Capturing Architectural Configurability: Variants, Options, and Evolution

André van der Hoek, Dennis Heimbigner, and Alexander L. Wolf

Software Engineering Research Laboratory
Department of Computer Science
University of Colorado
Boulder, CO 80309 USA

{andre,dennis,alw}@cs.colorado.edu

University of Colorado
Department of Computer Science
Technical Report CU-CS-895-99 December 1999

ABSTRACT

Although meant to be relatively stable, the architecture of a software system does, at times, change. This simple yet important observation immediately raises the question of how changes to an architecture should be captured. Current architecture description languages are not well-suited for this purpose, but existing techniques from the discipline of configuration management can be adapted to provide a solution. In particular, we propose a novel representation, called configurable software architecture*, that extends the traditional notion of software architecture with the concepts of variants, options, and evolution. We discuss the details of the representation, present an environment that allows the specification of configurable software architectures, and highlight a few of the opportunities that we believe arise once architectural configurability can be precisely captured.*

This work was supported in part by the Air Force Materiel Command, Rome Laboratory, and the Defense Advanced Research Projects Agency under Contract Numbers F30602-94-C-0253 and F30602-98-2-0163. The content of the information does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

1 Introduction

Rarely a software system has been constructed that, once delivered and in use, was never modified. In fact, it is well-known that most successful software systems undergo a large number of changes over time. Managing all these changes has almost exclusively been the domain of the discipline of configuration management for quite some time now. In particular, advanced techniques have been developed to address (among others) the following two problems: capturing the ever constant state of flux that the source code of a software system appears to be in, and selecting a particular system configuration out of the—potentially very large—set of available source code. In essence, these techniques can be characterized as managing the *configurability* of a software system.

The mere existence of configurability can basically be attributed to three phenomena, namely variability, optionality, and evolution. Although these three phenomena are closely related, it should be understood that they represent three very distinct concepts. *Variability* refers to the fact that a single software system can provide multiple, alternative ways of achieving the same functionality. As an example, consider a numerical optimization system [6] that has been engineered to operate either slow but very precise, or fast but only approximate. Depending on the desired mode of operation, the selection of source files included in the actual system configuration can be very different. *Optionality* means that a software system has one or more additional parts that each may or may not be incorporated in the system. For example, consider the fact that the numerical system can optionally gather statistics. The source files that implement the gathering and analyzing of the statistics at run-time are only included in the system configuration if the option to gather statistics is turned on. Finally, the term *evolution* is used to capture the notion that a software system changes over time to provide a related yet different set of capabilities. Once again, the numerical system provides an example in that it evolved from initially being capable of only optimizing rather simple functions to now optimizing large and very complex functions. Obviously, the system configuration has changed over time as the capabilities of the system have increased.

With the emergence of the discipline of software architecture, a new challenge has arrived with respect to managing configurability. It has been proposed to use the architecture of a software system to support various activities in the software life cycle. In fact, some infrastructure has already been developed to support this vision. In particular, a source code editor that is explicitly based on architecture [5], a tool specifically designed to test architectures [30], and a number of architecture-based activation systems [17, 22] have been constructed. With this kind of use of architectures, though, the architectural structure of a software system becomes as important as its source code; both are delivered and deployed. Moreover, both exist in multiple variants, both exhibit optionality, and both evolve over time. Therefore, the need arises to manage configurability in architectures analogous to the way configurability in source code is being managed [36]. Herein lies the aforementioned challenge: to investigate and develop techniques that are capable of managing and supporting configurability at the architectural level. A key observation to be made is that this challenge requires the management of the configurability of architectures themselves, not of the documents that describe the architecture. In fact, it is not sufficient to simply version an architectural description. Instead, to support some of the activities listed above, configurability has to be modeled inside an architecture itself.

Some aspects of architectural configurability are starting to be addressed. For example, changes at the architectural level have been used to dynamically reconfigure a software system at run time [13, 22]. As another example, it has been recognized that the differences between architectural configurations can be used as a basis for regression testing [25]. However, two problems persist in the work carried out so far.

- *Although certainly as important as evolution, neither variability nor optionality is supported.*
- *A suitable representation that captures architectural configurations and relates them to each other is lacking.*

This paper proposes a solution for these two problems by leveraging the modeling capabilities that have been developed in the field of configuration management. In particular, we propose a novel representation, called *configurable software architecture*, that integrates the traditional notion of software architecture with the concepts of variants, options, and evolution. Although this work is preliminary and we have not yet tested the use of the representation, we believe that the definition of the representation is a first step towards the structured use of architecture-level configurability in support of various software life cycle activities.

The remainder of this paper is organized as follows. We first, in Section 2, discuss a number of myths with respect to architectural configurability that need to be addressed before we introduce our representation in Section 3. We then discuss, in Section 4, an environment that we have created to support the specification of configurable software architectures. Subsequently, in Section 5, we present some of the potential uses of our representation. We discuss related work in Section 6 and conclude in Section 7 with a summary of our contributions and an outlook at future work.

2 Myths

Before we discuss our representation for configurable software architecture, we need to discuss a number of myths with respect to architectural configurability. Below, we introduce these myths and present a short rebuttal for each one.

Architectures do not change. The architecture of a software system is indeed meant to be relatively stable. However, during the life of a software system it is likely that changes occur that do have an impact on its architecture. Perhaps the most convincing argument is provided by Perry and Stieg, who showed that approximately 50 percent of all system changes are interface changes [23]. Given that a typical architectural description captures interfaces, we can, thus, expect an architecture to change.

