Elements of the Self-Healing System Problem Space

Philip Koopman
Institute for Software Research, International
& ECE Department
Carnegie Mellon University
Pittsburgh, PA, USA
koopman@cmu.edu

Abstract

One of the potential approaches to achieving dependable system operation is to incorporate so-called "self-healing" mechanisms into system architectures and implementations. A previous workshop on this topic exposed a wide diversity of researcher perspectives on what self-healing systems really are. This paper proposes a taxonomy for describing the problem space for self-healing systems including fault models, system responses, system completeness, and design context. It is hoped that this taxonomy will help researchers understand what aspects of the system dependability problem they are (and aren't) addressing with specific research projects.

1. Introduction

Self-healing systems form an area of research that is intuitively appealing and garnering increased attention, but not very well defined in terms of scope. At the 2002 Workshop on Self-Healing Systems [WOSS02], it became clear that researchers have differing views on what comprises research on self-healing systems. This paper attempts to document those views in the form of a description of the self-healing systems research problem space.

There is a rich set of existing knowledge on the general topic of dependable systems, and on techniques that can reasonably be considered to comprise "self-healing." For example, one view of self-healing systems is that they perform a reconfiguration step to heal a system having suffered a permanent fault. The use of standby spares in such a manner has been called "self-repair" [Bouricius69]. Systems that use modular redundancy (e.g., [vonNeuman56]) can tolerate component failures and might be considered to be self-healing.

It is premature to propose a consensus-based definition of the term "self-healing," so we do not attempt to do this beyond an appeal to intuition that such a system must somehow be able to "heal" itself. Whether this means that self-healing systems are really a subset of traditional fault-tolerant computing systems is unclear. However, the topic of self-healing systems has attracted a number of researchers who would not otherwise have been involved in the fault tolerant computing area. So, if nothing else, the label of self-healing has broadened the pool of researchers addressing the difficult problems of creating dependable systems.

To give researchers in this area a common basis for defining the scope of self-healing systems research, it seems worthwhile to set forth a description of issues being addressed by various research projects. This might provide a way for researchers to realize they mean considerably different things by their use of the phrase "self-healing," as well as to understand the similarities and differences in their approaches and domains. Toward that end, this paper attempts to describe the general problem space relevant to self-healing system research.

2. Elements of the model

Based on our experiences and observations at the WOSS02 workshop, we propose that there are four general categories of aspects to the self-healing system problem space: fault model, system response, system completeness, and design context (Table 1). (The particular categories are not important, but simply form a way to group related concepts for the purposes of discussion.) We shall discuss the elements of each category in turn.

[Avizienis01] contains an extensive taxonomy of fault tolerant computing terminology and approaches. We use this as the basis for terminology, and as the basis for the fault modeling portion of the taxonomy.

2.1. Fault model

Self-healing systems have similar goals to the general area of dependable computer systems. (Not all dependable computing research areas are "self-healing", but one can argue that all "self-healing" techniques ultimately are dependable computing techniques.)

One of the fundamental tenets of dependable computing is that a *fault hypothesis* (often called a fault model) must be specified for any fault tolerant system. The fault hypoth-

Table 1. Problem space model elements.

Fault model:	
	Fault duration
	Fault manifestation
	Fault source
	Granularity
	Fault profile expectations
System response:	
	Fault Detection
	Degradation
	Fault response
	Fault recovery
	Time constants
	Assurance
System completeness:	
	Architectural completeness
	Designer knowledge
	System self-knowledge
	System evolution
Design context:	
	Abstraction level
	Component homogeneity
	Behavioral predetermination
	User involvement in healing
	System linearity
	System scope

esis answers the question of what faults the system is to tolerate. (If one doesn't know what types of faults are to be tolerated, it is difficult to evaluate whether a given system is actually "fault tolerant.")

In a similar vein, self-healing systems must have a fault model in terms of what injuries (faults) they are expected to be able to self-heal. Without a fault model, there is no way to assess whether a system actually can heal itself in situations of interest. The following are typical fault model characteristics that seem relevant.

Fault duration: Faults can be permanent, intermittent (a fault that appears only occasionally), or transient (due to an environmental condition that appears only occasionally). Since it is widely believed that transient and intermittent faults outnumber permanent faults, it is important to state the fault duration assumption of a self-healing approach to understand what situations it addresses.

