Mae
An Architectural Evolution Environment

Roshanak Roshandel

Computer Science Department
University of Southern California
Los Angeles, CA 90089-0781 U.S.A.
roshande@usc.edu

ABSTRACT

We present Mae an architectural evolution environment, built upon a system model that combines architectural and configuration management concepts into a single representation. Through Mae, users can specify architectures (in terms of their constituent components, connectors, and interfaces) in a traditional manner, manage the evolution of the architectures using a check-out/check-in mechanism that tracks all changes, select a specific architectural configuration, and analyze the consistency of a selected configuration.

1. ARCHITECTURAL CHANGE

Consider the following scenario. An organization specializing in software development for mobile platforms is commissioned by a local fire department to produce an innovative application for "on the fly" deployment of personnel in situations such as natural disasters and search-and-rescue efforts. Following good software engineering practices, the organization first develops a proper architecture for the application in a suitable architectural style, then models this architecture in an architecture description language (ADL), refines the architecture into a module design, and, finally, implements the application impeccably. The new application is an instant hit, and fire and police departments across the country adopt it. Motivated by this success, as well as by demands for similar capabilities from the military, the organization enters a cycle of rapidly advancing the application, creating add-ons, selling upgrades, adapting the application to different hardware platforms (both stationary and mobile), specializing the application for its various customers, and generally increasing its revenue throughout this process.

Configuration management (CM) systems have long been used to provide support for these kinds of situations. This, however, leads to problems with the above scenario: as the application evolves, so does its architecture. These architectural changes must be managed in a manner much like source code, allowing the architecture to evolve into different versions and exhibit different variants. One solution is to store the entire architectural description in a single file and track its evolution using an existing CM system (called *coarse-grained versioning*). An alternative solution is to version each architectural element in a separate file (called *fine-grained versioning*). Problems associated with each of these approaches reduce their effectiveness in managing architectural evolution. These problems are briefly discussed in Section 2.

Any solution to managing architectural evolution must support an architect in using: 1) multiple versions of a single architectural element that are part of the same configuration, 2) optional elements, 3) variant elements, 4) elements that are both optional and variant, and 5) relations among optional and variant elements. To address these issues and to mitigate the problems associated with use of traditional CM systems, we have developed a novel approach called Mae. Mae combines techniques from the fields of software architecture and configuration management to make two unique contributions: 1) an architectural system model that facilitates capturing the evolution of an architecture and its constituent elements, and 2) an integrated environment that supports managing the evolution of architectures. Details of the system model may be found in [2]. We propose to demonstrate Mae's architectural evolution environment and its functionality in *designing*, *analyzing*, and *evolving* software architectures.

2. EXISTING APPROACHES

Even though it is possible to manage the evolution of the architectural artifacts using traditional CM systems, we argue that this cannot be done effectively.

2.1. Coarse-grained Versioning

One possible approach to using an existing CM system for managing architectural evolution is to store and version the entire architectural description as a single file. This solution is akin to storing and versioning the entire source code of a software program as a single file. Clearly, managing artifacts at such a coarse-grained level leads to severe problems since any single change would result in a new version of the entire architectural specification. Moreover, the presence of multiple optional and variant elements leads to a combinatorial explosion of branches, due to the fact that each potential combination must be explicitly specified. Finally, this approach prevents the use of multiple versions of the same artifact within a single architecture. In sum, these shortcomings make versioning an entire architectural specification as a single artifact a highly undesirable solution for managing architectural evolution.

2.2. Fine-grained Versioning

Versioning fine-grained artifacts is considered a better approach to managing source code evolution than coarse-grained versioning. However, the analogy does not hold when applied to architectural evolution. Fine-grained versioning leads to serious consistency problems due to the fact that the architectural specification and the CM system capture duplicate information about the architectural configuration. Any change in the composition of the architectural configuration must be reflected in the CM system, and vice versa. Given that much of architectural design resolves around composing an architectural configuration, this becomes a recurrent and potentially error-prone activity.

This approach also requires extensive use of branching to manage optionality and variability. Traditional CM techniques that support branching (e.g., differencing and merging) work well for source code. However, they simply do not work for

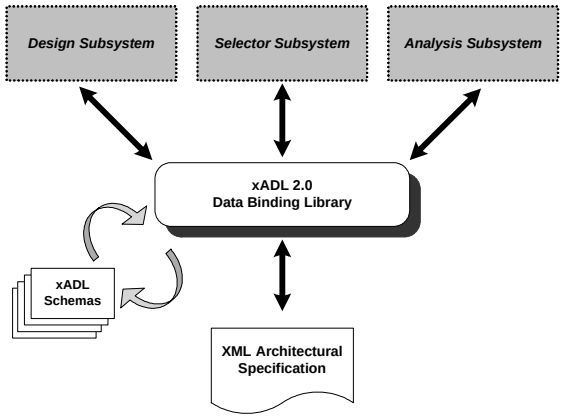

Figure 1. Mae's Architecture

architectural specifications, because of a difference in the level of granularity (i.e., lines of code in a source file versus components, connectors, links, and so on in an architectural specification). As a result, using a traditional CM system would force an architect into storing each potential architectural configuration on a separate branch. Finally, this approach requires breaking up an architectural specification into numerous small files to be managed separately. Even for a medium-sized application, this results in hundreds of small files that must be managed. While automated tools could be created to address this problem, the aforementioned problem of keeping the architectural specification and the CM system synchronized remains a significant obstacle.

