

Taming Architectural Evolution

André van der Hoek** **Marija Rakic***
**University of California, Irvine
Dept. of Information and Computer Science
444 Computer Science Building
Irvine, CA 92697 USA
Phone: 1-949-824-6326
andre@ics.uci.edu

Roshanak Roshandel* **Nenad Medvidovic***
*University of Southern California
Computer Science Department
Henry Salvatori Computer Center 300
Los Angeles, CA 90089 USA
Phone: 1-213-740-6504
{marija,roshande,veno}@usc.edu

ABSTRACT

In the world of software development *everything* evolves. So, then, do software architectures. Unlike source code, for which the use of a configuration management (CM) system is the predominant approach to capturing and managing evolution, approaches to capturing and managing architectural evolution span a wide range of disconnected alternatives. This paper contributes a novel architecture evolution environment, called Mae, which brings together a number of these alternatives. The environment facilitates an incremental design process in which all changes to all architectural elements are integrally captured and related. Key to the environment is a rich system model that combines architectural concepts with those from the field of CM. Not only does this system model form the basis for Mae, but in precisely capturing architectural evolution it also facilitates automated support for several innovative capabilities that rely on the integrated nature of the system model. This paper introduces three of those: the provision of design guidance at the architectural level, the use of specialized software connectors to ensure run-time reliability during component upgrades, and the creation of component-level patches to be applied to deployed system configurations.

Keywords

software architecture, configuration management, evolution, system model, design environment, Mae

1. INTRODUCTION

Consider a scenario in which an organization develops an innovative word processor. Following good software engineering practices, the organization first develops a proper architecture [28] for the word processor in a suitable architectural style [33], then models this architecture in an architecture description language (ADL) [23], refines the architecture into a module design, and, finally, implements the application impeccably. The new word processor is an instant hit and many copies are sold. Motivated by this success, the organization enters a continuous cycle of rapidly advancing the word processor, creating add-ons, selling upgrades, adapting the word processor to different hardware platforms, spe-

cializing the word processor for various customers, and generally increasing its revenue throughout this process.

Configuration management (CM) systems have long been used to provide support for these kinds of situations [5]. However, advanced CM support is only available for managing source code. It is possible to manage other kinds of artifacts, but present-day CM systems do not provide much support beyond storing different versions of those artifacts. Herein lies a problem with the above scenario: as the word processor evolves, so does its architecture. These architectural changes need to be managed in a manner much like source code, allowing the architecture to evolve into different versions and exhibit different variants [14].

This paper provides a solution to this problem. We have developed an architecture evolution environment, called Mae, which has two unique characteristics. First, it is centered on an architectural system model that tightly integrates architectural concepts with concepts from the field of CM. This integration captures the evolution and variability of architectures and is necessary to represent crosscutting relationships among evolving architectural elements. Second, Mae leverages and integrates many sources of auxiliary information used to describe architectural evolution, such as architectural styles (e.g., [2,35]), subtyping relations among architectural elements (e.g., [11,21]), and behavioral and constraint specifications (e.g., [18,41]). The addition of these information sources to the system model creates a rich information web that is used by Mae to guide and govern the architectural evolution process.

Because Mae uses a system model with integrated architectural and CM concepts, it provides enhanced support during the design process. The environment, for example, uses the integrated system model to suggest to an architect, candidate (versions of) components during design. As another example, Mae uses the integrated system model to validate the consistency of any local changes within the broader framework of an entire architecture.

Mae's integrated system model provides another benefit in the form of new capabilities in the domains of software deployment [12] and run-time change management [8,25]. Specifically, the availability of explicit relations among the multiple versions of architectural elements in the model allows the creation of patches at the architectural level, instead of at the source code level. These patches, in turn, can be automatically added to and removed from installed systems. They can even be used to provide "safe" upgrades by temporarily executing multiple versions of components

simultaneously via the use of special purpose software connectors [29]. Once the desired properties of an upgrade are established, old versions of components can be safely removed from the executing system.

In the remainder of this paper, we discuss Mae and the architectural system model upon which it is based. First, in Sections 2 and 3, we briefly present the background information and an example system that, together, set the stage for the ensuing discussion. Section 4 introduces our architectural system model and Section 5 discusses the implementation of Mae. Section 6 evaluates our work by demonstrating a set of new, advanced architecture-based CM capabilities enabled and supported by Mae. We discuss related work in Section 7 and present our conclusions in Section 8.

2. BACKGROUND

The architectural system model developed in Section 4 relies on concepts from the software architecture and CM fields. This section briefly discusses these concepts.

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 (1) descriptions of the elements from which systems are built, (2) interactions among those elements, (3) patterns that guide their composition, and (4) constraints on these patterns [28]. In general, a particular system is defined in terms of a collection of *components*, their interconnections (*configuration*), and interactions among them (*connectors*).

The field of software architecture is further characterized by several additional concepts. An *architectural style* defines a *vocabulary* of component and connector types and a set of *constraints* on how instances of these types can be combined in a system or family of systems [33]. When designing a software system, selection of an appropriate architectural style becomes a key determinant of the system's success. Styles also influence architectural evolution by restricting the possible changes an architect is allowed to make. Examples of styles include pipe and filter, layered, client-server [33], GenVoca [2], and C2 [35].

To date, many *architecture description languages* (ADLs) have been developed to aid architecture-based development [23]. 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 [22], Darwin [20], Rapide [19], UniCon [34], and Wright [1]. A number of these ADLs also provide extensive support for modeling *behaviors* and *constraints* on the properties of components and connectors [23]. 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 [21], the notion of subtyping adopted by ADLs is richer than that typically provided by programming lan-

guages: it involves constraints on both syntactic (e.g., naming and interface [11]) and semantic (e.g., behavior [19]) aspects of a component or connector. ADLs' supporting tools are used to ensure that the desired subtyping relationships are preserved at the architectural level. At the same time, it should be noted that ADL tools provide no assurance that the desired relationships will hold among the *implemented* components and connectors.

