cited works, our approach uses dynamic Web service discovery and selection as one of the main repair strategy. Given a faulty application, we are able to substitute one or more component Web services with compatible ones. The selection of compatible services is performed using semantic criteria based on service descriptors stored in an enhances UDDI registry. Moreover the selection involves both functional and non functional similarity evaluations.

3 Adaptive Web Service

WS-Diamond aims at realizing a set of tools that can be plugged into existing Web service environments and provide self-healing features during the Web service life cycle.

With respect to the traditional SOA, the adopted system also supports Web service adaptation (as studied in MAIS [12]). In particular, we suppose hat a set of actors collaborates according to a shared choreography, which defines how and when the actors are involved during this collaboration. Operationally, each actor declares its ability to perform one or more activities included in the choreography specification, offering one or more Web services. According to this scenario, in this work we consider a *Web application* as a collaboration. Thus, a Web application is a distributed application and the choreography is responsible for describing the collaboration between all the involved Web services.

The reference architectural aspects are illustrated in the following with respect to the dimension of *cooperation among distributed nodes*. Figure 1 shows a global self-healing system in which WS-Diamond-enabled nodes can cooperate with non WS-Diamond-enabled nodes. In addition, each WS-Diamond-enabled node may provide only a subset of the modules developed in the project. The main modules of a complete WS-Diamond-enabled node are: i) a management interface for Web services; ii) a process orchestration engine for enacting composed services; iii) a repair action selector; iv) a diagnosis infrastructure; v) a fault detection infrastructure.

In the paper, we illustrate how the different modules cooperate *inside a node*. Cooperation and negotiation of available substitute services occur via a contract-based approach.

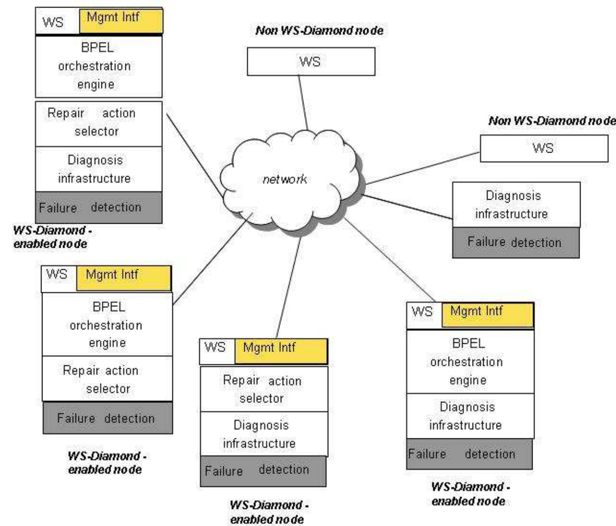

Fig. 1. Reference architecture

3.1 WS-Diamond Nodes

We assume that, within a WS-Diamond node, not all the diagnosys and repair features are available, but rather that each node is equipped with the fault and self-healing modules necessary to cooperate. The cooperation of the modules inside a node and within the execution environment main components views the diagnoser as the module to be notified of fault events through messages or events generated by the hardware/software infrastructure (including the self-healing system itself). By accessing messages, state logs and a fault database (all fault events are stored in a fault database), the diagnoser identifies which occurred fault needs to be recovered. The repair action selector performs a choice among a set of possible repair actions associated to each type of fault, as indicated in a *repair rules registry*. The selection triggers a repair action request to a repair module associated to the required action. A possible list of repair modules

contains: a substitution module (to replace services during the orchestration of a composed service), a wrapper generator (to change parameters to solve incompatibility during invocation), a quality module (to perform data quality checks and improvements), and a reallocation module (to reallocate the resources exploited by the services).

Our cooperating environment might be technologically heterogeneous. In fact, an actor involved in the cooperation can rely either on a classical node or on a WS-Diamond node. About the former, Web services run over typical application servers providing a traditional Web service container. About the latter, we suppose to use a special application server, called MAIS-P [13], specifically designed to support adaptive Web services.

