

WS-DIAMOND: An Approach to Web Services – DIAGNOSABILITY, MONITORING and DIAGNOSIS

Luca CONSOLE¹, Mariagrazia FUGINI² ⁱ

¹*Dipartimento di Informatica, Università di Torino, Corso Svizzera 185, I-10149 Torino, Italy*
Email: luca.console@di.unito.it

²*Politecnico di Milano, Piazza da Vinci, 32, Milan I-20133, Italy*
Tel: +39-02-23993405; Fax: +39-02-23993411; Email: fugini@elet.polimi.it

Abstract: Self-healing software is one of the challenges issues for IST research. The WS-Diamond project aims at making a step in this direction by developing a framework for self-healing Web Services. In this paper, we present the overall goals of the project, namely: i) defining an operational framework for self-healing service execution of conversationally complex Web Services, where monitoring, detection and diagnosis of anomalous situations, due to functional or non-functional errors (e.g., Quality of Service), are carried on and repair/reconfiguration is performed, thus guaranteeing reliability and availability of Web Services; and ii) defining a methodology and tools for service design that guarantee effective and efficient diagnosability/reparability during execution. Then the paper illustrates an application scenario showing principles of model-based diagnosis, and composed Web Service repair scenarios. An overall presentation of the architecture is given.

1. Introduction

The WS-Diamond Project, funded by the EU commission under the FET (Future and Emerging Technologies) – Open framework, aims at the development of a framework for *self-healing* Web Services, that is, services able to *self-monitor*, to *self-diagnose* the causes of a failure, and to *self-recover* from functional failures (e.g., the inability to provide a given service) and from non-functional failures (e.g., loss of Quality of Service – QoS). The focus of WS-Diamond is on composite and conversationally complex Web Services. The second goal of WS-Diamond is to devise guidelines and tools for designing services in such a way that they can be easily diagnosed and recovered at execution time, and tools to support the design of complex self-healing processes.

The project assumes that the availability and reliability of complex services will be of paramount importance in the near future. 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. Hence, WS-Diamond research concerns a number of “grand challenges” as described within the SOC research roadmap [1], at all levels: in the *Service foundations*, WS-Diamond studies dynamic connectivity capabilities, based on service discovery; at the *Service composition* level, it studies QoS-aware service composition, at the *Service management*, and *Service design and development levels*, it provides design principles for self-healability.

This paper focuses on general aspects of the project; then it presents an application scenario where repairs actions and plans [2] on complex composed services are described. In the literature, Web Service workflow languages and frameworks are being proposed, such as BPEL4WS (BPEL for short), allowing for a mixture of block and graph structured process models, thus making complex Web Service based applications available. Mechanisms for augmenting processes with monitoring functionality have been proposed in

[3], and in grid environments [4] to control access to relevant information about resource accessibility and state. However, a methodological approach is needed to design all aspects of such systems, focusing on exception handling and compensation mechanisms [5]. Methodologies and tools have been developed in the MAIS project, for adaptive web-based process execution based on flexible services [6]. Extended Petri Nets have been used as a modelling technique and as a basis for building analysis tools in process modelling [7]. However, little attention is paid to conform to patterns of interactions between organizations and to provide inherent flexibility and fault tolerance in process execution. More generally, attempts to provide self-healing capabilities to applications are studied in the field of autonomic computing, aimed at creating systems that can manage themselves when given high-level objectives from administrators [8], or in the field of organic computing. Both fields propose techniques for recovery actions in front of anomalous events occurring in the execution environment; WS-Diamond faces these same issues, but focuses on Web Services applications. Basic recovery is proposed in the literature with retry and substitution operations. Recovery in web-based process evolution is proposed in [9] based on adaptive service composition and service substitution. In grid services, retry and substitution operations are defined for recovery, but a more comprehensive approach to service repair is advocated in [10]. The mentioned approaches focus only of a direct repair of failed services, based on monitoring of failures, while the link between failures and faults is not considered as a basis for repair strategies.