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 [4]. Research and development over the past twenty-five years have produced numerous contributions within the field [7], evolving CM system functionality through three distinct generations. The first generation consists of such CM systems as SCCS [31], Sablime [3], and RCS [36]. The creation of this generation was a direct result of two immediate needs: to prevent multiple developers from making simultaneous changes to the same source file and to track the evolution over time of each source file. Both needs were satisfied through the automatic maintenance of *versioned archives*, where each archive contained a series of *revisions* to a single source file (using *delta storage* techniques to save disk space) 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 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 and advanced workspace management, research into *system models* [9,14,27,37] sparked the inception of the second generation of CM systems. System models and their associated modeling languages provide a way to capture the structure of the software being managed via *configurations*—sets of specific 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, thereby evolving both individual source files and configurations.

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. Different *CM policies* [26,38,40] are required, e.g., in situations where a large number of developers operate on a small set of source files or in cases where distributed groups of developers modify a single piece of software.

3. EXAMPLE APPLICATION

Throughout this paper, we use a simple word processor as an example application whose evolution will be managed by Mae, our architectural evolution environment. This example has been modeled and implemented in accordance with the C2 architectural

style [35]. The architecture of the word processor is shown in Figure 1. The word processor consists of a number of components that interact by exchanging messages via connectors. Of interest to this paper are three particular characteristics of the word processor (corresponding components are highlighted in the figure): (1) the *SpellChecker* component is optional; (2) the *SpellChecker* component exists in two different revisions, one of which uses the *SpellCheckRepository* component as its dictionary while the other uses arbitrary dictionaries stored on the Internet; and (3) the *WordCounter* component exists in multiple variants, each of which is designed and implemented differently, but intended to provide the same behavior. The combination of these three characteristics results in the availability of multiple versions of the word processor.

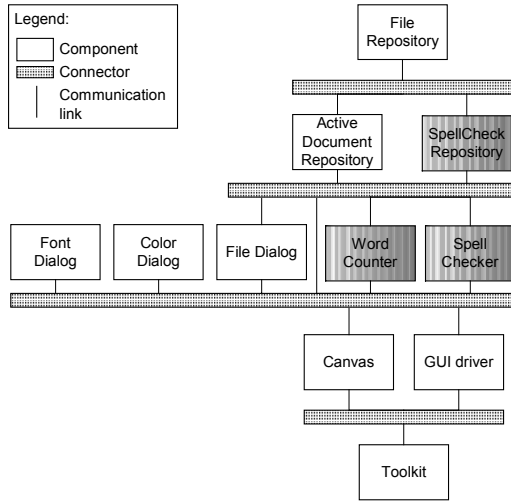


Figure 1. Example Word Processor Architecture.

4. ARCHITECTURAL SYSTEM MODEL

Mae is based upon an intimate integration of architectural and CM concepts. This integration creates a single system model in which the evolution of all architectural elements is captured in a natural and meaningful way—natural, because the system model inherently operates at the architectural level, and meaningful, because the system model relates changes to each other. Existing CM system models (e.g., ShapeTools [14] and PROTEUS [37]), are limited in this regard: with the exception of Adele (through interfaces [9]) and Inscape (through obligations and pre- and post-conditions [27]), these system models do not provide mechanisms beyond grouping, versioning, and selection.

The central role of architectural concepts in our system model changes that fact. Specifically, it allows us to leverage explicit architectural styles, subtyping relations among components, and behavior and constraint specifications for three important purposes. First, these concepts are used to meaningfully relate different versions of architectural elements. For example, subtype relations may be used to augment changes to components with information about the kind of subtype compatibility that is preserved with each change [22]. Second, these concepts can be used to ensure that particular configurations of architectural elements are consistent (e.g., matching the behavioral specifications of the architectural elements [6]). Third, the explicit and separate treatment of components and connectors creates a powerful composi-

tional modeling facility subsuming those found in existing CM system models. For example, existing CM system models only provide facilities for modeling the composition of a component out of other components, but do not provide mechanisms for precisely capturing how these other components interact with each other [7].

Figure 2 presents the details of our architecture-based system model. The model is not tied to any particular style or ADL; instead, it can be mapped onto different ADLs. In Section 5 we describe one such mapping used by Mae—to C2SADEL [22].

Types and Instances

Our architectural system model distinguishes types from instances. Every element, whether an interface, component, or connector, has to be defined as a type before it can be instantiated or used in the definition of other types. We version types to capture

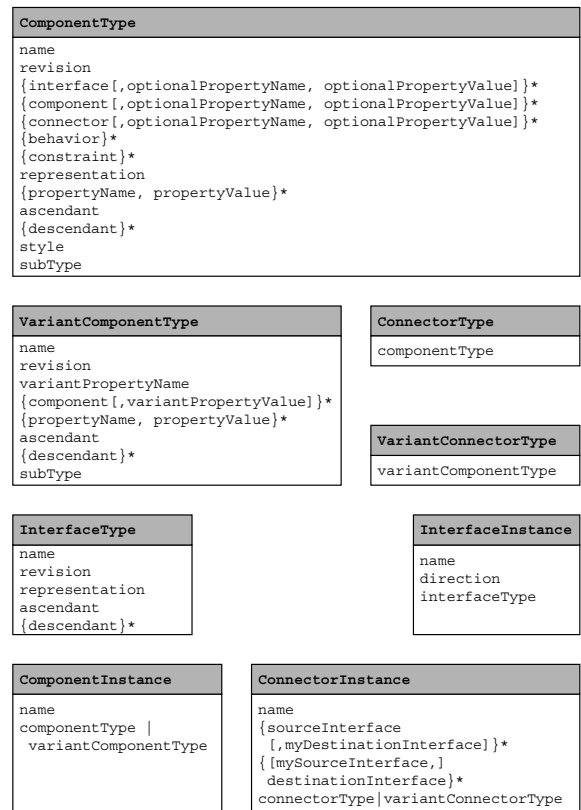


Figure 2. Architectural System Model.

the evolution of architectural elements. Each instance in our system model is, therefore, an instance of a specific version of a type.

