Mae—A System Model and Environment for Managing Architectural Evolution

ROSHANAK ROSHANDEL
University of Southern California
ANDRÉ VAN DER HOEK
University of California, Irvine
and
MARIJA MIKIC-RAKIC and NENAD MEDVIDOVIC
University of Southern California

As with any other artifact produced as part of the software life cycle, software architectures evolve and this evolution must be managed. One approach to doing so would be to apply any of a host of existing configuration management systems, which have long been used successfully at the level of source code. Unfortunately, such an approach leads to many problems that prevent effective management of architectural evolution. To overcome these problems, we have developed an alternative approach centered on the use of an integrated architectural and configuration management system model. Because the system model combines architectural and configuration management concepts in a single representation, it has the distinct benefit that all architectural changes can be precisely captured and clearly related to each other—both at the fine-grained level of individual architectural elements and at the coarse-grained level of architectural configurations. To support the use of the system model, we have developed Mae, an architectural evolution environment through which users can specify architectures in a traditional manner, manage the evolution of the architectures

---

This research was supported by the National Science Foundation under Grant numbers CCR-9985441, CCR-0093489, and IIS-0205724.
This effort was also sponsored by the Defense Advanced Research Projects Agency, Rome Laboratory, Air Force Materiel Command, USAF under agreement numbers F30602-00-2-0615, F30602-00-2-0599, and F30602-00-2-0607. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Rome Laboratory, or the U.S. Government.
This effort was also sponsored by the Jet Propulsion Laboratory, U.S. Army TACOM, Xerox Corporation, and Intel Corporation.
Authors' addresses: R. Roshandel, M. Mikic-Rakic, and N. Medvidovic, Computer Science Department, University of Southern California, Henry Salvatori Computer Center 300, Los Angeles, CA 90089; email: {roshande,marija,neno}@usc.edu; A. van der Hoek, Department of Informatics, University of California, Irvine, 444 Computer Science Building, Irvine, CA 92697-3425; email: andre@ics.uci.edu.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 1515 Broadway, New York, NY 10036 USA, fax: +1 (212) 869-0481, or permissions@acm.org.
© 2004 ACM 1049-331X/04/0400-0240 $5.00

using a check-out/check-in mechanism that tracks all changes, select a specific architectural configuration, and analyze the consistency of a selected configuration. We demonstrate the benefits of our approach by showing how the system model and its accompanying environment were used in the context of several representative projects.

Categories and Subject Descriptors: D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement; D.2.9 [**Software Engineering**]: Management; D.2.11 [**Software Engineering**]: Software Architectures

General Terms: Design

Additional Key Words and Phrases: Design environment, evolution, Mae, system model

1. INTRODUCTION

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 [Burrows and Wesley 2001] 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 [Kuusela 1999]. One solution is to store the entire architectural description in a single file and track its evolution using an existing CM system. The problem with this solution is that: (1) it only tracks changes at the level of an architecture as a whole; (2) it requires extensive use of branching to capture variations in an architecture; and (3) it does not support the use of multiple versions of the same artifact (e.g., a single component) within a single architecture. An alternative solution is to version each architectural element in a separate file. This leads to a different set of problems: (1) it results in potential inconsistencies due to duplication of information in the CM system and the architectural specification; (2) it requires breaking up an architectural specification into numerous small files; and (3) it still requires extensive use of branching to manage variability. Thus, while it is possible to use an existing CM system to manage architectural evolution, the problems associated with either solution prevent doing so effectively.

In this article, we introduce a novel approach for managing architectural evolution, called Mae, that overcomes these problems. Mae combines techniques from the fields of software architecture and configuration management to make two unique contributions: (1) an integrated architectural and configuration management 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.

Mae's first contribution, its system model, focuses on capturing architectural evolution. We formed the system model by extending a "notional" architectural model, based on the concepts commonly present in existing architecture description languages [Medvidovic and Taylor 2000], with selected configuration management modeling concepts. In particular, we carefully integrated the following concepts into the notional model to form a single architectural and CM system model: revisions to capture linear evolution, inter-file branching to capture diverging paths of evolution, guarded variants to capture alternatives, guarded options to capture nonmandatory architectural elements, and subtyping relations to capture the nature of changes. Tight integration of all these concepts into a single system model is necessary not only to capture the history of architectural evolution, but also to annotate that history with meaningful information regarding the compatibility among different versions of architectural elements as they have evolved over time.

Mae's second contribution, its architecture evolution environment, builds upon the system model to provide an architect with powerful capabilities for managing architectural evolution. The environment offers a carefully crafted combination of architectural and configuration management capabilities. Specifically, it supports architects in designing and visualizing a software architecture, checking out and checking in individual architectural elements and entire architectures, selecting a particular architectural configuration, and analyzing whether the behaviors and constraints of all selected architectural elements match in such a configuration.

Our usage of Mae demonstrates that it is an effective solution for managing architectural evolution, circumventing the problems that occur when using a traditional CM system for this purpose. In particular, we have applied Mae to manage the specification and evolution of three different systems: (1) an audio/video entertainment system—patterned after an existing architecture for consumer electronics, (2) a Troops Deployment and battle Simulations system—a research prototype designed by a different part of our research group at the University of Southern California, and (3) the SCRover project—a mobile robot system built in cooperation with the NASA Jet Propulsion Laboratory (JPL) using the JPL Mission Data System (MDS) framework. Collectively, these experiences show that Mae not only represents a feasible approach, but is also usable, scalable, and applicable to real world problems.

Our approach of tightly integrating CM concepts in the architectural representation (and, in parallel, CM functionality in the architectural evolution environment) represents a significant departure from traditional CM approaches, which strongly advocate keeping CM separate from the artifacts they version and manage [Estublier et al. 2002]. It also is at odds with research

that attempted to integrate fine-grained support for evolution into databases [Agrawal et al. 1991; Franconi et al. 2000; Wei and Elmasri 2000] and programming languages [Habermann and Perry 1981; Winkler 1986], arguably neither of which was very successful at addressing the breadth of issues that arise (e.g., at best, these solutions integrate a version identifier, but they do not address such issues as optionality and variability). In contrast to these attempts, we believe our approach has significant merit. First, architectures differ considerably from databases and programming languages in that they are structural and coarse-grain in nature, which makes them much more suitable for integrating configuration management concepts. Second, existing architectural and configuration management system models have began gravitating towards each other [van der Hoek et al. 1998b], a trend culminating to date in the approach presented in this article. Finally, our approach successfully overcomes the aforementioned problems, is easy to use, and even affords development of several new functional capabilities (as discussed in Section 5.4). Clearly, this has serious implications for CM research, which must rethink the position of always keeping CM separate from its target environment and must examine the potential implications of tightly integrating CM functionality with other representations and activities in the software lifecycle (e.g., requirements, test plans, cost models, etc.). While we do not know whether such integrations will have the same kinds of benefits as discussed here, the rest of this article demonstrates that at least in the case of software architecture tight integration is an advantageous endeavor.

The remainder of this article is organized as follows: First, in Sections 2 and 3, we briefly present background information and an example scenario that, together, set the stage for the ensuing discussion. Section 4 introduces our architectural system model and its realization in an extensible, XML-based, architectural modeling notation. Section 5 discusses the architecture and implementation of Mae's architectural evolution environment that leverages the system model to provide an architect with the actual ability of specifying and evolving an architecture. Section 6 evaluates the usability, scalability, and applicability of Mae in the context of three specific experiences. We discuss related work in Section 7 and present our conclusions in Section 8.

2. BACKGROUND

Our research builds upon concepts from the fields of software architecture and configuration management. This section briefly introduces the relevant concepts in each field. Given that the basis of our approach is an integrated architectural and configuration management system model, our discussion focuses on introducing and comparing the particular system modeling constructs used in each field.

2.1 Software Architecture

As software systems grew more complex, their design and specification in terms of coarse-grain building blocks became a necessity. The field of software architecture addresses this issue and provides high-level abstractions

for representing the structure, behavior, and key properties of a software system. Software architectures involve: descriptions of the elements from which systems are built, interactions among those elements, patterns that guide their composition, and constraints on these patterns [Perry and Wolf 1992]. In general, a particular system is defined in terms of a collection of *components* (loci of computation) and *connectors* (loci of communication) as organized in an architectural *configuration*.

To date, many *architecture description languages* (ADLs) have been developed to aid architecture-based development [Medvidovic and Taylor 2000]. ADLs provide formal notations for describing and analyzing software systems. They are usually accompanied by various tools for parsing, analysis, simulation, and code generation of the modeled systems. Examples of ADLs include C2SADEL [Medvidovic et al. 1999], Darwin [Magee and Kramer 1996], Rapide [Luckham and Vera 1995], UniCon [Shaw et al. 1995], xADL [Dashofy et al. 2002a], and Wright [Allen and Garlan 1997]. A number of these ADLs also provide extensive support for modeling *behaviors* and *constraints* on the properties of components and connectors [Medvidovic and Taylor 2000]. These behaviors and constraints can be leveraged to ensure the consistency of an architectural configuration throughout a system's lifespan (e.g., by establishing conformance between the services of interacting components).

Some ADLs also support *subtyping*, a particular class of constraints that may be used to aid the evolution of architectural elements. As shown in Medvidovic et al. [1998], the notion of subtyping adopted by ADLs is richer than that typically provided by programming languages: it involves constraints on both syntactic (e.g., naming and interface [Garlan et al. 1997]) and semantic (e.g., behavior [Luckham and Vera 1995]) aspects of a component or connector. ADLs' supporting tools are used to ensure that the desired subtyping relationships are preserved at the architectural level.

2.2 Configuration Management

The discipline of configuration management (CM) traditionally has been concerned with capturing the evolution of a software system at the *source code* level [Burrows and Wesley 2001]. Research and development over the past twenty-five years have produced numerous contributions in the field [Conradi and Westfechtel 1998; Estublier et al. 2002], evolving CM system functionality through three distinct generations. The first generation consists of such CM systems as SCCS [Rochkind 1975], Sablime [Bell Labs Lucent Technologies 1997], and RCS [Tichy 1985]. The creation of this generation was a direct result of two immediate needs: (1) to prevent multiple developers from making simultaneous changes to the same source file and (2) to track the evolution over time of each source file. Both needs were satisfied through the use of *versioned archives*. Each archive contained a series of *revisions* to a single source file (using *delta storage* techniques to save disk space [Hunt and Tichy 1998]) as well as *locks* to indicate modifications in progress. Recognizing the need for multiple lines of development as well as the need for temporary parallel work, RCS introduced the use of *branches* to store logical *variants* in a versioned archive

Table I. Comparison of System Modeling Capabilities (Adapted from van der Hoek et al. [1998b])

	Software Architecture	Configuration Management
Composition	* * *	*
Consistency	* * *	**
Evolution	*	* * *
Selection		**

Legend: 0 marks = *no support*; 1 mark = *some support*; 2 marks = *moderate support*; 3 mark = *extensive support*

file and *merging* as a method of moving changes from one branch to another. Combined, all revisions and variants create a *version tree*, which is the central entity through which users interact with a first-generation CM system.

