# A Modeling Framework for Self-Healing Software Systems

Michael Jiang, Jing Zhang, David Raymer, and John Strassner

Motorola Network Infrastructure Research Lab, Autonomics Research
{michael.jiang,j.zhang,david.raymer,john.strassner}@motorola.com

**Abstract.** For a system to be capable of self-healing, the system must be able to detect what has gone wrong and how to correct it. This paper presents a generic modeling framework to facilitate the development of self-healing software systems. A model-based approach is used to categorize software failures and specify their dispositions at the model level. Self-healing is then achieved by transforming the model of the system into platform-specific implementation instrumented with failure detection and resolution mechanisms to mitigate the effect of software failures and maintain the level of healthiness of the system.

**Key words:** Autonomics, modeling, AOM, model transformation

## 1 Introduction

When applied to computer-based systems and networks (CBSN), self-healing is one aspect of the set of capabilities exhibited by autonomic computational systems often represented by the phrase self-*, which most often includes self-protecting, self-configuring, self-healing, and self-optimizing. The basis of autonomic computing draws on biological analogies to describe an autonomic computer as a computer that is self-governing, in the same manner that a biological organism is self-governing [1][2].

A self-healing software system is one that has the ability to discover, diagnose, and repair (or at least mitigate) disruptions to the services that it delivers. For large scale systems, many different types of faults may exist, and their differing natures often require disparate, tailored approaches to detect, let alone fix them. Hence, for large scale systems, a self-healing system should also be able use multiple types of detection, diagnosis, and repair mechanisms.

Autonomic systems extend the above notion of self-healing to include the capabilities to adapt to changes in the environment, for example, to maintain its performance, or availability of resources.

This paper presents a modeling framework to specify and implement self-healing focusing on the software aspects of a system. Model constructs are used to classify software failures and specify their dispositions apart from the models that capture the base functionality of the system. Self-healing is achieved by transforming base models as well as self-healing models into platform-specific

implementation instrumented with fault detection and resolution, with intent to mitigate the effect of software failures and maintain the level of healthiness of the system.

## 2 Classification and resolution of software faults

### 2.1 Classification of software faults

The first step toward self-healing is the automatic recognition of software failures. The IEEE standard 1044 [3] documents a comprehensive list of categories and classifications for software anomalies. The taxonomies of software failures, errors, and faults have been discussed extensively for many different types of software systems. Commonly accepted definitions are as follows. A software fault refers to a defect in a system. An error is a discrepancy between the observed behavior of a system and its specified behavior. A software failure occurs when the delivered service deviates from correct service, a departure of system behavior from user requirements. A software fault or error may not necessarily cause a software failure [4].

The following are the major categories of software faults discussed in the literature for various types of software systems [5][6]:

- Syntactic faults: interface faults and parameter faults
- Semantic faults: inconsistent behavior and incorrect results
- Service faults: QoS faults, SLA related faults, and real-time violations
- Communication / interaction faults: time out and service unavailable
- Exceptions: I/O related exceptions and security-related exceptions

A syntactic fault occurs when the structures of messages or parameters of a requester do not agree with those of a provider. While compilers are able to capture these types of faults in a single program, they can happen in component-based software and web services where a requester and a provider may be constructed independently of each other. Semantic faults, on the other hand, represent inconsistencies between the original design and the programmers' intention. Semantic faults are more challenging to diagnose and often require specialized instrumentation to detect them.

A service fault occurs when the performance of the delivered service deviates from the required service performance as specified in the Quality of Service (QoS) or Service Level Agreement (SLA). These types of faults cause degraded performance of the system. Faults also occur due to communication and interaction among components, services, and subsystems in both distributed and centralized systems. The absence and unresponsiveness of providers and the timing of service requests are examples of communication and interaction faults.

Unhandled exceptions are another source of software faults that often lead to the abnormal termination of applications. Exception frameworks defined in languages, such as Java and C++, provide both built-in and user-defined exceptions. However, they do not provide a mechanism to prevent unexpected application termination due to unhandled exceptions. The different levels of granularity

in the exception framework facilitate the specification of exception handling to recover from specific and general software faults.