The paper is organized as follows. Section 2 presents the overall approach to faults, failures, and repair, and describes the WS-Diamond architecture. Section 3 describes an application scenario and presents examples of repair actions and plans executions. Section 4 concludes by discussing achievements and directions of research.

2. WS-Diamond Objectives: Approach and Architecture

The first goal of the project, addressed in the first phase of the project, is the design and development of a platform for supporting the self-healing execution of complex Web Services. This means that we focus on the problems that occur at run time, while the design issues will be faced in a second phase of the project. A second general consideration is that we are focusing on diagnosing problems that occur at run time and we are not considering the issue of debugging a service; in other words we assume that code has been debugged.

This led us to defining:

- the types of faults that can occur and we want to diagnose, that is:
 - functional faults and specifically semantic data errors (such as wrong data exchanges, wrong data in databases, wrong inputs from user, ...)
 - quality of service faults(In this paper we will focus on the former.)
- the types of observations/tests that can be available to the diagnostic process:
 - alarms raised by services during their executions
 - possible data exchanged by services
 - internal data to a service
- the types of repair/recovery actions that can be performed, such as compensating or re-doing activities.

In the first phase of the project, we concentrated on orchestrated services, even if some of the proposed solutions take into account the case of choreographed services.

In particular, we extended Web Service execution environments to include features that are useful to support the *diagnostic/fault recovery process*. Then, an architecture supporting self-healing service execution has been defined.

The architecture provides support for associating a diagnostic service with each application service, for gathering observations about service execution (e.g., data exchanged between services) and for recovery and repair actions. The architecture also includes a Monitoring Service for QoS problems, and a Repair Module supporting the execution of recovery plans on the basis of the diagnostic information. For QoS, attention is paid to temporal aspects and the monitoring of temporal constraints. We have characterized diagnosis and repair for Web Services, defining catalogue of faults and possible observations, and have proposed an architecture for the surveillance platform. The core of the platform is a diagnostic problem solver with algorithms for performing the diagnostic process, focusing on functional faults. The diagnostic architecture is decentralized: a diagnoser is associated with each individual service, while a supervisor is then associated with the orchestrator service or with the process owner [9]. We defined a communication protocol between local diagnosers and the supervisor, assuming that no knowledge about the internal mechanisms of a Web Service is disclosed by its local diagnoser. A global diagnosis can be computed by the supervisor after exchanging information with local diagnosers (invoking only those that are relevant to solve the specific problem under analysis). The correctness of the algorithms has been proved formally.

Repair has been then characterized as a planning problem, whose goal is to build the plan of the recovery actions to be performed to recovery from errors. The actions and plans are those described in detail in Section 3.

The WS-Diamond architecture is centred around a *self-healing layer* defined as a set of extensions to a Service Execution Engine designed in the project to enable monitoring, diagnosis and repair. The so called Diagnosis “Diamond” Services includes:

- Alarms and events generated by the service go to the “Diamond” (to Local Diagnoser)
- Local Diagnoser owns privately the model of the service and is in charge of explaining alarms (events) by either
 - explaining them with internal faults
 - blaming other services (from which inputs have been received) as the cause of the problem
- The Local Recovery Execution module receives recovery actions to be performed from the Global Recovery Planner. It is also admitted that repair actions are selected by the Local Diagnoser.
- Recovery actions are passed to the Self-healing layer.

The “Diamond” associated with the process orchestrator comprises:

- A Global Diagnoser and Recovery Planner
- A Local Diagnoser and Local Recovery Execution module

Figure 1 shows the overall architecture of WS-Diamond, with interactions among a service (and its diamond) and the orchestrator (and its diamond). In particular, it shows:

- two ways interaction between Local Diagnoser(s) and Global Diagnoser to compute a global diagnosis in a centralized way
- sequential interaction between Global Diagnoser and Recovery Planner
- one way interaction from the Recovery Planner to the local recovery execution module(s).