In order to support tracking of compound changes to groups of source files, research into *system models* [Estublier and Casalles 1994; Perry 1989; Tryggeseth et al. 1995] sparked the inception of the second generation of CM systems. System models and their associated modeling languages provide a way of capturing the structure of software via a *configuration*, which precisely specifies a set of versions of specific source files. To capture the potential evolution of the structure itself, configurations can exist in different revisions and variants, just as individual source files can. Through automation of workspace management via *configuration specifications* (sets of rules indicating which version of which source file to place in a workspace), changes to a multitude of source files can be stored back in a *repository* in a single step. To guide developers in maintaining consistent configurations in this process, some system models were enhanced to include such elements as *interfaces* [Estublier and Casalles 1994] and *behavioral specifications* [Perry 1989].

Flexibility was the key driving force behind the emergence of the third generation of CM systems. Researchers recognized that a single method of interaction (checking out artifacts into a workspace, modifying them as needed, and checking them back into the repository) was not adequate for all situations. Instead, different *CM policies* [Parisi and Wolf 2000; van der Hoek 2000; Wiborg Weber 1997] are required to support situations in which, for example, a large number of developers operate on a small set of source files or in which distributed groups of developers modify a single piece of software.

2.3 Comparison of System Modeling Capabilities

Of interest to this article is the fact that the system modeling capabilities offered by different approaches in the fields of software architecture and configuration management have, over time, gravitated toward each other. For instance, the Koala ADL incorporates modeling facilities for specifying variation points [van Ommering 2002]. Conversely, Ragnarok is a configuration management system modeling language that incorporates architectural constructs [Christensen 1998]. Many other such examples exist [Shaw et al. 1995; Tryggeseth 1995; Westfechtel and Conradi 2001].

As part of our previous work, we extensively studied this trend by comparing and contrasting in detail the capabilities provided by ADLs with those provided by CM system modeling languages [van der Hoek et al. 1998a, 1998b]. Table I

summarizes the results of this study as relevant to the work presented in this article. The number of bullets in a cell indicates the level of support for the particular concern. For instance, the field of software architecture provides an extensive array of modeling constructs to capture the composition of a software system (e.g., types and instances, components and connectors, configurations, interfaces). On the other hand, the field of configuration management only supports the concepts of configurations and interfaces.

A detailed review of our analysis of the two fields is outside the scope of this article. The conclusions of our previous work, however, represent a critical basis for the work presented here. In particular, our studies resulted in three conclusions regarding the system modeling constructs provided by the two fields:

(1) While the system modeling constructs developed by both fields are gravitating towards each other, they still are disparate and tend to be developed in isolation.
(2) When system modeling constructs differ, they tend to be developed for a different purpose and differ in an orthogonal manner; that is, they address different concerns and do not interfere with each other.
(3) When system modeling constructs are similar, they usually are developed for the same purpose and any differences are of a syntactic rather than semantic nature.

Our overall conclusion, therefore, is that it should be possible to meaningfully combine modeling constructs from the two fields to create a single representation that forms a system model for capturing architectural evolution. Moreover, we conclude that by first carefully unifying similar constructs and then adding the remaining, orthogonal constructs, the system model can be created such that it is free of conflicts and void of otherwise unforeseen problems. In the rest of the article, we substantiate these conclusions.

3. MOTIVATING EXAMPLE

One way of managing architectural evolution is to use an existing configuration management system. This would have the benefits of the system already being operational in the organization, well understood by its users, and used to manage other artifacts. However, such an approach leads to numerous problems. In this section, we discuss how several such problems may arise in the context of an example application.

3.1 Example Application

TDS, Troops Deployment and Battle Simulations, is an application that supports distributed, on-the-fly deployment of personnel in situations such as military crises and search-and-rescue efforts. TDS operates in a heterogeneous hardware environment and facilitates communication and interaction among a computer at *Headquarters* and a set of (handheld) devices used by *Commanders* and *Soldiers* in the battlefield. This application was designed and developed as

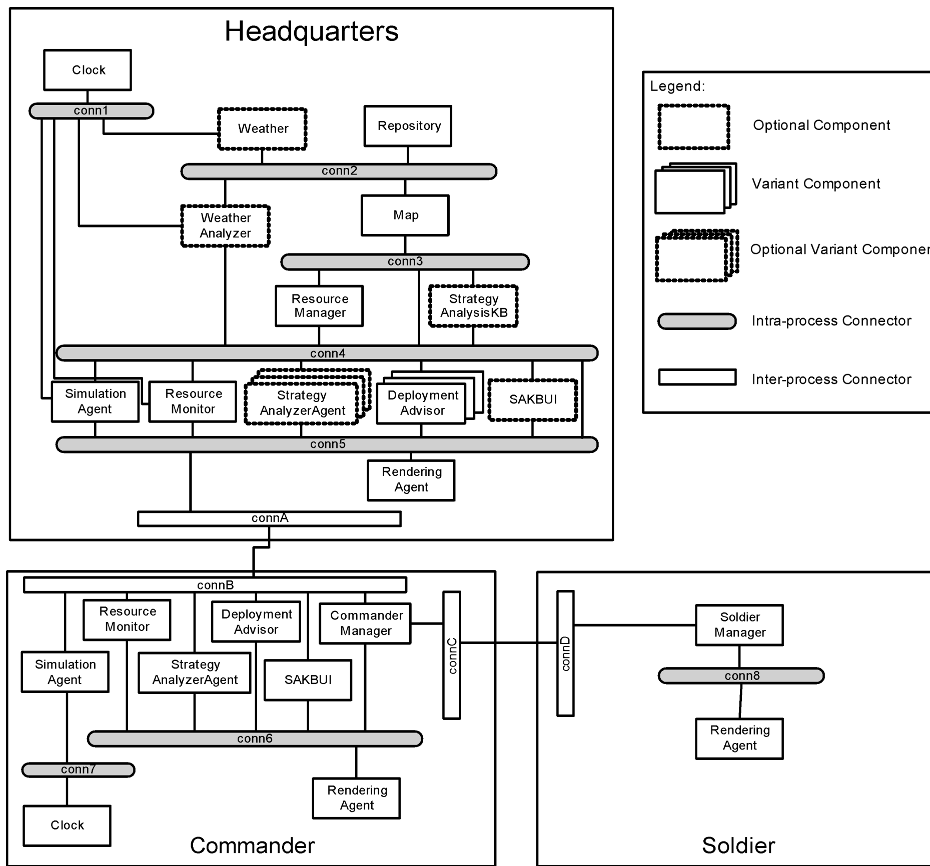

Fig. 1. TDS architecture.

part of a separate research effort and is similar to the application outlined in the Introduction.

Figure 1 illustrates the architecture of TDS, consisting of three subsystems, one for each type of user. A detailed discussion on all the details and design issues of TDS is provided elsewhere [Mikic-Rakic and Medvidovic 2002]. Here, we focus on the characteristics of the architecture of TDS that make managing its evolution challenging.

(1) Several components are part of more than one subsystem (e.g., *Resource-Monitor*, *Clock*). Each subsystem, however, may evolve separately. This can lead to situations in which multiple versions of the same component exist in the overall architecture. For instance, Headquarters may independently decide to upgrade its subsystem to include a newer version of *ResourceMonitor*, while the Commander subsystem retains the old version.
(2) Some components are optional. For example, the *Weather* and *WeatherAnalyzer* components are only used in cases in which weather is of importance to the battlefield situation.

(3) Several components exist in different variants. Different incarnations of *StrategyAnalyzerAgent* and *DeploymentAdvisor*, for example, provide alternative deployment strategies and associated analyses.
(4) Some components are both optional and variant, which is the case for the *StrategyAnalyzerAgent* component in the Headquarters subsystem. (Note that, at the same time, a single, specific variant of that component is a mandatory part of the Commander subsystem.)
(5) Finally, some relations exist among optional and variant components that, although not visible in a typical architectural diagram, determine their desired and/or valid combinations. For example, the optional *WeatherAnalyzer* and *Weather* components either both have to be instantiated or neither of them is to be present in the system.

These five characteristics are not unique to the TDS architecture. As demonstrated in other situations involving the evolution of other software architectures [Bosch 2001; Speck et al. 2002; van Ommering 2002], any evolving architecture introduces the same five kinds of issues. As such, any approach to managing architectural evolution must provide effective support for addressing all of these issues. One approach that has been attempted is to use an existing CM system to version the architectural specification. Below, however, we discuss in detail how such an approach fails, regardless of whether the architecture is versioned in a coarse-grained or fine-grained manner.

3.2 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. It is common knowledge in the field of CM that managing artifacts at such a coarse-grained level leads to severe problems. Projected to software architecture, this means tracking the evolution of an architecture as a whole—clearly an undesirable situation as any single change, no matter how localized, 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 of these elements results in a different architecture that must be explicitly specified. Finally, this approach prevents the use of multiple versions of the same artifact within a single architecture, something that a typical application such as TDS clearly requires. In sum, these shortcomings make versioning an entire architectural specification as a single artifact a highly undesirable solution for managing architectural evolution.

3.3 Fine-Grained Versioning

In the field of configuration management, versioning fine-grained artifacts is considered a better approach to managing source code evolution than coarse-grained versioning. This analogy, however, does not hold when applied to architectural evolution. In particular, fine-grained versioning leads to serious

consistency management 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, but simply do not work for architectural configurations. The issue is that these techniques operate on a line-by-line basis, whereas differencing and merging of architectures must be performed at the level of architectural elements such as components, connectors, interfaces, and so on. While a line-based approach may accidentally work, to guarantee proper operation, architecture-specific differencing and merging algorithms are needed [Van der Westhuizen and van der Hoek 2002]. As a result, the differencing and merging techniques embedded in current CM systems cannot be used, and an architect is forced 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 such as TDS, 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. In the next section, we present our solution, which relies on the use of an integrated architectural and CM system model to avoid the problems discussed here.

4. INTEGRATED ARCHITECTURAL AND CM SYSTEM MODEL

As discussed in the Introduction, the first part of our solution to managing architectural evolution consists of a novel system model that combines software architecture and configuration management concepts into a single representation. The integrated nature of the system model has the distinct benefit that all architectural changes can be precisely captured and clearly related to each other. As a result, architectural evolution is addressed in a natural and meaningful way—natural, because the system model inherently operates at the architectural level, and meaningful, because the system model annotates changes with semantic information describing the particular nature of the changes.

Table II presents the elements of Mae's system model. Each conceptual element is listed with actual capabilities that an architect would use when modeling the concept. For example, components, connectors, and interfaces are supported in the system model by allowing an architect to define types and use instances of those types to create architectures. As another example, compatibility rules among components can be specified using behaviors and constraints.

Table II. Elements of the Mae Integrated System Model

Concept	Modeling Capability
Components, connectors, interfaces	Types and instances
Compatibility rules & expectations	Behaviors and constraints
Hierarchical composition	Subarchitectures
Linear evolution	Revisions
Diverging paths of evolution	Inter-file branches
Alternatives (variation points)	Guarded variants
Non-mandatory architectural elements	Guarded options
Type evolution constraints	Subtyping

We note that our system model borrows from many previous contributions, as each of the concepts and its related modeling capabilities have been previously explored. Our system model is unique, however, in combining them in a single, flexible representation. The particular combination of concepts is a key contribution of our approach: it creates an advanced, rich kind of system model that remains centered on the explicit use of architectural elements, but incorporates sufficient information to effectively capture architectural evolution.