In the MAIS-P architecture, Web services run on the assumption that something might go wrong at invocation and execution time and, for this reason, a QoS driven approach is introduced to solve this possible critical situations. The service provider and a service requestor, in fact, agree on the *quality* that the Web service must guarantee during its execution. The requestor may specify quality requirements at Web service invocation time or these requirements may be implicitly specified in the user profile [5]. If a host realizes that the QoS of a Web service is decreasing to an unacceptable level, then two strategies can be adopted: channel/partner switching, to provide the Web service on a channel or through a partner with better QoS characteristics, or Web service substitution, selecting an alternative Web service for the user. Alternative Web services are searched by accessing an extended UDDI registry capable of assessing the similarity among Web services with respect to the provided functionality and the behavioral equivalence [4]. Similarity computation, supported by a Web service ontology, is based on a semantic-based analysis of the involved Web service. Since substituted and substituting Web services might have different signatures, an automatic wrapper generator can reconciliate differences in the provided interfaces, possibly with human intervention [8].

A sample application scenario may consist in four Web services: `BookShop WS`, `Publisher WS`, `Warehouse WS` and `Shipping WS`, supporting a book e-commerce process. In this scenario, a customer asks for a book invoking the `BookShop WS`, and submitting the book title. Each customer has a profile (custInf) which is used during Web service contracting and provisioning. The customer's profile collects various information about the customer, such as age, education, profession, interests, and book preferences, maximum budget. Such profile information is available to the `BookShop WS` using WS-enabled nodes to collect the customer's profile and to send it to the server [5]. We assume that the Web services in the application are *internally orchestrated*, while externally they are only *coordinated through the choreography.*

The goal of the set of interacting services is to send the right book to the right customer according to his preferences. In fact, a number of faults may occur during the execution of the process causing possible needs of substituting services and discovering new ones. For example, the wrong book is delivered [3], or a book is indefinitely reserved for a user who will not buy it, an order is

indefinitely delayed, or one or more of the involved services indefinitely runs waiting for a reply from possibly faulty services.

4 Self-healing platform

On one hand, we aim at realizing a platform to support a self-healing Web service execution where faulty services can be substituted by discovering compatible services. This means that Web services should be able to detect possible failures caused by the Web services layer and, consequently, to enact a recovery action transparently to the user standpoint. On the other hand, since during a Web application execution a Web service may invoke external (non self-healing) Web services, our upgraded MAIS-P platform should be able to detect failures even at application level, i.e., caused by the external Web service. In this case, according to the loosely-coupled philosophy underlying the Service Oriented Computing, we are not interested on fixing the failed Web service, but we try to react to such a failures possibly substituting the failed one.

Figure 2 shows the modules, embedded in the WS-Diamond nodes, that support our goals. The *Diagnoser* identifies fault events or receives fault event notifications. The identification task operates on the information coming from the infrastructure & middleware layer, to detect internal node failures, and on the exchanged messages, to detect failures on partner's nodes.
Our architecture provides four modules to handle recovery actions:

- *reallocation module*: it optimizes resource reallocation requests on the underlying hardware and software infrastructure; resource reallocation policies are proposed in [1];
- *substitution module*: this module, available in the MAIS-P platform, allows the selection of substitute services according to the characteristics of the service request context, such as services which are functionally similar [13];
- *wrapper generator module*: this module, available both in MAIS and VISPO platforms [13, 8], allows the completion of missing parameters during service invocation, or to convert parameters in different formats;
- *quality module*: the PoliQual system [6] provides functionalities to insert data blocks before Web service invocation or upon receipt of a message, to assess data quality on-line and to perform reactive and proactive repair actions.

We assume that faults have to be recovered one at a time, and that the *Recovery action selector* module, by accessing the Fault registry, invokes the corresponding fault repair action.

