Architectural Styles for Adaptable Self-Healing Dependable Systems

Matthew J. Hawthorne
Empirical Software Engineering Lab (ESEL)
ECE, The University of Texas at Austin
hawthorn@ece.utexas.edu

Dewayne E. Perry
Empirical Software Engineering Lab (ESEL)
ECE, The University of Texas at Austin
perry@ece.utexas.edu

ABSTRACT
Of all the possible architectural approaches to improving the dependability of software-based systems, only systems designed to be self-healing are able to adapt themselves at runtime in response to changing environmental or operational circumstances. In this paper, we discuss the basic functional requirements for self-healing systems, and explore a number of major issues related to architectural designs for incorporating runtime reflection and adaptation into software systems. We present several conceptual architectures for self-adaptation, and analyze the features, advantages and disadvantages of each architecture. Finally, we propose enhancements to currently used architectural description languages (ADLs) and system design tools to add explicit support for self-adaptive architectures.

Categories and Subject Descriptors
[**Software Engineering**]: D.2.11 Software Architectures – *patterns, domain-specific architectures*; D.2.9: Management – *software configuration management, life cycle*; D.2.7: Distribution, Maintenance, and Enhancement – *restructuring, reengineering*; D.2.4: Software/Program Verification – *reliability, programming by contract, model checking*; D.2.13: Reusable Software – *domain engineering*.

General Terms
Design, Management, Reliability, Performance, Security.

Keywords
Dependable Systems, Self-Adaptive Systems, Self-Healing Systems.

1. INTRODUCTION
Adaptive, self-healing systems are designed to reconfigure themselves in response to changing environmental or operational conditions, including errors or failures resulting from hardware or software faults, network problems, system loads, unexpected user or data inputs, and other runtime conditions. Self-adapting systems are intrinsically more flexible than fixed-configuration or human-configurable systems; the increase in flexibility is directly proportional to the number and scope of options the system adaptability framework supports for reconfiguring the system, along with the ability of the system's self-reflection mechanism to accurately detect and diagnose conditions for which reconfiguration is necessary or desirable, and the ability of the reconfiguration mechanism to effectively reason about input from self-reflection and select self-healing actions that utilize the configuration options the system provides. This is summarized in the following equation:

$$\Delta Flexibility \equiv Options \times Reflection \times Reasoning ,$$

where $\Delta Flexibility$ is the change in system flexibility as a result of adding self-healing functionality; *Options* is the number of different kinds of reconfiguration options the adaptation framework supports; *Reflection* is a measure of the ability of the self-reflection mechanism to accurately detect and diagnose problems; and *Reasoning* is a measure of the ability of the reconfiguration mechanism to effectively reconfigure the system using the available options.

Because of this potential increase in system flexibility, and by implication, dependability in the face of unexpected runtime conditions, designing self-healing capabilities into software-based systems can greatly enhance their dependability whenever all potential permutations of runtime conditions and failures cannot be known *a-priori* during system design, as is normally the case. In the remainder of this paper, we discuss how self-adaptation is related to system dependability, followed by a brief analysis of the basic architectural requirements for self-adaptive systems. We then discuss some of the more important architectural design issues for self-adaptive systems, and present several different architectural approaches for implementing self-healing systems. We analyze each of the architectures, using them to illustrate architectural directions that show the most promise for producing the greatest gains in dependability. We discuss related research and present our conclusions including several specific recommendations for enhancements in architectural description languages (ADLs) and system design environments to incorporate explicit support for self-adaptive architectural frameworks.

2. ADAPTATION AND DEPENDABILITY
Software-based systems are designed and implemented to operate under a given set of environmental and operational circumstances. Every system design is based on assumptions about the state of the system, the platforms on which it will execute, and the networks over which it will communicate. These assumptions may be either explicit (highly desirable), or implicit (undesirable, but common). Even well-designed systems tested against an explicit set of expected conditions often experience faults or failures when unforeseen circumstances violate one or more expectations. While robust architectures and good system design

ICSE'05, May 15–21, 2005, St. Louis, Missouri, USA.

practices have always led to systems that respond relatively gracefully to system or platform faults, even if that means little more than to say that well-designed systems fail relatively gracefully, for most applications and platforms, highly dependable systems must also be self-adaptive – that is, they must incorporate functionality that enables them to reconfigure themselves in response to unforeseen conditions – to actually achieve high levels of dependability.

