Towards Design for Self-healing

Alessandra Gorla
University of Lugano
via Buffi 13
6904 Lugano, Switzerland
alessandra.gorla@lu.unisi.ch

ABSTRACT

Self-healing mechanisms are increasingly attracting the interest of both industrial and research communities as a way of increasing reliability of software systems, while overcoming technical and cost limitations of classic analysis and testing techniques.

Many recent studies focus on techniques for enabling self-healing mechanisms independently from software design. These approaches are effective, but often limited by early design decisions. In this position paper, we argue that a disciplined design approach can enable a wide and effective range of self-healing mechanisms, thus overcoming many limitations of the current approaches.

We discuss the differences between design for testability and design for self-healing approaches, and we propose the foundation for a new *design for self-healing* methodology.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques; D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Design, Reliability

Keywords

Self-healing, design for self-healing

1. INTRODUCTION

Classic maintenance of software systems implies high costs and long repairing cycles due to the involvement of humans in detecting and repairing faults. Self-abilities enable software systems to automatically detect and repair faults without human intervention, thus reducing both costs and latency. Self-abilities may solve many different classes of problems that span from performance (self-optimization) to security (self-protection), architecture (self-adaptation), configuration (self-configuration) and functionality problems (self-healing) [9, 11].

In this paper we focus on *self-healing* mechanisms, that is mechanisms that detect and repair problems of functionality and services offered by software systems. While the behavior of single components and services is usually easy to test and analyze, often complex software systems suffer from unexpected integration problems that remain latent in the software despite accurate analysis and integration testing, and can lead to subtle run time problems difficult to prevent and repair. Self-healing mechanisms focus on detecting and overcoming such problems.

Although integration and system testing focuses on similar problems [13], they differ from self-healing mechanisms in goals, use and execution conditions: Integration and system testing aim to maximize failure occurrence, to identify and remove as many faults as possible before system deployment, while self-healing mechanisms aim to minimize the impact of failures when they occur in the deployed system; Testing aims to provide information to (manually) remove faults, while self-healing aims to prevent the effects of fault occurrence; Testing provides support to human intervention, while self-healing requires no human involvement; Test suites can be re-executed several times to reproduce failures, while self-healing mechanisms cannot reproduce executions, but try to avoid repeated occurrences of failures.

Although some simple mechanisms can be added to software systems independently from their development, suitable approaches to software design can enable powerful mechanisms that can deal with a large variety of problems. Such design approaches can be referred to as *design for self-healing*.

Design for self-healing introduces features that enable detecting run-time failures and identifying and isolating faulty behaviors of components or services. Due to the differences between testing and self-healing mechanisms, design for self-healing approaches cannot rely on the same assumptions and techniques developed over time for design for testability [1, 12, 5], but require new approaches.

The main goal of this position paper is to outline a research agenda for design for self-healing, by discussing the main requirements for design for self-healing techniques (Section 2), analyzing the limitations of design for testability approaches (Section 3), and proposing an initial set of design for self-healing methods (Section 4).

SOQUA '07 September 3-4, 2007, Dubrovnik, Croatia

2. REQUIREMENTS FOR SELF-HEALING

To detect and repair problems of software functionality and services, self-healing mechanisms must be able to predict or detect failure occurrences, identify their causes, and isolate them to prevent further failures, making sure that prevention mechanisms do not lead to unexpected faulty behaviors. Design for self-healing aims to support each of these main phases, *failure prediction or detection*, *fault diagnosis or localization*, *fault isolation or failure recovery*, and *validation*, by enabling supporting mechanisms. In this Section we summarize the main requirements for each phase to provide the background for discussing design for self-healing approaches.

Failure prediction or detection. Healing mechanisms are triggered by failures. Self-healing mechanisms must either detect or better predict failures. Detecting failures may be enough for some classes of reparable failures, but may be too late for other classes of failures. For example, detecting failing transactions while still keeping transaction information may be sufficient to trigger healing actions that bring the system back to a consistent state, while detecting system crashes may be easier, but not enough if the goal is to guarantee a minimum level of functionality.

Automatically detecting or predicting failures requires systems to be able to identify either explicitly or implicitly deviations from the expected behavior. Systems must (1) be able to observe their behavior by self-monitoring execution without significantly alter either functionality or non-functional properties like response time, and (2) infer actual or potential failures from the monitored data.