4.1 Fault Registry

A basic set of known service faults is classified and stored in a *Fault Registry* at three *system levels*, for which, examples are provided in Table 1:

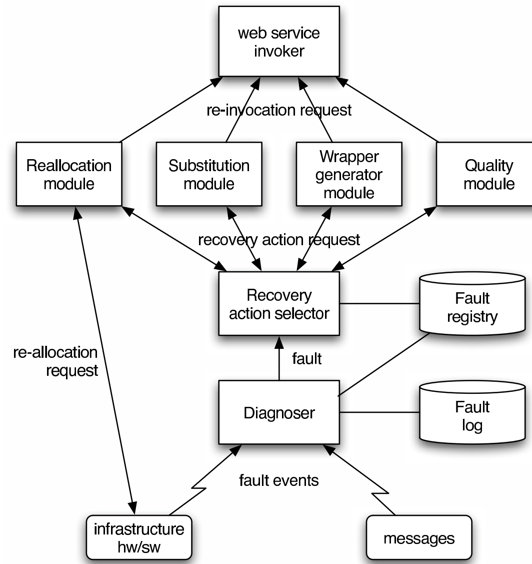

Fig. 2. Self-healing platform architecture

- the *Infrastructure & Middleware level* faults (not reported in the table) are due to failures in the underlying hardware, network, and system software infrastructure;
- the *Web service level* faults are due to failures in service invocation and service orchestration; these are captured by the WS-Diamond modules;
- the *Web application level* faults are malfunctions in the execution of Web applications due to data mismatches or coordination (choreography) failures. Also these faults are captured by the WS-Diamond modules.

Repair actions are designed according to the fault level and originate different recovery strategies according to the system components affected by the fault. For example, at the Web application level, the main goal is to provide: (i) services which respect the user requirements in terms of functionalities and QoS, (ii) business continuity, and (iii) fault masking. At the Web service level, the goal is to manage the service choreography correctly and to guarantee service continuity and QoS requirements, by substituting corrupted services with compatible ones available in the network.

4.2 Web service level faults

Faults at the Web service level are about the execution of a (set of) Web service(s). In particular, *Web service execution faults* raise either during service invocation or during execution of a simple Web service, while *Web Service coordination faults* result from composed Web services.

Level	Fault type	Examples
Web-application	Internal data faults	Data quality faults (value mismatch, e.g., inaccurate data in input parameter; missing data: null values).
	Application coordination faults	Application Failure due to reply timeout, resources not available at right time (Phase fault).
	Actor faults	Customer is not connected when a synchronous communication is needed.
	QoS violation faults	QoS value beyond threshold.
Web service	Web service execution faults	Missing parts in input message, wrong order of operation invocations (internal to a service).
	Web service coordination faults	Component service unavailable, process failure (time out).

Table 1. Fault Types

A *service execution* fault is raised during invocation if a service is not responding, or due to a wrong authorization of the end user, or to a parameter missing in the input SOAP message. A mismatch in the structure of messages to invoke a service may be due, for instance, to an update in the published service interfaces, which are not yet considered inside the invoking applications.
In our adaptive framework, a Web service execution fault may also occur when, upon substitution of a faulty Web service with a functionally equivalent one, no substitute is available. In that case, a discovery of a compatible service is needed in order to possibly substitute the faulty service. Discovery is performed using additional data about services, which are registered as elements of one or more *compatibility classes* [8], where each class identifies the required functionalities for the execution of a process activity. The rationale is that a provider associates a compatibility class with the service he is registering, if he thinks the service is able to satisfy the activity requirements. These requirements are stated when the class is firstly created and have to be satisfied by all class members. During the registration, compliance of the registered service to its compatibility classes is also evaluated and mapping information for semi-automatically building wrappers to adapt services to the process is generated.

A *service coordination fault* is typically due to a violation of the order of invocation of service operations or messages (e.g., a book payment is received before the corresponding book reservation). Coordination faults may occur when some of the Web services in a composed service are unavailable, or when a message is received that does not match the choreography protocol. Again, a phase of service discovery based on compatibility classes for substitution is needed.

4.3 Application level faults

Application level faults are related to the execution of a Web application. The mechanism used to detect a faulty service invocation is the *timeout*. If the result

of an operation is not received by the due time, the application argues that some of the invoked services are not working properly.

Examples of application faults in the bookshop example are: unavailable goods, wrong book identification data, or session faults.

Fault at this level belong to the following categories: 1) Application Coordination faults; 2) Actor faults; 3) QoS violation faults; 4) Internal Data faults. Examples of *Application Coordination faults* in the bookshop example are:

– A session fault: e.g., loss of an HTTP session.
– A phase time fault: e.g., the book has been paid for and hence the Payment phase of the application has completed, but the confirmation of payment is received after an internal time out.
– A resource booking fault: e.g., not the whole resource pool necessary to complete the service has been reserved, for example the shipping system has not been reserved in advance.
– Inter-process faults: e.g., service data regarding the customer address or customer credit card have not been received in the correct sequence or at the right time.