To summarize, neither coarse-grained nor fine-grained versioning provides an adequate solution for capturing and managing architectural evolution. We have developed a system model that addresses the above issues. The system model captures architecture in terms of types and instances of constituent components, connectors, and their interfaces; leverages behaviors, constraints, and subtyping relationships among them; employs revisions and inter-file branches to support linear and diverging paths of evolution; and uses guarded expressions to denote optionality and variability of the artifacts. Finally it supports hierarchical composition of components and connectors in the system. The full discussion may be found in [2].

In the next section we present our architectural evolution environment, which relies on the above system model, and addresses the architectural evolution problem.

3. MAE ENVIRONMENT

Mae's architecture evolution environment provides and enforces the specific procedures through which an architecture is created and evolved. The environment does so by providing a tightly-integrated combination of functionality that covers both architectural aspects, such as designing and specifying an architecture or analyzing an architecture for its consistency, and CM aspects, such as checking out and checking in elements that need to change or selecting a particular architectural configuration out of the available version space.

As shown in Figure 1, the Mae architectural evolution environment consists of four major subsystems. The first subsystem, the xADL 2.0 data binding library [1], forms the core of the environment. The data binding library is a standard part of the xADL 2.0 infrastructure that, given a set of XML schemas, provides a programmatic interface to access XML documents adhering to those schemas. In our case, the data binding library provides access to XML documents described by the XML schemas that represent Mae's integrated system model. In essence, thus, the xADL 2.0 data binding library encapsulates our system model by providing a programmatic interface to access, manipulate, and store evolving architecture specifications.

The three remaining subsystems each perform separate but complimentary tasks as part of the overall process of managing the evolution of a software architecture:

- The design subsystem combines functionality for graphically designing and editing an architecture with functionality for versioning the architectural elements. This subsystem supports architects in performing their day-to-day job of defining and maintaining architectural descriptions, while also providing them with the familiar check out/check in mechanism to create a historical archive of all changes they make.
- The selector subsystem enables a user to select one or more architectural configurations out of the available version space. Once an architecture has started to evolve, and once it contains a multitude of optional and variant elements, the burden of manually selecting an architectural configuration becomes too great. To overcome this burden and automatically extract a single architecture based upon a user-specified set of desired properties, Mae provides the subsystem as an integral part of its environment.
- Finally, the analysis subsystem provides sophisticated analyses for detecting inconsistencies in architectural configurations. This subsystem typically is used after a particular architectural configuration has been selected, and helps to ensure that the architectural configuration is not only structurally sound, but also consistent with the expected behaviors and constraints of each and every component and connector in the selected configuration.

4. EVALUATION

Mae has been successfully used in three different settings as the primary architectural development and evolution environment. The collective experiences not only show that Mae is effective in circumventing the problems that occur when using a traditional CM system, but also demonstrate that it is a usable and scalable solution that is applicable to real-world problems.

As a first experience, we used Mae to create and evolve the architecture of an audio/video entertainment system patterned after an existing architecture for consumer electronics. Our evaluation focused on usability, and in particular on whether the presence of configuration management functionality hinders or obscures the process of designing an architecture. Our second experience with Mae involved creating and evolving the software architecture of the Troops Deployment and battle Simulations system. The evaluation focused on evaluating the scalability of Mae. While the system contains a moderate number of component and connector types, the number of component and connector instances can be in the 100's. Finally, we evaluated Mae's applicability to real-world settings through independent use by another research group at the University of Southern California. This group collaborates with NASA's Jet Propulsion Laboratory (JPL) in modeling and analyzing the evolving software architecture of the SCrover application, which is the on-board software of a rover system built using JPL's Mission Data System (MDS) framework

5. REFERENCES

[1] Dashofy, E.M., van der Hoek. A., Taylor R.N., An Infrastructure for the Rapid Development of XML-based Architecture Description Languages, in *Proceedings of the 24th International Conference on Software Engineering (ICSE2002)*, Orlando, Florida.

[2] Roshandel R., van der Hoek A., Mikic-Rakic M., Medvidovic N., Mae - A System Model and Environment for Managing Architectural Evolution, *Submitted to ACM Transactions on Software Engineering and Methodology (In review)*, October 2002.