### 2.2 Fault detection

The detection methods for different software faults vary depending on the types of faults. For syntactic faults, the structures of messages or parameters are verified against design requirements and failures to conform to design requirements will result in syntactic faults. Communication and interaction faults may be detected by the involved entities. For instance, wrappers can be added to web service invocation to detect the availability and responsiveness of service providers. For faults related to exceptions, the exception framework defined in programming languages provides a uniform approach to fault detection. C++ and Java, for example, provide the "try - catch" constructs to capture both specific and general programming exceptions. The framework can be extended to capture user-defined exceptions as well.

Service faults require the monitoring of services to determine whether the provided services meet the required QoS or SLA agreement. Different types of service faults may require different detection mechanisms. For instance, if the invocation of a service must be completed within a specific time frame, the invocation can be timed such that the failure to respond within the required time frame is detected. The detection of semantic faults is more challenging. It requires specialized instrumentation and / or domain-specific knowledge to understand the correct vs. incorrect behavior of a component, service, or system.

Various generic approaches can also be applied to fault detection. One of the common approaches is to perform source code instrumentation. Extra code is inserted into the source code to facilitate the detection of software anomalies. Aspect-oriented programming (AOP) [7], for instance, can be applied to attach additional functions to the system for the purpose of detecting system performance and faults. Sensors and event collection and processing can also be applied to detect software faults. Majority voting [8], for instance, can be found in telecommunication systems to detect software or system anomaly to ensure high reliability and availability.

Failure prediction can also be used to enhance the above detection methods. This involves both learning and reasoning about failures. Preventative and corrective actions can be taken before faults occur to avoid the loss of functionality or disruption of services [5].

### 2.3 Fault resolution and self-healing

Various approaches have been described to handle software faults: fault prevention, fault removal, and fault tolerance. Fault prevention and removal aim to achieve fault-free software through robust design and rigorous testing. The goal of fault tolerance is to ensure the continual operation of a system in the presence of faults.

To achieve self-healing when a software fault occurs, the system must be able to get back to one of its normal operating states. This requires the system to know what the normal operating states are and what necessary actions should be performed to get back to the normal operating states. Error recoveries, forward or backward, can be employed to enable the system to reach an error-free state. Redundancy can also be used to compensate for errors. Check-pointing can be employed to save the state of a process or a component to be restored when faults occur, as often required in transaction processing.

There are different levels of granularities for fault resolution. Exception handling, for instance, can be declared for all types of exceptions, from most specific to the most general. For a particular exception, specific actions can be taken to recover from the exception effectively. For general exceptions, however, the action to recovery might be less efficient, less accurate, and more costly, such as resetting or restarting a complex subsystem.

## 3 Model-driven self-healing software system

We propose a modeling framework to facilitate the development of self-healing software systems. The framework consists of a generic self-healing model, a model composition mechanism for integration of base models (the ones that specify the base functionality of the system) and the self-healing models, as well as a code generation platform that supports automated generation of self-healing enabled software systems. The aim of this framework is to integrate various existing self-healing techniques (such as the ones mentioned in Section 2) with Model Driven Architecture (MDA) [9] approaches, with the intent to provide a unified platform to enable self-healing software systems to be engineered and implemented from model specifications.

### 3.1 Self-healing model

Figure 1 shows a fragment of the self-healing model for components and services. Different types of faults may be present for different types of software, such as a software component or a web service. A software fault can be detected with one or more detectors and recovered by using one or more resolution mechanisms. The resolution of a fault is modeled by a set of actions that transform the system from its faulty state to a specified operational state. Similarly, preventive actions can be taken when the occurrence of faults is predicted.

The self-healing model specifies the categories of faults that may occur for components and services: syntactic faults, semantic faults, service faults, communication/interaction faults, and exceptions. Each type of fault uses its specific fault detectors/predictors and resolution mechanisms. Each type of fault can also be decomposed into more specific faults depending on the nature of the component and application domain.