Architectural configurability can be expressed in existing architecture description languages. It is certainly possible to capture aspects of configurability in some existing architecture description languages (ADLs). In fact, certain ADLs, such as Rapide [15] or Darwin [16], are even powerful enough to explicitly program configurability. However, ADLs lack specific constructs for this purpose. In the face of a large number of variants, options, and evolutionary changes, this quickly leads to a complicated and chaotic architectural description that is rather difficult to understand.

Managing architectural configurability is simply applying a configuration management system to an architectural description. At first sight, the use of a configuration management tool (e.g., ClearCase [3], CVS [4], or Continuus [7]) to store and version architectural descriptions is an attractive and simple solution to capturing architectural configurability. However, if a configuration management system were to be used, the configurability aspects of an architecture would be stored as metadata separate from the architecture. Because the data is not an integral part of the architecture, such a solution prohibits the use of configuration data by the architecture itself. In particular, this becomes a problem in the case of dynamic, self-adaptive architectures, because these need the configuration data to govern the changes they undergo. Moreover, the creation of multi-version architectures, such as, for example, those required in Hercules [8] or the Simplex method [27], is not supported by this type of solution.

Managing architectural configurability is simply adding a version number to the entities in an architectural description. Adding version numbers to architectural entities is certainly a step towards capturing architectural configurability. However, a complete solution is more complicated. A version number is simply an identification mechanism that leaves the semantics of versioning (and versioning relationships) implicit. The traditional version tree is one way of capturing some of these semantics, but architecture has its own peculiarities that require the use of a different mechanism. Optionality, for example, is not supported by the version tree. In addition, the variant relationship needs to be extended because, in an architecture, independently developed components can be variants of each other. Thus, we require a more advanced solution than just a simple addition of a version number.

3 Configurable Software Architecture

As the name "configurable software architecture" already indicates, our representation for capturing architectural configurability is based on a number of concepts adopted from the discipline of configuration management. However, as discussed in the previous section, a straight adoption is not necessarily our best option; the unique aspects of software architecture as compared to source code require us to apply some of the traditional configuration management techniques in a rather different way. In this section, we discuss these aspects and their influences on the design of our representation for configurable software architecture. Before we do so, however, we first introduce a number of general principles that guided the design and then introduce an example system that we use throughout the remainder of the discussion.

3.1 Design Principles

Our representation for configurable software architecture has been influenced by a number of design principles. In the following, we enumerate these principles and discuss their impact on the representation.

- *Completeness.* Our representation needs to be able to capture all types of configurability to all architectural elements. Although a trivial observation, it influences our design considerably. In particular, it has been observed that the need to create a representation that involves hierarchies (an integral part of architectural specifications), variants, and evolution already brings forth a rather complicated design problem that is multi-dimensional in nature [33]. The addition of options further complicates this problem.

- *Simplicity.* Because we need to capture a large number of concepts in a single representation, a focus on the essential aspects to be modeled is an absolute necessity. If too many concepts are addressed via advanced modeling techniques, the representation becomes too complicated to use. In that case, it is likely that only a subset is used and that the full power of the representation is not taken advantage of.

- *Orthogonality.* The discipline of configuration management has traditionally addressed a number of different concepts through a single formalism: the version tree. In [10] it is argued that this is not a proper approach; instead, each concept should be handled in a separate manner. In the design of our representation for configurable software architecture we follow this advice and treat every concept orthogonally.

- *Language independence.* A large variety of architecture description languages exists [21]. We, obviously, would not like to tie our representation to a single language. Instead, our representation should make it possible to capture configurability for a multitude of ADLs. As demonstrated by Acme [11], this requires a careful design that reckons with the peculiarities of the various ADLs.

3.2 Example

Figure 1 presents a simplified version of an existing system that is currently in use to carry out research in the field of numerical analysis [6]. The purpose of the system is to globally optimize a mathematical function, i.e., to find the point in the domain of the function that yields the absolute lowest function value. The system consists of about 15,000 lines of Fortran and C code, and is modularized into a set of components. In the figure, each solid box represents such a component and each solid line indicates the existence of interaction between two components. For example, each `Optimizer` component interacts with a single `ComplexFunction` component. The dashed lines indicate a different kind of relationship among components, instantiation. As illustrated by the dashed boxes, the `Scheduler` component instantiates new `Optimizer` and `ComplexFunction` components in pairs.

In the system, the `GlobalOptimization` component manages the computation that takes place. It uses the `Scheduler` to create new `Optimizer` and `ComplexFunction` components, and allocates a particular interval of the domain to each `Optimizer` component. The `Optimizer` component carries out an optimization algorithm on the interval that it has been allocated, and uses its `ComplexFunction` component to evaluate the function at the particular points that the algorithm requires. The net effect is that the function is optimized by optimizing multiple intervals in parallel.

Throughout its lifetime, the system has been highly variable. Initially, the `ComplexFunction` component consisted of about 3,000 lines of Fortran code that were created at the local site, but it has since been replaced with a separate system, CHARMM, that was created at an external site. Also, alternative `Optimization` components exist that each exhibit unique characteristics with respect to the encapsulated optimization algorithm; some are fast but produce less precise results, whereas others are slow but very accurate. Moreover, the system is configurable to optionally generate statistics about its performance. Finally, new versions of the `GlobalOptimization` component are created on a regular basis as new approaches are being tried to find better results.