Fault manifestation: Intuitively, not all faults are as severe as others. Beyond that, components themselves can be designed to exhibit specific characteristics when they encounter faults that can make system-level self-healing simpler. A common approach is to design components that are fail-fast, fail-silent. However, other systems must tolerate Byzantine faults which are considered "arbitrary" faults. (It is worth noting that Byzantine faults exclude systematic software defects that occur in all nodes of a system, so the meaning of "arbitrary" is only with respect to an assumption of fault independence.)

Beyond the severity of the fault manifestation, there is the severity of how it affects the system in the absence of a self-healing response. Some faults cause immediate system crashes. But many faults cause less catastrophic consequences, such as system slow-down due to excessive CPU loads, thrashing due to memory hierarchy overloads, resource leakage, file system overflow, and so on.

Fault source: Assumptions about the source of faults can affect self-healing strategies. For example, faults can occur due to implementation defects, requirements defects, operational mistakes, and so on. Changes in operating environment can cause a previously working system to stop working, as can the onset of a malicious attack. While software is essentially deterministic, there are situations in which it can be argued that a random or "wear-out" model for failures is useful, suggesting techniques such as periodic rebooting as a self-healing mechanism. Finally, some self-healing software is designed only to withstand hardware failures such as loss of memory or CPU capacity, and not software failures.

Granularity: The granularity of a failure is the size of the component that is compromised by that fault. (The related notion of the size of a fault containment region is a key design parameter in fault tolerant computers.) A fault can cause the failure of a software module (causing an exception), a task, an entire CPU's computational set, or an entire computing site. Different self-healing mechanisms are probably appropriate depending on the granularity of the failures and hence the granularity of recovery actions.

Fault profile expectations: Beyond the source of the fault is the profile of fault occurrences that is expected. Faults considered for self-healing might be only expected faults (such as defined exceptions or historically observed faults), faults considered likely based on design analysis, or faults that are unexpected. Additionally, faults might be random and independent, might be correlated in space or time, or might even be intentional due to malicious intent.

2.2. System response

The first step in responding to a fault is, in most cases, actually detecting the fault. Beyond that there are various ways to degrade system operation as well as attempt recovery from or compensation for a fault. Each application domain has extra-functional aspects that are important, such as reliability, safety, or security. These extra-functional concerns influence desired system responses.

Fault Detection: Fault detection can be performed internally by a component, by comparing replicated components, by peer-to-peer checking, and by supervisory checks. Additionally, the intrusiveness of fault detection can vary from nonintrusive testing of results, to execution of audit or check tasks, redundant execution of tasks, on-line self-test, and even periodic reboots for the purpose of more thorough self tests. Systems might inject faults intentionally as on-line tests of fault detection mechanisms. A related area is that of ensuring that all aspects of a system are activated periodically so that any latent accumulated faults can be detected within a bounded time. Not all systems can achieve 100% fault detection in bounded time.

Degradation: Self-healing systems might not restore complete functionality after a fault. The degree of degraded operation provided by a self-healing system is its resilience to damage that exceeds built-in redundancy. Some systems must fail entirely operational (i.e., cannot fulfill their mission without full functionality). But many systems can degrade performance, shed some tasks, or perform failover to less computationally expensive degraded mode algorithms.

Fault Response: Once a fault has been detected, the system must select a response mechanism. Typical on-line responses include masking a fault (e.g., modular redundancy that performs a majority vote of independent computational results), rollback to a checkpoint, rollforward with compensation, or retrying an operation using the original or alternate resources. Heavier-weight responses include system architectural reconfiguration (on-the-fly or involving a reboot), invoking alternate versions of tasks, killing less important tasks, and requesting assistance from outside the system. The fault response might be optimized to maintain desired properties such as correctness, quality of service contracts, transactional integrity, or safety. Fault responses might also be preventative (such as a periodic system reboot), proactive (such as an action triggered by a burst of faults which were tolerated but are indicative of a possible near-term failure), or reactive.

Recovery: After a system has detected a fault, potentially degraded, and invoked a fault response, it must recover operation to complete the self-healing process. Recovery involves issues such as integrating newly committed resources into ongoing processes, "warming up" resources by transferring system state into them, or taking action to bring the system to a clean known state before proceeding with operations. A component might be hot-swapped, require a warm system reboot, or require a cold system reboot to finish recovery.

Time constants: The time constants of a system, along with the fault distribution assumptions, play a large role in determining what types of self-healing are feasible. The time constant of faults with respect to the forward progress of computations determines things like the frequency at which checkpoints must be taken, or whether a system can reboot itself quickly enough to prevent an overall system outage. Additionally, if intermittent or transient faults are in the majority as is typical in many systems, the speed of the detection-response-recovery cycle might need to be faster than typical fault arrival periods to avoid system thrashing.