The increase in dependability from self-healing is proportional to the probability that a given explicit or implicit assumption will be violated, and the probability of a system failure given such a violation, and the probability that self-healing will prevent the failure, divided by the overall probability of system failure for any reason.

$$\Delta Dependability|Self\text{-}Healing \equiv (P(violation) \times P(fail|violation) \times P(adapt|violation)) / P(failure_{system}),$$

where $\Delta Dependability|Self\text{-}Healing$ is the change in system dependability as a result of introducing self-healing functionality that may address a given runtime violation of the system assumptions; if $P(x|y)$ is the probability that x will occur given y, then $P(violation)$ is the unconditional probability that a given violation will occur, $P(fail|violation)$ is the probability that the violation will cause a system failure if it does occur, $P(adapt|violation)$ is the probability that the self-healing mechanism will successfully adapt the system in response to the violation, and $P(failure_{system})$ is the overall probability that the system will fail for any reason.

By extension, the overall change in system dependability is proportional to the sum of the increases in dependability over every possible violation of the system's operational assumptions.

For each $i \in \{assumptions\}$, where $\{assumptions\}$ is the set of all explicit and implicit design and implementation-time system assumptions that may be violated at runtime:

$$\Delta Dependability_{system}|Self\text{-}Healing \equiv \sum_i (P(violation_i) \times P(fail|violation_i) \times P(adapt|violation_i)) / P(failure_{system}),$$

where $\Delta Dependability_{system}|Self\text{-}Healing$ is the overall change in system dependability as a result of introducing self-healing functionality.

Note that neither of these equations takes into account the possibility that a poorly designed self-healing mechanism might degrade system dependability in other ways.

While the preceding equations are admittedly simplistic, even such simplistic equations demonstrate the potential for adaptive self-healing systems to enhance dependability where any of the assumptions underlying the system implementation may be violated. And unless the system is being built on a completely deterministic execution platform, using machine language or a real-time operating system, and all possible combinations of data input and system state are able to be generated and tested before the system is deployed, self-healing is a critical strategy for enhancing system dependability for any applications where the cost of system failure would be prohibitive.

3. ADAPTIVE SYSTEM REQUIREMENTS

Since self-adaptive systems must detect runtime conditions for which some kind of adaptation should be done, and then perform some kind of adaptive configuration in response, it follows that adaptive system architectures have three basic requirements: a *reflection* mechanism to detect internal or external conditions to which the system should respond; a *reasoning* mechanism to determine what actions should be performed in response to input from the reflection mechanism; and a *configuration* mechanism to perform the necessary changes to repair or optimize the system as directed by the reasoning mechanism. Adaptation implementations can range from simple, ad-hoc solutions consisting of hard-coded inline program statements that check for and respond to specific runtime conditions, to comprehensive hierarchical agent-based monitor-configurator frameworks.

Figure 1 shows a conceptual model for self-healing architectures. The adaptation mechanism consists of components that perform *reflection*, *reasoning* and *configuration*. Each adaptation component either monitors or configures one or more aspects of the system or the system environment.

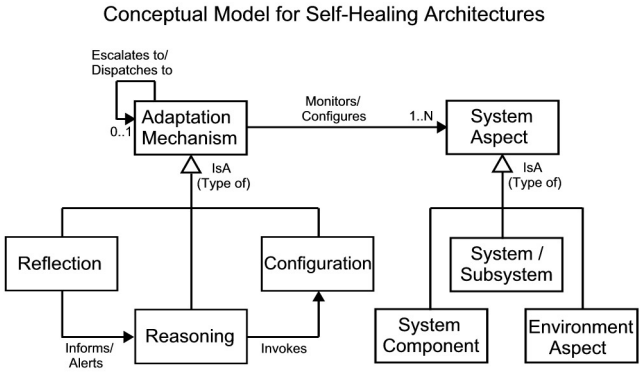

Figure 1. Conceptual architectural model.

