Towards a Self-Healing approach to sustain Web Services Reliability.

Mohamed Hedi Karray, Chirine Ghedira, Zakaria Maamar

Towards a Self-Healing Approach to Sustain Web Services Reliability

Mohamed-Hedi Karray
Femto-st Institute, Besançon, France
hedi.karray@femto-st.fr

Chirine Ghedira
Lyon 1 University, Lyon, France
chirine.ghedira@liris.cnrs.fr

Zakaria Maamar
Zayed University, Dubai, U.A.E
zakaria.maamar@zu.ac.ae

***Abstract*—Web service technology expands the role of the Web from a simple data carrier to a service provider. To sustain this role, some issues such as reliability continue to hurdle Web services widespread use, and thus need to be addressed. Autonomic computing seems offering solutions to the specific issue of reliability. These solutions let Web services self-heal in response to the errors that are detected and then fixed. Self-healing is simply defined as the capacity of a system to restore itself to a normal state without human intervention. In this paper, we design and implement a self-healing approach to achieve Web services reliability. Two steps are identified in this approach: (1) model a Web service using two behaviors known as operational and control; and (2) monitor the execution of a Web service using a control interface that sits between these two behaviors. This control interface is implemented in compliance with the principles of aspect-oriented programming and case-based reasoning.**

Keywords- Web service; reliability; self-healing; case-based reasoning; AOP.

I. INTRODUCTION

We all agree that the Web is dynamic by nature; new services are offered, some services cease to exist without prior notice, new business opportunities arise, etc. This nature puts a lot of pressure on those who are in charge of developing business applications that should be loosely coupled and spread over enterprises' organizational boundaries. Making Web services the technology of choice upon which these applications could be built would require looking into some issues, with emphasis on reliability in this paper, that still hinder the acceptance of Web services. Reliability is the ability to perform independently of the current execution circumstances, which permits to guarantee business continuity. To address Web services reliability, several works are reported in the literature [1]. Recently, self-healing seems leading these solutions [2,3]. In information technology, "*self-healing describes any device or system that has the ability to perceive that it is not operating correctly and, without human intervention, make the necessary adjustments to restore itself to normal operation*"[1].

In [4], we started examining the reliability of Web services through the use of two behaviors, which we denoted by control and operational. Both behaviors are specifically used to specify the functioning of a Web service and comply with separation of concerns principle. By doing this, the development and maintenance of Web services is made simple. On the one hand, the operational behavior illustrates the business logic that underpins the functioning of a Web service, i.e., how the functionality of a Web service is achieved. On the other hand, the control behavior guides in a controlled way the progress of executing the operational behavior (i.e., business logic) by stating the actions to take and the constraints to put on this progress. In this paper, we capitalize on both behaviors to let Web services self-heal and hence, achieve the reliability of the business applications they implement. Mainly injecting self-healing mechanisms into Web services should help in discovering, diagnosing, and reacting to disruptions that affect Web services operation [5]. We discuss how our self-healing Web services are developed and oversee the progress of both behaviors towards completion.

Enhancing a system with self-healing capacities could be based on internal or external mechanisms [6]. The former refer to trapping errors (includes exceptions: we consider an exception as an exceptional error) when happened like modern programming languages (e.g., Java exceptions, assertion checking) and run-time libraries (e.g., timeouts for RPC) do. The latter refer to monitoring a system using some "outsider" components (e.g., monitoring, recovery, etc.) that determine when a system's behavior is acceptable and whether self-healing should be initiated or not.

Given the black box nature of a Web service, its implementation details are only known to those who took part in its development. Therefore, these persons would be in charge of developing the self-healing functionalities as well [7]. Since the external components' features (monitoring, recovery, etc.) are more attractive and effective [6,8], we adopt an hybrid approach that combines the benefits of both internal and external mechanisms, by first, separating the external features of self-healing (monitoring, recovery, etc.) from the execution of the Web service (which is not the case with, for instance, Java exceptions), and second, encapsulating these features into modules that run internally and in parallel with the execution of the operations of the Web service.