Assurance: Every domain has a specific set of system properties of importance. Every system requires assurances of some level of functional and extra-functional correctness for normal operation. Self-healing systems additionally require a way to assure that such properties are maintained during and after fault occurrences. Challenges in this area include assurance between the time a fault occurs and the time the fault is detected (keeping in mind that not all faults are detected); assurance during degraded mode operations; and assurance during recovery operations. This assurance might be provided at design time or might involve checks at run time. Finally, the assurance might be absolute or probabilistic, and might involve all functionality or partial assurance of only a few key system properties.

2.3. System completeness

Real systems are seldom complete in every sense. Self-healing approaches must be able to deal with the reality of limits to knowledge, incomplete specifications, and incomplete designs.

Architectural completeness: Few system architectures are completely elaborated when the first implementation is built. Architectures and implementations evolve over time. Many systems are "open" in that third-party components can be added during or after system deployment. And, many systems are designed using prebuilt components that have details and behavior so opaque to the overall system designer that the architecture might as well be considered incomplete. Finally, a system might be built upon discovery mechanisms which are intended to extend the architecture or implementation at run-time. A related issue is that implementations of components evolve, are patched, suffer configuration management problems, and so on.

Designer knowledge: Designers in the typical case do not have complete knowledge of the systems they design. Any system is designed using a set of abstractions about underlying components. But beyond that the designer must deal with missing knowledge about aspects of components, and in all likelihood incorrect knowledge about system components due to documentation and implementation defects. It is common for designers to have a thorough understanding of typical system behaviors, but to have little or no understanding of atypical system behaviors – especially system behaviors in the presence of faults. A vital aspect of designer knowledge is how well the fault model for the system is characterized and whether field information about faults is fed back to the system designer.

System self-knowledge: Systems must have some level of knowledge about themselves and their environment in order to provide self-healing. This self-knowledge is limited by the aspects of knowledge built into a component (for example, a component might or might not be able to predict its execution time in advance), the accessibility of knowledge about one component to another component, and defects in representation of such knowledge either due to initial design defects or staleness caused by system evolution. The concept of reflection is often discussed in the context of system self-knowledge; however it also seems possible to build systems that have no awareness of their state but rather exhibit emergent correctness as a consequence of the interaction of their component behaviors.

System evolution: Self-healing systems must deal with the fact that they change over time. Sources of change include designed operating mode changes, accumulated component and resource faults, adaptations to external environments, component evolution, and changes in system usage. Making use of available information on system dynamics might help with self-healing, such as being able to count on a scheduled system outage (or self-schedule an outage) to perform healing.

2.4. Design context

There are several other factors that influence the scope of self-healing capabilities that could be considered to form the design context of the system.

Abstraction level: Systems can attempt to perform various forms of self-healing to application software, middleware mechanisms, operating systems, or hardware. Self-healing techniques can be applied to implementations (such as wrappers to deal with unhandled exceptions) or architectural components.

Component homogeneity: While some systems have completely homogenous components, it is common to have systems that are heterogeneous to some degree. Server farms often have different versions of processing hardware, and might well have different versions of operating systems or other software installed, especially when changes are applied incrementally across a fleet of components as a risk management technique. Homogeneity can consist of exact component duplicates, or components that are "plug-compatible" even though they have differing implementations. Some systems are inherently heterogenous, such as the computational components within embedded systems such as automobiles. The heterogeneity of a system tends to limit its ability to simply migrate computational tasks as a self-healing strategy and requires that self-healing approaches deal with the issue of configuration management of systems both before and after healing.

Behavioral predetermination: Most systems do not have perfectly predetermined and deterministic behavior, and some self-healing approaches must be able to accommodate this. Non-deterministic behavior abounds in hardware and in software infrastructure. But, beyond that, it is often impractical to quantify things such as absolute worst-case execution time. Even things that might seem determinable in theory such as enumeration of all possible exceptions that can be generated by a software component might be impractical due to obscure component interactions or defects. In the time dimension, system tasks might be event-based or periodic, necessitating differing assumptions and approaches by healing mechanisms.

Both the system and the self-healing mechanism can have differing levels of behavioral predeterminism. For example, a rule-based application or one that employs neural networks might not be readily analyzed for behavior. Similarly, a self-healing mechanism might employ nondeterministic or analytically complex approaches that make design-time analysis of behavior impractical.