4. DESIGN ISSUES

Of the many aspects of self-healing architectural design that could be discussed, this section focuses on four major design issues related to the organization and communication among the major adaptive entities in the architecture (reflection, reasoning and configuration). Architectural aspects explored include the degree of inter-component coupling, the direction and level of inter-component communications, configuration dispatch styles, and ways to specify configuration functionality. In this paper, runtime reflection components are called *monitors*, and configuration components are called *configurators*. Configurators may also encapsulate reasoning functionality; separate system-level reasoning components are called *configuration managers*.

4.1 Tight vs. Loose Coupling

The first major architectural design issue for adaptive systems is the level of inter-component coupling. In a tightly coupled adaptation system, the reflection, reasoning and configuration components have direct, explicit interdependencies. Examples of tight coupling include systems where the monitoring, configuration and configuration management implementation entities (components, objects, or methods) explicitly invoke one another without an intervening layer of logical abstraction.

Tightly coupled systems may be somewhat simpler to implement, but this apparent simplicity comes at the cost of flexibility and scalability. Tight coupling is a particularly risky architectural style for a self-healing system because certain types of tight coupling can make the self-healing mechanism itself vulnerable to being disabled when system components fail, leading to the paradoxical situation where the self-healing system itself may need healing. And since any system where hard-coded references must be recoded in order to change implementation objects is probably too inflexible to use for serious large-scale system development anyway, loosely coupled adaptation frameworks are far more practical for large, complex systems. Loosely coupled systems can also exploit the full abstractive and adaptive power of the component-connector abstraction, which allows connectors to handle and encapsulate inter-component connection and communication concerns, freeing system components to focus on application-related functionality.

4.2 Instance vs. Intent-based Selection

How the adaptation system or its components specify one another when they need to request a service (e.g., reconfiguration) is another important aspect of a self-healing architecture, and is closely related to the degree of inter-component coupling. Specifying a particular implementation of a monitor or configurator leads to logical dependencies, or *logical tight coupling*, among the adaptation mechanism entities. A better design is for the adaptation mechanism to be designed so that its components identify one another by functional *intent* [8], i.e., by their logical or functional role in the system. This requires all adaptation entities to be identifiable by functional role, and it requires the system to include a directory service, service provider request mechanism, or similar functionality that the system or its components can use to locate and establish a connection with the appropriate component or components that will fulfill the required service. Note that intent- or functionality-based specification only results in looser coupling if the supporting directory or service request mechanism is flexible (i.e., functional role to component instance mappings are not hard-coded, the system supports adding and removing component instances at runtime, etc.).

4.3 Peer-to-Peer vs. Hierarchical Organization

Peer-to-peer approaches allocate self-adaptive functionality to symmetrical sets of monitor and configurator peer components. Each monitor interacts with a peer-level configurator, and vice-versa. The configurator peer may delegate configuration duties, or escalate to a higher-level configurator if necessary (see the aggregator-escalator-peer style, below). Similarly, the monitor peer may pass alerts and other messages on to a higher-level aggregator monitor, which communicates with its higher-level configurator peer.

Peer-to-peer styles are simple and symmetrical, and are similar in many ways to common network protocol architectures. However, more hierarchical approaches allow monitor messages to reach higher-level configurator or configuration manager components, which are able to make more comprehensive subsystem- or system-level reconfiguration decisions if needed.

4.4 Single vs. Multiple Dispatch Configuration

It may be desirable to use a "chain of commands" style of configurators; if one configurator cannot handle a given configuration request, or if a configurator unsuccessfully attempts to handle a request, the configurator or the configuration manager can pass the request to the next configurator in the chain. Alternately, the configuration functionality can be allocated to a sequential or branching chain of configurators; each configurator performs its part of the configuration and then passes the request to the next configurator until the request has traversed the entire chain or tree, and all the configuration actions have been performed.

5. ARCHITECTURAL APPROACHES

In this section, we present and discuss several reference architectural styles that illustrate the architectural design issues from the previous section. Each architecture includes a diagram, a brief explanation of how the architectural style works, and a short discussion of some of its strengths and weaknesses as they relate to the previously discussed design issues, and any other issues that may affect the utility of the style for practical system design.

