An Architecture for Constructing Self-Evolving Software Systems

Chrysanthos Dellarocas
Center for Coordination Science
Massachusetts Institute of Technology
MIT Room E53-315
Cambridge, MA 02139, USA
+1 (617) 258-8115
dell@mit.edu

Mark Klein
Center for Coordination Science
Massachusetts Institute of Technology
MIT Room E40-169
Cambridge, MA 02139, USA
+1 (617) 253-6796
m_klein@mit.edu

Howard Shrobe
Artificial Intelligence Laboratory
Massachusetts Institute of Technology
MIT Room NE43-839
Cambridge, MA 02139, USA
+1 (617) 253-7877
hes@ai.mit.edu

1. ABSTRACT

This paper proposes an architecture for "closing the feedback loop" over the entire software evolution process and enabling the construction of *self-evolving software systems*. Self-evolving software systems are capable of automatically detecting when changing external circumstances or internal conditions can be better handled by alternate software modules and able to dynamically swap these modules into place. Our approach integrates results of recent work in software architecture and dynamic reconfiguration. Furthermore, it introduces the novel concept of an *evolution engine*, which sits alongside a running application, oversees its execution and automatically decides when and how to evolve it. The evolution engine relies on models of the current and alternative system configurations as well as on a generic base of reusable knowledge about software exceptions.

1.1 Keywords

Dynamic architectures and reconfiguration, architectural evolution

2. INTRODUCTION

Most of the recent work in dynamic software evolution focuses on dynamic reconfiguration and assumes that the decisions of *when* and *how* to evolve a software system are performed manually (for example [6,8,11]). In a number of highly volatile and time-critical domains this is not good enough. Automatic target recognition, military logistics and air-traffic control are some examples of domains where faulty or sub-optimal software behavior can cause disaster in a matter of seconds. Software systems in such domains must be *self-evolving*, that is, prepared to automatically detect when changing external circumstances or internal conditions can be better handled by alternate software modules and able to dynamically swap these modules into place.

This paper proposes an architecture for "closing the feedback loop" over the entire software evolution process and enabling the construction of *self-evolving software systems*. Our approach integrates results of recent work in software architecture and dynamic reconfiguration. Furthermore, it introduces the novel concept of an *evolution engine*, which sits alongside a running application, oversees its execution and automatically decides when and how to evolve it. The evolution engine relies on models of the current and alternative system configurations as well as on a generic base of reusable knowledge about software failures and exceptions.

We are finding that software architecture has an important role to play in the context of self-evolving software but that its definition has to be extended. In order to be able to automatically detect *when* a software system is not meeting its goals and decide *how* to evolve it into a more viable state, rich run-time information structures, which combine together descriptions of a system's *structure*, *intended behavior*, *design rationale* and *design alternatives* are proving to be fundamental prerequisites.

3. OVERVIEW

Our architecture for constructing self-evolving software systems consists of four components (Figure 1):

