Automatic Workarounds as Failure Recoveries*

Alessandra Gorla
University of Lugano
Faculty of Informatics
Lugano, Switzerland
alessandra.gorla@lu.unisi.ch

ABSTRACT

Mechanisms to automatically recover from problems are key elements to designing self-managed software systems. So far most research on self-managed systems focused on non-functional problems, such as architectural mismatches, performance problems and configuration incompatibilities.

In our work, we focus on techniques for automatically recovering from functional failures. We aim to exploit the intrinsic redundancy of many complex software systems that can produce the same results in several ways. We are investigating techniques that, in case of failure, look for execution sequences that are equivalent to the one that leads to the failure, and can thus be executed in alternative to the failing one to produce the expected results. We refer to sequences that are equivalent to failing ones and can recover from failures as *automatic workarounds*.

The contribution of the thesis will be the definition of techniques to automatically generate equivalent sequences from different models, and to identify automatic workarounds within sets of equivalent sequences, and a thorough evaluation of the techniques in the context of self-healing software systems.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Reliability*; D.2.5 [**Software Engineering**]: Testing and Debugging—*Error Handling and Recovery*

General Terms

Reliability

Keywords

Self-healing, autonomic computing, workarounds, equivalent sequences, fault recovery

*This work is supported by the project PerSeoS funded by the Swiss National Fund

FSE-16 Doctoral Symp., November 10, Atlanta, Georgia, USA

1. RESEARCH PROBLEM

Many modern software systems are increasingly complex and hard to manage with classic engineering approaches. Autonomic and self-managed techniques aim to reduce the cost of software failures and increase software reliability by providing mechanisms to automatically recover from runtime problems [15, 16, 18].

Autonomic and self-managed approaches cope with several types of problems that span from performance (*self-optimization*) to security (*self-protection*), architecture mismatches (*self-adaptation*), configuration (*self-configuration*) and functional failures (*self-healing*).

In our work, we focus on techniques for healing from functional failures (*self-healing*). Such techniques must automatically *detect failures*, that is identify wrong behaviors of the software system, *diagnose faults*, that is localize the causes of failures, *recover from faults*, that is repair the problems, and *verify* that the system behaves as expected after repairing.

Within self-healing approaches, we investigate fault recovery mechanisms. Fixing faults, that is modifying the incorrect statements in the code, can be very hard without human judgment. We do not aim to automatically fix faults, but we investigate recovery techniques that automatically mask the faults, and thus avoid failures without modifying the actual code.

Some software systems are redundant, in the sense that the same functionality can be obtained through different sequences of operations. We call different sequences of operations that implement the same functionality *equivalent sequences*. Our research aims to exploit the intrinsic redundancy of many software systems to automatically derive *workarounds* that can produce correct results by avoiding the execution of faulty components.

We assume the availability of a model that may be derived from software specifications or correct software executions [19, 7], the presence of a failure detection mechanism that can signal failures, and a roll-back mechanism that can bring the software systems back to a consistent state.

Our research hypothesis is that we can identify equivalent sequences from models, and we can select effective workarounds by matching the set of equivalent sequences with the failing execution sequence that is identified by the failure detection and the roll-back mechanisms.

The main expected contribution of our research will be the definition and experimentation of a fault recovery mechanism based on the automatic identification and execution of workarounds to mask faults. This includes the defini-

tion of techniques for automatically identifying equivalent execution sequences based on different kinds of models, and techniques for dynamically selecting sequences that represent effective workarounds of software faults.

2. BACKGROUND AND RELATED WORK

In the context of self-managed systems, many of the approaches proposed so far focus on software architecture problems, and in particular on how systems should adapt to meet environment changes [22], to solve architectural mismatches [6, 8], and to fix architectural faults [12]. Other approaches deal with performance problems, mainly focusing on self-optimization of component based systems [9] and Web applications [21].

Techniques to automatically recover from functional faults have been historically investigated in the context of fault tolerant systems [23, 17]. Most fault tolerant approaches either imply high development costs or target specific classes of failures. For example, *n-version programming* techniques, that rely on multiple versions of the same software subsystem designed and implemented by different engineers to reduce the probability of coincidental failures, increase development costs. On the other hand, *rebooting* and (more efficient) micro rebooting techniques, that periodically reboot (part of) the system, target specific classes of failures, namely non deterministic failures, like the ones due to memory leaks [2, 3, 24].

In the context of self-managed software systems, little work has addressed functional faults so far. Fuad et al. [11] save the system administrator's recovery actions in a repository, add wrappers to Java classes to catch run time exceptions and look for suitable recovery actions in the recovery repository when a failure occurs. This work shares our goals, but relies on recorded human actions, that may not be available in several important classes of systems.