Useful self-healing systems do not have to predict all possible failures, but may focus on either detecting or predicting only some classes of failures. Detection and prediction mechanisms may co-exist and deal with different as well as similar classes of failures.

Fault diagnosis or localization. To recover from failures, self-healing mechanisms must localize their causes. While debugging aims to completely identify the faults [15], self-healing mechanisms may not be able to fully diagnose the actual faults automatically, but may be able to localize faults only with some level of approximation, still providing enough information to trigger suitable healing actions [2]. For example, self-healing systems may identify incompatibilities between components or parameters that can lead to failures, but may not be able to localize the actual faults. Although the diagnosis is incomplete, the information may be sufficient to trigger healing mechanisms that prevent calling incompatible components or using incompatible parameters, thus avoiding system failures.

Tracking failures to faults may become very difficult when failures manifest themselves far from the fault occurrences. Fault diagnosis and localization require reducing the gap between fault occurrence and failure manifestation, that is, occurrences of faults should produce revealable effects as soon as possible. Moreover, once failures are predicted they should be prevented before their occurrence without significantly altering system performances, while when detected, their effects should be neutralized before causing irreparable damages to the system. Faults must be diagnosed early enough to trigger failure prevention mechanisms in time. For example, self-healing systems should turn occurrences of faults that cause memory leaks into diagnosable effects early enough to prevent system crashes due to memory exhaustion.

Fault isolation or failure recovery. When faults are diagnosed or localized before the actual failures, healing mechanisms can try to either remove the faults, or, more realistically, isolate them to prevent their effects (*fault isolation*) [6]. When failures cannot be prevented, healing mechanisms can try to partially or totally eliminate their effects (*failure recovery*). For example, invocations of methods with incompatible values for their parameters may be substituted with different invocation sequences that produce the same final effect, thus avoiding the predicted failure. For instance, if a method to add an individual to a set fails when the individual already belongs to the set, a self-healing mechanism may automatically substitute the invocation of the `add` method with the invocation of a `check-for-membership` method followed by the `add` method if the individual is not a member of the set. If the method invocation cannot be prevented and the system fails, e.g., by corrupting the set of individuals, the self-healing mechanism can try to backup to a consistent state of the set.

Isolating faults and recovering from failures require systems to modify their execution by either altering the sequence of statement executions or the state or both. Executions and states may be altered at different levels and with different temporal validity. Alterations may be permanent, when the diagnosis clearly indicates permanent faults that can be eliminated, or temporary, when the diagnosis does not provide enough information to completely localize and correct the faults, or the faults cannot be automatically corrected with the available healing mechanisms.

Validation. Healing mechanisms may produce undesirable effects. Similarly to classic debugging approaches, self-healing mechanisms may suffer from diagnosis errors that may fail in identifying the actual faults, or may produce unexpected side-effects that may cause further failures. Moreover, since in the same system, there may coexist many independent self-healing mechanisms, healing actions may unexpectedly interact causing new failures or leading to system instability. For example, self-healing clients and servers may adapt their interfaces to overcome mutual protocol mismatch problems in an infinite cascade of adaptation actions.

Verifying the adequacy of the healing actions is complicated by the fact that contrary to classic testing and debugging, self-healing systems cannot execute expensive regression test suites to verify the suitability of the corrections. Self-healing systems must be able to verify their behavior without requiring expensive test executions, and must be able to coordinate autonomous mechanisms that may exist at different system level or in different subsystems.

3. DESIGN FOR TESTABILITY

Although not immediately applicable to designing self-healing software systems, design for testability approaches provide interesting hints. In this section, we discuss uses and limitations of the main design for testability guidelines in the context of design for self-healing.

While design for testability is widely applied in hard-

ware with techniques as BIST (Built-In Self Test) and BISR (Built-In Self Repair) [10, 3], the applications to software design are still limited. The main design for testability guidelines require *visibility* and *controllability* [1, 12, 5]. Visibility is the ability to observe states, outputs and resource usage during software execution. It is instantiated in many ways, for example, verbose outputs, event logging, resource monitoring, and run-time verification of design by contract assertions. Controllability is the ability to modify inputs and states during software execution, to study different behaviors and what-if situations. Design for testability is often paired with test automation, and is related to the automatic generation of scaffolding and test oracles.