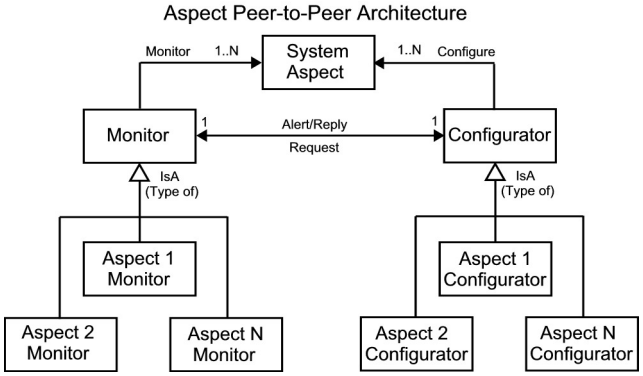

Figure 2. Aspect Peer-to-Peer architectural style.

5.1 Aspect Peer-to-Peer

The aspect peer-to-peer architectural style is a simple approach that allocates a monitor component to monitor each aspect of the system or environment that needs to be monitored, and a peer configurator component to reconfigure the system (or delegate configuration; see, e.g., the aggregator-escalator-peer style below). This style is similar to network protocol architectures where each level in the protocol stack has a peer-level counterpart at the other end of the communication link. Such a strict peer-to-peer approach, while conceptually simple, has serious drawbacks for most self-healing applications, because an actual configurator or configuration manager may require output from more than one monitor to make a decision about the optimal reconfiguration actions to take. As a result, the aspect peer-to-peer approach needs to be combined with one or more additional strategies (e.g., aggregator-escalator-peer or chain-of-configurators, described below) to be useful. *Figure* 2 shows the aspect peer-to-peer architectural style.

5.2 Aggregator-Escalator-Peer

The aggregator-escalator-peer adaptation style overcomes some of the limitations of a strictly peer-to-peer monitor-configurator approach by allowing monitors to pass their outputs to higher-level aggregator monitors, which then package the combined

output from the lower-level monitors into a coherent composite package. This composite output is then passed to a peer configurator, which benefits from getting a single consistent picture of related monitor outputs, instead of receiving all the outputs separately, and having to either cache the data until it has received enough (maintaining data ages and TTL timers as well), or worse, having to respond to each low-level monitor alert separately.

Figure 3 shows a simple example that demonstrates how a self-healing system can use an aggregator-escalator-peer adaptation architecture to aggregate output from multiple environmental monitors, and pass a comprehensive set of monitor output data to a higher-level configurator that will be able to make better configuration decisions and operate more efficiently than it would if it received and responded to each low-level monitor alert separately.

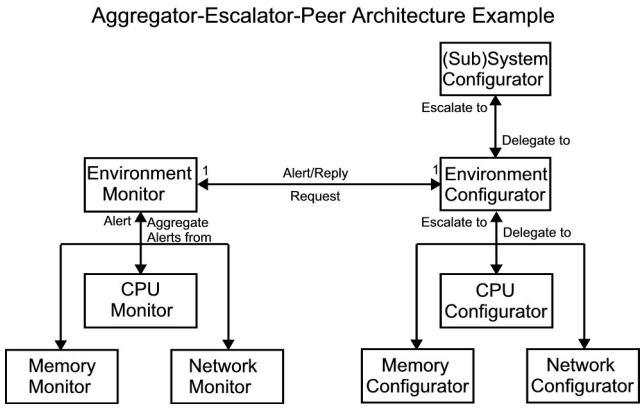

Figure 3. Aggregator-Escalator-Peer architectural style.

5.3 Chain-of-Configurators

The chain-of-configurators architectural style is really two different styles. In the first variation, similar to the Chain-of-Responsibility design pattern, multiple configurators are chained together in a linear or other traversable structure (e.g., tree). The configuration request is passed along the configurator chain until a configurator is able to successfully handle the request. This enhances loose coupling, and also makes it easy to implement runtime addition and removal of configurator instances. The chain-of-configurators style allows self-healing systems to try all available configuration strategies for repairing a given problem. The system can then promote successful strategies and demote or prune less-successful strategies while the system is running, just by manipulating the list.