Other self-healing approaches rely on registries containing sets of rules that code failures and corresponding recovery actions. Examples of these approaches are Dynamo [1] and SH-BPEL [20]. Dynamo monitors BPEL processes, checks assertions on functional and non-functional requirements, and executes the recovery actions specified in the registry. SH-BPEL offers the possibility of specifying simple recovery actions like alternative services, or process restart, as well as recovery actions in the form of alternative execution sequences. In SH-BPEL alternative executions sequences are coded by designers in the recovery registry. We aim to automatically derive sequences and workarounds from models, to cope with a larger set of failures, including failures not expected by designers.

We based our idea of looking for equivalent sequences on background work that proposed equivalent sequences to cope with different classes of problems. In the early nineties, Doong and Frankl [10] proposed to automatically generate test oracles from algebraic specifications by using equivalent scenarios. Doong and Frankl's work aims to automatically generate sets of both equivalent and non-equivalent scenarios to verify the results of test executions, and focuses on the effectiveness of the generated scenarios to validate the test results. Later, Henkel and Diwan [14] defined a technique to automatically derive algebraic specifications from Java code. They infer axioms based on the observational equivalence of methods execution. Mocci and Ghezzi [13] proposed a technique that relies on classes behavioral models to optimize Henkel and Diwan's approach to infer algebraic specifications.

We work in a different context and for different goals: Doongl and Frankl require the presence of detailed algebraic specifications, while Henkel and Diwan, and Mocci and Ghezzi derive executions sequences from code to maximize behavioral coverage, and thus produce complete specifications; We start with a failing execution and an overall model of the expected behavior.

3. HYPOTHESES AND APPROACH

As stated in the introduction, our research hypotheses are the following:

- Complex software systems usually have intrinsic redundancy at some abstraction level, in the sense that the same functionality can be achieved in several ways;
- This intrinsic redundancy can be exploited to identify equivalent execution sequences, that is, executions that produce equivalent results from the user viewpoint. Equivalent sequences can be automatically derived from a model of the program behavior;
- Given a failure detection mechanism, and a roll back mechanism that brings the software system back to a consistent state after a failure occurrence, the execution of sequences equivalent to the failing one can produce effective workarounds that mask software faults.

Let us consider for example a text editor that offers *append* and *set* operations. The set operation sets the value of the text in a document (overwriting the text present in the document), while append adds new text to the document at the end of the existing text. Appending text `a` to text `b`, is equivalent to setting the text to `ba`. From our viewpoint the software system is redundant, in the sense that there is more than one way to set the text to `ba`. The two execution sequences $\langle$*set(b) append(a)*$\rangle$ and $\langle$*set(ba)*$\rangle$ are equivalent, since they produce the same results. If we can detect a failure of the software system after calling an *append()*, and we can bring the system back to the state before the failure, we can mask the fault in the *append* by invoking an equivalent *set* operation.

Several interesting research challenges come out of this scenario:

- Verify our hypothesis about the existence of redundancy in some interesting classes of software systems. We plan to focus on component- and service-based software systems, where components and services often provide partially overlapping functionalities, and we plan to experimentally validate the hypothesis.
- Define a technique to automatically identify execution sequences equivalent to failing ones starting from models of the expected behavior of the software. We plan to focus on different models, starting from operational models, like finite state machines and statecharts, to later include other kinds of models, like algebraic models.

- Define strategies to identify effective workarounds from equivalent sequences. We plan to investigate prioritization schemas that sort equivalent sequences according to the likelihood of representing effective workarounds.
- Experimentally verify the effectiveness of the approach. We plan to conduct experiments with increasingly complex software systems.

The core of the automatic workarounds approach is the ability of identifying sequences that are equivalent to the one that led to a failure.

To identify equivalent sequences we rely on models of the application. So far we studied finite state machines. Consider, for example, the finite state machine in Figure 1 that represents a simple sale functionality. Customers can add items to their wish list, put the items in their cart, and buy them.

Two sequences are equivalent if they have the same *intended* effect, that is if, starting from the same state, the *correct* execution of the two sequences leads to the same final state. Notice that we do not consider equivalent two sequences that have the same *actual* effect, since in presence of failures the final state of the two sequences can be different. For these reasons, the models we rely on for the derivation of equivalent sequences must represent correct behaviors. We consider either specification models, or dynamic models derived from correct executions.

Given a failing sequence of operations, we identify the *initial state* (the state from where the failing sequence has been invoked), the *intended final state* (the state where the sequence was supposed to take the system to), and the *fallback state* (the state where the failing sequence actually brought the system to). Then we generate all the possible sequences equivalent to the failing one.