While a large number of CM concepts are available for potential use in our integrated system model [Conradi and Westfechtel 1998], we only selected few based upon the following criteria:

—*Simplicity.* The combination of selected CM concepts should address all of the problems raised in Section 3, but without introducing unnecessary complexity. Ideally, therefore, the resulting system model contains the smallest possible set of constructs that combined are expressive enough to capture all aspects of architectural evolution.
—*Focus on Architecture.* Each concept should preserve the architecture-centric nature of the system model. In particular, the addition of CM concepts should not take the main focus away from architecture, since the primary activity of an architect is design and he or she should not be distracted from that activity by having to specify a multitude of CM concepts.
—*No Interference.* The CM concepts should not interfere either with one another or with the existing architectural concepts in the system model. In particular, each of the CM concepts should be integrated in a manner that avoids overlaps and prevents any undesirable interactions that may result from such overlap.

As a result of stringently applying these criteria while forming our system model, some advanced CM concepts such as change sets [Munch 1993], change packages [Wiborg Weber 1997], and advanced branching strategies [Walrad and Strom 2002] are not included. Our aim was to first build and demonstrate the feasibility of a system model that addresses the concerns raised in Section 3. While an exploration of more advanced techniques may lead to a more advanced solution, we leave such an exploration to future work.

Throughout the rest of this section, we discuss the details of Mae's system model by first introducing a "notional" architectural model and then detailing

our extensions for capturing evolution. We conclude the section with a discussion on the realization of the system model as a series of xADL 2.0 schemas.

4.1 Notional Architectural Model

Rather than tying our system model to one particular ADL, we wish to demonstrate how our approach may be applied to arbitrary ADLs. We therefore build our system model around a *notional architectural model*, which captures the elements commonly found in most existing ADLs [Medvidovic and Taylor 2000]. In particular, the notional architectural model is based upon the familiar concepts of components, connectors, and interfaces, supports the specification of subarchitectures, strongly separates types from instances, and leverages behavioral and constraint specifications to help enforce architectural consistency. The result is shown in the first two columns of Table III, which present the detailed constructs used to model each architectural element. The notation in Table III uses underlining to designate a set of fields that uniquely identify the element ("keys"); italics to denote identifiers that point to other elements in the model ("foreign keys"); asterisks to indicate zero or more elements; square brackets to indicate fields that may or may not be present; and curly braces to indicate groups of elements. In the notional architectural model, for example, a component type is identified by its name, has zero or more interfaces that are defined separately, may be hierarchically composed out of a set of independently defined components and connectors, and could have associated behavioral and constraint specifications that detail its functionality and interactions.

4.1.1 *Interface Types.* At its core, the notional architectural model leverages interface types to define abstract sets of services that a component may provide to, or require from, the system. Moreover, an interface type may be specified using a set of *interfaceElements* that detail the methods' signature exposed by a component or connector. To actually use an interface type, an instance of that type must be created. Such an instance is typically defined in terms of a *name* that distinguishes it from other instances of the same type. Additionally, each interface instance has a *direction*: "in" for provided services, "out" for required services, and "in/out" for services that are both provided and required. Finally, an interface instance maintains a pointer to its originating *interfaceType*, and thus automatically inherits the *interfaceElements* from the interface type.

4.1.2 *Component Types.* Component types are the primary building block of all ADLs and thus, the primary building block of our notional architectural model. A component type provides and/or requires a set of services via its *interfaces,* each of which is an instance of an existing *interfaceType*. It also may hierarchically contain other *component* and *connector instances* that, together, form its internal subarchitecture. In such cases, a set of *interfaceMappings* maps interfaces from the composite component to the interfaces on its constituent component or connector instances. Finally, *behaviors* and *constraints* specify the rules and limitations by which a component's interfaces may be accessed. Particularly, behaviors describe externally visible protocols of interaction as

Table III. Comparison of Notional Architectural Model with Mae's Integrated System Model

<table>
<tr><th>Element</th><th>Notional Architectural Model</th><th>Mae Integrated System Model</th></tr>
<tr><td>Interface Type</td><td>name
{interfaceElement}*</td><td>name
revision
{interfaceElement}*
*ascendant**
*descendant**</td></tr>
<tr><td>Interface instance</td><td>name
direction
interfaceType</td><td>name
direction
interfaceType</td></tr>
<tr><td>Component type</td><td>name
*interface**
*component**
*connector**
{interfaceMapping}*
behavior*
constraint*</td><td>name
revision
{*interface* [, guard]}*
{*component* [, guard]}*
{*connector* [, guard]}*
{interfaceMapping}*
behavior*
constraint*
subtype*
*ascendant**
*descendant**</td></tr>
<tr><td>Component instance</td><td>name
componentType</td><td>name
componentType | *variantComponentType*</td></tr>
<tr><td>Connector Type</td><td>name
*interface**
*component**
*connector**
{interfaceMapping}*
behavior*
constraint*</td><td>name
revision
{*interface* [, guard]}*
{*component* [, guard]}*
{*connector* [, guard]}*
{interfaceMapping}*
behavior*
constraint*
subtype*
*ascendant**
*descendant**</td></tr>
<tr><td>Connector instance</td><td>name
{*sourceInterface*
[, *myDestInstance*]}*
{[*mySourceInterface*,]
destInterface}*
connectorType</td><td>name
{*sourceInterface* [, *myDestInterface*] [,guard]}*
{[*mySourceInterface*,] *destInterface* [, guard]}*
connectorType | *variantConnectorType*</td></tr>
<tr><td>Variant component type</td><td>—</td><td>name
revision
{*interface* [, guard]}*
{*componentType* | *variantComponentType*, guard}*
subtype*
*ascendant**
*descendant**</td></tr>
<tr><td>Variant connector type</td><td>—</td><td>name
revision
{*interface* [, guard]}*
{*connectorType*, | *variantConnectorType*, guard}*
subtype*
*ascendant**
*descendant**</td></tr>
</table>

cause-and-effect relations among incoming and outgoing services. Constraints restrict behaviors by specifying the internal conditions of a component or connector that must hold for a service to be invoked. For example, a behavioral rule may specify that before a *divide* service can be used, an *init* service has to be invoked. Furthermore, it may state that an invocation of *divide* will always be followed by an invocation to other components to communicate the results. A constraint rule, on the other hand, may state that an invocation of a *divide* service can only be performed if the denominator is nonzero. In essence, constraints are used to specify and limit the internal functional conditions that must be satisfied before and after invocation of services.

4.1.3 *Connector Types.* The final key concept in our notional architectural model is that of a connector type. It is generally accepted that components and connectors do not differ in their structure, but only differ in their particular purpose. As such, they should be explicitly distinguished. We therefore treat connector types as first-class elements in the notional architectural model. However, unlike instances of component types, instances of connector types often include some additional information that further defines their role in an architecture. In particular, since connector instances form the bindings among components, a connector instance must define links from source interfaces on some component instances to destination interfaces on other component instances. Depending on the nature of a connector, the connector's own interfaces may or may not be used in this process. Since two schools of thought exist in the field of software architecture (one in which connectors do not have interfaces [Taylor et al. 1996] and one in which they do [Allen and Garlan 1997]), we designed the notional architectural model so that it can support both cases: connector instances may link component interfaces either directly or via the connector's interfaces.

Overall, the particular combination of concepts in the notional architectural model creates a representation that is similar, if not equivalent, to the core of most ADLs [Medvidovic and Taylor 2000]. While the remainder of our discussion builds upon this notional model, we believe that our contributions can, therefore, be applied equally effectively to other specific ADLs.

4.2 Mae's Architectural System Model

Recall from the discussion in Section 3 that 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 achieve this in our approach, we have extended the notional architectural model with three broad categories of modeling support: optionality, variability, and evolution. Below, we discuss each category in detail and describe how their combination creates an effective solution that provides support for all five modeling needs.

4.2.1 *Optionality.* As an architecture evolves, some elements may become useful only in certain architectural configurations and thus may or may not

need to be present when a particular configuration is instantiated. Mae's system model supports optionality through the use of Boolean guards that are attached to particular elements in the system model. In particular, Boolean guards specify conditions that can be associated with the interfaces, components, and connectors that constitute a component or connector type. To determine whether an optional element is included in a specific architectural configuration, a selection mechanism is required to evaluate the guards. (Section 5.2 introduces an implementation of such a mechanism.)

Optionality also plays an important role in the specification of connector instances. In particular, a connector instance specifies links to and from interfaces on components. If such an interface is optional, the link must inherit the Boolean guard of the optional interface. This avoids the problem of dangling links when the interface is not included in a particular architectural configuration. Architects, of course, can correlate any set of elements by using the same Boolean guard for each element in the set. This creates the effect of a logically coherent group of elements for which it is guaranteed that either all or none of the elements are present in any selected configuration.

To illustrate the concept of optionality, Table IV contains excerpts from the actual specification of the Troops Deployment and Battle Simulations (TDS) system discussed in Section 3. Shown are the specifications of an interface type, an interface instance, a component type, and a variant component type (some of which will be discussed in later subsections to illustrate other aspects of the system model). Consider the *SimulationAgent* component type, which is hierarchically constructed from the *Agent*, *Knowledgebase*, and *ResourceFetcher* component instances, which in turn are connected by the *FetcherBus* connector instance. (Although not included in the example, the specification of this connector instance establishes the actual links among the component instances, thereby defining the internal topology of the component type.) The *interfaceMapping* field maps interface instances on the overarching component type to interface instances on the constituent component instances. For instance, the *iGetResources* interface instance is mapped to the interface instance *iGetAllResourceInfo* on the component instance *ResourceFetcher*.

Of interest to this discussion are the *ResourceFetcher* component and the *iGetResources* interface. They both are optional, and their instantiation depends on evaluation of their corresponding guards. Since the two guards are identical, setting the variable *fetchLocally* to true and the variable *timeToDisconnection* to a value less than 10 will result in inclusion of both optional elements as a group. On the other hand, violation of either one of those conditions will result in omission of both optional elements from the instantiated configuration.

4.2.2 *Variability.* Traditional CM systems address two complementary forms of variability: *divergence* and *convergence*. Divergence concerns elements for which two or more separate paths of evolution are needed (e.g., to address different demands by different customers). Convergence brings diverging paths of evolution together at specific variation points, which designate places in an architecture where one of multiple, logically alternative elements can be

Table IV. Example Excerpts from TDS Architecture Specification

Element	TDS Example
Interface type	name = iAnalyzeDefensiveStrategy revision = 1 interfaceElement = {PerformDefensiveAnalysis (newRegion : Region) } ascendant = {iAnalyzeStrategy 1.4} descendant = { }
Interface instance	name = iAnalyze direction = in interfaceType = {iAnalyzeDefensiveStrategy 1}
Component type	name = SimulationAgent revision = 3 interface = { {iGetResources, fetchLocally & timeToDisconnection < 10}, iGetRules, iSimulateBattle } component = {Agent, Knowledgebase, {ResourceFetcher, fetchLocally & timeToDisconnection < 10} } connector = {FetcherBus} interfaceMapping = {IGetResources -> ResourceFetcher.iGetAllResourceInfo} behavior = {(iGetRules, iGetResources, iSimulateBattle*) } constraint = {iGetRules.PostCond(RulesInstantiated == true), iGetResources.PreCond (timeToDisconnection < 10), iSimulateBattle.PreCond(RulesInstantiated == true) } ascendant = {SimulationAgent 2} descendant = {SimulateAndFight 1 } subtype = {beh and int, SimulationAgent 2 }
Variant component type	name = DeploymentAdvisor revision = 2 interface = {iDeployTroops, {iDeployAmmunition, military_mode == true}, {iDeployTents, military_mode == false }} componentType = { {HumanitarianDeployment 1, military_mode == false}, {MilitaryDeployment 2, military_mode == true }} ascendant = {DeploymentAdvisor 1, StrategyAnalysisKB 5} descendant = { } subtype = {{ beh DeploymentAdvisor 1}, {int StrategyAnalysisKB 5 } }

included. Our system model must naturally support both forms of variability at the architectural level.