User involvement in healing: While the goal of much thinking about self-healing systems is to achieve complete autonomy, this might be an over-ambitious goal. Most systems have a limit to healing ability, beyond which users must become involved in system repair. The opportunity for self-healing system collaborations with users are two-fold: users can adapt their behavior to help systems function despite failures, and users can provide advice to systems to guide aspects of their self-healing behavior.

System linearity: Overall system linearity and component coupling can greatly affect the ability of a system to self-heal. If a system is completely linear (i.e., all aspects of the system are completely composable from component aspects) then self-healing of one component can be carried out without concern for its effect upon other components. While many well-architected systems have good linearity, component interaction is a typical situation that must be addressed by self-healing approaches.

System scope: How big is the system? A single-node computing system does not have all the self-healing possibilities available to a geographically distributed computing system. Similarly, portions of the system might be considered out-of-bounds when creating a self-healing mechanism, such as a requirement to use an off-the-shelf operating system or existing Internet communication protocols. The scope of system self-healing might therefore be a single component; a computer system; a computer system plus a person; an enterprise automation suite; or the computer in the context of society including regulatory agencies, maintenance groups, and insurance mechanisms.

3. Examples of use

Because the purpose of this paper is to propose a way of structuring a complex and still relatively unexplored research area, it is unlikely that the results are complete or in-

Table 2. Self-Healing Problem Spaces Addressed By Example Research Projects.

	Property	RoSES Graceful Degradation	Semantic Anomaly Detection	Amaranth QoS
Fault Model	Fault Duration	Permanent	Permanent+Intermittent	Permanent+Intermittent
	Fault Manifestation	Fail fast+silent components Potentially correlated	Unexpected data feed values; Recovery only if uncorrelated	Resource exhaustion Potentially correlated
	Fault Source	All non-malicious sources	Representable by templates; Non-malicious	Peak resource demand; Non-malicious
	Granularity	Component failure in distributed embedded system	Failure of Internet data feed	Depletion of memory, CPU, etc. in distributed system
	Fault Profile Expectations	Random; arbitrary; unforeseen	Anomalies compared to prior experience	Random; resource consumption only
System Response	Fault Detection	State variable staleness	Anomaly detection	Resource monitoring alarm
	Degradation	Fail-operational; Maximize system utility	Not addressed	Preserve predetermined baseline functions; eject nonessential tasks
	Fault Response	Reconfigure SW based on data and control flow graphs	Substitute redundant data feed	Admission control policy: Admit “baseline” tasks and reject some enhanced tasks
	Recovery	Reconfigure SW & reboot system	On-the-fly data feed switch	Terminate enhanced tasks as necessary
	Time constants	Long time between failures; Can handle multiple failures	Valid data samples occur much more often than anomalies	Can handle multiple failures; Tasks can be terminated instantly
	Assurance	Future work; reliability-driven	“Good enough” data quality	Static analysis of baseline load

System Completeness	Architecture Completeness	Closed, complete system;
		Graceful upgrade/downgrade;
		System must work in worst case
			Dynamic Internet data feeds;
			Unknown gaps & defects;
			Common case handling complete
				Closed, complete system;
				System must work in worst case

	Designer Knowledge	Assumed to be complete	Component specifications unknown -- must be inferred	Complete; workload information is statistical distribution
	System Self-Knowledge	System knows component presence & failure; data/control flow	History used as basis for anomaly detection	Available resources and approximate task resource consumption
	System Dynamicism	Upgrades & downgrades; System stable during mission	Data feeds come and go	Workload is stochastic
Design Context	Abstraction Level	HW & SW components within distributed system	Nodes on Internet	Tasks within distributed system
	Component Homogeneity	Heterogenous components and resources	Redundant or correlated data feeds	Homogenous resources; heterogeneous tasks
	Behavioral Predetermination	Components characterized; Functions must be composable	Predetermined data feed type; Behavior of data feed discovered	System design predetermined; Workload is stochastic
	User Involvement	Fully automatic	User accepts/rejects templates	Fully automatic

	System Linearity	Multiattribute utility theory;
		Scalability assumes linearity;
		Bin-packing task approach
			Not applicable	Tasks have discrete operating points;
				Bin-packing approach

	System Scope	Multiple computers in embedded control system	Multiple computers + user on Internet-based system	Multiple computers on Internet or closed network system

deed even apply to all research projects. Additionally, the type of information required to describe many projects is not fully available from published sources. In the interest of providing concrete examples, three of our own research projects are briefly described in terms of the proposed categories in Table 2.