The second chain-of-configurators variant is similar to the first, except that it utilizes a Visitor pattern, in which configurator chaining is used to compose higher-order configuration functionality using a group of lower-order configurators. This visitor-style variant enhances the power and flexibility of the adaptability mechanism by allowing the adaptation reasoning or configuration management mechanism to construct new configuration solutions at runtime from existing lower-level solutions. While the visitor variant does not enhance loose coupling by itself, combining the two approaches can lead to powerful and flexible configuration solutions.

The visitor-style variant of the chain-of-configurators can also be used to implement configuration functionality similar to the aggregator-escalator-peer style in peer-to-peer monitor-configurator systems; in this case, the configurator chain aggregates the output from all the necessary peer-level monitors, and passes it on to the next higher-level configurator.

Since both variants of the chain-of-configurators style lack any intrinsic organization, beyond the traversal order of the list or tree data structures used, the configuration mechanism or human system engineer must enforce any ordering constraints, etc., on the set of configurator components in the chain. For example, to implement a strategy that tries all localized configuration strategies first, then escalates to a higher-order configurator or the system configuration manager, the last-resort configuration entity must be at the end of the chain.

Figure 4 depicts the chain-of-configurators architectural styles for self-adaptive systems, including an optional configuration manager.

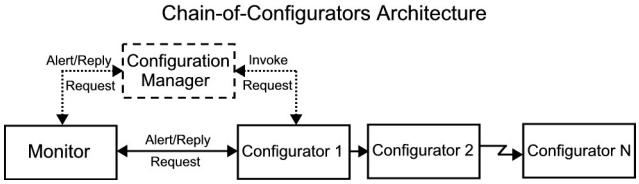

Figure 4. Chain-of-Configurators architectural style.

5.4 Configuration Manager

The configuration manager architectural style utilizes a configuration manager component to centralize and encapsulate the logic involved with reasoning about configuration changes. In addition, the configuration manager is typically also the central communication clearinghouse for the other adaptation entities (e.g., monitors and configurators), since the configuration manager must ultimately make any reconfiguration decisions and communicate those decisions to the configurator components that will be involved in making the required changes.

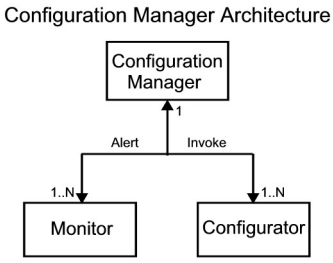

Figure 5. Configuration Manager architectural style.

Configuration manager architectures can be combined with other architectures, e.g., aggregator-escalator or chain-of-configurators), but regardless of other architectural styles used, the centralized control imposed by a configuration manager strongly affects the fundamental nature of the self-configuration mechanism. While the peer-to-peer approaches are intended to support more decentralized and distributed modes of inter-component communication and configuration reasoning, a configuration manager centralizes at least the system's self-healing logic, and closely related supporting functionality such as learning and

planning. A configuration manager may still utilize loose coupling if it delegates monitoring and reconfiguration, and the system includes a mechanism that supports functional intent-based component specification or a similar technique. Figure 5 shows the basic configuration manager style for self-adaptation.

6. DISCUSSION

Loosely coupled adaptation architectures are better than tightly coupled architectures. In particular, systems where the components that reason about and perform reconfiguration can be specified by function instead of explicitly by implementation type or instance will be much more flexible than systems where the configuration structure is deterministically programmed prior to deployment.

Low-level monitors and configurators combined with aggregation and escalation to enable higher-level solutions can be a powerful and flexible architectural style for self-healing systems. Configurator chaining gives the system even more flexibility (chain of responsibility) or expressive power (visitor style) for performing configurations.

Largely because a centralized configuration control component is easier to implement than, e.g., a distributed agent-based service, configuration managers are common architectural features in self-healing systems. A central configuration manager also offers some intrinsic advantages for aspects of system ownership like system administration and security. Depending on the design details of the configuration manager, it may introduce a potential single point of failure, or alternately, the configuration manager itself may be designed to be self-healing. Mirrored servers and similar redundancy strategies may also be employed to mitigate against single-point failure concerns. Care must be taken to ensure that a single configuration manager-based architecture will be scalable enough for the system being designed, since scalability concerns may weigh in favor of more federated or distributed configuration approaches as systems become very large or distributed.