To capture diverging paths of evolution, we adopt interfile branching [Seiwald 1996][1] as applied to each of the types in the system model. Interfile branching prescribes that each new branch is formed by creating a new type that has its own unique name and follows its own linear path of evolution. To still maintain the relationship with the original type, the definition of each type in Table II is extended with the fields *ascendant* and *descendant*. Upon creation of a new branch, the ascendant of the new type is set to the original type (which creates a "back" pointer). In addition, the set of descendants of the original type

[1]Because our system model does not pertain to files, interfile branching is not an ideal term here. Nonetheless, we adopt the term because it identifies a well-known branching strategy.

is updated with the new type (which creates a "forward" pointer). In essence, the ascendant and descendant fields implicitly maintain a decentralized version tree. Compared to the traditional mechanism of using a centralized version tree, interfile branching has the distinct advantage of separating concepts (linear evolution as described in Section 4.2.3 and diverging paths of evolution as described here) and identifying branches via an explicit type name rather than an arbitrary number in a version tree [Seiwald 1996; Walrad and Strom 2002].

To capture convergence, we use variation points, which designate specific places in an architecture where one of multiple, logically alternative elements (termed variants) can be included. To represent variation points, Mae defines *variant component types* and *variant connector types*. A variant component type is specified as a set of other component types, each with an associated, mutually exclusive Boolean guard. Instances of variant component types can be used in exactly the same way as instances of "regular" component types. The only time that a difference is noted is upon instantiation of a particular architectural configuration. At that time, each instance of a variant component type must be resolved to be of a specific type. To that purpose, each of the Boolean guards within a variant component type is evaluated and the instance is configured to be of the type for which the Boolean guard evaluates to *true*.

The interfaces on a variant component type are not necessarily the same as the interfaces on each of its variants. It may therefore happen that inclusion of one particular variant in a configuration leads to a different set of interfaces being available than inclusion of another variant. To avoid other component or connector instances relying on the presence of such interfaces, Mae prescribes that the overarching variant component type must use optional interfaces to identify those interfaces that are not exposed by all of its variants. While this incurs some extra work and care by the architect, it has already been demonstrated by Koala that the added benefits of flexibility and evolvability outweigh this shortcoming [van Ommering 2002].

Table IV shows the variant component type *DeploymentAdvisor*, which has three interface instances and consists of two variants. One variant provides advice concerning humanitarian deployment (*HumanitarianDeployment*) and the other concerning military deployment (*MilitaryDeployment*). When instantiating a configuration containing an instance of this variant component type, one of the variants is selected based on the value of the variable *military_mode* as supplied by the architect: if the value is *true*, *MilitaryDeployment* will be instantiated; if the value is *false*, *HumanitarianDeployment* will be instantiated. Note that two of the interface instances of *DeploymentAdvisor* are optional to reflect the fact that not all interface instances are provided by all of the variants. In particular, the interface instance *iDeployAmmunication* is only available when the *militaryDeployment* variant is chosen, and the interface instance *iDeployTents* is only available when the *humanitarianDeployment* variant is selected.

Combined, divergence and convergence allow efficient specification of variability: elements common to every configuration are modeled only once, while variant types capture the exact differences among the configurations as isolated

variation points. Furthermore, divergence and convergence integrate orthogonally with our approach to capturing optionality; that is, the two coexist and can be combined without interfering with each other. For instance, it is possible to specify optional instances of variant component types or optional interfaces in variant component types.

It is important to note that the use of Boolean guards in specifying optionality and variability is key to an architect's ability to establish relationships among those elements within an architectural configuration. By using the same Boolean guards to specify variants in two different variant component types, for instance, an architect can ensure pair-wise inclusion of specific variants (e.g., each variant in one of the variant component types is associated with a particular variant in the other variant component type). As another example, an architect could use mutually exclusive Boolean guards for some optional elements to guarantee that only one of the options is included. Similar relationships can be established among arbitrary sets of optional or variant elements.

4.2.3 *Evolution.* An architecture can change in many different ways. New elements can be added, existing elements can be changed, elements can be removed, elements can be converted into optional or variants elements (and back), and so on. Mae's system model captures all these changes using *versioning* and *subtyping*. Versioning is used to identify different (historical) incarnations of the same element; subtyping is used to annotate each change to indicate the nature of that change.

Every type that is part of the system model is versioned through the use of the *revision* field. The use of the word revision—and not version—is an intentional choice. Since diverging paths of evolution are captured using interfile branches, the revision field does not have to serve "double-duty". In particular, whereas existing CM systems use a detailed versioning scheme that encodes both linear evolution and branching (e.g., a version number such as 1.3.4.2 indicates that an element evolved to revision 4.2 on a separate branch that started from revision 1.3 of the main branch), our system model voids the need for such a versioning scheme and captures each evolutionary concern using a separate mechanism.

The use of versioning alone is not sufficient. Even in a traditional CM system, each change (theoretically, at least) is accompanied by an unstructured textual comment describing the nature of the change. In our system model, we provide a more systematic approach: each change to an architectural element can be annotated with an indication of whether the change preserves certain properties with respect to its predecessor. In particular, Mae uses the *subtype* field to indicate whether the change preserves the name, interfaces, and/or behavioral constraints of its predecessor [Medvidovic et al. 1998]. For instance, if a newer revision of a component type is both an interface and behavioral subtype [Liskov and Wing 1994] of its predecessor, its instances can be substituted for the old component instances in the architecture. On the other hand, if no subtype relation exists, substitutability may not be inferred. In other words, subtyping information annotates the history of each type with meaningful information regarding the particular nature of its changes, and helps an architect in understanding how an architecture has evolved.

A subtyping relation can be specified not just for successive revisions of a type, but also for arbitrary (variant) component and (variant) connector types that are not necessarily related in a single version tree. This, again, preserves the orthogonal nature of our system model and also allows for the system model to contain much useful information about compatibility among architectural elements that otherwise would have to be captured elsewhere.

To illustrate these concepts, we first note that every type in Table IV is versioned. For instance, the interface type is *iAnalyzeDefensiveStrategy* revision *1*. Additionally, all references to types are versioned as well. For example, *iAnalyzeDefensiveStrategy* revision *1* is derived from the interface type *iAnalyzeStrategy* revision *1.4*, and the variant component type *iDeploymentAdvisor* revision *2* consists of *humanitarianDeployment* revision 1 and *militaryDeployment* revision *2*.

The component type *SimulateAndFight* revision *3* illustrates the use of subtypes. It is a descendent of *SimulationAgent* revision *2* and, as a subtype, maintains interface and behavior compliance with that component type. *DeploymentAdvisor* revision *2* also shows the use of subtypes and has two particular subtype relationships: it is a behavior subtype of *DeploymentAdvisor* revision *1* and an interface subtype of *StrategyAnalysisKB* revision *5*. This dual relationship indicates that the *DeploymentAdvisor* component type exposes interfaces that are compatible with those implemented by revision *5* of the *StrategyAnalysisKB* component type, and provides behaviors that are compatible with those exposed by revision *1* of the *DeploymentAdvisor* component type.

4.2.4 *Concluding Remarks.* Together, the examples shown in the discussion in Section 4.2 illustrate each of the modeling concepts that we have added to the notional system model: revisions to capture linear evolution, interfile branching to capture diverging paths of evolution, guarded variants to capture alternatives, guarded options to capture nonmandatory architectural elements, and subtyping relations to capture the compatibility of different component and connector types with their predecessors. Furthermore, we note that the examples illustrate how the newly introduced concepts address all of the problems introduced in Section 3, that they preserve the architecture-centric nature of the original notional model, and that they continue the tradition of orthogonal design that is established by the notional model. In essence, their combination allows for an effective approach to capturing architectural evolution that eliminates the problems associated with the use of traditional CM systems.

We should also note that the examples use a particular semantics for some of the fields. For instance, we use first-order logic to specify the constraints of the component type *SimulationAgent* revision *1*, and we use a particular numbering scheme to designate the revisions of each type element. The system model itself, however, is agnostic with respect to the semantics of its individual elements. Just as the notional model is neutral with respect to existing ADLs (e.g., it does not prescribe the actual notation or format used for specifying interfaces or behaviors), our system model is neutral with respect to the details of the configuration management extensions. For example, we do not prescribe a particular version numbering scheme or a notation for Boolean guards. Rather,

the system model only enforces their presence; exact notations, format, and conventions will be imposed by the support environment described in Section 5. Our examples shown in Table IV follow those conventions.

4.3 Realization of the System Model

Mae's system model may be implemented in a variety of ways. For instance, it could be implemented as a specialized library in a widely-used programming language such as Java or C++, or, as another example, it could be implemented using database schemas. However, such implementations would limit future extensions and adaptation for different domains. For this reason, we have concretely realized all of the system model's concepts using xADL 2.0 [Dashofy et al. 2002a], an extensible, XML-based [Extensible Markup Language 2004] architecture description language. xADL 2.0 is a collection of modularly-organized XML schemas that represent architectural elements. It provides a base set of features that support representation of components, connectors, and interfaces through both types and their instances. Furthermore, as inspired by Mae's architectural system model, xADL 2.0 already includes facilities for capturing some CM-related information (e.g., optional elements; variant component and connector types; versions of component, connector, and interface types). Finally, the xADL 2.0 XML schemas that capture these concepts have been intentionally designed to be extensible. Implementing our system model, therefore, was a relatively straightforward activity.

To complete Mae's system model realization as an extension to xADL 2.0, we needed to build five additional kinds of modeling concepts upon the core functionality provided by xADL 2.0:

(1) *Boolean guards*, to guide the inclusion of optional and variant elements;
(2) *Subtyping*, to model the relationships among different versions of component and connector types;
(3) *Services*, to model the details of interface types as sets of interface elements with typed parameters and return values;
(4) *Behaviors* to model the interactions of components and connectors in the system;
(5) *Constraints*, to restrict execution of components' services using a set of pre- and post-conditions.

To add these concepts, we created two XML schemas. The first schema simply extends the optional and variant definitions of xADL 2.0 with an actual Boolean expression language as used by Mae to model guards. A Boolean expression is a valid combination of operands and logical operators (*and*, *or*, *not*), comparisons (*equal*, *greater than, less than, not equal*), Boolean values (*true*, *false*), and operators on ordered and unordered collections (*in-range*, *in-set*).