The main principles of visibility and controllability can be useful also in the context of design for self-healing, but most instantiations for design for testability focus on human visibility (for example, verbose outputs, event logging, resource monitoring), and on controlled executions and re-executions of software, while in the context of self-healing, humans are out of the loop, and executions cannot be repeated for the sake of testing.

Visibility can enable failure detection, fault diagnosis and verification of the behavior after healing, but can be useful when instantiated into inter-system notifications that do not involve humans. We may for example take advantage of modules that expose part of their state in a computer readable form, to enable healing features to detect failures and diagnose faults, but verbose outputs and event logging in human readable form may be of little if no relevance.

Controllability can enable mechanisms for healing or isolating faults, by allowing automatic reschedule of module or service executions to circumvent faults and faulty modules. Controllability mechanisms may for example enable rolling back to correct states, healing corrupted states, or disabling faulty modules, but do not produce benefits when used to re-execute faulty modules to expose the fault for off-line diagnosis, while the system is in operation.

When used during operation, as in the case of self-healing, both visibility and controllability mechanisms must obey strong overhead constraints and must prevent major failures, while during testing, there are no relevant constraints on execution overhead, and exposing major failures is not only allowed, but often desirable to avoid their occurrence after deployment. Thus some design for testability approaches may not only be of little use, but even undesirable when designing self-healing systems.

4. APPROACHING DESIGN FOR S-H

In the former section we have argued that the main design for testability principles, visibility and controllability, continue to be valid for designing self-healing software systems, but call for different instantiations. In this section, we propose some guidelines for enhancing self-healing capabilities of software systems during the design. Self-healing systems require abilities for detecting failures, diagnosing and healing faults, and verifying the changes. Each phase needs different abilities and design approaches.

Failure detection. To enable failure detection, we must be able to efficiently detect requirement violations at run time. The strong limitations to run time overhead make it impossible to check for "global" properties, that is properties that require non-local view of the computation state, and call for "locally verifiable" properties, that is assertions that refer only to the "local" computation state, as in the design-by-contract paradigm.

The design must facilitate accessibility to the information involved in the checks (values of variables, parameters, etc). Design for self-healing may also identify "history" values, that is values of some variables useful in later checks. Such values can be properly stored in "history" variables to make them locally accessible in later checks. We may for example introduce "before-method-call" variables that record the value of the parameters before calling methods, to make such values locally available for checks during and after the method execution.

Self-healing mechanisms may evolve during the life of the software system by learning from system execution. The system may for example trace the causes of an unexpected run-time failure to a given combination of values, and may decide to automatically trace those values to prevent further failures of the same type, or it may identify values useless to trace after fixing some faults. This call for automatically adaptable tracing mechanisms that can dynamically change depending on the values to be traced.

Detecting failures long after fault occurrences complicates diagnosis and healing. For example, when buffer overflows or memory leaks cause failures long after their occurrences, they become very difficult to diagnose and often almost impossible to fix. Failure detection mechanisms must reduce the gap between fault occurrence and failure detection. This can be achieved by introducing design practices that insert probes at convenient granularity levels. In order to detect memory leaks, we may for example inspire from the Purify approach that records and checks the state of memory locations as soon as accessed, thus detecting memory faults when they occur and not much later, when they lead to a visible failure [8].

Fault diagnosis. Self-healing systems aim to reduce both the time overhead required to diagnose faults and the effects of fault propagation. It may be preferable to quickly isolate a faulty component, rather than fully identify the fault after a long and expensive process.

Fault propagation can be limited by design mechanisms that support "system standby", by allowing (1) system execution to be temporary suspended between failure detection and fault healing, and (2) fault propagation analysis, such as data flow dynamic slicing, to identify portions of the code potentially affected by the fault.

Fault diagnosis is supported by design decisions that simplify the reconstruction of recent past execution, and promote separation of concerns, to isolate fault effects.

Fault healing. Healing mechanisms depend on the kind of faults and the class of systems. In this paragraph, we identify a preliminary set of mechanisms that can be adapted to different classes of faults and can influence design decisions.

Once diagnosed, faults can be isolated by activating alternative computation paths built-in at design time, which may be designed as: redundant modules, following the direction of fault tolerant systems, alternative modules, by adapting equivalent scenarios as proposed by Doong and Frankl for testing [7], version rollback, that consists of redeploying former versions of the faulty components, when the faults are diagnosed as caused by the updated versions.