In this paper, we propose a self-healing approach that is built upon a set of dedicated modules that would support the reliability of Web services. These modules are part of a "control interface'' that ensures the monitoring of a Web service's behavior, the catching of errors, and the recovery from these errors. The design of this control interface complies with Aspect-Oriented Programming (AOP) and Case-Based Reasoning (CBR) to benefit from the dynamic weaving principles and previous similar recovery cases in order to use previously adapted solutions.

[1] (www.bitpipe.com/tlist/Autonomic-Computing.html)

Section 2 presents some related work on Web services self-healing. Section 3 suggests a motivating example to highlight the run-time errors problem that hinder Web services execution Section 4 presents our approach to set up the control interface. Section 5 discusses our implementation. A brief discussion about the approach is presented in section 6. Finally, Section 7 concludes the paper and sets guidelines for future work.

II. RELATED WORK

To make Web services the technology of choice when developing critical applications, it is important to enhance them with mechanisms that guarantee continuity of operations despite failure. Self-healing is among these mechanisms and could fall into the research theme of reliability as reported in [12]. We identify two categories of works on the topic, works that are based on models and works that based on intelligence and technology.

Concerning the first category, we find the work presented by Dabrowski et al. who use architectural models to characterize how different elements such as architecture, topology, consistency-maintenance mechanism, and failure-recovery strategy could contribute to self-healing during communication failure [14]. In this specific failure and using notification as a consistency-maintenance mechanism, the authors divided self-healing properties into recovery techniques and topology. In [3], Ben Halima et al. propose a self-healing framework that is capable of managing Web services-based distributed interactive applications. This framework focuses on QoS monitoring and uses models for QoS analysis. In fact, the framework considers the communication level monitoring while intercepting exchanged SOAP messages and extending them with QoS parameter values. Glosh et al. classify self-healing systems based on similarities or relationships between approaches, mechanisms, architectures and technologies applied in these systems. In this classification, the authors indicate that such systems use models, whether external or internal, to monitor behaviors so these systems can adapt themselves to the run-time environment [13].

Regarding the second category, Gurguis et al. present an approach to achieve the autonomic computing of Web services. They divide Web services into functional Web services providing computing functionalities over the Internet, and autonomic Web services encapsulating atomic attributes such as self-configuration, self-healing, and self-optimization [2]. Monatni and Anglano present a CBR approach for providing large-scale, distributed software systems with self-healing capabilities [18]. However, the approach does not use structured knowledge such as models of the system behavior, thus easing its applicability to large-scale in complex software systems. Friese et al. present the design and implementation of a Robust Execution Layer that acts as a transparent, configurable add-on to any BPEL4WS execution engine to support self-healing business processes. The continuity of process execution is achieved through service replacement in case of communication failures [16]. Baresi and Guinea identify and classify the major faults in service-oriented systems and draw some solutions that allow recovery strategies using pre and post-conditions for the required and provided operations based on service and process description via BPEL [15]. As for us, our approach is in another context. Indeed, given the black box nature of the web services, knowing the service behaviors (i.e., control and operational) is essential.

The EU SHADOWS can be considered as an hybrid work classified into the two categories. It concentrates on self-healing of complex systems using a model base approach [19]. The project introduces pioneering technologies such as the automatic concurrent debugging and the data race detection to enable the systematic self-healing of failures classes and an approach to the integration of several self-healing technologies into a common framework solution. In this approach a game-based model-checking technique is used to the verification and the adaptation task: the system acts as a player in a hostile world. Based on this model, if anything goes wrong, the system adapts its behavior to accomplish its task in a different way. The system doesn't try to recover the confronted problem. In our approach, when the service catches any error, it tries to recover the error in the aim to not change its behavior.

III. MOTIVATING SCENARIO

We choose WeatherWS whose functionality is to return a 5-day weather-forecast report on a certain city. In [4], we argue why this Web service is not simple and can be used to illustrate different types of errors.

We assume that WeatherWS requires two inputs: city name and date. At run-time, one of the following cases happens knowing that WeatherWS searches the city's name in a dedicated database: (1)The access to the database fails; (2)The city requested does not exist in the database; (3)The city requested exists in the database; WeatherWS submits a report to the user.