3.3 Basic Architectural Skeleton

Figure 2 shows the basic architectural skeleton upon which we have constructed our representation for configurable software architecture. Although syntactically different, the skeleton is largely patterned after the semantics that were introduced by Acme [11] and Darwin [16]. It contains the common architectural concepts of interfaces, components, and connections. Although the skeleton is similar to some of the system models provided in traditional configuration management systems (e.g., Adele [9], DSEE [14], and Jasmine [19]), some important differences exist. Since

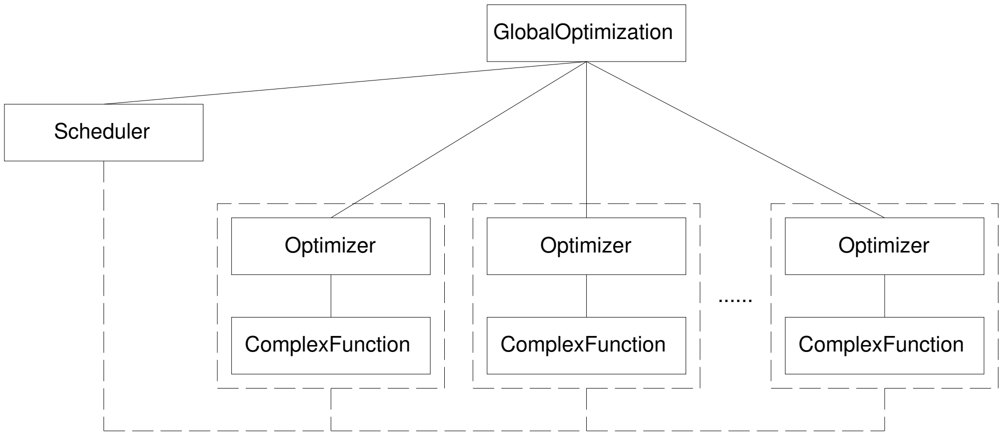

Figure 1: Example System.

these differences have been further elaborated upon in the existing software architecture literature [12, 24, 29, 34], we only briefly mention the core differences here: besides components and their interfaces, connections are explicitly modeled as well; the behavior and constraints of components and connections are explicitly modeled; both components and connections can have multiple interfaces; and interfaces are directed.

Before we move on to our discussion on configurability, several words are in place about the skeleton representation. First, we note that the representation is based on a type-instance mechanism. However, because instances define additional fields besides the ones defined in their types, they also exhibit an inheritance relationship with respect to their types. Thus, our representation does not reflect a type-instance system in the truest sense of the word. To indicate this difference, we use the term *exemplar* as opposed to the term *type*.

The second observation we need to make is that, because of language independence reasons, connections are not required to have interfaces. Some ADLs have what can be considered anonymous connections whose sole purpose is to connect components via component interfaces (consider the buses used in C2 [31] or the use of the `bind` statement in Darwin [16]). Other ADLs raise the level of connections to what are called connectors [2, 28]. In those ADLs, a connector has its own interfaces and connects components by binding the interfaces of the components to its own interfaces. To support both kinds of ADLs, the use of interfaces in connections is optional in our representation.

Two additional observations need to be made about the skeleton representation. The first is that the hierarchical composition of architectures is supported through the construction of exemplars out of instances of other exemplars. That is, an exemplar is defined as a set of interfaces that it exposes, a set of components that it consists of, and a set of connections that (internally) connect the contained components and the exposed interfaces. The second observation to be made is that, even though connection exemplars are semantically different from component exemplars, they are syntactically identical. This follows the traditional notion that connections are nothing but components, but need to be recognized as separate entities.

To illustrate the basic skeleton, Figure 3 shows a partial specification of the global optimization system. We first notice the exemplar `Point`, which defines an interface exemplar that represents a geographical location. As described previously, the representation of an exemplar is ADL-specific.

Figure 2: Basic Software Architecture Representation.

In this particular case, we have chosen Darwin as the language to use. An instance of the exemplar `Point` is given by the interface `result`, which is an `in` interface, indicating it consumes a result that is produced by some other component.

Hierarchical composition is demonstrated by the component exemplar `GlobalOptimization`. It contains the interface `result`, as well as two components, `opt` and `eval`, and two connections, `bus1` and `pipe1`. For brevity, we do not show the definitions of these components and connections, but only note that they are defined in terms of other exemplars. The only instance we do describe, though, is `bus1`. This connection illustrates how the instances contained by an exemplar can be connected. In this particular case, the connection connects the interface `result` of the exemplar `GlobalOptimization` to the interface `minimum` of the component `opt`. The behavior, constraints, and representation of the exemplar `GlobalOptimization` are, once again, language dependent and not further defined here.

InterfaceExemplar
name = Point;
representation = {
integer x;
}

Interface
name = result;
direction = in;
interfaceExemplar = Point;

ComponentExemplar
name = GlobalOptimization;
interfaces = result;
components = opt, eval;
connections = bus1, pipe1;

Connection
name = bus1;
sourceInterfaces = minimum;
destinationInterfaces = result;
connectionExemplar = Bus;

Figure 3: Basic Software Architecture Representation Example.