Our second XML schema is more complex: it provides constructs to model subtyping relationships and enhances a component's service specification with modeling constructs for interfaces, behaviors, and constraints. Table V shows selected extracts from the schema along with example TDS specifications related to these extracts. Specifically, the table shows the XML definitions of

Table V. Mae Schema Extracts and Examples

Interface Element

```
<xsd:complexType name="InterfaceElementDecl">
  <xsd:complexContent>
    <xsd:extension base="archtypes:Signature">
      <xsd:sequence>
      <xsd:element name="parameter" type="Variable" minOccurs="0"
           maxOccurs="unbounded"/>
      <xsd:element name="result" type="xsd:string" minOccurs="0"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

Operations

```
<xsd:complexType name="OperationDecl">
  <xsd:sequence>
    <xsd:element name="dir" type="archinstance:Direction"/>
    <xsd:element name="localVars" type="Variable" minOccurs="0"
          maxOccurs="unbounded"/>
    <xsd:element name="preDecl" type="xsd:string"/>
    <xsd:element name="postDecl" type="xsd:string"/>
  </xsd:sequence>
  <xsd:attribute name="id" type="archinstance:Identifier"/>
</xsd:complexType>
```

TDS Examples

```
<beh:interfaceDecl xsi:type="c2:InterfaceElementDecl"
types:id="igetMap">
  <types:description xsi:type="instance:Description"> iGetMap
</types:description>
  <types:direction xsi:type="instance:Direction"> prov
</types:direction>
  <beh:parameter xsi:type="beh:VarDecl">
    <beh:name xsi:type="xsd:string"> submapID </beh:name>
    <beh:type xsi:type="xsd:string"> Integer </beh:type>
  </beh:parameter>
  <beh:result xsi:type="xsd:string"> MapType </beh:result>
</beh:interfaceDecl>
```