Interfaces

The basic building blocks of our architectural system model are *interface types*, which define abstract sets of services that a component may provide or require. Each interface type is defined by a unique name, a revision number, and a representation. The *name* and *revision number* are used to uniquely distinguish each incarnation of an interface type as it evolves over time. The *representation* is used to capture a detailed specification of the interface in

the specific ADL onto which the system model is mapped (e.g., Mae currently stores C2SADEL interface specifications).

To capture diverging paths of evolution, we have adopted interfile branching [32]: a new interface type is created that has its own unique name and follows its own linear path of evolution. The fields *ascendant* and *descendant* are used to keep track of interfile branches: upon creation of a new branch, the ascendant of the new interface type is set to the original interface type. In addition, the set of descendants of the original interface type is updated with the new interface type.

As an example, consider the following definition of revision 1 of the *tWordCount* interface type used by the word processor example discussed in Section 3.

```
Name = tWordCount
Revision = 1
Representation = { CountWords ( text : String ); }
Ascendant = { }
Descendant = { tFastWordCount 1 }
```

In this example, C2SADEL is used to specify the details of the interface type in the representation field. The definition states that *tFastWordCount* revision 1 is a new interface type derived from *tWordCount* revision 1. Subsequent evolution of *tFastWordCount* is captured in a separate branch.

An *interface instance* is defined in terms of a specific revision of an interface type. Each instance has 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.

Components

Component types are used to hierarchically model the composition of an architecture. In the definition of a component type, *name*, *revision number*, *representation*, *ascendant*, and *descendant* serve the same purpose as they do in the definition of an interface type. Our system model captures many additional aspects of a component type. Specifically, each component type is defined in terms of interface instances, component instances, connector instances, behaviors, constraints, subtype, and style. *Interface instances* describe the services that are provided and/or required by a component type. *Component* and *connector instances* define a component type, resulting in the hierarchical construction of an architecture out of finer grain elements. *Behavior* and *constraint* specifications can be associated with each component type and the relation of the component type to its predecessor (whether a branch or revision) can be captured via the *subtype* field. The subtype relations that are currently captured allow a subtype to preserve its supertype’s interface, behavior, or both [21]. Similarly to the representation, the language in which behaviors, constraints, and subtypes are specified is opaque to the system model and mapped onto a specific ADL by the tools that use the system model. Finally, the specific rules of architectural style to which the internal composition of a component type has to adhere are captured in the *style* field. Therefore, styles can be defined on a per-component type basis.

The interface, component, and connector instances that constitute a component type may be *optional*. Specifically, each instance

may be guarded by a *property* consisting of a name/value pair. Depending on the actual property value provided by an architect, the instance may or may not be included in the architecture. Note that multiple instances may depend on the same name/value pair, allowing the inclusion or exclusion of multiple architectural elements through the specification of a single property.

Consider the following definition of revision 3 of the *tSpellChecker* component type.

```
Name = tSpellChecker
Revision = 3
Interface = { iSpellCheck }
Component = { iTokenizer,
              iResultCollector,
              iStatistics, collectStatistics, true }
Connector = { iC2bus1,
              iC2bus2, collectStatistics, true,
              iC2bus3 }
Behavior = { iSpellCheck* }
Constraint = { }
Representation = { <<omitted for brevity>> }
Ascendant = { tSpellChecker 2 }
Descendant = { }
Style = { C2 }
SubType = { beh \and int }
```

In this example, the *tSpellChecker* component type exposes one interface and has a defined behavior in which the interface can be invoked over and over again. The component type is hierarchically constructed out of the *iTokenizer*, *iResultCollector*, and *iStatistics* component instances, which are connected by the *iC2bus1*, *iC2bus2*, and *iC2bus3* connector instances. Although not included in the example, the specifications of these connector instances establish connections among the component instances, thereby implicitly defining the topology of the *tSpellChecker* component type. The *iStatistics* component instance and the *iC2bus2* connector instance are optional, depending on the value of the property *collectStatistics*. Note that the instances do not have revision numbers associated with them because each instance is defined elsewhere in terms of a specific revision of a specific type.

Variants

In addition to “regular” component types, our system model is able to represent *variant component types*. These component types encapsulate sets of alternative component instances, one of which is used at a time. For example, revision 2 of the *tWordCounter* variant component type is defined as follows.

```
Name = tWordCounter
Revision = 2
VariantPropertyName = method
Component = { iCounterViaWhiteSpace, quickdirty,
              iTokenizingCounter, quick,
              iSpecialCharacterCounter, accurate }
Ascendant = { tWordCounter 1 }
Descendant = { }
SubType = { beh }
```

Three alternative component instances constitute the *tWordCounter* variant component type. One instance is selected to be included in an architecture based on the value of the variant prop-

erty method (e.g., if the value is *quick*, *iTokenizingCounter* will be instantiated).

Isolating variability as a separate type represents a departure from most CM system models, which manage variability and evolution in a single data structure—the version tree. However, our choice is consistent with Adele [9] and Koala [39], which successfully have used similar approaches in isolating variant handling as specific variant “points” in an architectural configuration.