Simpler and somehow less intrusive recovery mechanisms may be based on simple rollback [14] or reboot mechanisms, following atomic transaction design approaches that are common practice in data base design (atomic transactions), and microreboot mechanisms [4] common practice in operating system design.

Validation. Program changes may fail in restoring correct execution either because of bad diagnosis or undesired interactions.

Changes based on bad diagnosis may not address the actual fault and sometimes even affect correct portions of the program. Design can enable easy discovery of such cases by providing mechanisms for recording the faulty behaviors and checking for the absence of such behaviors after changes.

Undesired interactions may involve subsystems that share the same self-healing facilities or subsystems with independent self-healing mechanisms. In the first case, undesired interactions may derive from unexpected side effects, which can be either avoided or isolated by adopting clean design practices that explicitly identify all side effects. In the second case, undesired interactions may derive from independent healing mechanisms that try to heal the same fault. For example, different subsystems may detect failures that derive from exchanging semantically incompatible values, and may independently trigger healing mechanisms to solve the diagnosed incompatibilities, leading to newly incompatible situations.

Simple monitoring and recovery procedures may be based on design mechanisms that prescribe a sort of publish-subscribe mechanisms between independent subsystems to notify changes and thus react properly when detecting interfering changes implemented by independent modules.

5. CONCLUSIONS

Designing self-healing systems poses new challenges and problems. In this position paper, we draft a first set of requirements for design for self-healing techniques, we discuss why the main principle underlying design for testing approaches still hold, but most design for testing techniques do not apply straightforwardly, and we propose a set of guidelines for defining design for self-healing approaches. The research area is open and wide, and we are currently working on instantiating the guidelines on concrete recommendations and design methodologies.

6. ACKNOWLEDGMENTS

The work presented in this paper is supported by the Founds National Suisse de la Researche Scientifique as part of the project PerSeoS: Pervasive self-adaptive software systems.

7. REFERENCES

[1] R. V. Binder. Design for testability in object-oriented systems. *Communications of the ACM*, 37(9):87–101, 1994.

[2] P. Bodic, G. Friedman, L. Biewald, H. Levine, G. Candea, K. Patel, G. Tolle, J. Hui, A. Fox, M. I. Jordan, and D. Patterson. Combining visualization and statistical analysis to improve operator confidence and efficiency for failure detection and localization. In *ICAC '05: Proceedings of the Second International Conference on Automatic Computing*, pages 89–100, Washington, DC, USA, 2005. IEEE Computer Society.

[3] M. Bushnell and V. Agrawal. *Essentials of Electronic Testing for Digital, Memory and Mixed-Signal VLSI Circuits.*, pages 463–488. Kluwer Academic Publishers, 2000.

[4] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot - a technique for cheap recovery. In *OSDI '04: Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, pages 31–44, San Francisco, CA, USA, 2004.

[5] R. Coelho, U. Kulesza, and A. von Staa. Improving architecture testability with patterns. In *OOPSLA '05: Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 114–115, New York, NY, USA, 2005. ACM Press.

[6] R. de Lemos and J. L. Fiadeiro. An architectural support for self-adaptive software for treating faults. In *WOSS '02: Proceedings of the first workshop on Self-healing systems*, pages 39–42, New York, NY, USA, 2002. ACM Press.

[7] R.-K. Doong and P. G. Frankl. The ASTOOT approach to testing object-oriented programs. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 3(2):101–130, April 1994.

[8] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Winter USENIX Conference*, pages 125–136. USENIX Association, January 1992.

[9] P. Horn. Autonomic computing manifesto - IBM's perspective on the state of information technology. IBM Research, October 2001.

[10] N. K. Jha and S. Gupta. *Testing of Digital Systems*, pages 560–679. Cambridge University Press, New York, NY, USA, 2002.

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

[12] B. Pettichord. Design for testability. In *PNSQC '02: Proceedings of the 20th Annual Pacific Northwest Software Quality Conference*, pages 243–270, 2002.

[13] M. Pezzè and M. Young. *Software Testing and Analysis.* John Wiley & Sons, 2007.

[14] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: treating bugs as allergies—a safe method to survive software failures. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 235–248, New York, NY, USA, 2005. ACM Press.

[15] A. Zeller. *Why Programs Fail: A Guide to Systematic Debugging.* Morgan Kaufmann, October 2005.