The execution of WeatherWS can be reflected using different states. We refer to these states as business and use them to form the operational behavior of a Web service. "City located", "weather collected", and "report delivered" are examples of business states in the operational behavior of WeatherWS. This latter passes from one state to another subject to first, completing the operations that are included in each state and second, satisfying the transitions that connect states. To follow up the execution progress of a Web service, we use the control behavior along with a specific set of states that are extracted from the field of transactional Web services.

To achieve self-healing Web services, we look into the interactions occurring between these two behaviors. Different types of failure could lead to self-healing such as bugs in the business logic and resource removal.

IV. OUR PROPOSED SELF-HEALING APPROACH

Our proposed self-healing approach for Web services takes place over two steps: how to model the behaviors of a Web service, and how to set up the self-healing mechanisms in terms of operation safety and error recovery at run-time. The identification of these mechanisms is built upon the closed-loop of Garlan et al. in the control system paradigm [6] which consists to monitoring, interpretation, resolution and adaptation.

A. Behavior modeling

A behavior illustrates the actions that a Web service takes in response to some event occurrence and condition satisfaction. In [4], the operational behavior shows the business logic that underlies the functioning of a Web service, and the control behavior guides the execution progress of the business logic (i.e., operational behavior) of this Web service. As briefly reported in Section 2, the control behavior uses a set of states that are reported in the literature of transactional Web services [9]. The complete list of these states is as follows: ‘‘activated’’, ‘‘not-activated’’, ‘‘done’’, ‘‘aborted’’, ‘‘suspended’’, and ‘‘compensated’’. The state chart diagrams is use to represent both behaviors.

In addition to the control and operational behaviors, Maamar et al. developed mechanisms that support the interactions between them. These mechanisms are used to convey details from one behavior to another and *vice versa*. For example, a message from the control to the operational behaviors carryies a temporal event that permits to trigger the execution of a Web service.

The use of state chart diagrams to model the control and operational behaviors shows what should happen at run-time but does not permit to follow up the execution progress at the operation level. In fact, questions like which operation was recently executed, what dependency exists between operations, and what operation failed cannot be tracked if state chart diagrams are used. Any self-healing exercise requires a clear access to the operations that were executed and the operations that encountered problems [17]. To overcome this limitation of state chart diagrams, we decided to use activity charts to model the operations of a Web service.

After modeling both behaviors, the next step consists of mapping some states in the control behavior of a Web service onto other appropriate states in the operational behavior as discussed in [4].

The control and operational behaviors of a Web service are based on a finite set of sequences. In [4], these sequences are called path and defined as follows: A path $p_{i\rightarrow j}$ in a Web service behavior B is a finished sequence of states and transitions starting with state S_i and finishing at state S_j noted: $p_{i\rightarrow j}= s_i\rightarrow(l_i)\ s_{i+1} \rightarrow(l_{i+1})\ s_{i+2}\ldots s_{j-1}\rightarrow(l_{j-1})\ s_j$; such that $\forall k \in \{i, j-1\}:(s_k, l_k, k+1) \in T$.

In the other hand, we define an execution scenario in a Web service as the association of a control state and a path in the operational behavior along with an execution priority defined by the developer of the Web service. This priority defines the recommended paths that need to be executed in order to satisfy a user’s needs. Scenarios having the smallest value are considered as the more adequate to meet a user’s expectations.

Also, a function ***Next*** was defined in [4]. This function specifies in which control state the Web service must go after taking a final state in the operational path. We redefine this function to specify the control state that needs to be taken following the execution of a scenario.

Scenarios and Next function are used to oversee the progress of the execution of a Web service. Our self-healing approach relies on the interactions that exist between behaviors and is built upon a control interface that drives these interactions.

B. Control Interface

Like any other program, Web services may be subject to events that could affect their normal execution progress. Our self-healing approach is based on a control interface (see Fig.1) that contains the following modules: monitoring, interpretation, resolution, and adaptation. These modules support synchronization, verification, detection, and recovery.