Note that the definition of a variant component type does not contain interface instances. As a rule, a variant component type exhibits the (unique) interfaces exposed by all of its components. Selection of a particular variant may result in an illegal architectural configuration if other components use interfaces that are not exposed by the selected variant component. It has already been demonstrated by Koala that the added benefits of flexibility and evolvability outweigh this problem [39]. Moreover, behavior and constraint specifications can be leveraged to ensure consistency of a particular configuration once a variant has been instantiated.

Component instances, connector types, variant connector types, and connector instances complete Mae’s architectural system model. *Component instances* are simply named instances of a component type or variant component type. Following the view of Darwin [20], our system model defines *connector types* (*variant connector types*) in the same hierarchical manner as component types (variant component types). As can be seen in Figure 2, however, *connector instances* are different from component instances in that each connector instance has an associated set of links that connect sets of component instances. Since two schools of thought exist in the field of software architecture (one in which connectors do not have interfaces [35] and one in which they do [1]), our system model supports connector instances that link component interfaces, both directly and via the connector’s interfaces.

Discussion

Our system model borrows from many previous contributions and unifies them in a single, flexible representation that is unique in marrying architecture and CM concepts. This combination of concepts is a key contribution of our approach: it creates an advanced, rich kind of system model that is centered on the explicit use of architectural entities. Although clearly related, each concept in this model serves a very distinct purpose:

- Revisions capture *linear* evolution;
- Inter-file branches capture *diverging* paths of evolution;
- Subtyping captures the specific *evolution constraints* between two successive versions (whether revisions or branches) of an architectural element, demanding the preservation of specific properties (interface and/or behavior);
- Variant component and connector types capture *alternatives* within an architecture in specific, localized variation points;
- Options capture architectural elements whose inclusion in a configuration is *not mandatory*;
- Behaviors and constraints capture *compatibility rules* and *expectations* of each component and connector type; and
- Styles capture the *composition rules* of each component and connector type.

A strength of our model is that the orthogonal nature of the concepts allows them to be meaningfully combined. For example, a

variant component type such as *tWordCounter* can evolve just like a “regular” component type (i.e., one that is not a variant). As another example, it is possible to use subtyping to capture additional information about the relationships among the multiple variants of *tWordCounter*: *iTokenizingCounter* may be an interface subtype of *iCounterViaWhiteSpace*.¹ In this case, capturing and exploiting the relationship between variants (a CM concept) and subtyping (an architectural concept) provides a benefit that neither was able to provide alone: variants afford architects with flexibility in specifying and evolving an architecture, while subtyping constrains that flexibility to ensure desired system properties during evolution.

5. IMPLEMENTATION

To demonstrate the utilities of the architectural system model, we have developed a prototype architecture evolution environment called Mae. The current implementation of Mae consists of an extension and loose integration of DRADEL, an environment for supporting architecture-based analysis and development, and Mé-nage, a graphical environment for specifying versioned software architectures. This particular combination provides the functionality needed to manage architectural evolution: like any architecture development environment, architectures can be created, manipulated, and analyzed; in addition, all changes to the architecture are captured and related to each other using the system model described in the previous section.

Shown in Figure 3, the graphical user interface of Mae consists of two windows through which all user interaction is coordinated. The first window is a generic interface for the specification of evolving architectures. It is complemented by the second window that creates the binding from the generic interface to a specific architecture description language—in this case, C2SADEL [22]. With this explicit separation of concerns, our environment is more amenable to adapting to other ADLs: only the language binding needs to be replaced while the generic interface may be reused.

The generic interface for specifying evolving software architectures is divided into three parts: the *canvas*, which allows an architect to define a new (version of a) component, connector, or interface type; the *version tree* (top) associated with the artifact being defined; and the *design palette* (left), which lists all versions of all types already defined. In the figure, version 3 of the *tSpellChecker* component type is being defined in terms of specific instances of several other component types, including *tTokenizer* version 5, *tResultCollector* version 1, and (optionally, as indicated by the white borders) *tStatistics* version 2. Note the use of specific version identifiers in the hierarchical construction of the *tSpellChecker* component type: not only is a particular version of *tSpellChecker* defined (3), but the versions of the instances out of which it is created are also explicitly defined as prescribed by our system model.

To manage the multitude of changes that may occur when an architecture evolves, Mae supports a check-out/check-in policy for all architectural elements; architects check out one or more elements, make changes to them, and check them back in once the modifications are complete. As a result, a version history of archi-

¹ Interface subtyping preserves the original interface, but is allowed to alter the behavior accessed via that interface [21].

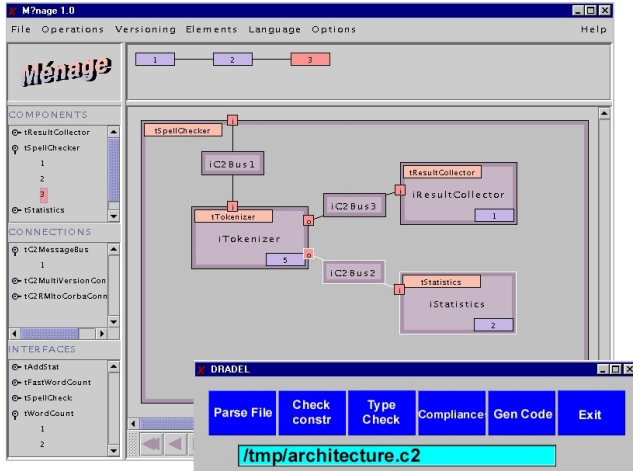


Figure 3. The Mae environment

tectural elements is incrementally created, allowing the architect to retrieve and examine previous versions, and also to undo changes. Note that changes may pertain to any aspect of our system model. Besides the “normal” types of changes regarding the actual architectural hierarchy and types, behaviors, constraints, subtype relations, styles used in a component or connector type, and even the set of properties associated with a particular version of a component or connector type may all change. In all these cases, however, the change has to be explicitly captured and a new version of the architectural element has to be created.