An example of *Actor fault* is an authorization fault, e.g., the customer has entered a wrong password three times. *QoS violation faults* are related to local and global constraints specified by the user or by the application designer. For example, the user can require that the delivery time of the ordered book is lower than a given threshold (e.g., five days) or that the total price of the ordered book is lower than a given amount (e.g. 20$). Another category of QoS faults is bound to the *process design*, that is, to the way the application workflow has been developed. For example, an Unavailable goods fault, or a Payment failure fault should be treated by *exception handlers* specifically included in the process workflow by the designer. If some handlers have not been designed, the application could experience a deadlock or a total crash. Being the system self-healing, we expect the fault log to gather information useful to subsequently design the necessary handler.
Internal Data faults include data quality faults related to data manipulated during the execution of a service. Examples are the wrong title of an ordered book, or mismatched customer data. These faults may be regarded as QoS faults, but, since they specifically regard data internal to a service, they are evidenced as possibly bounded to specific filters and options to be treated apart by specific recovery actions.

5 Self-Healing Web services

The architecture shown in Figure 2, currently under development, connects the schema of the fault types with the schema of the recovery actions. Namely, the fault registry will be accessed by the diagnosys module in order to blame the faulty Web service. Once that the fault type has been identified, the Recovery Action Selector accesses the recovery actions table (see Table 2) to select the

proper repair action that has to be undertaken. This selection is performed at runtime, taking into account context information (e.g., the network status or the workflow execution status). Hence, the same fault type can be handled by different recevery action modules, according to the particular context in which the fault has occurred.

Recovery action type	**Actions**	**Fault type**	**Type**
Service-oriented recovery action	Retry	Infrastructural, WS execution	Reactive
	Substitute	WS execution, WS coordination	Reactive
	Completion of missing message parts	WS execution	Reactive
	Reallocate	QoS	Reactive/proactive
	Change process structure	All	Proactive
	Process-oriented methods	Application level	Proactive
Data quality recovery actions	Insert data quality block	Internal data	Reactive
	Process-oriented methods	Application level	Proactive

Table 2. Recovery actions

This Section presents the set of recovery actions that can be employed to recover from failures. In particular, we focus on the management of Web application and Web service level faults. With respect to the way in which these actions are performed, two types of recovery actions are identified: *reactive recovery actions* and *proactive recovery actions*.

Reactive recovery actions are performed along with the execution of Web services and try/allow the recovery of running services. *Proactive recovery actions* are mostly based on data mining techniques and can mainly be executed in an off-line mode; proactive recovery actions are complex and require the support of an environment able to execute services, to detect runtime faults, and to perform recovery actions with no damage to the running instances of the monitored Web services. A long term approach to self-healing is adopted, where recovery actions have the goal of improving Web services and Web applications in order to avoid future failures. These actions can be oriented to provide a one-shot improvement action or to modify the service provisioning process for a permanent data and process improvement.

Recovery actions can be also classified in *service-oriented recovery actions* and *data quality recovery actions*. While the former deals with invocation, orchestration and choreography aspects of Web services, the latter pertains mainly to the management of data quality faults. For each fault type, several candidate recovery actions may be proposed, depending on the fault type and whether a reactive or proactive approach is taken, as synthesized in Table 2.

In the bookshop example, assume the invocation of the *sendBook* operation on the `Publisher WS` fails. The Diagnoser detects that the `Publisher WS` is experiencing a quality degradation that slows down its execution so that it cannot be used by other Web services anylonger. Consequently, the Diagnoser stores the detected fault in the fault log and invokes the Recovery Action Selector module. This module repairs the fault using the Substitution module to search for a compatible Web service to substitute the `Publisher WS`. The workflow execution is then restarted by invoking the *sendBook* operation on the substitute Web service. If the interface of the substitute service is different from the interface of the original service, the invocation is mediated by the Wrapper generator module.