In the control interface, the **Mapping Module (MAM)** is a repository of XML schemas and XML data that result from the mapping between the control and the operational behaviors. The MAM contains, also, additional elements such as matching paths, execution scenarios, and the results of the *Next* function. In addition, the MAM provides data regarding the expected behaviors during the execution of other modules to (i) instantiate the conversations between the two behaviors by the **Conversation Management Module (CMM)** or (ii) create recovery by the **Error Recovery Module (ERM)** in case of error.

In the control interface, the CMM instantiates, manages, and checks the conversational messages that are exchanged between the two behaviors. The CMM collects the scenario execution priority based on the current state in the control behavior of the Web service and initiates conversations with each state in the operational path that is included in this scenario. In this work we adapt the conversational messages that are reported by Maamar et al in [4] like *Sync*, *Success, Fail, Ack, just to cite some*.

Through the management and monitoring of the different conversational messages that are exchanged, the CMM catches errors that interrupt the normal progress of the execution of a Web service. These errors are usually detected using *Fail* message.

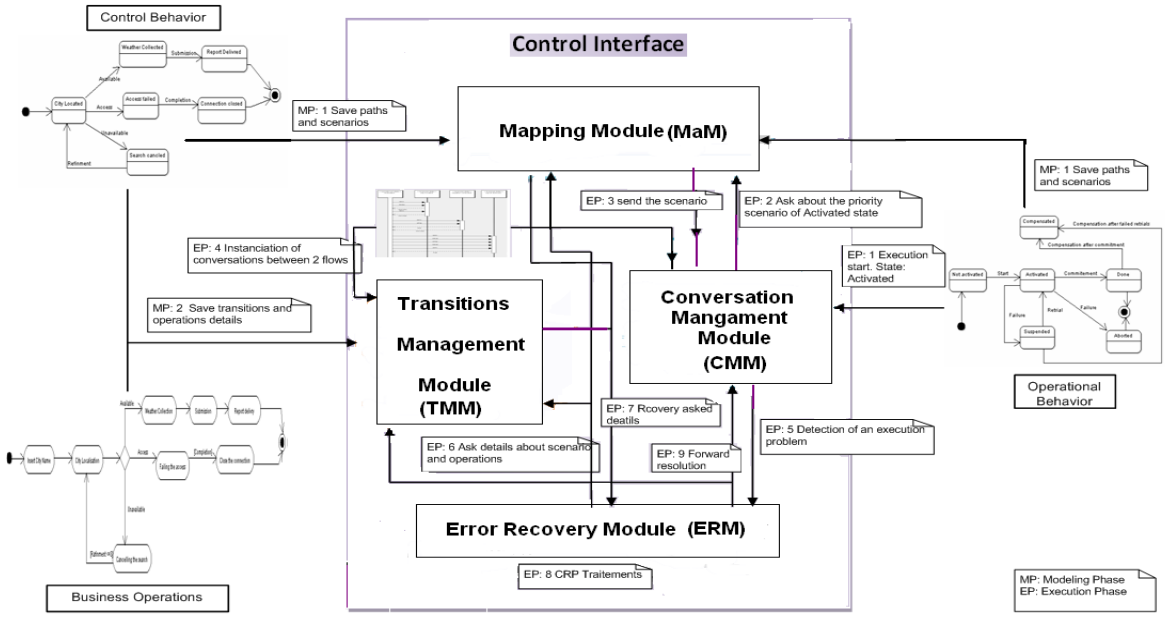

Fig.1. Self-healing architecture

The CMM keeps track of different elements that help a Web service self-heal. These elements include (1) a component that instantiates conversations, (2) a list including all messages types, (3) details of messages related to the conversation in progress such as message Id, message origin, message destination, etc., and (4) log of the previous conversations. It should be noticed that there is a watchdog that monitors the messages of each conversation and raises alerts for the benefit of the CMM when it catches a failure message.

In the control interface, the **Transition Management Module (TMM)** comes into play before claiming the successful execution of a scenario. This claim depends on the operations to execute per state as well as the transitions that connect states. The TMM stores the intra-behavior transitions (i.e., transitions from one state to another in the same behavior) and all the information about the business operations (i.e., constraints, functioning description, implementation and execution). We define this information using pre- and post-conditions and constraints.