We first produce a new finite state machine from the original one by removing all the invocations of the failing sequence that do not affect the reachability of the intended final state from the fall-back state. We then generate sequences that connect the fall-back state to the intended final state. For example, if the failing sequence of the sale function of Figure 1 consists of the only invocation of function *addToCart* from state *onSale*, we can remove transition

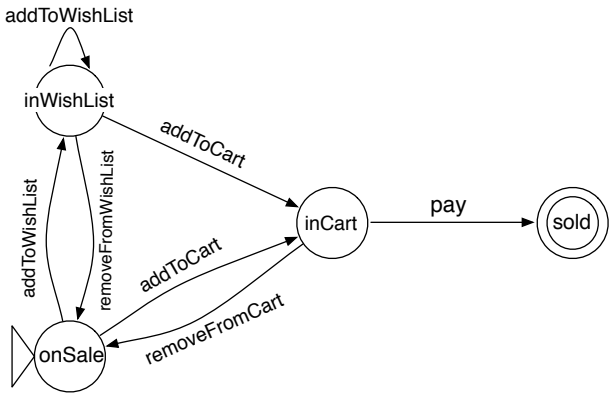

Figure 1: FSM model of a simple sale functionality

addToCart from state *onSale* to state *inCart*, without affecting the reachability of the intended final state, and identify sequences equivalent to the failing one, for instance:

addToWL, addToCart
addToWL, removeFromWL, addToWL, addToCart
addToWL, addToCart, removeFromCart, addToWL, addToCart

If none of the invocations that belong to the failing sequence can be removed from the FSM without making the intended final state unreachable from the fall-back state, we interleave invocations of the failing sequence with *indifferent* invocation sequences. Indifferent sequences are sequences that, according to the specification, do not affect either the application behavior or the final result, and that, when invoked in presence of failure, may mask or avoid the problem. Such invocations can involve single functions that alter only the *timing* or *scheduling*, and thus have no functional effect (for example, delay the execution of a function), or *maintenance* actions (for example, clean the browser cache) that have no direct functional effect on the application. They can also involve sequences of two or more services that *mutually mask* their effects (for example, pairs of add and remove, or load and unload operations). Thus, it is possible to generate sequences equivalent to the failing one by interleaving the failing sequence of invocations with indifferent invocations. For example, if the failing sequence of the sale function of Figure 1 consists of the only invocation of function *pay* from state *inCart*, we cannot remove transition *pay* from state *inCart* to state *Sold*, without affecting the reachability of the intended final state. Thus, we generate sequences equivalent to the failing one by adding indifferent invocations, for instance:

removeFromCart, addToCart, pay
removeFromCart, addToWL, removeFromWL, addToCart, pay

The number of equivalent sequences can be very high, and sometimes can even be infinite. Thus, it is important to select sequences that are highly likely to be valid workarounds. We plan to investigate different strategies to prioritize equivalent sequences to select likely workarounds. Possible strategies include prioritization according to:
Length, under the hypothesis that short sequences that do not include the faulty invocation are likely to be as effective as long sequences.
Weight, under the hypothesis that we can weight sequences according to the similarity to the failing one and select the least similar sequences.
History, under the hypothesis that sequences that share elements with successful workarounds are likely to work also for new failures.
Information about fault localization, assuming the availability of mechanisms that provide information about the actual fault, and can thus be used to discriminate among equivalent sequences.
Combination of different strategies, under the hypothesis that none of the strategies provides conclusive information, but combined with others may be more effective.

4. EVALUATION PLAN

To validate our approach we plan to (1) identify classes of software systems that have intrinsic redundancy at some abstraction level, (2) verify that we can automatically identify

equivalent sequences from different models, and (3) verify that we can automatically select valid workarounds.

We plan to perform several experiments on increasingly complex software systems. First we plan to study classes of systems, starting from component- and service-based systems, to verify the presence of intrinsic redundancy at the right abstraction level. We then plan to work on known faults, derive models from specifications or with dynamic analysis tools, reproduce the known failures, identify equivalent sequences from the models, select automatic workarounds and measure the effectiveness of the technique as the ranking of the first effective workaround. Finally, we plan to apply the technique to a set of applications to measure the number of failures that can be automatically prevented.

We conducted a set of preliminary experiments with some popular applications like Apache Tomcat, Google maps and Flickr [4, 5]. We have reproduced known failures; we have generated sequences equivalent to the failing one from finite state machine models; and we have ranked the sequences according to their length to gather preliminary data on the approach. The results are positive: we have been able to automatically generate equivalent sequences, and identify suitable workarounds.