Once an evolving architecture has been specified, Mae supports the selection of a particular configuration out of the multitude of available architectural configurations. First, a specific revision of the architecture has to be selected. Within this revision, variants and options may still be undefined. The variants and options that are included in the final configuration are determined by the property values supplied to Mae by a developer: each variant and option whose property values match those supplied by the developer are included. To support a developer in this selection process, Mae provides a number of analyses. First, a developer is able to request the set of currently undefined properties, which are in effect, the set of decisions that still need to be made before a single architectural configuration can be selected. Second, a developer can use Mae to analyze the properties and determine any conflicting properties. In the example of Section 3, one of the *SpellChecker* variants requires the inclusion of the *SpellCheckRepository* component and, as such, this component specifies the property “includeSpellCheckRepository” to be *true*. If a developer, on the other hand, selects the value of this property to *false*, effectively overwriting the desired value, Mae warns the developer that this selection may lead to problems.

After a particular version of the architecture has been selected, Mae generates the C2SADEL [22] specification for the selection. To do so, Mae maps the generic concepts of the model (e.g., representation, behavior, and constraints) onto specific language constructs in C2SADEL. The functions displayed in the bottom window of figure 3 become available for various purposes. First, the generated C2SADEL specification can be parsed and checked

for adherence to the topological rules of the C2 style.² Then, analysis tools can be used to further verify the semantic correctness of the C2SADEL specification. Specifically, type checking can be used for two kinds of analyses: (1) given a specification of an architecture, Mae can analyze each component to ensure that the interfaces and behaviors it requires are satisfied by the components along its communication links [41], and (2) given a set of component specifications, Mae can analyze whether their specified interface and/or behavior subtyping relationships hold [18].

In addition to verifying C2SADEL specifications, Mae inherits a number of facilities from DRADEL that can actively assist architects and software developers. First, an architect can use the compliance checker to uncover whether any subtyping relationships among arbitrary components exist (see Section 6). Second, software developers can generate, from the C2SADEL specification, a partial Java implementation of the system. Specifically, Mae generates templates for the components that make up the system, reuses standard implementations for the connectors, and automatically generates the component that represents and instantiates the system configuration. In doing so, Mae uses a version of the C2 implementation and execution framework [25].

Our prototype implementation of Mae is not yet complete as of yet. The *integration* between the two components (DRADEL and M?nage) can be made much tighter such that, for example, type checking occurs automatically and instantaneously rather than “after the fact.” Moreover, we intend to significantly extend the number of *analyses* provided by Mae, since the particular combination of features captured in the system model requires the creation of new analyses that relate and explore information not commonly available before (one such analysis is discussed in the next section). It is important to observe that Mae already provides complete support for *capturing* all aspects of the architectural system model discussed in the previous section. As such, we have used Mae, e.g., to capture the evolution of our example word processor system, and have been able to use Mae’s system model as the basis of the research directions introduced in the next section.

6. DEMONSTRATIONS

Mae and its underlying system model were devised for a single purpose only: to manage architectural evolution. However, as our research progressed, we recognized that the particular combination of architectural and CM information captured in our system model opens up a number of new opportunities in the field of software architecture. For example, the integrated nature of the system model affords the introduction of enhanced support for architectural design, the creation and use of multi-versioning connectors, and the automatic derivation of architectural change scripts. Although our research in each of these areas is preliminary, we include their discussion here as a further demonstration of the usefulness of Mae’s system model.

² Although Mae’s system model supports the specification of different styles, the reuse of DRADEL in our prototype currently limits the verification of styles to C2 only.

Enhanced Design Experience

Existing CM systems and architectural design environments do not provide functionality to actively support their users in choosing appropriate versions of the artifacts to be incorporated in a configuration. Consider, for example, a developer who has to replace a faulty new version of a component with an older version. Usually, the overall configuration has evolved in parallel with individual components and connectors and 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. If the tests fail, the developer checks out yet another version of the component and repeats the process until eventually (and hopefully) a suitable version is found. In a typical architectural design environment even this process cannot be supported: because versioning information is not captured in these environments, the search normally relies on the memory of the developer and the sporadic commentary that may have been captured to select appropriate components as candidates to replace the faulty component.

Compared to most CM systems, however, architectural design environments do have available much auxiliary information that is usually reserved for analysis purposes. For example, once a particular architectural configuration has been created, it can be analyzed for behavioral consistency. These kinds of analyses have thus far been performed *after the fact*.³

One of the opportunities we are exploring with Mae is that the particular combination of information that is captured in its architectural system model is well suited for analyses of particular system model actions *beforehand*. In particular, we are exploring the ways in which Mae can enhance the design process by allowing designers to ask “questions” regarding the architecture. One such question leverages Mae’s integration of versioning and subtyping information: developers can ask Mae to list those versions in the version history of a component that exhibit a particular subtyping relationship. This kind of inquiry helps in the replacement problem described above: Mae is able to suggest candidate versions of components to replace an existing version. For example, a developer can ask Mae for those versions that are behaviorally consistent with the existing version—in effect asking for those versions that are guaranteed to have no adverse influence on the remainder of the architecture.