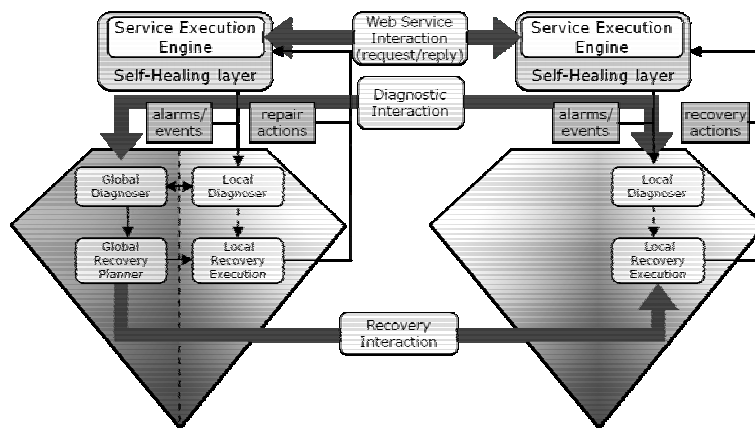


Figure 1: WS-Diamond Overall Architecture

In the self-healing layer, anomalous situations may become evident at run time as the inability to provide a service, or to fulfil a contracted QoS. To recover from these situations, various actions can be undertaken, depending on *where* the fault has occurred in the application flow, and on the type of fault (blocking, inter-application, intra-application, due to missing data or to faulty actions performed by human actors, and so on).

At this level, methodologies and tools are provided for designing self-healing services, supporting service design and execution mechanisms [12]. Moreover, services are of managed type: a *Management Interface*, built according to WSDM standards [13] describes which business operations (e.g., repair actions) can be defined on which state of the application/service execution.

3. An Application Scenario

To assess the project, an e-commerce scenario was selected, namely a FoodShop company that sells typical food products on the Web. The company has an online shop and several warehouses (WH1, ..., WHn) located in different areas, responsible for stocking unperishable goods and delivering items to customers. Customers interact with the FoodShop Company by placing their orders, paying the bills and receiving goods. In case of perishable items, which cannot be stocked, or in case of out-of-stock items, the FoodShop Company must interact with one or more Suppliers (SUP1, ..., SUPm). The business process, executed through cooperation of several services, covers the whole path from the customer order to the parcel delivery. 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. The business process includes activities carried out by humans, such as the preparation of the supply package. However, we assume that these activities have an electronic counterpart (a so called “wrapper”) in the Web Services, whose goal is to track the process execution. The diagnosis approach specifies the execution of distributed conversations among local and global diagnosis services. Details about the strategies adopted for diagnosis can be found in [11], where also properties about the correctness of the adopted algorithms are proved. In this paper, we focus on the repair phase, assuming that the diagnosis steps are executed according to such framework. We assume that the Shop is the process and each service has its own local database, where faulty data can exist.

3.1 Framework and Definitions

In general, a process P based on Web Services in an organization invokes both operations of *internal services*, performing business process logic, and operations of *external services*, which are invoked sending messages and receiving response messages synchronously or

asynchronously. In this paper, we consider that the 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 on-line or pre-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. 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* an action (invoking an operation which is defined as a compensation for a given one in a given state),
- *substitute* a service.

In short, *retry* addresses temporary faults of services, *redo* addresses temporary faults on data (e.g. a wrong item code), *compensate* can be a complex sequence of actions aimed at rolling back to a safe process state, and finally *substitute* addresses permanent faults (e.g. 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. Faults can be either *Permanent* or *Temporary*. If a fault is permanent, invoking the same operation of the service again will result again in a failure. If a fault is temporary, re-invocation of the operation originating the erroneous message in the $F = \langle S, M \rangle$ pair may result in a correct message.

3.2 Sample Failures and Faults

In Figure 2, we show the Web Services of the FoodShop example, interacting to fulfil an order from a customer. For the sake of simplicity, we consider that 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 2) has originated the failure, and then a repair action or plan is executed.