It should be noticed that the control interface does not address hardware failure problems that could affect client or server sides. Nevertheless, we designed the CI in a way that it records a Web service's execution states in case interruptions arise due to external events. We provided a buffer part of the TMM.

In the control interface, the ERM receives alerts of execution errors that CMM submits and takes corrective actions in response to these alerts. The implementation of these actions complies with the AOP principles [10] by using the following three components: *Aspects base, Patterns base, and Case base.*

Aspects base. AOP is a paradigm that captures and modularizes concerns that split a software system into modules called Aspects. Aspects can be integrated into a system using dynamic weaving [11]. An aspect contains different code fragments (advices) and location descriptions (pointcuts) to identify where code fragments should be plugged. Our use of AOP is motivated by the dynamic weaving of aspects. An aspect can be enabled and disabled at run-time. In our self-healing approach, we define a base that contains different types of aspects that could characterize the different errors during a Web service execution. These aspects are triggered by an aspect weaver that exists in the TMM when an error arises. We identify aspects with a triple (Name of the module including the aspect, Set of advices, Set of pointcuts).

Patterns base. It is a container of execution patterns for business operations. Each business operation is associated with a set of execution patterns. A pattern is used to decompose an operation into segments according to a certain semantics and implementation constraints. We define three types of patterns. Two are defined at design time (*normal patterns* and *error patterns*) and one at run time.

Case base. It contains cases of errors along with their solutions. Whenever the ERM receives an alert, it creates a new solution by assembling an error pattern with an aspect and records this case in the base if it does not already exist. A case is characterized by the 3-uplet <***Sy, Con, R***> where ***Sy*** presents sets of the problem symptoms and the "error patron"; ***Con*** is the case context, that means which aspect in which business operations; ***R*** is the carried out treatment i.e. resolution made.

When the ERM receives an alert, it recovers the execution scenario from the MAM as well as the details from the *fail* message that is related to the control state. This indicates that there is an execution problem that the CMM reports. First, the CMM starts to synchronize itself with the MAM and TMM to retrieve information related to the current scenario and the operations in the control state that is affected by this error. The ERM consults its base of patterns in order to compare the Log pattern received with "normal'' and "error'' patterns so that the aspect related to this error is detected. If the pattern is already in the database, the ERM consults its base cases to see if a similar error has already been treated and solved. If yes, the ERM sends a solution to the CMM and the TMM for deployment.

Otherwise, if it is not according to this pattern and the base of aspects, the ERM selects the associated aspect. Then the ERM sends the solution to the TMM module to apply it. In the case where the Log pattern does not exist in the error patterns list, the ERM adds this new pattern to the patterns base and sends an alert to the Web Service developer, asking its (new pattern) assignment to one or many aspect. After the application of the solution, the ERM updates its case base.

V. EXPERIMENTS

The feasibility of our self-healing approach was tested by implemenying the control interface. We used C # from .Net beta 2005 platform to program the different modules and XML to define behaviors, conversation messages, execution scenarios, patterns, and last but not least the manipulated data at run time.

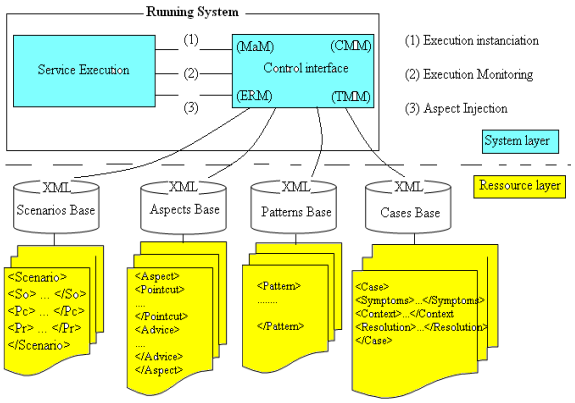

Fig.2. Prototype architecture

Our experiments started by injecting errors to check the mechanisms for error detection and aspect activation. An example of error was an empty object (NULL) that was returned following the execution of "weather collection'' operation and then submitted to another operation namely "report delivery''. Our prototype catches, collects, and locates the error, identifies the appropriate error pattern, and finally determines the aspect that is associated with this error. All this happens without propagating the error to the client. We are now working on the injection of the set of advices related to aspects, to test the reaction of the web service behavior after this modification.