We have implemented this idea in Mae. Specifically, Mae traverses the ascendant and descendant relations of a component and determines the kind of subtype relation that each of the traversed versions exhibits with respect to the original version of the component. In doing so, it not only traverses the revision history, but also follows branches “backwards” and “forwards.” For each visited component version, Mae determines the subtyping relationship. As illustrated in Figure 4, Mae then displays the resulting list of subtype relations. This significantly narrows the search space for a suitable component version. In the example shown, none of the other versions of the *tSpellChecker* component exhib-

its the desired subtyping relationship (preservation of interface *and* behavior) with version 3, but in a larger version tree several versions are likely to be behavior or interface compliant with the component version to be replaced.



Figure 4. Mae’s status window

This small enhancement to Mae represents only our first attempt at leveraging Mae’s system model to enhance the design experience. We plan to develop and implement other analyses that help a developer in decision making. Example queries will allow a developer to ask such questions as “Which component version preserves the desired architectural style?” or “Which components can be suitably used as variants while being behaviorally consistent?” Rather than being a passive environment that only stores the effects of changes, we plan to further grow Mae into an environment that uses advanced architectural analyses to actively participate in ensuring correct architectural evolution.

Multi-Versioning Connectors

We have already discussed how consistent evolution via subtyping is ensured by Mae at the architectural level. However, there is no guarantee that *implemented* components will preserve the properties and relationships established at the architectural level. As discussed in Sections 4 and 5, Mae’s system model can be leveraged to address this problem. Specifically, the model provides the following three capabilities: (1) an infrastructure that allows implementation and execution of architectures; (2) the ability to generate a (partial) implementation of an architecture from its ADL model; and (3) the ability to associate implemented modules (e.g., Java classes and interfaces) with the elements in Mae’s architectural system model (e.g., components, connectors, interfaces, and configurations).

In this section we focus on a novel capability enabled by Mae that directly leverages its implementation/execution infrastructure. We have enhanced that infrastructure with special-purpose software connectors intended to aid component testing and reliable component upgrades. These connectors, called *multi-versioning connectors (MVC)*,⁴ allow any component in a system to be replaced with a set of its versions (variants) that will execute in parallel. This capability was inspired by the approach for reliable component upgrade suggested by Cook and Dage [8]. Their approach treats individual procedures as components, allows multiple such procedures to be executed simultaneously, and provides a means for comparing their execution results. We have realized that a similar capability is needed at the level of coarser-grained components to

³ Argo’s design critics [30] are an exception since they perform analyses *while* an architecture is being designed.

⁴ This usage of the *MVC* acronym is entirely unrelated to the Model-View-Controller approach to constructing Smalltalk applications [13].

aid architectural evolution. At the same time, the increased granularity of the involved components has resulted in several challenges that Cook and Dage never faced, but that we had to overcome in our implementation of MVC [29].

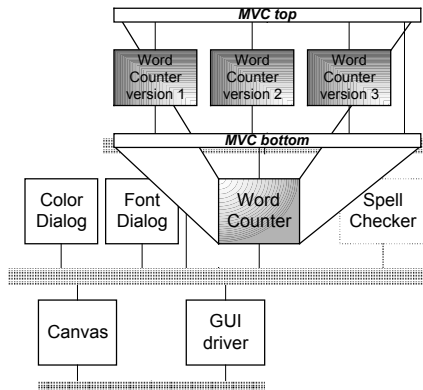


Figure 5. Multi-Versioning Connectors.

To enable architecture-level “multi-versioning” of components, such component versions are “wrapped” by a pair of MVCs as shown in Figure 5. MVCs insulate the rest of an application from the fact that a given component is multi-versioned. The role of an MVC is to invoke all component versions for each message it receives from below (*MVC bottom*) or above (*MVC top*), and to propagate to the rest of the system any messages created by the multi-versioned components. Each MVC propagates messages from only one of the multi-versioned components, designated as *authoritative* (i.e., nominally correct) with respect to the invoked operation. At the same time, each MVC logs the results of all the multi-versioned components’ invocations and compares them to the results produced by the authoritative version. Additionally, MVCs perform comparisons of the multi-versioned components’ performance (i.e., execution speed), relative correctness (i.e., whether they are producing the same results as the authoritative version) and reliability (i.e., number of failures during an execution). MVCs allow component authority for a given operation to be changed at any point in time.

MVCs allow the insertion of new component versions into a system during runtime (e.g., versions 2 and 3 of *Word Counter* in Figure 5), without removing the old version. MVCs also allow multiple components that provide complementary functionality to be used in concert to accomplish the desired functionality of a *single* component. Finally, MVCs leverage the ability of the implementation infrastructure to log execution history in order to undo runtime changes to an application. MVCs support two kinds of architecture-level undo: restoring a multi-versioned component’s state to any point in the past and reversing the insertion and removal of component versions.

Our support for MVCs directly leverages Mae’s system model, which relates component versions at the *architectural* level, to ensure the preservation of these relationships in the *implemented* components. For example, if a subtyping relationship is defined between two versions of a component, MVC can be used to ensure that the implemented versions indeed preserve the specified relationship. Mae’s versioning information can aid this task; e.g.,

by ensuring that the tested versions belong to the same version tree.

Automated Change Script Generation

Explicit software architecture descriptions have already been used to assist in the management of dynamic, run-time change. One particularly promising approach uses extension wizards to dynamically update Java systems [25]. These wizards can be considered architecture-level patches: they contain a series of differences between an actual architecture and a new, desired version of the same architecture. A typical wizard is divided into two separate parts: one part contains the logical “recipe” describing the modifications to be made in terms of architectural operations, while the other part contains the Java class files that are to be added to the system. A trivial example of an extension wizard is provided by the following logic to dynamically add the optional *iSpellChecker* component to an instance of the word processor:

```
Add iSpellChecker
Weld iSpellChecker.top to Conn1.bottom
Weld iSpellChecker.bottom to Conn2.top
Start iSpellChecker
```

First, the component is added to the architecture, then it is properly connected to the remainder of the architecture, and, finally, it is activated. Other primitives are available to replace and remove components (and connectors).