In the example, we consider the following elements:

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

We work under the assumption of single faults, and that retry and substitute actions are always successful. Considering the *failures*, the scenario assumes the following ones:

Fai1: ForwardOrder results in a fault message from WAREHOUSE

Fai2: Check&Reserve results in fault message from SUPPLIER (item_code does not exist).

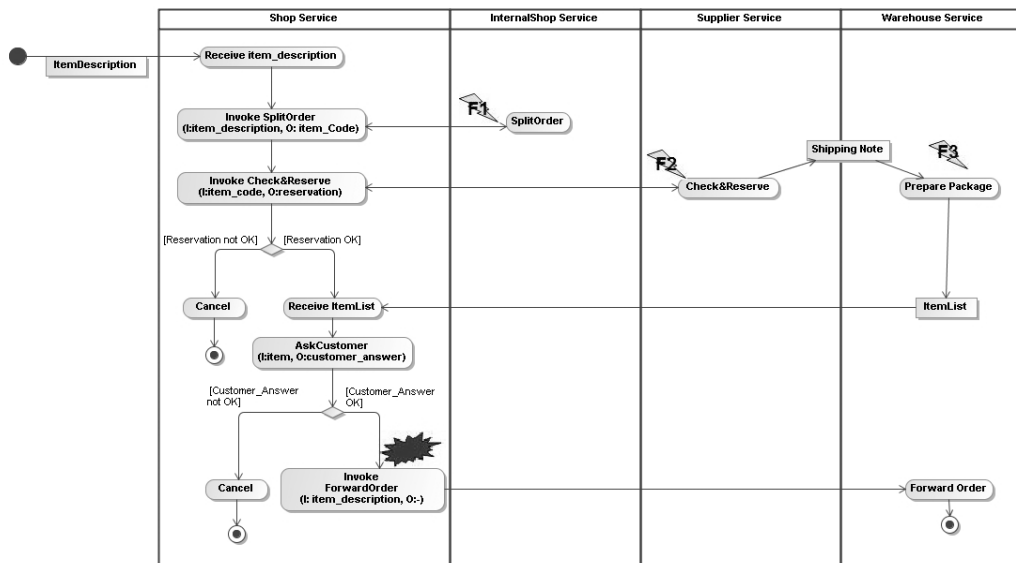


Figure 2: UML-like Representation of the Example Scenario: Interacting Web Services, Symptoms and Faults

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

Possible faults $F = \langle S, M \rangle$

F1 = $\langle \text{SHOP}, \text{Item_code} \rangle$

F2 = $\langle \text{SUPPLIER}, \text{reservation} \rangle$

F3 = $\langle \text{WAREHOUSE}, \text{ItemList} \rangle$ - the WAREHOUSE fills in 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 fault in the SUPPLIER service. This service has a wrong correspondence between the item name 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 the ordered goods (item_description in Figure 2), the shipping note, and the package are not consistent, thus raising the failure *Fail*. However, exactly the same failure can occur also in case the SHOP sent the wrong code to the supplier, and in the case the warehouse prepared the wrong package.

When coming 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 in the three cases. For instance, if the error were in the SHOP, the shop can send the correct data to the supplier and compensate affected actions (see for instance plan P1 below), while if the 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.

3.3 Execution and Repair Actions in Plans

Table 1 summarizes the possible situations of faults in the various Web Services of the example. Also a fault type characterization is given.

Only the case in which a wrong code corresponds to an existing goods item is shown. Otherwise, the failures Fai2 and Fai3 are easier to be diagnosed and repaired (a fault message is raised immediately upon the fault). Four different plans can be generated,

depending on the fault and on the infected Web Services. In Table 2, the possible repair plans are reported. Then, the repair strategies that can be adopted are reported in Table 3.