The prototype architecture as Fig.2 presents is decomposed in two layers: "system layer" and "resource layer". The running system consists of two components: "service execution" and "control interface". An internally exchange for instantiation, monitoring and recovery (i.e. aspects injection) is made in the running system between the two components. The control interface is connected to the different XML data bases (Scenarios, Aspects, Patterns, Cases) managed by its modules.

VI. DISCUSSIONS

When it comes to self-healing, several requirements are suggested in the literature such as adaptability, dynamicity, and autonomy [20]. We took into account these requirements while working on our approach to self-heal Web services. For example, the Web services adapt their course of actions in reaction to the errors that are detected. We, also, took into account other guidelines such as failure detection, fault diagnosis, fault healing, and validation [21]. The purpose of these guidelines is to attempt the completeness, soundness, and robustness of any self-healing approach. If one of these guidelines is not satisfied, the usability of the approach can be questioned. The externalization of self-healing mechanisms is generic in dealing with multiple Web services at once. This approach coupled with the black box nature of Web services increases the complexity of making them self-heal independently of any human assistance [18], contrarily to the internal approach that is specified within the Web service itself. This approach relies on the knowledge of a Web service's behaviors. Although most of the aforementioned works adopt an external approach, we adopted an hybrid approach taking advantage of both approaches' benefits through the use of first, the closed loop that the external approach offers [6] and second, the visibility and controllability elements that the internal approach offers [21]. Visibility is the ability to observe states, outputs, and resource usage during a Web service execution. Controllability is the ability to modify inputs and states during service running, to study different behaviors, and what-if situations. Hence, given the black box nature of a Web service, these two functionalities cannot be ensured only by an internal approach. To this purpose, our approach is based on modeling Web service behaviors. It implements a model-based approach, where models of a desired Web service behavior governs the self-healing process throughout the service design and implementation phase and the service deployment phase as used in the EU SHADOWS project [19]. We based our approach on two types of models: *control/operational* and *fault.* The *control/operational model* specifies the nominal behavior that must be satisfied by the Web service and provided by the Web service developers. This model ensures the behavior synchronization in order to facilitate

the monitoring, control of states, and fault localization. The fault model specifies the types of faults that can be identified and repaired by our control interface. We specifically address fault types related to aspects, which were designed and implemented in a separated module in of this control interface.

In term of CBR, our approach differs from the work of Montani and Anglano in [18]. These ones infer new cases from previous resolutions using similarity mechanisms. Contrarily, in our approach we verify if a case is resolved before any similarity reasoning is carried out. In the future, we aim at enhancing this part to handle the cases that require more than one repair action.

Regarding the effectiveness of our approach, we could not evaluate it in this paper which applies only the feasibility and implementation aspects. All other points such as the calculation of metrics will be addressed in future work.

VII. CONCLUSION

Self-healing is one of the important elements that could enhance the reliability of Web services [21]. In this paper, we examined self-healing Web services by first, describing their control and operational behaviors and second, implementing a control interface that oversees the performance of these Web services and takes corrective actions when necessary. This interface was implemented in compliance with the principles of aspect-oriented programming and case-based reasoning.

Our future work revolves around different aspects. Firstly, we will continue enhancing the prototype to conclude additional tests about WeatherWS, and more examples of real Web services will be developed to identify the failures that are not detected as expected. Secondly, we would like to make the control interface ``learn'' new patterns and study failure possibilities using proactive and predictive methods to predict when a failure would occur so that corrective actions are taken. Moreover, we plan to apply our proposed self-healing approach to the composition level by looking into the combination of the respective self-healing mechanisms of component Web services.

REFERENCES

[1] A. Erradi, P. Maheshwari, "A broker-based approach for improving Web services reliability Web Services", ICWS 2005, Proceedings. 2005 IEEE International Conference, july 2005, pp355- 362.

[2] SA. Gurguis, A. Zeid , "Towards Autonomic Web Services:Achieving Self-Healing Using Web Services," ACM SIGSOFT Software Engineering Notes, 2005 - portal.acm.org.