Meanwhile, since the Diagnoser has detected that the quality of service degradation was due to a server crash at the publisher site, the WS-Diamond architecture proactively activates also the Reallocation module. Based on the analysis of the fault logs, the Reallocation module reallocates all the Web services that were running on the crashed server.

5.1 Service-oriented recovery actions

The techniques employed to realize this kind of recovery strictly depend on the model used to describe Web services. Some models describe how Web services act internally (i.e., orchestration), while other models only describe how different Web services collaborate (i.e., choreography). While both choreography and orchestration are exploited to detect faults, reactive recovery actions only rely on the orchestration model of the service. With our architecture, if a Web service has an internal behavior specified using a WS-BPEL process that composes other services, we are able to perform recovery actions over the Web service controlling the execution of its internal process. Under our approach, the recovery actions that can be applied over a WS-BPEL process in a MAIS-enabled node are the following: i) *retry* the invocation of a failed Web service, ii) *substitute* a failed Web service, iii) *reallocate* a failed Web service, and iv) *change* the structure of the process.

Retry Web services invocation
This recovery action is applied when faults point out a temporary unavailability of one or more services that compose the internal process of the analyzed Web service. In this case, the solution is to suspend the execution of the process and retry the invocation of the unavailable services until they return available. This solution is quite simple and does not require any sophisticated methodologies to manage the service invocation.

Substitute Web services
A more complicated situation is the case where one or more services are considered as definitely unavailable and, in order to complete the process execution, it is necessary to substitute each failed service.

It is possible to overcome this problem using the MAIS-P architecture that supports Web service compatibility evaluation and Web service substitution.

The compatibility evaluation between two Web services is performed comparing their functional interfaces (i.e., WSDL documents) and their provided QoS. If two services are defined as compatible, MAIS-P is able to automatically create a *service wrapper*, starting from their WSDL descriptions. Once that the wrapper is created, service substitution can be easily performed. The process structure is not modified, and the WS-BPEL orchestration engine continues the execution of the process without considering the service substitution. The management of the substitution is left to the MAIS-P architecture which exploits the wrapper to translate the parameters sent by the orchestrator into the parameters accepted by the substitute service and vice-versa.

Completion of missing parameters
Service invocation may fail when some of the input message parts are missing. Possible recovery actions may be based on knowledge of the role of parameters. In [8], we propose a technique to dynamically evaluate message composition of invoked Web service operations and look for missing information when parameters are necessary for message execution, while optional parts are ignored. The technique is based on an adaptive service invocation infrastructure.

Reallocate Web services
This type of recovery action is very useful for the particular subset of QoS violation faults that derives from a lack of hardware or software resources on the service provider side. In this situation, reallocating and executing the service on different machines or application servers can solve the problem. Reallocation is possible only if Web services are provided with an ad-hoc management interface and the recovery manager has free access to all the resources (e.g., the recovery manager can determine the load balancing or the application priority in the operating system).
Reallocation may be performed as reactive actions, when QoS violations are detected, but also as proactive actions, when optimization of service execution plans is performed using predictive techniques on future states of the execution environment.

Change the process structure
When service substitution or service reallocation are not enough for resuming a failed process, another recovery action that can be applied consists in modifying the structure of the process itself. According to [7], data mining techniques can be exploited in order to proactively identify possible anomalous situations during process execution. Workflow systems, in fact, produce a large quantity of log information which states how the old processes have been executed and how the current process is going on. In this case, the process designer, possibly supported by a tool, can manually fix or modify the process in order to delete erroneous activities. Once that the process is modified, it can be resumed or re-executed and previously described recovery actions can be employed.

5.2 Data quality recovery actions

Run time recovery actions in data quality require the identification of the causes of data errors and their permanent elimination through an observation of the whole process where data are involved. As an example, we refer to the Information Product Map (IP-MAP) methodology [14] which graphically describes the process by which the information product is manufactured. Error detection and the correction can be performed using different methods:

- *Data cleaning by manual identification*: comparison between value stored in the database and value in the real world;
- *Data bashing (or Multiple sources identification)*: comparison of the values stored in different databases;
- *Cleaning using Data edits*: automatic procedures that verify that data representation satisfies specific requirements.