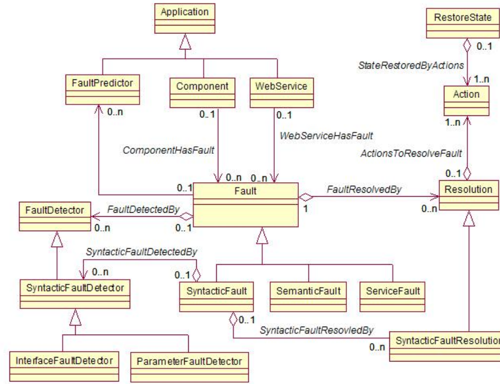


**Fig. 1.** Self-healing model fragment

### 3.2 Model composition

A self-healing model is described as a set of abstract constructs that are instantiated when it is associated with the base system models. In this way, the self-healing feature is kept separate from the base functionality of the software system. The decomposition of the base functionality and the self-healing feature improves modularity and re-usability of the self-healing modules. The main advantage of this approach is that it allows developing and maintaining the self-healing models independently from the base model. Furthermore, self-healing models can be applied to all levels in the software system hierarchy. Figure 2 illustrates a scenario in which different self-healing models (denoted by dotted rectangles) can be composed from different software models, from class level through component level to the system level.

Deploying a self-healing model to the base application model is done by model composition techniques. Typically, model composition involves merging two or more models to obtain a single integrated model.

Aspect-Oriented Modeling (AOM) [10] is a promising research area that supports model composition. Self-healing models can be specified as crosscutting aspects that are embedded in the base models. Fault detectors are intended to capture anomalous status of the software (what can go wrong, as well as when and where it did go wrong). These patterns can be denoted using pointcut descriptors. Resolutions are actions that will be taken once a certain type of failure occurs. These actions can be encapsulated within the advice of an aspect model. An aspect weaver instantiates aspect models and binds them to the base models.

Model weaving [11] is another approach for model composition, wherein the relationships between the models are captured in a weaving model. A weaving model is one that specifies different kinds of mappings between model elements. By adopting this approach, a separate weaving model is created that explicitly specifies the links between base models and self-healing models. All three kinds

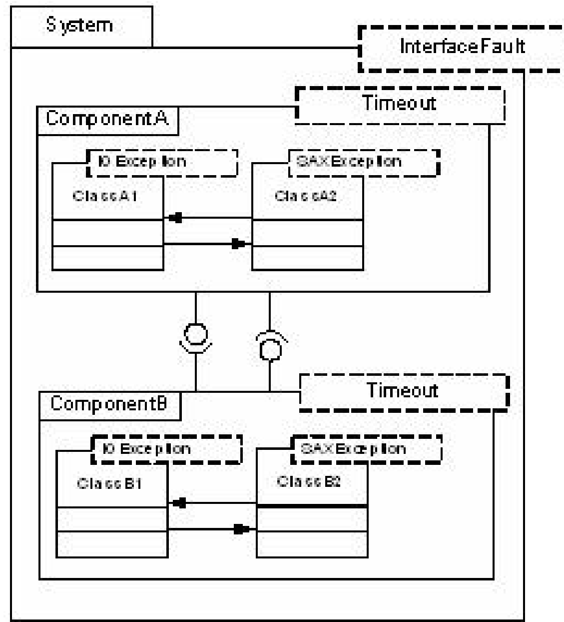


**Fig. 2.** Composition of base models and self-healing models

of models (i.e., base, self-healing and weaving models) will be fed to a weaver engine and composed into a single integrated model.

### 3.3 Code generation

The self-healing models are specified in a platform-independent manner. To support a complete self-healing application, a family of code generators [12] needs to be employed for transforming platform-independent models into platform-specific code, each describing how the self-healing features are implemented on a different platform.