3.4 Representation

Figure 4 shows our representation for configurable software architecture as an extension to the basic architectural skeleton that we introduced in Figure 2. Below, we discuss one by one how this extended representation addresses the problems of capturing evolution, optionality, and variability.

3.4.1 Evolution

Our first step towards building a representation for configurable software architecture is the addition of a revision number to each of the exemplars. To capture the evolution of an exemplar, the revision numbers of subsequent editions of the exemplar are related to each other in a version tree. This scheme follows the practice of many standard configuration management solutions. However, our solution deviates in two important ways. First of all, we note the intentional use of the term *revision number* as opposed to the more conventional term *version number*. In traditional CM systems, the version tree captured both evolution and variability. In our representation, we separate these two and only support evolution through the version tree. In essence, we expect the version tree to evolve linearly. To indicate this expectation, we use the more specific term *revision number*. Of course, branches are still allowed in the version tree, but only to represent parallel work. We expect such parallel work to be temporarily and eventually be merged in with the main line of development.

The second deviation also concerns branches. A different kind of branch from the branch for parallel work is the one where a separate line of development starts (e.g., a new baseline). Traditionally, such a branch is labeled in the version tree as a special branch by using some kind of attribute mechanism. In our representation, we separate out this kind of branching and adopt the solution used in Perforce, namely inter-file branching [26]. When a new baseline is created for an exemplar, the exemplar is copied to a new name space with a new version tree. Thus, the new baseline can evolve separately from the old baseline. Of course, we track the ascendant/descendant relationship among the old and new baselines in our representation.

Once exemplars evolve, the question arises as to how particular revisions of instances are selected during hierarchical composition. In our representation we have chosen a rather simple scheme: instances are instances of particular revisions of exemplars, and exemplars are hierarchically constructed based on these instances. This implies that exemplars are constructed in terms of specific revisions. Although more complicated schemes based on advanced selection rules are

Component Exemplar	Variant Component Exemplar
name revision {interface [, optionalPropertyName, optionalPropertyValue]}* {component [, optionalPropertyName, optionalPropertyValue]}* {connection [, optionalPropertyName, optionalPropertyValue]}* behavior constraints representation {propertyName, propertyValue}* ascendant {descendant}*	name revision {interface [, optionalPropertyName, optionalPropertyValue]}* variantPropertyName {component, variantPropertyValue}* representation {propertyName, propertyValue}* ascendant {descendant}*
Connection Exemplar	**Variant Connection Exemplar**
componentExemplar	variantComponentExemplar
Interface Exemplar	
name revision representation ascendant {descendant}*	
Component	**Connection**
name componentExemplar | variantComponentExemplar	name {sourceInterface [, myDestinationInterface]}* {[mySourceInterface,] destinationInterface}* connectionExemplar | variantConnectionExemplar
Interface	
name direction interfaceExemplar	

Figure 4: Configurable Software Architecture Representation.

certainly a possibility, for simplicity we have chosen to not incorporate such a scheme yet.

Figure 5 illustrates how the addition of evolution influences our example. We first note that each exemplar is versioned and identified by a name and a revision number. In the example, we are defining revision 3 of the exemplar `GlobalOptimization`. Additionally, we note that each instance is defined in terms of a specific revision of its exemplar. For example, the interface instance `result` is based on the exemplar `Point` revision 2. The last observation we make is that the exemplar `GlobalOptimization` has a descendant, namely `ComplexGlobalOptimization` revision 1. The descendant indicates the beginning of a separate baseline, that is branched off from revision 3 of the exemplar `GlobalOptimization`.

3.4.2 Optionality

Optionality is an underdeveloped area in both the disciplines of software architecture and configuration management. The sole architecture description language that explicitly supports optionality is Koala [37]. Even so, only the optionality of interfaces can be specified, and the optional primitive does not extend to components or connections. With respect to configuration management, none

```
InterfaceExemplar
  name = Point;
  revision = 2;
  representation = {
    integer x;
  }
```

```
Interface
  name = result
  direction = in
  interfaceExemplar = Point(2);
```

```
ComponentExemplar
  name = GlobalOptimization;
  revision = 3;
  interfaces = result;
  components = opt, eval;
  connections = bus1, pipe1;
  ...
  ascendant = (none);
  descendant =
    ComplexGlobalOptimization(1);
```

```
ComponentExemplar
  name = ComplexGlobalOptimization;
  revision = 1;
  interfaces = result;
  components = complexOpt, eval;
  connections = fastBus1, pipe1;
  ...
  ascendant = GlobalOptimization(3);
  descendant = (none);
```

Figure 5: Architectural Evolution Example.

of the system modeling languages developed to date explicitly supports optionality. The attribute mechanism used for selection purposes can be leveraged to provide a rudimentary way of supporting optionality, but the optionality aspects of the system model are relatively hidden this way.

In our representation, we explicitly model optionality through the use of properties. As shown in Figure 4, each interface, component, or connection instance that is used in the hierarchical construction of an exemplar can be guarded with an optional property name/value pair. The property name specifies the name of the property that the inclusion of an instance depends on, whereas the property value specifies the exact value that the property should have to actually include the instance in the specification.

To determine which options are included in a (partial) system, an exemplar sets the values for the properties that it desires to use. These values are propagated down the hierarchical chain of inclusion to properly select all parts of the system. Currently, an exact match is required between the optional property value and the selected value to include the optional interface, component, or connection. Of course, more powerful logic-based schemes can be devised, but we have not implemented any of these yet.