Table 1: Faults Characterization

Correct execution	F1 (SHOP)	F2 (SUPPLIER)	F3 (WAREHOUSE)
Receive (lasagna) SplitOrder (lasagna, O4) Check&Reserve (O4) Prepare-package (O4,lasagna) Package: lasagna ForwardOrder: lasagna, OK	Receive (lasagna) SplitOrder (lasagna, O5) Check&Reserve (O5) (IF item-code not existent <i>failure Fai2</i>) Prepare-package (O5,spaghetti) Package: spaghetti ForwardOrder: lasagna, <i>Failure Fail</i>	Receive (lasagna) SplitOrder (lasagna, O4) Check&Reserve (O4) Prepare-package (O4,spaghetti) (IF O4, spaghetti not existent or incoherent <i>failure Fai3</i>) Package: spaghetti ForwardOrder: lasagna <i>Failure Fail</i>	Receive (lasagna) SplitOrder (lasagna, O4) Check&Reserve (O4) Prepare-package (O4,lasagna) Package: spaghetti ForwardOrder: lasagna <i>Failure Fail</i>
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 2: Four Possible Repair Plans

P1	P2	P3	P4
Repair after SplitOrder Change value: item-code=O4 Compensate(Check&Reserve) Redo Check&Reserve(O4) IF reservation=OK Retry ForwardOrder(lasagna, OK) Resume after ForwardOrder OTHERWISE Compensate(AskCustomer) Compensate(ForwardOrder) Invoke Cancel	Repair after SplitOrder Compensate(Check&Reserve) Substitute SUPPLIER Invoke Check&Reserve(O4) IF reservation=OK Retry ForwardOrder (lasagna, OK) Resume after ForwardOrder OTHERWISE Compensate(AskCustomer) Compensate(ForwardOrder) Invoke Cancel	Retry ForwardOrder (lasagna)	Repair after SplitOrder Change value: item-code=O4 Compensate(Check&Reserve) Redo Check&Reserve(O4) Resume after Check&Reserve

Table 3: Repair Strategies

Failure	Fault	Plan
Fai1	F1: SHOP, item-code	P1
Fai1	F2: SUPPLIER, reservation	P2
Fai1	F3: WAREHOUSE, ItemList	P3
Fai2	F1: SHOP, item-code	P4

4. Conclusions and Future Work

In this paper, we have presented the WS-Diamond approach to self-healing Web Services 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.

Designing self-healing Web Services requires being able to evaluate the joint degree of diagnosability and reparability of the services [14]. Reparability is a property of a faulted composition of Web Services stating they can be repaired, i.e., repair goals can be achieved by synthesizing repair plans [15]. Since diagnosability and reparability are not inherent properties of a system, WS-Diamond aims at analyzing the requirements for self-healability depending on the way the system is designed. This means to analyze the available service descriptions, the composition mechanisms and the description provided for composition, the communication language and protocols supporting complex conversations (extending proposals such as [16]), the available observations, and the adopted repair actions and

diagnostic algorithms. Self-healability is hence defined as a joint property that achieves a bridge between diagnosability and reparability [17]. The second phase of WS-Diamond is devising guidelines for designing services that can be easily diagnosed and recovered during execution [18]. Other directions of work are: operational methods for checking diagnosability [19], identification of factors that influence reparability and reparability degrees, and identification of monitors and repair handlers for self-healability.