As shown in Figure 3, the base and self-healing models are translated into platform-specific implementations by their own code generators, respectively. Specifically, the representation of software faults and their responding resolutions are transformed to implementation code that detects the occurrence of faults and takes the appropriate actions to restore the system to the specified operational state. The base code is then augmented with the self-healing instrumentation by using program composition techniques [13]. As a result, a complete self-healing enabled software system is constructed and directly mapped to the representation of the composed self-healing enabled software model, upon which model-based analysis can be performed for system verification and validation.

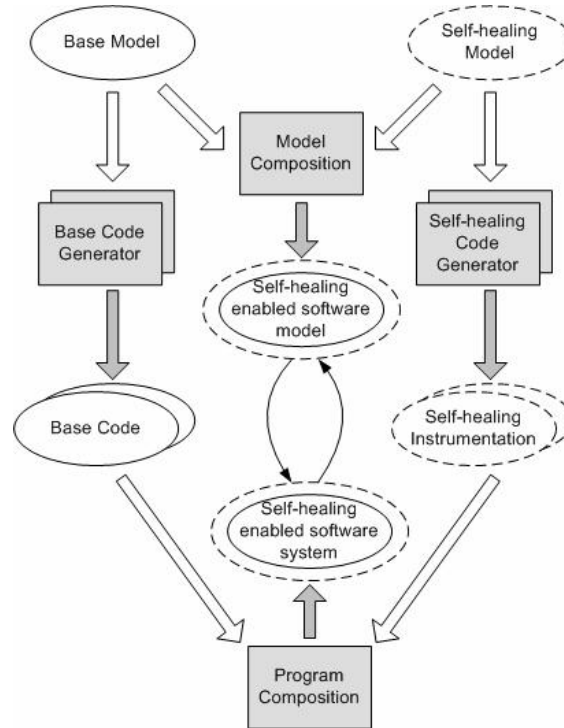


**Fig. 3.** Self-healing software generation

## 4 Related work

The IEEE Standard Classification for Software Anomalies [3] provides a comprehensive list of software anomaly classification and related data items that are helpful to identify and track anomalies. The methodology of this standard is based on a process (sequence of steps) that pursues a logical progression from the initial recognition of an anomaly to its final disposition. In [6], a fault taxonomy of software components is proposed to facilitate the identification and classification of common faults in component-based software. Authors in [5] describe the taxonomy of failures, errors, and faults for dependable and secure computing. The means to attain dependability and security are categorized as fault prevention, fault tolerance, fault removal, and fault forecasting. In [14], tree-based techniques are proposed for the classification of software failures based on execution profiles. The UML profile for modeling QoS and fault tolerance [15] defines a set of UML extensions to represent QoS and fault tolerance concepts based on object replications.

Projects in the DARPA DASADA program [16][17] describe an architecture that uses probes and gauges to monitor the execution of programs, generate events containing measurements, and take actions based on the interpretation

of events. Effectors interact with the system to ensure that system's runtime behavior fits within the envelope of acceptable behavior. Authors in [18] propose the generation of proxies and wrappers to add autonomic functionalities to object-oriented applications to cope with failures without source code adaptation. In [19], authors describe the use of code transformation to instrument Java byte-code by adding fault analysis and healing actions. Authors in [20] present a connector-based self-healing mechanism for software components. A component in a self-healing system is designed as a layered architecture, structured with the healing layer and the service layer. The role of connectors between tasks in a component is extended to support the self-healing mechanism in detecting, reconfiguring, and repairing anomalous objects in components.

The novelty of our approach to self-healing software systems lies in the modeling framework to integrate functional and self-healing specifications, as well as the application of model transformation to realize self-healing capabilities. Our modeling framework aims to be more general than the prior approaches in that we synthesize various self-healing techniques and employ model composition and transformation mechanisms to support full-fledged self-healing software systems from the abstract model specifications down to the system implementations.

## 5 Conclusion and discussion

This paper presents a generic framework for modeling self-healing software systems. Modeling constructs are used to capture software faults and their detection and resolutions to transform the system from faulty states to operational states specified in the model. The classification of software faults and the modeling of fault detection and resolution facilitate the construction of self-healing software systems. Self-healing is achieved by transforming the self-healing models into platform-specific implementation, which is then composed with the base module to form an integrated software system that provides failure resolutions to mitigate the effect of software faults.