5. REFERENCES

[1] L. Baresi, S. Guinea, and L. Pasquale. Self-healing BPEL processes with Dynamo and the JBoss rule engine. In *ESSPE '07: Intl. workshop on Engineering of software services for pervasive environments*, pages 11–20, 2007. ACM.

[2] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot - a technique for cheap recovery. In *OSDI'04: Proc. of the 6th Conf. on Symposium on Operating Systems Design & Implementation*, 2004.

[3] G. Candea, E. Kiciman, S. Zhang, P. Keyani, and A. Fox. JAGR: An autonomous self-recovering application server. In *Active Middleware Services*, pages 168–178. IEEE Computer Society, 2003.

[4] A. Carzaniga, A. Gorla, and M. Pezzè. Self-healing by means of automatic workarounds. In *SEAMS '08: Proc. of the 2008 Intl. Workshop on Software Engineering for Adaptive and Self-Managing Systems*, pages 17–24, 2008. ACM.

[5] A. Carzaniga, A. Gorla, and M. Pezzè. Healing web applications through automatic workarounds. *Intl. Journal on Software Tools for Technology Transfer*, to appear.

[6] L. Cavallaro and E. D. Nitto. An approach to adapt service requests to actual service interfaces. In *SEAMS '08: Proc. of the 2008 Intl. workshop on Software engineering for adaptive and self-managing systems*, pages 129–136, 2008. ACM.

[7] V. Dallmeier, C. Lindig, A. Wasylkowski, and A. Zeller. Mining object behavior with ADABU. In *WODA '06: Proc. of the 2006 Intl. workshop on Dynamic systems analysis*, pages 17–24, 2006. ACM.

[8] G. Denaro, M. Pezzè, and D. Tosi. Adaptive integration of third-party web services. In *DEAS '05: Proc. of the 2005 workshop on Design and evolution of autonomic application software*, pages 1–6, 2005. ACM.

[9] A. Diaconescu and J. Murphy. A framework for using component redundancy for self-optimising and self-healing component based systems. In *WADS '03: Proc. of the Workshop on Software Architectures for Dependable Systems*, 2003.

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

[11] M. M. Fuad and M. J. Oudshoorn. Transformation of existing programs into autonomic and self-healing entities. In *ECBS '07: Proc. of the 14th Annual IEEE Intl. Conf. and Workshops on the Engineering of Computer-Based Systems*, pages 133–144, 2007. IEEE Computer Society.

[12] D. Garlan, S. W. Cheng, and B. Schmerl. Increasing system dependability through architecture-based self-repair. In *Architecting Dependable Systems*, volume 2677 of *Lecture Notes in Computer Science*, pages 175–194. Springer, 2003.

[13] C. Ghezzi, A. Mocci, and M. Monga. Efficient recovery of algebraic specifications for stateful components. In *IWPSE '07: Ninth Intl. workshop on Principles of software evolution*, pages 98–105, 2007. ACM.

[14] J. Henkel, C. Reichenbach, and A. Diwan. Discovering documentation for java container classes. *IEEE Transactions on Software Engineering*, 33(8):526–543, Aug. 2007.

[15] P. Horn. Autonomic computing: IBM perspective on the state of information technology. In *AGENDA 01*, Scottsdale, AR, 2001.

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

[17] I. Koren and C. M. Krishna. *Fault Tolerant Systems*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[18] J. Kramer and J. Magee. Self-managed systems: an architectural challenge. In *FOSE '07: 2007 Future of Software Engineering*, pages 259–268, 2007. IEEE Computer Society.

[19] D. Lorenzoli, L. Mariani, and M. Pezzè. Inferring state-based behavior models. In *WODA '06: Proc. of the 2006 Intl. workshop on Dynamic systems analysis*, pages 25–32, 2006. ACM.

[20] S. Modafferi, E. Mussi, and B. Pernici. SH-BPEL: a self-healing plug-in for ws-bpel engines. In *MW4SOC '06: Proc. of the 1st workshop on Middleware for Service Oriented Computing*, pages 48–53, 2006.

[21] H. Naccache and G. Gannod. A self-healing framework for web services. In *ICWS '07: Proc. of the 2007 IEEE Intl. Conf. on Web Services*, pages 398–345, July 2007.

[22] P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum, and A. L. Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, 14(3):54–62, 1999.

[23] L. L. Pullum. *Software Fault Tolerance Techniques and Implementation*. Artech House, Inc., Norwood, MA, USA, 2001.

[24] R. Zhang. Modeling autonomic recovery in web services with multi-tier reboots. In *ICWS'07: Proc. of the IEEE Intl. Conf. on Web Services*, 2007.