[3] R. Ben Halima, K. Guennoun, K. Drira and M. Jmaiel, " Providing Predictive Self-Healing for Web Services: A QoS Monitoring and Analysis-based Approach," Journal of Information Assurance and Security 3 (2008) pp 175-184.

[4] M. Sheng, Z. Maamar, H. Yahyaoui, J. Bentahar, and K. Boukadi, "Separating Operational and Control Behaviors: A New Approach to Web Services Modeling", IEEE IC magazine, 2009.

[5] M. P. Papazoglou, P. Traverso, S. Dustdar and F. Leymann, "Service-Oriented Computing: State of the Art and Research Challenges," IEEE COMPUTER SOCIETY, 2007.

[6] D. Garlan, B. Schmerl, "Model-based Adaptation for Self-Healing Systems," Proceedings of the first workshop on Self-healing systems, 2002.

[7] P. Koopman, "Elements of the Self-Healing System Problem Space," ICSE Workshop on Software Architectures for Dependable Systems WADS 2003

[8] D. S. Wile, A. Egyed, "An Externalized Infrastructure for Self-Healing Systems," Proceedings of the Fourth Working IEEE/IFIP Conference on Software Architecture (WICSA'04), 2004.

[9] Z. Maamar, N.C. Narendra, D. Benslimane, S. Sattanathan, "Policies for Context-driven Transactional Web Services", Proceedings of The 19th International Conference on Advanced Information Systems (CAiSE'2007), Trondheim, Norway, 2007.

[10] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Videira Lopes, J-C. Loingtier, J. Irwin, "Aspect-Oriented Programming", In European Conference on Object-Oriented Programming (ECOOP), Springer-Verlag LNCS 1241. Finland, June 1997.

[11] C. Bockisch, M. Haupt, M. Mezini, K. Ostermann, "Virtual Machine Support for Dynamic Join points", Proceedings of the 3rd International Conference on Aspect Oriented Software Development -- AOSD 04, Lancaster UK, 2004, pp. 83--92.

[12] R. de Lemos, C. Gacek, A. Romanovsky, "ICSE 2003 Workshop on Software Architectures for Dependable Systems and self-healing," ACM SIGSOFT Software Engineering Notes, 2003.

[13] D. Ghosh, R. Sharman, H. Raghav Rao, S. Upadhyaya," Self-healing systems: survey and synthesis," Decision Support Systems, 2007.

[14] C. Dabrowski, K. Mills, "Understanding Self-healing in Service-Discovery Systems," Proceedings of the first workshop on Self-healing systems, 2002.

[15] L. Baresi, C. Ghezzi, and S. Guinea, "Towards Selfhealing Service Compositions," In Proceedings of the First Conference on the PRInciples of Software Engineering, Buenos Aires, Argentina, 2004.

[16] T. Friese, J. Muller, B. Freisleben, "Self-Healing Execution of Business Processes Based on a Peer-to-Peer Service Architecture," Procidings of The 2nd IEEE International Conference on Autonomic Computing ICAC, Springer, 2005.

[17] A. Carzaniga, A. Gorla and M. Pezzè, "Self-Healing by Means of Automatic Workarounds, Proceedings of the 2008 international workshop on Software engineering for adaptive and self-managing systems", Leipzig, Germany

[18] S. Montani, C. Anglano, Achieving Self-Healing in Service Delivery Software Systems by Means of Case-Based Reasoning, Journal of Applied Intelligence Volume 28 , Issue 2 (April 2008) pp139 - 152

[19] 24. O. Shehory, "A Self-healing Approach to Designing and Deploying Complex, Distributed and Concurrent Software Systems", Lecture Notes in Computer Science, Programming Multi-Agent Systems, 2007, pp3-13

[20] M. Mikic-Rakic, N. Mehta, N. Medvidovic, "Architectural Style Requirements for Self- Healing Systems", Proceedings of the first workshop on Self-healing systems, South Carolina, 2002, pp49 -54

[21] A. Gorla, Towards Design for Self-healing, Fourth international workshop on Software quality assurance: in conjunction with the 6th ESEC/FSE joint meeting Dubrovnik, Croatia, 2007, pp86 –89.