7. CONCLUSIONS

Since dependable systems must often be implemented using less-than-dependable platforms and networks, and even dependable hardware and software still fails occasionally, especially in the presence of unexpected user input or other data- and state-related issues, adaptive self-healing systems are an important strategy for software engineers who develop such systems. We argue for the potential of self-healing mechanisms to enhance system dependability; then, starting from the basic requirements of self-healing systems, we present a general conceptual architecture for adaptive systems, discuss several important architectural design issues, briefly analyze several architectural styles for self-healing systems, and use that analysis to recommend several specific architectural style directions that show the most promise for implementing adaptive self-healing systems.

Finally, to enable system architects to more easily select and incorporate appropriate adaptation architectures, and to promote separation of concerns between application and adaptation architectures, we propose the following enhancements to current architectural description languages (ADLs) and system design environments to incorporate explicit support for self-healing architectural frameworks:

- *Self-healing ADLs*: Add new syntactic constructs to specify, define and combine architectural strategies for self-healing systems, along with component-level interfaces for runtime monitoring and configuration, etc.
- *System design toolsets*: Enhance with explicit self-healing system support, including self-adaptive architectural styles.
- *Visualization*: Add self-healing design views such system *aspects views*; *monitor views*; *alert condition editors*; *configurator views*; and *alert type views*.
- *Additional requirements*: Enhance *meta-data* support, e.g., for system- and component-level properties that may be monitored.

Adding capabilities such as these to ADLs and system design tools promises to contribute significantly to the productivity of system designers who need to design architectures for self-healing systems. But even more importantly, this kind of explicit design environment support will immediately help engineers design better-quality self-healing architectures, as well as contributing to improved documentation, sharing and standardization of architectural styles for self-healing systems, all of which will serve to enhance system dependability.

8. REFERENCES

[1] Blair, G., Coulson, G., Blair, L., Duran-Limon, H., Grace, P., Moreira, R., Parlavantzas, N. *Reflection, Self-Awareness, and Self-Healing.* WOSS'02, Nov. 18, 2002.

[2] Brandozzi, M. and Perry, D. *Architectural Prescriptions for Dependable Systems*. WADS'2002, May 2002

[3] Cheng, S., Huang, A., Garlan, D., Schmerl, B., and Steenkiste, P. *An Architecture for Coordinating Multiple Self-Management Systems*. WICSA-4, 2004.

[4] Garlan, D., Cheng, S., Schmerl, B. "Increasing System Dependability through Architecture-based Self-Repair," *Architecting Dependable Systems*. de Lemos, R., Gacec, C., and Romanovsky, A., eds., LNCS 2677, Springer-Verlag, 2003.

[5] Garlan, D., Schmerl, B. *Model-Based Adaptation for Self-Healing Systems*. WOSS'02, 2002.

[6] Hansson, H., Akerholm, M., Crnkovic, I., and Torngren, M. "SaveCCM – a Component Model for Safety-Critical Real-Time Systems." In *Proceedings of 30 Euromicro Conference, Special Session Component Models for Dependable Systems*, Sept. 2004.

[7] Hawthorne, M., Perry, D. *Applying Design Diversity to Aspects of System Architectures and Deployment Configurations to Enhance System Dependability*. WADS'04, June 30, 2004.

[8] Hawthorne, M., Perry, D. *Exploiting Architectural Prescriptions for Self-Managing, Self-Adaptive Systems: A Position Paper*. WOSS'04, Oct. 31-Nov. 1, 2004.

[9] Klein, F., Giese, H., "Separation of concerns for mechatronic multi-agent systems through dynamic communities." *Software Engineering for Multi-Agent Systems III: Research Issues and Practical Applications*. Choren, Ricardo, Garcia, Alessandro, Lucena, Carlos, Romanovsky, Alexander, eds., LNCS, Springer-Verlag, Dec. 2004.

[10] Koopman, P. Elements of the Self-Healing System Problem Space. WADS'03, May, 2003.

[11] de Lemos, R., Fiadeiro, J. *An Architectural Support for Self-Adaptive Software for Treating Faults*. WOSS'02, 2002.

[12] Perry, D., Wolf, A. *Foundations for the Study of Software Architecture*. ACM SIGSOFT Software Engineering Notes, 17(4):40, Oct. 1992.