Figure 6 demonstrates the use of optionality in our representation. We have extended the example of Figure 5 with an optional part that gathers statistics. A component `stat` and a connection `bus2` are now included in the exemplar `GlobalOptimization`, but it should be noted that both are guarded by a property called `statistics` that has to have the value `true` for either to be included. Inside the definition of the exemplar `Optimizer`, the interface `stat` is optional as well, depending on the same property `statistics` as the other optional parts of the architecture. Whether or not the optional parts are included in the architecture depends on the value of the property `statistics`. In this case, the exemplar `GlobalOptimization` defines the property to be `true`, which results in the inclusion of all optional parts.

```
ComponentExemplar
  name = GlobalOptimization;
  revision = 3;
  interfaces = result;
  components = opt, eval,
    stat(statistics == true);
  connections = bus1, pipe1,
    bus2(statistics == true);
  properties = {
    statistics = true;
  }
```

```
ComponentExemplar
  name = Optimizer;
  revision = 1;
  interfaces = stat(statistics == true);
  components = (none);
  connections = (none);
```

Figure 6: Architectural Optionality Example.

3.4.3 Variability

Rather than following an implicit approach in which variability is locally defined inside a component (such as, for example, required in PCL [32]), our representation derives from the approach taken by Adele [9] and explicitly defines variability as a separate entity inside a specification. In particular, we define two new types of exemplars: a variant component exemplar and a variant connection exemplar. Given that, analogous to the way a connection exemplar is syntactically equivalent to a component exemplar, the syntax of a variant connection exemplar is identical to the syntax of a variant component exemplar, we only discuss the specifics of a variant component exemplar here.

Our decision to define a variant component exemplar as a separate entity in our representation stems from the following three requirements.

- The representation must support the ability to define independently developed component exemplars as variants of each other.
- The representation must support the ability to evolve the definition of a variant just like any other regular component exemplar or interface exemplar can evolve.
- The representation must be able to define a variant once and use its definition many times.

To satisfy these requirements, a variant component exemplar is a separate entity that can evolve and contain optional interfaces just like a regular component exemplar. However, its hierarchical containment is defined rather differently. As opposed to being hierarchically constructed in terms of components and connections, a variant component exemplar is defined by a number of components that each have the same interfaces as the variant component exemplar. In addition, the variant component exemplar specifies a property name for which each contained component has an associated property value. Based on the value of the property as defined in a "higher-level" component exemplar, the variant component selects one of the contained components as the one that is selected to be included in the system.

To account for the existence of both component exemplars and variant component exemplars, a component instance is now either an instance of a component exemplar or an instance of a variant component exemplar. Because instances, when used in hierarchical composition, are only connected through interfaces, and both regular and variant component exemplars are defined in terms of their

ComponentExemplar	VariantComponentExemplar
name = GlobalOptimization;	name = Optimizer;
revision = 3;	revision = 1;
interfaces = result;	interfaces = stat(statistics == true);
components = opt, eval, stat(statistics == true);	optionalProperty = speed;
connections = bus1, pipe1, bus2(statistics == true);	components = slowOpt(slow), fastOpt(fast);
properties = { statistics = true; speed = fast; }	

Figure 7: Architectural Variability Example.

interfaces, component instances can be used in the same way regardless of whether they represent a regular or a variant component exemplar.

To illustrate variability in our example, we take a look at the exemplar `Optimizer`. Figure 7 illustrates how this exemplar is defined in terms of two variant components: an instance of the exemplar `FastOptimizer`, called `fastOpt` and an instance of the exemplar `SlowOptimizer`, called `slowOpt`. Selection of one of the two instances depends on the property `speed`, which can have the values `slow` and `fast`. In this particular case, the exemplar `GlobalOptimization` sets the value of the property to `fast`, which through hierarchical propagation causes the `fastOpt` component instance to be included in the architecture.

4 Environment

Maintaining a configurable software architecture by writing specifications by hand is, of course, a difficult and error-prone task. Therefore, we have constructed a simple environment, called Ménage, that supports the graphical construction of configurable software architectures. Shown in Figure 8, the environment is divided into three separate areas. The first area, displayed on the left, contains lists of available component, connection, and interface exemplars. These exemplars have been previously created, and can be used in the hierarchical construction of new exemplars. The largest available area of the environment is used exactly for this purpose; i.e., it allows new exemplars to be specified. These exemplars can be completely new exemplars, or exemplars that are new revisions of existing ones. As an example, the Figure displays the specification of revision 3 of the component exemplar `GlobalOptimization`, which is constructed out of instances of an `Optimizer`, a `FunctionEvalation`, and a `Statistics` exemplar. The interfaces of the components are connected via connections of two kinds: buses and pipes. It should be noted that the component `stat`, the connection `bus2`, and the interface `stats` on the component `opt` are highlighted with a white border to indicate their optionality. Depending on the value of the optional property `statistics`, these instances are included in the architecture or not.