```
<beh:operationDecl xsi:type="beh:OperationDecl" beh:id="ogetMap">
  <beh:dir xsi:type="instance:Direction"> prov </beh:dir>
  <beh:localVars xsi:type="beh:Variable">
    <beh:name xsi:type="xsd:string"> id </beh :name>
    <beh:type xsi:type="xsd:string"> Integer </beh:type>
  </beh:localVars >
  <beh:preDecl xsi:type="xsd:string">(id \greater 0)\and(id \less 17)
</beh:preDecl>
  <beh:postDecl xsi:type="xsd:string"> (\result = theMap)
</beh:postDecl>
</beh:operationDecl>
```

interfaceElements and *operations* that together describe part of a component's external behavior. *InterfaceElements* enhance a standard xADL 2.0 interface with a sequence of services, each defined as a set of input parameters and their types, as well as the service's return type. Operation declarations capture constraints on the execution behavior of *interfaceElements* as pre- and post-conditions. Our schemas do not enforce the semantics of these fields. However, the Mae environment (described in Section 5) requires them to be specified using first-order logic.

In the example shown in Table V, the *iGetMap* interface element takes a *submapID* of type *Integer* as an input parameter while its return type is *MapType*. Its pre-condition requires that the input parameter has a value between 1 and 16 (the TDS application handles up to 16 different submaps), and the post-condition specifies that the result of this operation is stored in variable *theMap.* Note that an architect does not have to directly edit these kinds of XML specifications; the Mae architecture evolution environment discussed in the next section shields the architect from all of the XML details and provides them with a high-level graphical user interface for specifying all aspects of the system model.

5. ARCHITECTURE EVOLUTION ENVIRONMENT

The second part of our solution for managing architectural evolution is Mae's architecture evolution environment. While the system model provides the facilities for capturing architectural evolution, the environment provides and enforces the specific procedures through which an architecture is created and evolved. Not surprisingly, 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 it 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 2, the Mae architectural evolution environment consists of four major subsystems. The first subsystem, the *xADL 2.0 data binding library* [Dashofy et al. 2002a], 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 discussed in Section 4.3. Therefore, the xADL 2.0 data binding library, in essence, 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 complementary 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 may become too great. To overcome this burden and automatically extract a

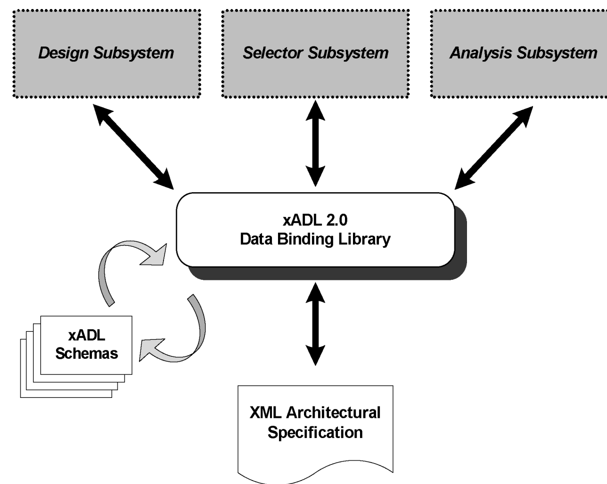

Fig. 2. Mae's architecture.

single architecture based upon a user-specified set of desired properties, Mae provides the selector 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 component and connector in the configuration.

Below, we first detail each of these subsystems and their role in managing architectural evolution. We then highlight several new capabilities that are facilitated by our system model and architecture evolution environment. These novel capabilities directly leverage our system model and would be more difficult to attain using alternative approaches.

5.1 Design Subsystem

The design subsystem of Mae supports an architect in his or her day-to-day activities. At its core, this subsystem operates as any other graphical design editor: an architect defines new interface types, component types, and connector types, and instantiates these types to define an architectural configuration. Mae distinguishes itself, however, by integrally supporting a check-out/check-in policy to manage the changes that occur when an architecture evolves. Each architectural element, whether a simple interface type or a complex composite component type, must be checked out first, then modified, and finally checked back in once the modifications are complete. As a result, a version history is incrementally created, allowing an architect to retrieve and examine previous versions of architectural elements. Note that changes may pertain to any aspect of our system model. In addition to coarse-grained changes to architectures, components, connectors, and interface types, Mae integrally supports

fine-grained changes to behaviors, constraints, subtype relations, and other properties associated with a particular version of an architectural element.

Once a version of an element has been checked in, that version becomes immutable. It can no longer be modified in order to protect any other parts of the architecture that depend on the immutable element. This feature guarantees incremental stability as the architecture is designed. It also guarantees the integrity of the old versions of the architecture during maintenance.

The user interface of the design subsystem is shown in Figure 3. The GUI is divided into three parts: the design palette (left), listing all versions of all types currently defined; the version pane (top), showing the version tree associated with the currently selected type; and the canvas (main area), supporting an architect in viewing or modifying an architecture. It should be noted that every element is shown with its associated version number. For example, the component *TopLevelArchitecture* that is currently being modified is version *2.1*, and consists of, among others, a component instance *clock* (of type *Clock* version *1*) and component instance *weatherComp* (of type *Weather* version *2*). Having this information visually present ensures that an architect is always aware of exactly which element he or she is changing, and which elements are being used in the process. More importantly, it makes it possible for an architect to incorporate more than one version of a single component, connector, or interface type in the same architecture—something that cannot be done using an existing CM system. Finally, it makes it possible for an architect to spot elements that are out-of-date and for which newer versions are available.[2]

In the figure, the architect is in the process of changing the architecture by adding an optional component instance. After the architect selects the type and version of the desired component type to be instantiated, inputs the name under which the component instance will be created, and specifies its Boolean guard, the new optional component is added to the architecture. Note that it is not immediately linked to the other components. To do so, a new connector may be added or an existing connector may be modified to add links to the new optional component. Other types of elements (e.g., nonoptional component instances, interface instances, connector instances, and so on) are added in an analogous way. Naturally, the environment also supports an architect in removing and modifying elements.

While designing an architecture, behaviors and constraints can be specified using ArchEdit [Dashofy et al. 2002a], a syntax-based editor that is a standard part of the xADL 2.0 toolkit. We have integrated ArchEdit with our design environment to allow easy editing of behaviors and constraints. Similarly, subtype relations can also be specified using ArchEdit.

5.2 Selector Subsystem

Mae supports an architect in selecting a specific configuration using its selector subsystem. Selection of a configuration is performed in two phases: first, an

[2]While it is possible to do so manually and an architect will often have sufficient knowledge to do so, automated support is desired. Mae provides this support in the form of a toggle that turns on or off automatic highlighting of all out-of-date elements.

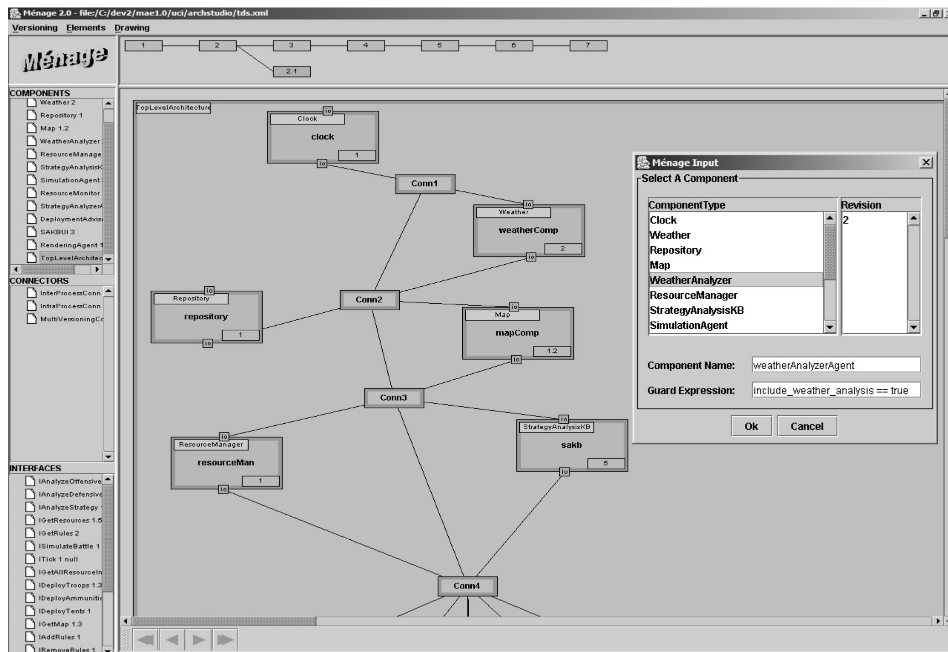

Fig. 3. Mae's design subsystem as applied to TDS.

architect selects a "starting point" by choosing a particular version of an architecture or subarchitecture (shown in the top part of Figure 4); then, the architect specifies a set of properties as name-value pairs (shown in the middle part of Figure 4). Using these properties, Mae hierarchically traverses the architecture and attempts to resolve each of the Boolean guards that it encounters. If it can fully resolve a Boolean guard to *true*, the respective element is included. If it can fully resolve a Boolean guard to *false*, the respective element is removed. If a Boolean guard can only be partially resolved, the element is included with the reduced Boolean guard attached. A single selection, thus, may not always result in resolution of all optional and variant elements. Iterative use of the selector subsystem, however, should eventually result in the selection of a fully resolved architecture if the user is careful to input properties that resolve the remaining Boolean guards.

5.3 Analysis Subsystem

The analysis subsystem completes Mae's architectural evolution environment. It complements the other components by providing an architect with the ability to ensure the consistency of a selected architectural configuration. Consistency can be checked at both the syntactic and semantic level. At the syntactic level, the analysis subsystem checks for simple constraints concerning the topological and structural specification of an architecture. This includes, among others, verifying whether the names of all component and connector instances are unique and (optionally) ensuring that the topology of the architecture adheres to a

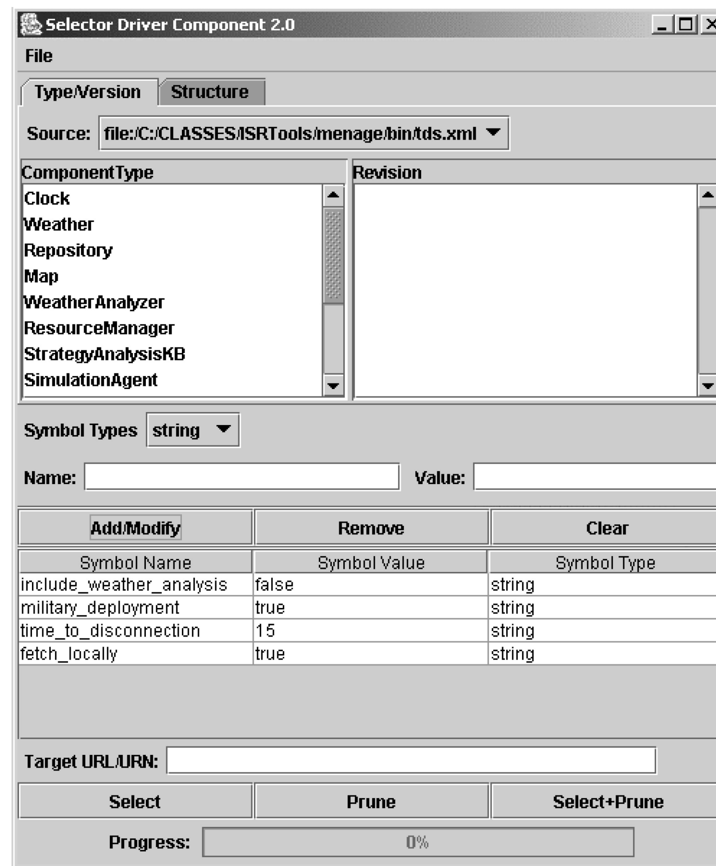

Fig. 4. Mae analysis subsystem as applied to TDS.

particular style. The latter is achieved using a pluggable topological constraint checker component with a generic API that can be adapted to any specific stylistic set of constraints.

To verify the semantic correctness of an architectural configuration, the analysis subsystem employs type-checking techniques. Given an architecture, Mae analyzes each component to ensure that its interfaces, behaviors, and constraints are satisfied by other components along its communication links [Medvidovic et al. 1999]. First, Mae makes sure that all provided and required services of communicating components properly match, and then it interprets the behaviors and constraints of each component (specified in first-order predicate logic) to verify whether they are satisfied. Finally, Mae verifies whether the subtyping relations specified within the architectural configuration hold.

Figure 5 shows the UI of the analysis subsystem. The top window indicates the progress of each step that comprises the process of verifying the overall consistency of an architecture. The bottom window shows the analysis output of each step. In this particular case, a mismatch is detected: the components *SimulationAgent* version *3, StrategyAnalysisKB* version *5*, and *Weather* version *2*

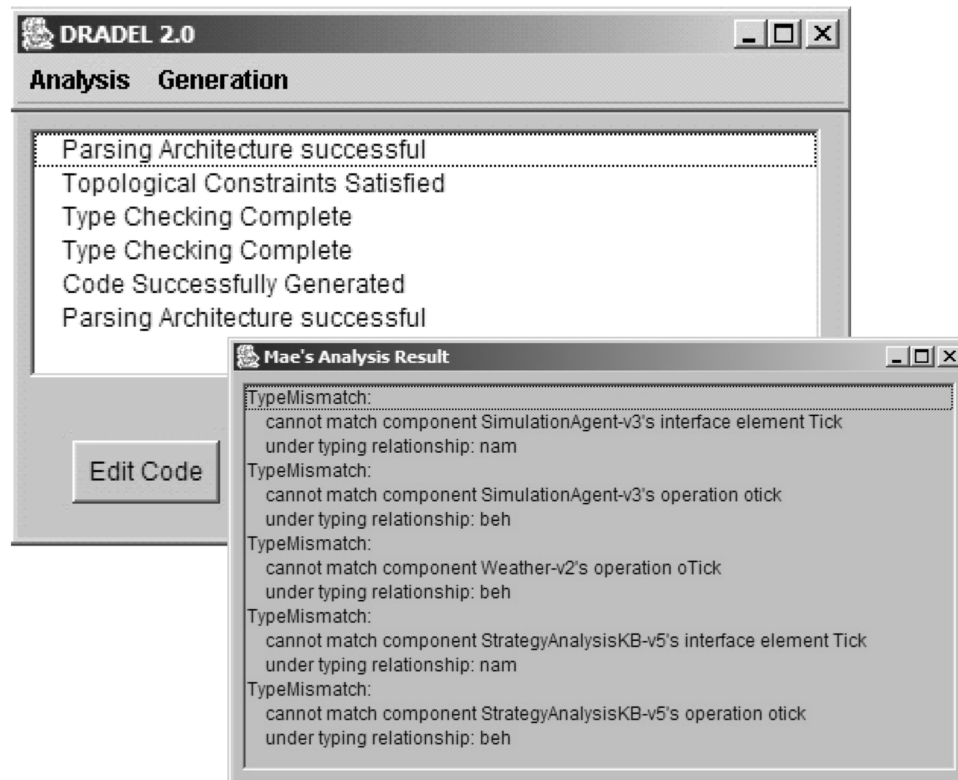

Fig. 5. Mae analysis subsystem as applied to TDS.

all require the interface *Tick*, but none of them is connected to a component that provides that interface.

5.4 Additional Capabilities

In building Mae, we discovered that its integrated system model enabled us to build several novel capabilities that significantly enhance the environment. While these capabilities are considered to be add-ons rather than core functionality of the environment, we discuss them here to further demonstrate how our integrated system model provides significant benefits in terms of the management of architectural evolution.

5.4.1 *Version-Aware Design Assistance.* Consider a situation in which a developer needs to replace a faulty new version of a component with an older version. Since the overall configuration may also have evolved, the developer has to *search* for a version of the component that is compatible with the rest of the configuration. In a typical CM system, such a search involves the developer checking out a version of the component, compiling the system, and subsequently executing and testing the system to verify its correctness. This process may need to be repeated multiple times until eventually (and hopefully) a suitable version is found.

Mae automates this process by leveraging the integrated nature of its system model. In particular, we have implemented a preliminary version of a *Compliance Analyzer* subsystem that, given a particular version of a component or connector, presents an architect with a list of related versions that exhibit a desired subtyping relationship to the original version. For instance, an architect may request a list of all versions that are interface compliant or just a list of those versions that are behaviorally compliant.

The Compliance Analyzer implements this functionality by traversing the ascendant and descendant relations of the element in question and, for each version, analyzing whether the desired relationship holds. This clearly relies on the integrated nature of the system model, since the Compliance Analyzer needs both architectural information (i.e., behaviors, constraints, and interfaces) and CM information (i.e., ascendants, descendants) to properly operate.

Note that when the Compliance Analyzer traverses the version tree of an architectural type, it repeatedly determines the subtyping relationship that may exist among two versions of that type. This feature is useful in and of itself, and we are currently investigating the adaptation of that feature in the Mae environment such that it is automatically invoked each time a new version of a component or connector is checked in. This would allow automated annotation of every change with the proper subtyping relationship, and would relieve a developer from having to document this fact manually.

5.4.2 *Component-Level Patches.* By relying on a mapping from architectural components to implementation classes, several approaches have used software architecture to interpret change descriptions at the architectural level and dynamically administer the corresponding changes at run-time [Gorlick and Razouk 1991; Magee and Kramer 1996, Oreizy et al. 1998]. For instance, the use of extension wizards offers architecture-level patches that contain a series of differences between an actual architecture and a new, desired version of the same architecture [Oreizy et al. 1998]. The differences are precisely specified in terms of additions and removals of components and connectors, and form a logical recipe for updating a running system. One drawback of the current approach to extension wizards is that they have to be created by hand. In case a large number of differences exists between the actual and desired version of an architecture, this can be a cumbersome and error-prone task.