One drawback of the current approach to extension wizards is that they have to be created by hand, without any automated assistance. In the case of large software architectures and many differences between versions, this can be a very cumbersome and error-prone task. Mae provides an opportunity to alleviate this problem: its system model contains all the information necessary to automate *architectural differencing*. Specifically, Mae integrates the architectural entities that change with the versioning information that describes the changes. It is this integration that facilitates the automation.

We have developed and are currently implementing a differencing algorithm that, given two versions of an architecture, automatically generates the logical part of an extension wizard. Thanks to Mae’s integrated architectural system model, the algorithm itself is fairly simple: it fully exploits both the architectural and CM information in performing a comparative hierarchical traversal of the composition of two versions of an architecture. In doing so, it follows the standard practice adhered to in the differencing techniques used in the field of CM [4].

The use of the integrated system model also ensures a degree of confidence in a change, not present if the information comes from different sources. For example, a CM system used to manage different versions of an architectural description may not be able to guarantee that component *foo* in version 1 is the same as component *foo* in version 2. Mae’s system model naturally provides this guarantee with the information that it captures.

7. RELATED WORK

Few approaches have combined architecture and CM concepts in addressing architectural evolution. Two notable exceptions are UniCon [34] and Koala [39]. 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 and options. Moreover, the primary focus of UniCon is on implementation-level variability, not on variability at the level of the definitions of architectural elements.

Koala is closest to Mae in the advanced modeling facilities that it provides for capturing product family architectures. 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 a product family architecture. 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, behavior and constraint specifications, and styles. Mae extensively uses this kind of information in providing the functionality described in Sections 5 and 6.

Several other approaches in the fields of CM and software architecture laid the foundation for the work presented in this paper. In the field of CM, PROTEUS introduced a system model in which components were explicitly recognized [37]. Adele introduced interfaces to be used in verifying the consistency of selected configurations [9]. Finally, Inscape used obligations, 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 [27].

In the field of software architecture, several approaches have extensively used explicit software connectors; prominent examples are Wright [1], UniCon [34], and C2 [35]. Rapide [19] uses an OO-like inheritance mechanism to support component evolution, while Acme [11] supports structural subtyping. DRADEL [22] was the first approach to introduce behavioral subtyping [18,41] into an ADL. Finally, GenVoca [2] and Aesop [10] were early examples of approaches that used styles as a means of guiding and controlling architectural evolution.

Our approach leverages and tightly integrates all these contributions to provide a novel environment in which the problem of architectural evolution is managed in a meaningful, useful, and complete fashion.

8. CONCLUSIONS

This paper has presented a novel approach to managing architectural evolution. The essence of the approach lies in the use of a generic system model that integrates CM concepts, such as revisions, variants, and configurations, with architectural concepts, such as components, connectors, subtypes, and styles. By mapping the generic system model onto a specific ADL, many different analyses become available that can be adapted for the purpose of maintaining the consistency of the architectural configurations captured by the model.

With the advent of our system model, 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. Our evolutionary design environment, Mae, and its novel capabilities discussed in Sec-

tion 6 represent only the starting point: even though each capability is certainly useful at this moment, none is complete as of yet. We continue to explore more advanced functionality by further enhancing our prototypes. Specifically, we intend to address the following three issues in the near future: (1) extending Mae with additional design-time functionality by providing additional types of analyses and advanced change-based versioning support [7]; (2) tightly integrating development-time architectural evolution with the evolution of a deployed system at run-time; 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 configuration management in addressing evolution: these techniques are related and, at times, equivalent. A deeper understanding of their relationship and tradeoffs among them is much needed. We believe that the current Mae prototype forms a solid foundation upon which we can perform all of these investigations.

9. ACKNOWLEDGEMENTS

This material is based upon work supported by the National Science Foundation under Grant No. CCR-9985441. Effort also sponsored by the Defense Advanced Research Projects Agency, Rome Laboratory, Air Force Materiel Command, USAF under agreement numbers F30602-99-C-0174 and F30602-00-2-0599. 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.

REFERENCES

1. Allen R., and Garlan D., A Formal Basis for Architecture Connection. *ACM Transactions on Software Engineering and Methodology*, 6(3): p.213-249, 1997.
2. Batory D., and O'Malley S., The Design and Implementation of Hierarchical Software Systems with Reusable Components. *ACM Transactions on Software Engineering and Methodology*, 1(4), October 1992.
3. Bell Labs Lucent Technologies, Sablime v5.0 User's Reference Manual: Murray Hill, New Jersey, 1997.
4. Buffenbarger J, Syntactic software merging, In *Software Conguration Management: ICSE SCM-4 and SCM-5*, Springer-Verlag. pp.153-172, 1995
5. Burrows C., and Wesley I., Ovum Evaluates Configuration Management, Burlington, Massachusetts: Ovum Ltd., 1998.
6. Compare D., Inverardi P., and Wolf A.L., Uncovering Architectural Mismatch in Component Behavior. *Science of Computer Programming*, 33(2): pp.101-131, 1999.
7. Conradi R., and Westfechtel B., Version Models for Software Configuration Management. *ACM Computing Surveys*, 30(2): pp.232-282, 1998.
8. Cook J.E., and Dage J.A., Highly Reliable Upgrading of Components, in *Proceedings of the 1999 International Conference on Software Engineering*, pp.203-212, 1999.
9. Estublier, J. and Casalles, R., The Adele Configuration Manager, in *Configuration Management*, W.F. Tichy, Editor, Wiley: London, Great Britain. pp.99-134, 1994.