The third area of the environment shows the evolution of the exemplar that is currently being defined. In the version tree displayed at the top of the figure, we see that the exemplar has evolved through three generations. New revisions are created via a traditional check-out/check-in mechanism that is supported by the environment. Change comments are associated with each check-in, and a history of changes can be obtained. The special mark on revision 3 indi-

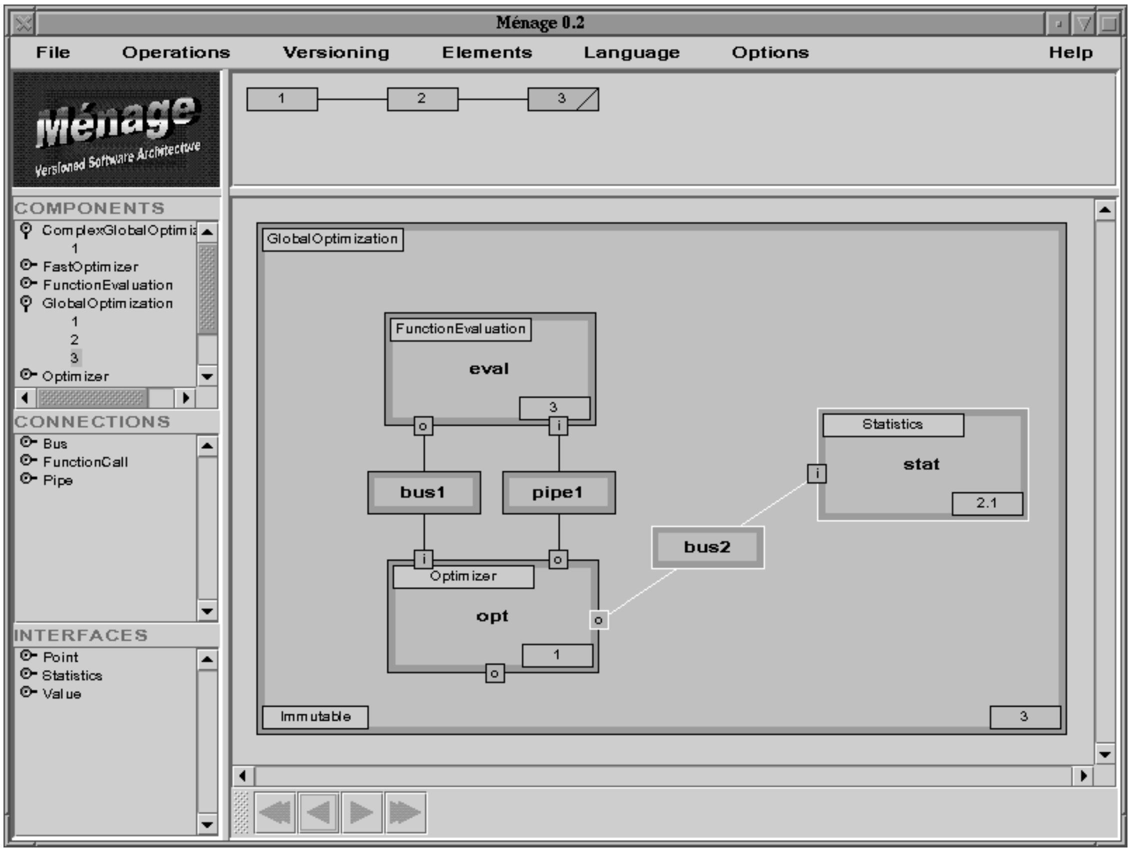

Figure 8: Ménage Configurable Software Architecture Environment.

cates that this revision serves as a branching point for a new baseline . In this case, the exemplar `ComplexGlobalOptimization` evolved out of the (immutable) revision 3 of the exemplar `GlobalOptimization`.

Figure 9 shows how the environment supports variability. A variant component exemplar, `Optimizer`, is being defined in terms of instances of two other component exemplars that each have the exact same interfaces as the `Optimizer` exemplar. The variant property is `speed`, and one of the contained component instances, `slowOpt`, is a slow optimizer (as indicated by the variant property value `slow` that is associated with the component), whereas the other instance, `fastOpt`, is a fast optimizer (as indicated by the variant property value `fast` that is associated with the component). Depending on the value of the property `speed` as given to it by a higher-level exemplar in the architecture, the `Optimizer` selects either the `slowOpt` or the `fastOpt` component to be included in the architecture.

Of course, configurable software architectures that are specified in the environment need to be verified for correctness. Two types of verification processes are supported. The first is incorporated in the environment itself and is able to verify the correct use of properties and variants; conflicting properties in different parts of the architecture can be detected, undefined properties can be uncovered, and improper variant specifications can be discovered. The second verification process is based on a reduction to a specification in a "native" ADL. A configurable software architecture in our specification is reduced to a specification in a particular ADL (such as, for example, C2 or