Mae's system model contains all the information necessary to automate the processes of *architectural differencing* (to create an architecture-level patch) and *architectural merging* (to apply an architecture-level patch). To explore this opportunity, we first created a xADL 2.0 extension for capturing architectural differences [Van der Westhuizen and van der Hoek 2002]. A unique aspect of this representation is the fact that it not only identifies instances of the architectural elements that need change, but also contains the types of those elements. As a result, the patch is self contained and can be applied independently of the originating architecture.

The second part of our solution consists of the implementation of an architectural differencing algorithm and an architectural merging algorithm

[Van der Westhuizen and van der Hoek 2002]. By mapping the merging algorithm onto a run-time management system [Dashofy et al. 2002b], we successfully replaced the manual approach of extension wizards with a fully automated run-time patch creation and patch application mechanism. While the details of the approach are beyond the scope of this article, and further information and detailed algorithms can be found elsewhere [Van der Westhuizen and van der Hoek 2002], it is important to note that the algorithms are rather simple and that the overall approach could be quickly constructed. Mae's integrated system model provides all the information necessary and guarantees critical properties, such as unique identifiers and an accurate versioning history, which the algorithms leverage.

5.4.3 *Multiversioning Connectors.* A special case of run-time upgrade occurs when a new version of an already existing component must be incorporated into a running system. Traditional approaches, discussed in Section 5.4.2, will simply swap out the old component for the new one. Prior to performing this swap, however, it is important to assess its impact onto the running system. In Section 5.3, we already discussed how Mae's analysis subsystem can be used to ensure that the new version of the component matches with the rest of the system. However, there is no guarantee that the *implemented* version of the component will preserve the desired properties and relationships established at the architectural level.

Mae's system model can be leveraged to address this problem. Because Mae is built on top of xADL 2.0, it inherits the implementation mappings to relate architecture-level elements with implemented Java classes and interfaces [Dashofy et al. 2002a]. Using these facilities, we built special-purpose software connectors to address the problem of ensuring reliable component upgrades. These connectors, called *multiversioning connectors (MVC)*,[3] allow any component in a system to be replaced with a set of its versions. Although executing in parallel, these components are wrapped by MVC's to appear to the rest of the system as a single component. The role of each MVC is to relay any invocation that it receives from the system to all component versions it encapsulates, and to propagate the generated result(s) to the rest of the system.

Each component version may produce some result in response to an invocation. The MVCs allow a system's architect to specify the component authority [Cook and Dage 1999] for different operations. A component designated as authoritative for a given operation will be considered nominally correct with respect to that operation. Should any of the other versions deviate from these results, this inconsistency will be detected and reported by the MVC—allowing an architect to discover unexpected run-time deviations in functionality and behavior. Further details and examples of the usage of MVC may be found in Rakic and Medvidovic [2001].

[3]This usage of the *MVC* acronym is entirely unrelated to the Model-View-Controller approach to constructing Smalltalk applications [Krasner and Pope 1998].

6. EXPERIENCE

In this section, we discuss our experience with Mae to demonstrate its utility with respect to critical attributes such as usability, scalability, and applicability to real-world problems. Particularly, Mae has been successfully used in three different settings as the primary architectural development and evolution environment. Our first experience involved applying Mae to an audio/video entertainment system, patterned after an existing architecture for consumer electronics [Van der Westhuizen and van der Hoek 2002]. We next applied Mae to create and evolve the software architecture of the Troops Deployment and Battle Simulations system (TDS) [Mikic-Rakic and Medvidovic 2002] that was introduced in Section 3.1. Finally, we applied Mae to SCRover, an independent research project at the University of Southern California (USC) conducted in collaboration with NASA's Jet Propulsion Laboratory (JPL). These settings collectively represent three very different applications, and include an in-house application with extensive architectural variation, a highly scalable application with potentially hundreds of component and connector instances, and an application to an embedded system.

6.1 Usability

In using Mae for the specification and evolution of our three example systems, we paid particular attention to whether the presence of configuration management functionality hinders or obscures the process of designing an architecture. Our evaluation of usability, therefore, focuses on whether it was possible to create the three respective architectures much like one would create those architectures in existing environments such as ArchStudio [Dashofy et al. 2002a] or AcmeStudio [Garlan et al. 1997]. In general, our experience shows this to be the case. A good example is the action of adding a connector to an existing configuration. In existing environments, this is performed by selecting a connector type and choosing the interfaces to which it connects. In Mae, this action is performed in exactly the same way should a regular connector be added. Mae's functionality only deviates if it is desired that an optional connector is added. This deviation is minor, however, as the only additional action to be performed by the user is to specify the Boolean guard governing the inclusion of the connector.

We compared each of the actions supported by Mae to their counterparts in ArchStudio and AcmeStudio. We learned that some special attention has to be paid at certain times. For example, instead of simply choosing a type, an architect must choose a type and a version; a Boolean guard must be specified when an optional or variant element is included in the architecture; and finally, elements must be checked out and checked in. The fundamentals of the process of designing an architecture, however, does not change. Moreover, the user only needs to do something different when the unique features of Mae are used; at other times, the process remains exactly the same. The inclusion of configuration management functionality, thus, incurs minimal overhead as compared to traditional architectural design.

The only exception to having minimal overhead is the check out operation: an architect must manually check out, one by one, all the elements he/she intends to modify. This clearly is cumbersome, and will be addressed in an upcoming version of Mae's design subsystem. In particular, we intend to implement a modified check-out operation that, instead of requiring manual checkouts, automatically and transparently checks out artifacts when an architect begins to modify them. This should further reduce the impact of the presence of configuration management functionality on the core tasks of architectural design.

6.2 Scalability

The scalability of Mae was demonstrated in all three contexts. The architecture of the audio/video entertainment system consists of 25 component types, 3 connector types, and 3 interface types, all available in a number of different versions. Numerous variation points exist in the architecture, resulting in the availability of about 100 different configurations. The structure of the Troops Deployment and Battle Simulations system (TDS) contains a moderate number of component types (20) and connector types (3). However, the number of component and connector instances can be in the hundreds, depending on the number of *Commander* and *Soldier* subsystems included, since those subsystems can be replicated multiple times. Finally, the SCRover architecture contains about 30 component types, 9 interfaces types, and 3 connector types that are all designed to strictly adhere to the stylistic rules defined by NASA JPL's Mission Data Systems (MDS) framework [Dvorak et al. 1999].

Mae successfully supported the initial specification and subsequent evolution of all three systems, and did so regardless of the number of architectural types and instances. The only issue that surfaced was that Mae operated slower when it had to graphically depict a large number of architectural elements. This is due to Mae's event-based communication, which requires the retrieval of the necessary data from the underlying xADL 2.0 libraries before it can be displayed. We are investigating ways of improving Mae's response time by enabling an internal cache. Other, more minor issues, were resolved as they occurred, leading to the final incarnation of Mae presented in this article.

6.3 Applicability

We evaluated Mae's applicability to real-world settings in the context of SCRover, a testbed for NASA's High Dependability Computing Program (HDCP). An independent research group at USC collaborated with JPL to model and analyze the evolving software architecture of the SCRover application, a mobile robot system built using JPL's MDS framework [Dvorak et al. 1999]. The SCRover architecture is designed to strictly adhere to the stylistic rules defined by the MDS framework. To support the specification and analysis of SCRover using Mae, we reconfigured and enhanced the Mae environment to support the MDS architectural style. Reconfiguration involved adding xADL 2.0 extensions that model MDS-specific constructs. Because of Mae's extensibility, which is inherited from the xADL 2.0 approach, no new code needed to be added to the design subsystem to absorb the additional extensions. Enhancements

involved modifying Mae's analysis subsystem to perform consistency analysis on architectural models adhering to the MDS style, which specifies the details of component and connector interfaces differently and in more detail than our approach outlined in Section 4. The details of these modifications are beyond the scope of this paper, but can be found in Roshandel et al. [2004].

The two most important aspects of the SCRover experience are that Mae is being used by an external group of users, and that it is being applied to a real-world application. While this is an ongoing project, so far the experience has been extremely positive. All relevant aspects of the SCRover application could be modeled, and the users were able to do so in a straightforward way. As a result, Mae is currently being evaluated by JPL and several other universities and research centers involved with the SCRover project.

7. RELATED WORK

Few approaches have combined software architecture and CM concepts in addressing architectural evolution. Two notable exceptions are UniCon [Shaw et al. 1995] and Koala [van Ommering 2002]. UniCon was the first ADL to incorporate constructs for capturing variant component implementations. Based on a property selection mechanism, each component in a given architectural configuration is instantiated with a particular variant implementation. Compared to Mae, UniCon is limited: its system model does not provide facilities for capturing architectural revisions or options. Moreover, the primary focus of UniCon is implementation-level variability, not variability at the level of the definitions of architectural elements.

On the other hand, the Koala ADL is specifically designed for capturing product family architectures. While other approaches to capturing product family architectures exist [Asikainen et al. 2002, 2003; Krueger 2002, Rochkind 1975], Koala is the most advanced and provides the most complete set of features. Specifically, Koala naturally models variability and optionality via a property mechanism similar to Mae's. Using a versioning system, Koala is even able to capture the evolution of an application family. However, two critical differences exist between Koala and Mae. First, Koala does not integrate versioning information inside its representation; it uses an external CM system instead. This has the drawback of creating another, independent source of information to be used in capturing architectural evolution. Second, Koala does not provide mechanisms for capturing subtypes, behaviors, and constraints. Mae extensively uses this kind of information in providing the functionality described in Sections 5 and 6.

Mens and D'Hondt [2000] leverage the Object Constraint Language (OCL) to capture the evolution of UML models. Specific constraints describe how one model is different from another. Their technique formalizes the evolution of UML models, but it is unclear whether it could be extended to other modeling techniques. In comparison to Mae, their approach operates at a lower level of abstraction (i.e., classes, objects) and does not explicitly address the issue of versions, variants, and optionality.

Several other approaches in the fields of CM and software architecture laid the foundation for the work presented in this article. In the field of CM, Proteus PCL introduced a system model in which components were explicitly recognized [Tryggeseth et al. 1995]. Adele introduced interfaces to be used in verifying the consistency of selected configurations [Estublier and Casalles 1994]. Finally, Inscape used obligations and pre- and post-conditions in a manner similar to Mae's use of behavior and constraint specifications to guarantee proper interactions among components put together in a configuration [Perry 1989]. Compared to Mae, these approaches provide some of the building blocks, but none of them integrates all the necessary facilities to manage architectural evolution.

In the field of software architecture, several approaches have extensively used explicit software connectors; prominent examples are Wright [Allen and Garlan 1997], UniCon [Shaw et al. 1995], and C2 [Taylor et al. 1996]. Rapide [Luckham and Vera 1995] uses an OO-like inheritance mechanism to support component evolution, while Acme [Garlan et al. 1997] supports structural subtyping. Finally, GenVoca [Batory and O'Malley 1992] and Aesop [Garlan et al. 1994] were early examples of approaches that used styles as a means of guiding and controlling architectural evolution. Once again, each approach provides one or more building blocks, but none of them integrates them to precisely capture all aspects of architectural evolution.