To achieve system-wide optimization beyond self-healing, policies are required to orchestrate the behavior of the system and adjust its operations to meet business objectives. Various self-* aspects need to be integrated to achieve autonomic computing. This will be the focus of our future research, which will also investigate the integration of ontology and production rules to enhance the fault modeling with reasoning capabilities to improve fault analysis and classification.

## References

1. John Strassner and Jeffrey O. Kephart, "Autonomic Systems and Networks: Theory and Practice," Network Operations and Management Symposium (NOMS), 2006.
2. Jeffrey O. Kephart and David M. Chess, "The Vision of Autonomic Computing," IEEE Computer, Vol. 36(1), January 2003.
3. IEEE Standard Classification for Software Anomalies, IEEE Std 1044-1993, 1993.

4. John Musa, "Software Reliability Engineering", McGraw-Hill, 1999.
5. Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, Carl Landwehr, "Basic concepts and taxonomy of dependable and secure computing," IEEE Transactions on Dependable and Secure Computing, Volume 1, Issue 1, 2004, pp. 11-33.
6. Leonardo Mariani, "A fault taxonomy for component-based software," Proceedings of International Workshop on Test and Analysis of Component-Based Systems (TACoS), Electronic Notes in Theoretical Computer Science, Vol. 82, Elsevier Science, 2003.
7. Gregor Kiczales, et al, "Aspect-Oriented Programming," European Conference on Object-Oriented Programming, Finland, June 1997.
8. Salim Hariri, Alok Choudhary, and Behcet Sarikaya, "Architectural Support for Designing Fault-Tolerant Open Distributed Systems". IEEE Computer, 1992.
9. Richard Soley and the OMG Staff Strategy Group, "Model-Driven Architecture," ftp://ftp.omg.org/pub/docs/omg/00-11-05.pdf.
10. AOM: http://www.aspect-modeling.org/
11. Didonet Del Fabro Marcos, Bzivin Jean, Jouault Frdric, Breton Erwan, Gueltas Guillaume, "AMW: A Generic Model Weaver," In Proceedings of the 1re Journe sur l'Ingnierie Dirige par les Modles (IDM05), Paris, France, June-July 2005.
12. Markus Voelter, "A Collection of Patterns for Program Generation," Eighth European Conference on Pattern Languages of Programs, Irsee, Germany, June 2003.
13. Uwe Amann, Invasive Software Composition, Springer-Verlag, 2003.
14. Patrick Francis, David Leon, Melinda Minch, Andy Podgurski, "Tree-Based Methods for Classifying Software Failures," Proceedings of the 15th International Symposium on Software Reliability Engineering (ISSRE'04), Saint-Malo, France, November 2004.
15. Object Management Group, "UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms," http://www.omg.org/cgi-bin/doc?formal/06-05-02, 2006.
16. David Garlan, Bradley Schmerl and Jichuan Chang, "Using Gauges for Architecture-based Monitoring and Adaptation," Working Conference on Complex and Dynamic Systems Architecture, Australia, 2001.
17. Janak Parekh, et al, "Retrofitting autonomic capabilities onto legacy systems," Journal of Cluster Computing, 2005, pp. 141-159.
18. A. Reza Haydarlou, et al, "A Self-healing Approach for Object-Oriented Applications," 3rd International Workshop on Self-Adaptive and Autonomic Computing Systems, 2005.
19. M. Muztaba Fuad and Michael J. Oudshoorn, "Transformation of Existing Programs into Autonomic and Self-healing Entities," 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'07), 2007.
20. Michael E. Shin and Daniel Cooke, "Connector-Based Self-Healing Mechanism for Components of a Reliable System," Workshop on the Design and Evolution of Autonomic Application Software (DEAS 2005), St. Louis, Missouri, May 21, 2005.