Acknowledgement

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

References

- [1] M.P. Papazoglou, P. Traverso, S. Dustdar, F. Leymann, Service-Oriented Computing Research Roadmap, ftp://ftp.cordis.europa.eu/pub/ist/docs/directorate_d/st-ds/services-research-roadmap_en.pdf.
- [2] M. Fugini, E. Mussi. Recovery of Faulty Web Applications through Service Discovery. 1st International Workshop on Semantic Matchmaking and Resource Retrieval (SMR 2006), Seoul, Korea, 2006.
- [3] L. Baresi, S. Guinea and P. Plebani. WS-Policy for Service Monitoring. Technologies for E-Services. 6th International Workshop, TES 2005, Trondheim, Norway, pp. 72-83, 2005.
- [4] CoreGRID European Research Network, web site: <http://www.coregrid.net/>
- [5] W. Bausch, C. Pautasso, G. Alonso. Programming for Dependability in a Service-based Grid. 3rd IEEE International Symposium on Cluster Computing and the Grid, Tokyo, Japan, 2003, pp. 164-171.
- [6] B. Pernici (Editor). Mobile Information Systems, Springer, 2006.
- [7] W. M. P. van der Aalst. Matching observed behaviour and modelled behaviour. An approach based on Petri nets and integer programming, in Decision Support Systems, Vol. 42(3), 2006, pp. 1843-18592.
- [8] IBM, Autonomic Computing: IBM's Perspective on the State of Information Technology. <http://www-1.ibm.com/industries/government/doc/content/resource/thought/278606109.html>.
- [9] A. Erradi, P. Maheshwari, V. Tosic. Policy-Driven Middleware for Self-adaptation of Web Services Compositions. 7th International Middleware Conference, Melbourne, Australia, 2006, vol. 4290, pp. 62-80.
- [10] G.C. Fox, D. Gannon. Workflow in Grid Systems. Concurrency and Computation: Practice and Experience, 18(10), 2006, pp. 1009-1019.
- [11] L. Console, C. Picardi, D. Theseider Dupre'. A Framework for Decentralized Qualitative Model-Based Diagnosis. In Proc. of the Twentieth International Joint Conference on Artificial Intelligence, IJCAI-07. Hyderabad, India, 2007, pp. 286-291.
- [12] S. Modafferi, E. Mussi, B. Pernici. SH-BPEL - A Self-Healing plug-in for Ws-BPEL engines. Middleware for Service Oriented Computing (MW4SOC) Workshop of the 7th International Middleware Conference, November 2006, Melbourne, Australia.
- [13] W. Vambenepe. Web Services Distributed Management: Management Using Web Services (MUWS 1.0) Part 1 and Part 2 OASIS, 2005.
- [14] M.-O. Cordier, L. Travé-Massuyès, X. Pucel. Comparing diagnosability in Continuous and Discrete-Event Systems, in 17th International Workshop on Principles of Diagnosis DX'06, Spain, 2006, pp. 55-60.
- [15] H. Asher, H. Feingold. Repairable Systems Reliability: Modelling, Inference, Misconceptions and Their Causes, Journal of the American Statistical Association, 1987, vol. 82 (397).
- [16] C. Ferris, D. Langworthy. Web Services Reliable Messaging Protocol, 2005.
- [17] M. Cordier, Y. Pencilé, L. Travé-Massuyès, T. Vidal. Self-Healability = diagnosability + reparability, Proc. of the 18th International Workshop on Principles of Diagnosis, DX'07, May 2007 Nashville, USA.
- [18] D. Ghosh, R. Sharman, H.R. Rao, S. Upadhyaya. Self-healing systems: survey and synthesis, Decision Support Systems, Vol 42(4), 2007, pp. 2164-2185.
- [19] X. Pucel, S. Bocconi, C. Picardi, D. Theseider-Dupré, L. Travé-Massuyès. Diagnosability analysis for Web Services with constraint-based models, Proc. of the 18th International Workshop on Principles of Diagnosis, DX'07, May 2007 Nashville, USA.

ⁱ Other contributors include: L. Ardissono, S. Bocconi, R. Furnari, A Goy, G. Petrone, C. Picardi, M. Segnan and D. Theseider Dupre of Dipartimento di Informatica, Università di Torino, Italy; C. Cappiello, S. Modafferi, E. Mussi, B. Pernici and F. Ramoni of Politecnico di Milano; G. Friedrich and V. Ivanchenko of Universität Klagenfurt; K. Guennoun, Y. Pencole, X. Pucel, A. Subias and L. Travé Massuyès of LAAS-CNRS, Université de Toulouse; M.O. Cordier, X. Le Guillou and T. Vidal of IRISA Université Rennes 1 and J. Eder of Universität Wien.