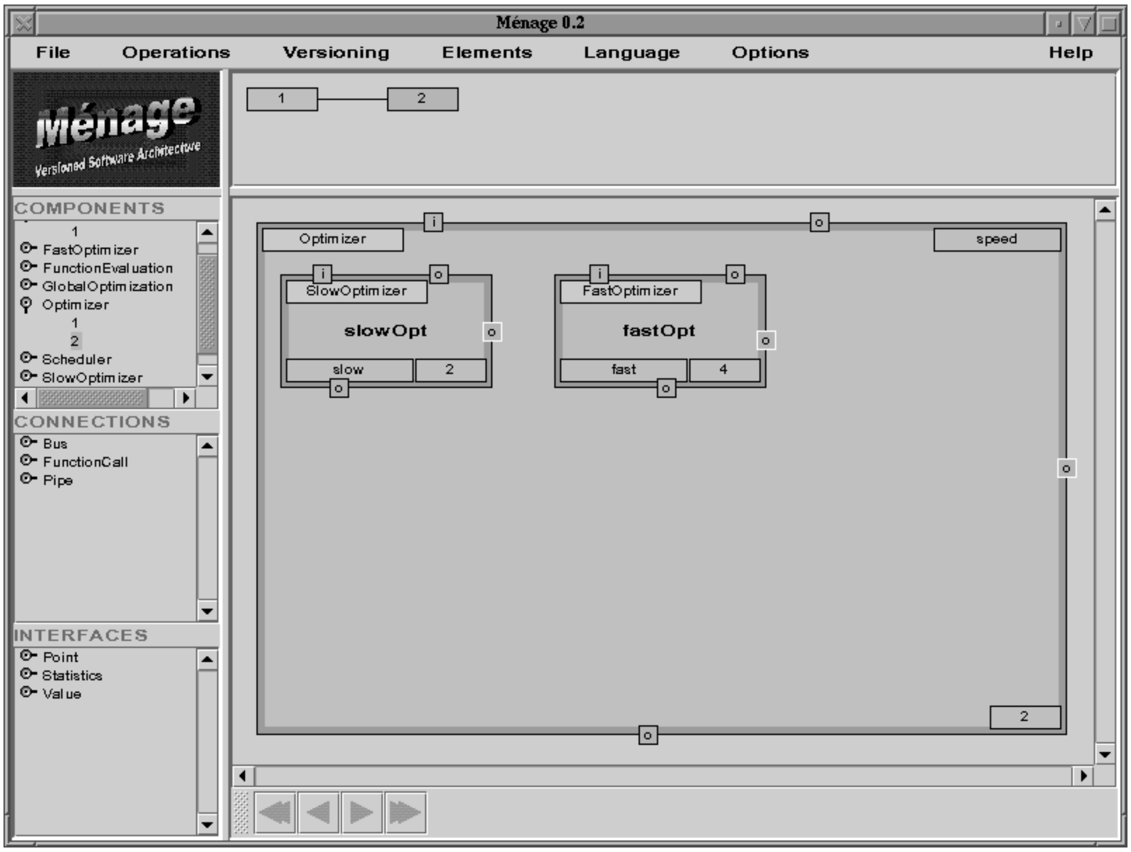

Figure 9: Specifying Variability in Ménage.

Darwin) and the verifier of the particular ADL is invoked to determine whether the specification is sound.

5 Opportunities

As already stated in the introduction, the development of our representation is preliminary in terms of experience. We have not been able to put the representation in actual use just yet, but we do have identified some of the opportunities that we believe can benefit from the availability of a representation for configurable software architecture. In this section, we briefly discuss three of those opportunities.

5.1 Active Patches

One of the activities that is currently supported by some of the architectural approaches is architectural reconfigurability. In C2, for example, wizard scripts contain algorithms to update the architecture of a system at run time [22]. Until now, these wizard scripts had to be created by hand; a designer has to study the differences between the actual architecture and the architecture that is desired. Obviously, this can be a rather laborious and potentially error-prone process. Given a representation for configurable software architecture, the differences between the actual and desired architecture can be calculated using a difference algorithm. It should be noted that the result of the difference algorithm can be rather distinct from the classical output of a difference between two source files. Whereas in the latter case the output is a simple listing of textual differences, the structural nature of an architecture facilitates a semantic interpretation of the differences. In particular, it seems to be possible to generate active patches that, rather than being a listing of differences, are much like a wizard script; the active patch contains an algorithm, that, if applied to a particular architecture, transforms that architecture into a different architecture [35].

5.2 Dynamic, Multi-Version Systems

Highly reliable and fault-tolerant applications are often constructed as self-configuring, multi-version systems in which critical functionality is implemented in alternative ways in different components. In these applications, an arbitrator typically determines which component are active and which results are used. Depending on the quality of the results from the various components, the arbitrator has the ability to reconfigure the application to include new components, remove components, activate or deactivate components, and sometimes even to rewire how the components are connected. Obviously, a strong relationship exists between these types of applications and our representation. In particular, the variant relationship that is captured by our representation, combined with our property-based optionality mechanism, could be used to support the definition and reconfiguration of these types of applications.

An alternative way of improving the reliability of an application is described in [8]. Rather than using variants, the approach is based on the incremental inclusion of multiple revisions of the same component. Once again, an arbitrator is used to determine which results from which component revisions are used in the application. Of course, our representation could serve the same role in this scenario as it does in the above: because the evolution of a component is precisely captured, the representation could be used to support the definition, evolution, and reconfiguration of these types of multi-version applications.

5.3 Architectural Erosion

One of the main problems identified in the software architecture literature is architectural erosion: once the conceptual architecture of a system has been created, it becomes out of date with respect to the actual architecture that is embedded in the implementation of the system [24]. To remedy this situation, it is often attempted to organize the structure of the source code of a system along its architectural components. However, this turns out to be a rather difficult exercise that is not very well supported by the current configuration management tools; these tools are geared towards managing the structure of the source code, not the structure of the system itself.

Our representation offers a unique opportunity to approach the problem of architectural erosion. In particular, the representation not only captures the structure of a system, but also its evolution over time. This complements the functionality of traditional configuration management systems which only capture the structure and evolution of the system implementation. We believe it is possible to build a configuration management system that utilities our representation for configurable software architecture to support its activities. In particular, we believe it is possible to organize these activities in terms of the architecture of the system that is being managed. This makes the architecture a first-class citizen in the configuration management system, thereby allowing it to be managed and evolved at the same time as the implementation. This, in turn, allows the architecture and implementation to be kept in sync over time.