10. Garlan D., Allen R., and Ockerbloom J., Exploiting Style in Architectural Design Environments in *Proceedings of SIGSOFT'94: Foundations of Software Engineering*, pp.175–188, New Orleans, Louisiana, USA, December 1994.
11. Garlan D., Monroe R., and Wile D., ACME: An Architecture Description Interchange Language in *Proceedings of CASCON'97*, November 1997.
12. Hall, R.S., Heimbigner, D.M., and Wolf, A.L., A Cooperative Approach to Support Software Deployment Using the Software Dock, in *Proceedings of the 1999 International Conference on Software Engineering*, pp.174-183, 1999.
13. Krasner G. E., and Pope S. T, A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*, 1(3):26–49, August/September 1988.
14. Kuusela, J., Architectural Evolution, in *Proceedings of the First Working IFIP Conference on Software Architecture*, Kluwer Academic: Boston, Massachusetts. 1999.
15. Lampen, A. and Mahler, A., An Object Base for Attributed Software Objects, in *Proceedings of the EUUG Autumn'88 Conference*: Cascais, Portugal. pp.95-105, 1988.
16. Larsson, M. and Crnkovic, I., New Challenges for Configuration Management, in *Proceedings of the Ninth International Symposium on System Configuration Management*. pp.232-243, 1999.
17. Le Metayer D., Software Architecture Styles as Graph Grammars, in *Proceedings of FSE4*, San Francisco, p.15-23, October 1996.
18. Liskov B. H., and Wing J. M., A Behavioral Notion of Subtyping. *ACM Transactions on Programming Languages and Systems*, vol. 16, no. 6, pp. 1811-1841, November 1994.
19. Luckham D. C., and Vera J., An Event-Based Architecture Definition Language. *IEEE Transactions on Software Engineering*, vol. 21, no. 9, pp. 717-734, September 1995.
20. Magee J., and Kramer J., Dynamic Structure in Software Architectures, in *Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp.3-13, 1996.
21. Medvidovic, N., Rosenblum, D. S., and Taylor, R. N. A Type Theory for Software Architectures. Technical Report, UCI-ICS-98-14, University of California, Irvine, April 1998.
22. Medvidovic N., Rosenblum D. S., and Taylor R. N., A Language and Environment for Architecture-Based Software Development and Evolution, in *Proceedings of the 1999 International Conference on Software Engineering*, pp.44-53, 1999.
23. Medvidovic N., and Taylor R.N., A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering* 26(1), pp. 70–93, January 2000.
24. Mehta N., Medvidovic N., and Phadke S., Towards a Taxonomy of Software Connector, in *Proceedings of the 22nd International Conference on Software Engineering (ICSE 2000)*, pp. 178–187, Limerick, Ireland, June 2000.
25. Oreizy P., Medvidovic N., and Taylor R. N., Architecture-Based Runtime Software Evolution in *Proceedings of the 20th International Conference on Software Engineering*, pp.177-186, Kyoto, Japan, April 1998.
26. Parisi F., and Wolf A.L., Foundations for Software Configuration Management Policies Using Graph Transformations, in *Fundamental Approaches to Software Engineering 2000*, Springer-Verlag. pp. 304-318, 2000.
27. Perry D.E., The Inscope Environment, in *Proceedings of the Eleventh International Conference on Software Engineering*, pp. 2-11, 1989.
28. Perry D.E., and Wolf A.L. Foundations for the Study of Software Architectures. *ACM SIGSOFT Software Engineering Notes*, vol. 17, no. 4, pp. 40-52, October 1992.
29. Rakic M., and Medvidovic N., Increasing the Confidence in Off-the-Shelf Components: A Software Connector-Based Approach. To appear in *Proceedings of the 2001 Symposium on Software Reusability*, Toronto, Canada, May 2001.
30. Robbins J., Redmiles D., Software Architecture Critics in the Argo Design Environment. *Knowledge-Based Systems*. Special issue: The Best of IUI'98.
31. Rochkind M.J., The Source Code Control System. *IEEE Transactions on Software Engineering*, SE-1(4): 1975.
32. Seiwald C., Inter-file Branching - A Practical Method for Representing Variants, in *Proceedings of the Sixth International Workshop on Software Configuration Management*, Springer-Verlag. pp. 67-75, 1996.
33. Shaw M., and Garlan, D., *Software Architecture: Perspectives on an Emerging Discipline*: Prentice Hall, 1996.
34. Shaw, M., et al., Abstractions for Software Architecture and Tools to Support Them. *IEEE Transactions on Software Engineering*, 21(4): pp. 314-335. 1995
35. Taylor R.N., et al., A Component- and Message-Based Architectural Style for GUI Software. *IEEE-TSE*. 22(6), 1996.
36. Tichy W.F., RCS, A System for Version Control. *Software - Practice and Experience*. 15(7): pp. 637-654, 1985.
37. Tryggeseth E., Gulla B., and Conradi R., Modeling Systems with Variability Using the PROTEUS Configuration Language, in *Proceedings of the Fifth International Workshop on Software Configuration Management*, Springer-Verlag. pp. 216-240, 1995.
38. van der Hoek A., A Generic, Reusable Repository for Configuration Management Policy Programming, University of Colorado at Boulder: Boulder, Colorado, 2000.
39. van Ommering R., et al., The Koala Component Model for Product Families in Consumer Electronics Software. *IEEE Computer*, 33(2): pp. 78-85, 2000.
40. Wiborg Weber D., Change Sets versus Change Packages: Comparing Implementations of Change-Based SCM, in *Proceedings of the Seventh International Workshop on Software Configuration Management*, pp. 25-35, 1997.
41. Zaremski A. M. and Wing J. M. Specification Matching of Software Components. *ACM Transactions on Software Engineering and Methodology*, vol. 6, no. 4, pp. 333-369, October 1997.