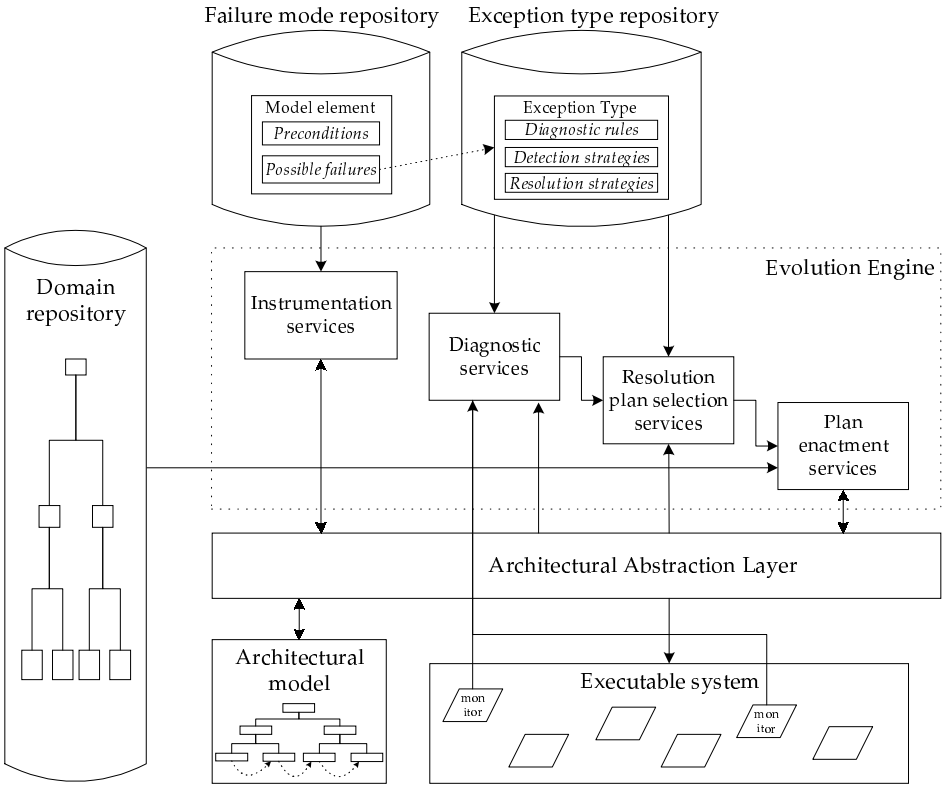

Figure 1: Overview of the proposed architecture

- A *domain repository*, which captures the space of alternative solutions in a given application domain.
- An *architectural model*, which reflects the current goals, structure and design rationale of the running system at any given time.
- An *evolution engine*, which monitors whether the system is meeting its goals, detects exceptional conditions, diagnoses the underlying causes and then selects and enacts an evolution plan for resolving them.
- An *architectural abstraction layer*, which allows the evolution engine to express dynamic system changes at the architectural level and ensures that these changes are properly applied to the running system.

3.1 The Domain Repository

Our approach is intended for application domains where, although the space of alternative solutions is generally well understood, insufficient information exists during design-time for deciding which solution within that space would be most appropriate in a given context. For example, in the domain of automatic target recognition, although the space of alternative algorithms is well understood, the decision of which algorithm to use each time often cannot be reliably made until after a missile has been fired.

To develop self-evolving systems in such domains, the first prerequisite is to capture an adequate portion of the design space in a domain repository. Such a domain repository will store information about the structure, behavior and design tradeoffs (performance, requirements, limitations, etc.) of alternative software configurations in the given application domain. The human designer will use this information in order to derive the initial executable system configuration. During run-time, the evolution engine (see below) will consult the domain repository in order to select an alternative system configuration after some condition has signaled the need for it.

3.2 The Architectural Model

To generate a software system in a highly volatile domain, an initial set of design choices, however uncertain, must be made and an initial configuration of components must be put together. In addition to the domain repository, an architectural model is needed to capture information about the current configuration of the system. In addition to information about the *structure* of the system, the architectural model must also contain information about the system's *intended behavior* and the *design rationale* that has lead to it. The evolution engine will make use of that model in order to understand what goals a computation is supposed to achieve (and under which assumptions the

system believes that it can achieve them) so that it can monitor whether it has been successful.

3.3 The Evolution Engine

The evolution engine sits alongside a running application, oversees its execution and automatically decides when and how to evolve it. It relies on information contained in the architectural model and design repository as well as on a generic base of reusable knowledge about software failure modes and exception types. The evolution engine removes the need for the system designer to anticipate all possible exceptions that might occur during the system's operation and explicitly "hardcode" ways to deal with them.

Anticipating and Detecting Exceptions

The first step in enabling self-evolution is to determine, given a model of the system's current configuration, the ways in which the system might fail and then to instrument the system with additional monitors so that these failures can be detected. One approach for achieving this is to compare every element of the architectural model against a repository of generic architectural patterns, resource types and constraint types annotated with the ways in which they can fail, i.e. with their characteristic *exception types*. Failure modes for a given element can be uncovered using failure mode analysis [12]. In the case of software architecture, more specialized analysis techniques can uncover certain possible failures, such as deadlock and race conditions (for example [1,5,7]).

While systems can fail in many different ways, such failures have a relatively limited number of different manifestations, including timeouts, constraint and resource violations, etc. Every exception type includes pointers to *exception detection* strategies. Such strategies are parameterized architecture modification scripts. They specify how to instrument the system with additional monitors that check for signs of actual or impending failure. The instrumentation component of the evolution engine instantiates and applies these scripts to the architectural model through the Architectural Abstraction Layer (see below) in order to augment the base system with appropriate monitor components.

Diagnosing Exceptions

During run-time, monitor components generate appropriate events when exception manifestations are detected. Just as in medical domains, selecting an appropriate intervention requires understanding the underlying cause of the problem, i.e. its *diagnosis*. A key challenge here, however, is that the symptoms revealed by the exception detection processes can suggest a wide variety of possible underlying causes. Many different exceptions (e.g. "resource not available", "message misrouted" etc.) typically manifest themselves, for example, as timeouts.

Our approach is based on heuristic classification [2]. This approach works by traversing a taxonomy of generic exception types. When an exception is detected, the evolution engine traverses the exception type taxonomy top-down like a decision tree. It starts from the diagnoses implied by the manifest symptoms and then iteratively refines the specificity of the diagnoses by eliminating exception types whose defining characteristics are not satisfied.

Selecting a Resolution Strategy

Once an exception has been detected and at least tentatively diagnosed, the evolution engine must prescribe a plan that resolves the exception and evolves the system to a viable state. This can be achieved, in our approach, by selecting and instantiating one of the generic *exception resolution* strategies that are associated with the hypothesized exception type. Examples of resolution strategies include: replacing a faulty resource with an alternative resource, replacing the current configuration with an alternative configuration from the domain repository, adjusting the set of assumptions and constraints in the architectural snapshot to reflect the new state of the environment, etc. Since an exception can have several alternative resolution strategies, each suitable for different situations, the evolution engine uses a decision procedure identical to that used in diagnosis in order to find the right one.