RoSES (Robust Self-configuring Embedded Systems) [Shelton03] is a project that is exploring graceful degradation as a means to achieve dependable systems. It concentrates on allocating software components to a distributed embedded control hardware infrastructure, and is concerned with systems that are entirely within the designer's control.

The semantic anomaly detection research project [Raz02] seeks to use on-line techniques to infer specifications from underspecified components (e.g., Internet data feeds) and trigger an alarm when anomalous behavior is observed. An emphasis of the research is using a template-based approach to make it feasible for ordinary users to provide human guidance to the automated system to improve effectiveness.

The Amaranth project [Hoover01] is a Quality of Service project that emphasizes admission policies. A key idea is to have tasks with at least two levels of service: baseline and optimized. A system could thus be operated to guarantee critical baseline functionality via static system sizing, with idle resources employed to provide optimized performance on an opportunistic per-task basis.

All three projects are, in our opinion, "self-healing software system" research projects. But as shown by Table 2 they have widely varying areas of exploration, assumptions, and areas that are unaddressed. The area in which all three projects are substantially similar is the last attribute, in which all three systems assume a distributed computing environment. It is worth noting that the categories were created before Table 2 was constructed, so this provides initial evidence that the categories capture differences among general projects rather than being specific to just these projects. But of course since the people involved in the three projects discussed overlap, this does not prove generality and certainly does not demonstrate completeness.

4. Conclusions

It is too soon to tell whether "self-healing" system approaches are just a different perspective on the area of fault tolerant computing, and whether that perspective brings significant benefits. Resolving this issue requires better understanding of what is meant by the term "self-healing" in the first place. To that end, this paper proposes a taxonomy for describing the problem space for self-healing systems.

Relevant aspects of self-healing system approaches include fault models, system responses, system completeness, and design context. It is of course unreasonable to expect every research paper on self-healing systems to address every possible aspect discussed, and no doubt some important aspects are yet to be discovered. It remains to be seen how different aspects interact in various domains, and which aspects matter the most in practice. However, it is hoped that this taxonomy will provide a checklist for researchers to use in explaining the part of the problem space they are addressing, and perhaps to help avoid inadvertent holes in self-healing system approaches.

5. Acknowledgments

I would like to thank the participants of WOSS02 for stimulating discussions that made this paper possible, the anonymous reviewers for their comments, and the members of my research group for their additional suggestions. The techniques in this paper come from the experience of others and decades of research, especially in the fault tolerant/dependable computing community. Interested readers should consult the proceedings of the Fault Tolerant Computing Symposium (FTCS) and Dependable Systems & Networks (DSN) conference for further information.

This work was supported in part by the High Dependability Computing Program via NASA Ames cooperative agreement NCC-2-1298, and in part by the General Motors Collaborative Research Laboratory at Carnegie Mellon University.

6. References

[Avizienis01] A. Avizienis, J.-C. Laprie and B. Randell, *Fundamental Concepts of Dependability*, Research Report N01145, LAAS-CNRS, April 2001.

[Bouricius69] Bouricius, W.G., Carter, W.C. & Schneider, P.R, "Reliability modeling techniques for self-repairing computer systems," *Proceedings of 24th National Conference*, ACM, 1969, pp. 395-309.

[Hoover01] Hoover, C., Hansen, J., Koopman, P. & Tamboli, S., "The Amaranth Framework: policy-based quality of service management for high-assurance computing," *International Journal of Reliability, Quality, and Safety Engineering*, Vol. 8, No. 4, 2001, pp. 1-28.

[Raz02] Raz, O., Koopman, P., & Shaw, M., "Enabling Automatic Adaptation in Systems with Under-Specified Elements," *1st Workshop on Self-Healing Systems (WOSS'02)*, Charleston, South Carolina, November 2002.

[Shelton03] Shelton, C., Koopman, P. & Nace, W., "A framework for scalable analysis and design of system-wide graceful degradation in distributed embedded systems," *WORDS03*, January 2003.

[vonNeuman56] von Neumann, J., "Probabilistic logics and the synthesis of reliable organisms from unreliable components," 1956, in Taub, A. H., (ed.), *John von Neumann: collected works*, Volume V, pp. 329-378, New York: Pergamon Press, 1963.

[WOSS02] *1st Workshop on Self-Healing Systems (WOSS'02)*, Charleston, South Carolina, November 2002, ACM Press.