6 Concluding Remarks

We have presented our approach for fault management of Web applications based on self-healing systems. A reference architecture for faults treatment and a classification of faults have been given, together with a set of strategies for recovery. Fault occurring during Web service and Web application execution have been studied and discussed within a proposed architecture where faults detection and interpretation for repair requiring search of substitutive services have been presented. Mutual dependencies among faults originated at these different system abstraction levels are a relevant issue to be further investigated in our research. Moreover, the distribution vs. central coordination of fault events detection and repair action execution are an important aspect to be studied for the proposed architecture.

Future work will focus on careful design of repair strategies, and on the evaluation of the proposed approach in the Project testbed environment.

Acknowledgments
Part of this work has been supported by EU Commission within the FET-STREP Project WS-Diamond.

References

1. D. Ardagna, M. Trubian, and L. Zhang. SLA Based Profit Optimization in Multi-tier Systems. In *NCA 2005 Proc.*, 2005.
2. L. Ardissono, L. Console, A. Goy, G. Petrone, C. Picardi, M. Segnan, and D. T. Dupre: Advanced Fault Analysis in Web Service Composition. In *International WWW Conference, poster session*, pages 1090–1091, 2005.
3. L. Ardissono, L. Console, A. Goy, G. Petrone, C. Picardi, M. Segnan, and D. T. Dupre: Enhancing Web Services with Diagnostic Capabilities. In *ECOWS05 Proc.*, 2005.
4. D. Bianchini, V. De Antonellis, B. Pernici, and P. Plebani. Ontology-based methodology for e-service discovery. *Accepted for publication on Journal of Information Systems, Special Issue on Semantic Web and Web Services*, 2004.
5. C. Cappiello, M. Comuzzi, E. Mussi, and B. Pernici. Context Management for Adaptive Information Systems. In *International Workshop on Context for Web Services (CWS-05)*. Elsevier, 2005.
6. C. Cappiello, C. Francalanci, and B. Pernici. A self-monitoring system to satisfy data quality requirements. In *Proc. ODBase05*, 2005.
7. M. Castellanos, F. Casati, M.-C. Shan, and U. Dayal. iBOM: A Platform for Intelligent Business Operation Management. In *Proc. of the 21st International Conference on Data Engineering, ICDE 2005, 5-8 April 2005, Tokyo, Japan*, 2005.
8. V. De Antonellis, M. Melchiori, L. D. Santis, M. Mecella, E. Mussi, B. Pernici, and P. Plebani. A Layered Architecture for Flexible Web Service Invocation. *Software: Practice and Experience*, 36(2):191–223, February 2006.
9. E. Esfandiari and V. Tosic. Towards a Web Service Composition Management Framework. In *In ICWS05 Proc.*, 2005.
10. R. Hamadi and B. Benatallah. Recovery Nets: Towards Self-Adaptive Workflow Systems. In *WISE*, pages 439–453, 2004.
11. P. Koopman. Elements of the Self-Healing System Problem Space. In *ICSE WADS03 Proc.*, 2003.
12. MAIS Web Site. http://www.mais.project.it.
13. B. Pernici, editor. *Mobile Information Systems. Infrastructure and Design for Adaptivity and Flexibility.* Springer, April 2006.
14. G. Shankaranarayan, R. Y. Wang, and M. Ziad. Modeling the Manufacture of an Information Product with IP-MAP. In *Proceedings of the 6th International Conference on Information Quality*, 2000.
15. G. Wang, C. Wang, A. Chen, H. Wang, C. Fung, S. Uczekaj, Y. L. Chen, W. Guthemiller, and J. Lee. Service Level Management using QoS Monitoring, Diagnostic, and Adaptation for Networked Enterprise Systems. In *EDOC 2005 Proc.*, pages 239–248, 2005.
16. D. S. Wile and A. Egyed. An Externalized Infrastructure for Self-Healing Systems. In *Fourth Working IEEE/IFIP Conference on Software Architecture (WICSA'04)*, 2004.
17. Y. Yan, M. Cordier, Y. Pencolé, and A. Grastien. Monitoring Web Service Networks in a Model-based Approach. In *ECOWS05 Proc.*, 2005.