Evolving the System

After a resolution strategy has been selected, the evolution engine enacts it. A typical resolution strategy would traverse the domain repository and select an alternative configuration, which appears to be better suited to the latest information about the system requirements and environment. Following that, an evolution plan is constructed for moving the system from the current configuration to the new configuration. The evolution plan is expressed as a sequence of architecture modification operations. Examples of such operations include: adding, removing or replacing a component with an alternative component, adding, removing or replacing a set of constraints with alternative constraints, etc. The evolution plan is applied to the architectural model through the Architectural Abstraction Layer.

3.4 The Architectural Abstraction Layer

The goal of the Architectural Abstraction Layer is to isolate the evolution engine from low-level synchronization and consistency issues that arise during dynamic reconfiguration [6,8] and to allow it to access information about the running system and express configuration changes at the architectural level. More specifically, the AAL provides an interface through which the evolution engine can access and modify the architectural model of the running application using a set of high-level architecture modification operations, such as, add/remove a component, add/remove a constraint, etc. The AAL then ensures that these changes are correctly applied to the running system and manages the

synchronization and consistency issues associated with dynamic change.

4. CURRENT STATUS

The Adaptive Systems and Evolutionary Software Group at MIT (`http://ccs.mit.edu/ases`) is in the process of constructing a prototype implementation of the architecture presented in this paper. Our efforts are focused on the following three areas:

- designing and implementing the various services of the evolution engine; experimenting with alternative approaches towards monitoring, exception diagnosis, plan selection and plan enactment
- developing prototype versions of the failure mode and exception type repositories; exploring the extent to which we can define generic exception detection and resolution strategies, which can then be combined with a domain repository to provide application-specific exception handling
- designing the interfaces of the evolution engine with the architectural and domain models; understanding how much information about the system's structure, goals and design rationale is necessary to enable self-evolution; integrating this information with existing architectural description notations

Our intention is to base the remaining components of our architecture on existing research prototypes in the areas of domain and requirements engineering (e.g. [3,4]), architectural and functional description languages (e.g. [9,10]) and dynamic reconfiguration (e.g. [6,8,11]).

5. REFERENCES

[1] R. Allen and D. Garlan. A Formal Basis for Architectural Connection. *ACM Trans. on Software Eng. Method. (TOSEM)*, 6 (3), July 1997, pp. 213-249.

[2] W. J. Clancey. Heuristic Classification. *Artificial Intelligence*, 27(3), 1985, pp. 289-350.

[3] A. Dardenne, A. van Lamsweerde and S. Fickas. Goal-directed requirements acquisition. *Science of Computer Programming*, Vol. 20, 1993, pp. 3-50.

[4] C. Dellarocas, J. Lee, T.W. Malone, K. Crowston and B. Pentland. Using a Process Handbook to Design Organizational Processes. *Proc. AAAI 1994 Spring Symposium on Computational Organization Design*, Stanford, CA, March 21-23, 1994, pp. 50-56.

[5] P. Inverardi and A.L. Wolf. Formal Specification and analysis of software architectures using the chemical abstract machine. *IEEE Trans. on Software Eng.*, 21 (4), Apr. 1995, pp. 373-386.

[6] J. Kramer and J. Magee The Evolving Philosophers Problem: Dynamic Change Management. *IEEE Trans. on Software Eng*., 16 (11), Nov. 1990, pp. 1293-1306.

[7] D.C .Luckham et. al. Specification and Analysis of Software Architecture using Rapide. *IEEE Trans. on Software Eng*., 21 (4), April 1995, pp. 336-355.

[8] K. Moazami-Goudarzi and J. Kramer. Maintaining Node Consistency in the Face of Dynamic Change. *Proc. of 3rd Int'l Conf. on Config. Distrib. Systems (CDS '96)*, Annapolis, MD, May 1996; pp. 62-69.

[9] N. Medvidovic and R.N. Taylor. A Framework for Classifying and Comparing Architecture Description Languages. *Proc. 6th European Soft. Eng. Conf*., Zurich, Switzerland, Sept. 22-25, 1997, pp. 60-76.

[10] J.W. Murdock. Modeling Computation: A Comparative Synthesis of TMK and ZD. Georgia Inst. Of Tech. College of Computing Technical Report GIT-CC-98-13, April 1998.

[11] P. Oreizy, N. Medvidovic, and R.N. Taylor. Architecture-Based Runtime Software Evolution. *Proc. 20th Int'l Conf. on Soft. Eng. (ICSE'98)*, Kyoto, Japan, April 19-25, 1998, pp. 177-186.

[12] D. Raheja. Software system failure mode and effects analysis (SSFMEA)-a tool for reliability growth. *Proc. of Int'l Symp. On Reliability and Maintainability (ISRM'90)*, Tokyo, Japan, June 1990, pp. 271-277.