8. CONCLUSION

Perhaps the most challenging aspect of software engineering is effective management of software systems' evolution. This is evidenced by statistics that place the costs of evolution at well over 50% of all development costs [Ghezzi et al. 1991]. It is also evidenced by a large and growing number of academic and practitioner-oriented conferences, symposia, and workshops dealing with different aspects of software maintenance and evolution.

However, current approaches to software evolution exhibit an interesting idiosyncrasy. On the one hand, it is widely accepted that architecture is a software system's centerpiece. As such, it would appear to be the natural starting point for software evolution. On the other hand, aside from some notable exceptions [Magee and Kramer 1996; Oreizy et al. 1998; van Ommering 2002], software architecture has not been leveraged as the basis of software system evolution. Further compounded by the fact that existing CM solutions are unable to effectively deal with architectural evolution, this effectively negates the central importance of software architecture.

Our work presented in this article has been targeted precisely at this issue. We have presented a novel approach to managing architecture-centered software evolution. Our approach has indeed anchored the evolution process to the architectural concepts of components, connectors, subtypes, and interfaces, enhancing it with the power and flexibility of the CM concepts of revisions, variants, options, and configurations. The result of this approach is a novel architectural system model with a novel associated architecture evolution environment that, combined, effectively address architectural evolution.

With the advent of our system model and the implementation of the Mae environment, we have only begun to explore the richness of the problem of properly managing architectural evolution from the time a system is designed to its eventual deployment and execution in the field. For example, Mae's novel capabilities discussed in Section 5.4 represent only a set of starting points for our work: although each capability already has been shown useful, none should be considered complete as of yet. We continue to explore more advanced functionality by further enhancing and applying our solutions. Specifically, we have begun addressing the following three issues: (1) extending Mae with additional design-time functionality by providing additional types of analyses and advanced change-based versioning support; (2) tightly integrating development-time architectural evolution with the evolution of a deployed system at run-time [Mikic-Rakic and Medvidovic 2002]; and (3) enhancing multi-versioning connectors and the run-time infrastructure to further increase the reliability of system upgrades. Moreover, our long-term interests are to further investigate the relationship among typing, software architecture, and CM in addressing evolution: these techniques are related and, at times, equivalent. A deeper understanding of their relationship and the tradeoffs among them is much needed. We believe that Mae forms a solid foundation upon which we can perform these investigations.

AVAILABILITY

For more information on the Mae environment, its xADL 2.0 extension schemas, and the example applications presented in this article, visit the Mae Web Site at:
http://cse.usc.edu/~softarch/mae.html

For information on the SCRover project visit USC's High Dependability Computing Web Site at:
http://cse.usc.edu/hdcp

Details of xADL 2.0 core schemas may be found at:
http://www.isr.uci.edu/projects/xarchuci/

ACKNOWLEDGMENTS

We wish to thank the following individuals for their involvement in the development of Mae: Marwan Abi-Antoun, Ping Chen, Matt Critchlow, Eric Dashofy, Ebru Dincel, Rob Egelink, Akash Garg, and Christopher Van der Westhuizen.

REFERENCES

AGRAWAL R., BUROFF S., GEHANI N. H., AND SHASHA, D. 1991. Object versioning in ODE. In *Proceedings of the 7th International Conference on Data Engineering* (Kobe, Japan), pp. 446–455.

ALLEN R. AND GARLAN D. 1997. A formal basis for architecture connection. *ACM Trans. Softw. Eng. Meth. 6*, 3, 213–249.

ASIKAINEN, T., SOININEN, T., AND MÄNNISTÖ, T. 2003. Towards intelligent support for managing evolution of configurable software product families. In *Proceedings of the ICSE Workshops SCM 2001 and SCM 2003 Selected Papers*. 86–101.

ATKINSON, C., BAYER, J., BUNSE, C., KAMSTIES, E., LAITENBERGER, O., LAQUA, R., MUTHIG, D., PAECH, B., WUST, J., AND ZETTEL, J. 2002. *Component-Based Product Line Engineering with UML*. Addison-Wesley, Reading, Mass.

BATORY, D., AND O'MALLEY, S. 1992. The design and implementation of hierarchical software systems with reusable components. *ACM Trans. Softw. Eng. Meth. 1*, 4.

BELL LABS LUCENT TECHNOLOGIES. 1997. *Sablime v5.0 User's Reference Manual.* Lucent Technologies, Murray Hill, N.J.

BOSCH J., FLORIJN, G., GREEFHORST, D., KUUSELA, J., OBBINK, J. H., AND POHL, K. 2001. Variability issues in software product lines. In *Proceedings of the Product Family Architecture Workshop*. pp. 13–21.

BURROWS, C. AND WESLEY, I. 2001. Ovum evaluates configuration management. Ovum, Ltd., Burlington, Mass.

CHRISTENSEN, H. B. 1998. Experiences with architectural software configuration management in Ragnarok. In *Proceedings of the 8th International Symposium on System Configuration Management*.

CONRADI, R. AND WESTFECHTEL, B. 1998. Version models for software configuration management. *ACM Computing Surveys 30*, 2, 232–282.

COOK, J. E. AND DAGE, J. A. 1999. Highly reliable upgrading of components. In *Proceedings of the 1999 International Conference on Software Engineering*. pp. 203–212.

DASHOFY, E. M., VAN DER HOEK, A., AND TAYLOR, R. N. 2002. 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.

DASHOFY, E. M., VAN DER HOEK, A., AND TAYLOR, R. N. 2002. Towards architecture-based self-healing systems. In *Proceedings of the 1st ACM SIGSOFT Workshop on Self-Healing Systems*. ACM, New York.

DVORAK, D., RASMUSSEN, R., REEVES, AND SACKS, A. 1999. Software architecture themes in JPL's mission data system. In *Proceedings of the AIAA Space Technology Conference and Exposition* (Albuquerque, N.M.).

ESTUBLIER, J. AND CASALLES, R. 1994. The adele configuration manager. In *Configuration Management*. W. F. Tichy, Ed. Wiley: London, Great Britain, pp. 99–134.

ESTUBLIER, J., LEBLANG, D., VAN DER HOEK, A., CONRADI, R., CLEMM, G., TICHY, W., AND WIBORG-WEBER, D. 2002. Impact of the research community on the field of software configuration management: Summary of an impact project report. *ACM SIGSOFT Softw. Eng. Notes 27*, 5, 31–39.

EXTENSIBLE MARKUP LANGUAGE (XML). 2004. http://www.w3.org/XML/.

FRANCONI, E., GRANDI, F., AND MANDREOLI, F. 2000. A semantic approach for schema evolution and versioning in object-oriented databases. In *Proceedings of the 6th International Conference on Rules and Objects in Databases (DOOD 2000)*.

GARLAN, D., ALLEN, R., AND OCKERBLOOM, J. 1994. Exploiting style in architectural design environments. In *Proceedings of SIGSOFT'94: Foundations of Software Engineering* (New Orleans, La.). ACM, New York, 175–188.

GARLAN, D., MONROE, R., AND WILE, D. 1997. ACME: An architecture description interchange language. In *Proceedings of CASCON'97*.

GHEZZI, C., JAZAYERI, M., AND MANDRIOLI, D. 1991. *Fundamentals of Software Engineering*, Prentice-Hall, Englewood Cliffs, New Jersey.

GORLICK, M. M. AND RAZOUK, R. R. 1991. Using weaves for software construction and analysis. In *Proceedings of the 13th International Conference on Software Engineering (ICSE13)* (Austin, Tex.).

HABERMANN, A. N. AND PERRY, D. E. 1981. System composition and version control for Ada. In *Software Engineering Environments*, H. Huenke, Ed. North-Holland, Amsterdam, The Netherlands, pp. 331–343.

HUNT, J. J. AND TICHY, W. F. 1998. Delta algorithms: An empirical analysis. *ACM Trans. Softw. Eng. Meth. 7*, 2, 192–214.

KRASNER, G. E. AND POPE, S. T. 1988. A cookbook for using the model-view-controller user interface paradigm in smalltalk-80. *J. Obj.-Orient. Prog. 1*, 3, 26–49.

KRÜEGER, C. W. 2002. Variation management for software production lines. In *Proceedings of the 2nd International Software Product Line Conference*. pp. 37–48.

KUUSELA, J. 1999. Architectural evolution. In *Proceedings of the 1st Working IFIP Conference on Software Architecture*. Kluwer Academic: Boston, Mass.

LISKOV, B. H. AND WING, J. M. 1994. A behavioral notion of subtyping. *ACM Trans. Prog. Lang. Syst. 16*, 6, 1811–1841.

LUCKHAM, D. C. AND VERA, J. 1995. An event-based architecture definition language. *IEEE Trans. Softw. Eng. 21*, 9, 717–734.

MAGEE, J. AND KRAMER, J. 1996. Dynamic structure in software architectures. In *Proceedings of the 4th ACM SIGSOFT Symposium on the Foundations of Software Engineering*. ACM New York, pp. 3–13.

MEDVIDOVIC, N., ROSENBLUM, D. S., AND TAYLOR, R. N. 1998. A type theory for software architectures. Tech. Rep. UCI-ICS-98-14, University of California, Irvine, Irvine, Calif.

MEDVIDOVIC, N., ROSENBLUM, D. S., AND TAYLOR, R. N. 1999. A language and environment for architecture-based software development and evolution. In *Proceedings of the 1999 International Conference on Software Engineering*. pp. 44–53.

MEDVIDOVIC, N. AND TAYLOR, R. N. 2000. A classification and comparison framework for software architecture description languages. *IEEE Trans. Softw. Eng. 26*, 1, 70–93.

MENS, T., D'HONDT, T. 2000. Automating support for evolution in UML. *Auto, Softw. Eng. 7*, 1, 39–59.

MIKIC-RAKIC, M., MEDVIDOVIC, N. 2002. Architecture-level support for software component deployment in resource constrained environments. In *Proceedings of 1st International IFIP/ACM Working Conference on Component Deployment* (Berlin, Germany). ACM, New York.

MUNCH, B. P. 1993. Versioning in a software engineering database—The change-oriented Way. Ph.D. dissertation. Division of Computer Systems and Telematics, The Norwegian Institute of Technology.

OREIZY, P., MEDVIDOVIC, N., AND TAYLOR, R. N. 1998. Architecture-based runtime software evolution. In *Proceedings of the 20th International Conference on Software Engineering* (Kyoto, Japan). 177–186.

PARISI, F. AND WOLF, A. L. 2000. Foundations for software configuration management policies using graph transformations. In *Fundamental Approaches to Software Engineering 2000*. Springer-Verlag, New York, pp. 304–318.

PERRY, D. E. 1989. The inscape environment. In *Proceedings of the 11th International Conference on Software Engineering*. pp. 2–11.

PERRY, D. E. AND WOLF, A. L. 1992. Foundations for the study of software architectures. *ACM SIGSOFT Softw. Eng. Notes 17*, 4, 40–52.

RAKIC, M. AND MEDVIDOVIC, N. 2001. Increasing the confidence in off-the-shelf components: A software connector-based approach. In *Proceedings of the 2001 Symposium on Software Reusability* (Toronto, Canada).

ROSHANDEL, R., SCHMERL, B., MEDVIDOVIC, N., GARLAN, D., AND ZHANG, D. 2004. Understanding tradeoffs among different architectural modeling approaches. In *Proceedings of the 4th Working IEEE/IFIP Conference on Software Architecture* (WICSA 2004), (Oslo, Norway). IEEE Computer Society Press, Los Alamitos, Calif.

ROCHKIND, M. J. 1975. The source code control system. *IEEE Trans. Softw. Eng. SE-1*, 4.

SEIWALD, C. 1996. Inter-file branching—A practical method for representing variants. In *Proceedings of the 6th International Workshop on Software Configuration Management*. Springer-Verlag, New York, pp. 67–75.

SHAW, M., DELINE, R., KLEIN, D., ROSS, T., YOUNG, D., AND ZELESNIK, G. 1995. Abstractions for software architecture and tools to support them. *IEEE Trans. Softw. Eng. 21*, 4.

SPECK, A., PULVERMÜLLER, E., AND CLAUSS, M. 2002. Versioning in software modeling. In *Proceedings of the 6th International Conference on Integrated Design and Process Technology*.

TAYLOR, R. N, MEDVIDOVIC, N., ANDERSON, K., WHITEHEAD, J. AND, ROBBINS, J. 1996. A component- and message-based architectural style for GUI software. *IEEE-Trans. Softw. Eng. 22*, 6.

TICHY, W. F. 1985. RCS, A system for version control. *Softw.–Pract. Exper. 15*, 7, 637–654.

TRYGGESETH, E., GULLA, B., AND CONRADI, R. 1995. Modeling systems with variability using the PROTEUS configuration language. In *Proceedings of the 5th International Workshop on Software Configuration Management*, Springer-Verlag, New York, pp. 216–240.

VAN DER HOEK, A. 2000. A generic, reusable repository for configuration management policy programming. Ph.D dissertation. Dept. Computer Sci., Univ. Colorado, Boulder, Boulder, Col., Jan.

VAN DER HOEK, A., HEIMBIGNER, D. H, AND WOLF, A. L. 1998a. Investigating the applicability of architecture description in configuration management and software deployment. Tech. Rep. CU-CS-862-98. University of Colorado at Boulder, Boulder, Col.

VAN DER HOEK, A., HEIMBIGNER, D. H, AND WOLF, A. L. 1998b. Software architecture, configuration management, and configurable distributed systems: A ménage a trois, Tech. Rep. CU-CS-849-98, University of Colorado, Boulder, Boulder, Col.

VAN DER WESTHUIZEN, C. AND VAN DER HOEK, A. 2002. Understanding and propagating architectural change. In *Proceedings of the 3rd IEEE/IFIP Working International Conference on Software Architectures* (Montreal, Ont., Canada). IEEE Computer Society Press, Los Alamitos, Calif.

VAN OMMERING, R. 2002. Building product populations with software components. In *Proceedings of the 24th International Conference on Software Engineering (ICSE2002)*, (Orlando, Fla).

WALRAD, C. AND STROM, D. 2002. The importance of branching models in SCM. *IEEE Comput. 35*, 9 31–38.

WESTFECHTEL, B. AND CONRADI, R. 2001. Software architecture and software configuration management. In *Proceedings of the International Workshop on Software Configuration Management*.

WEI, H. AND ELMASRI, R. 2000. PMTV: A schema versioning approach for bi-temporal databases. In *Proceedings of the 7th International Workshop on Temporal Representation and Reasoning (TIME '00)* (Nova Scotia, Canada). pp. 143–151.

WIBORG WEBER, D. 1997. Change sets versus change packages: Comparing implementations of change-based SCM. In *Proceedings of the 7th International Workshop on Software Configuration Management*. pp. 25–35.

WINKLER, J. F. H. 1986. The integration of version control into programming languages. In *Proceedings of the International Workshop on Advanced Programming Environments* (Trondheim, Norway), pp. 230–250.

Received November 2002; revised August 2003; accepted April 2004