6 Related Work

Perhaps most surprising is the limited amount of attention that has been paid by the discipline of software architecture to properly capturing configurability. Despite all the work in architectural reconfigurability that could obviously benefit from the availability of such a representation (e.g., [1, 18, 22, 27]), Unicon [28] and Koala [37] are the only two ADLs that have explicitly addressed part of the problem. Unicon incorporates a mechanism to specify variant implementations of components, whereas Koala allows the specification of components that have optional interfaces. However, in both cases only a small part of the larger problem of capturing configurability is addressed, and our work subsumes either solution.

One additional architectural effort to support evolution should be mentioned. DRADEL [20] is an environment and language that supports evolution through subtyping. Although different in nature from our solution, it does provide an alternative mechanism to capture evolution. However, neither optionality nor variability is addressed by DRADEL.

From a configuration management perspective, much work is (of course) related to our efforts of capturing architectural configurability. As previously described, we have borrowed and adapted a variety of concepts that have been established in the discipline for quite some time now. Still, our representation differs considerably from most system modeling languages developed to date. Perhaps the one system modeling language that is closest to ours in terms of expressive capabilities is Adele [9]. However, compared to its capabilities, our representation supports optionality, allows the definition of variability in terms of multiple interfaces as opposed to just one, facilitates the evolution of variable components just like regular components, and incorporates the notion of connections.

7 Conclusions

In this paper, we have discussed a rather different topic: the application of configuration management techniques to software architecture. We have developed the abstraction of configurable software architecture and created a representation that captures the variability, optionality, and evolution of architectures. In addition, we have described an environment that can be used to graphically specify configurable software architectures. Although the research is still in its preliminary stages, we believe the work presented here does make a number of contributions. The most important contribution is a precise representation for capturing configurability as an integral part of software architecture. Not only has such a representation been lacking until now, but our representation explicitly recognizes the concepts of variability, optionality, and evolution as equally important entities in the representation. A secondary contribution is the fact that we have successfully applied and adapted configuration management techniques to operate outside of the traditional domain of managing source code.

Much work still remains to be done. Our immediate concerns are to put our representation in actual use. In particular, our plans are to build a configuration management and software deployment system that are explicitly based on configurable software architecture.

REFERENCES

[1] B. Agnew, C. Hofmeister, and J. Purtilo. Planning for Change: A Reconfiguration Language for Distributed Systems. In *Proceedings of the Second International Workshop on Configurable Distributed Systems*, pages 15–22, Los Alamitos, California, March 1994. IEEE Computer Society Press.

[2] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, July 1997.

[3] Atria Software, Natick, Massachusetts. *ClearCase Concepts Manual*, 1992.

[4] B. Berliner. CVS II: Parallelizing software development. In *Proceedings of 1990 Winter USENIX Conference*, Washington, D.C., 1990.

[5] P. Bucci, T.J. Long, and B.W. Weide. Teaching software architecture principles in CS1/CS2. In *Proceedings of the Third International Software Architecture Workshop*, pages 9–12, November 1998.

[6] R.H. Byrd, E. Eskow, A. van der Hoek, R.B. Schnabel, and K.P.B. Oldenkamp. A parallel global optimization method for solving molecular cluster and polymer conformation problems. In *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, pages 72–77. SIAM, 1995.

[7] Continuus Software Corporation, Irvine, California. *Continuus Task Reference*, 1994.

[8] J.E. Cook and J.A. Dage. Highly Reliable Upgrading of Components. In *Proceedings of the 1999 International Conference on Software Engineering*, May 1999. To appear.

[9] J. Estublier and R. Casallas. The Adele configuration manager. In W. Tichy, editor, *Configuration Management*, number 2 in Trends in Software, pages 99–134. Wiley, London, Great Britain, 1994.

[10] J. Estublier and R. Casallas. Three dimensional versioning. In *Software Configuration Management: ICSE SCM-4 and SCM-5 Workshops Selected Papers*, number 1005 in Lecture Notes in Computer Science, pages 118– 135, New York, New York, 1995. Springer-Verlag.

[11] D. Garlan, R. Monroe, and D. Wile. ACME: An architecture description interchange language. In *Proceedings of CASCON'97*. IBM Center for Advanced Studies, November 1997.

[12] D. Garlan, W. Tichy, and F. Paulisch. Summary of the dagstuhl workshop on software architecture. *SIGSOFT Software Engineering Notes*, 20(3):63–83, July 1995.

[13] J. Kramer and J.N. Magee. Analysing Dynamic Change in Software Architectures: A Case Study. In *Proceedings of the Fourth International Conference on Configurable Distributed Systems*, pages 91–100, Los Alamitos, California, May 1998. IEEE Computer Society Press.

[14] D.B. Leblang, R.P. Chase, Jr., and H. Spilke. Increasing productivity with a parallel configuration manager. In *Proceedings of the International Workshop on Software Versioning and Configuration Control*, pages 21–37, 1988.

[15] D.C. Luckham, J.J. Kenney, L.M. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and analysis of system architecture using rapide. *IEEE Transactions on Software Engineering*, 21(4):336–355, April 1995.

[16] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In *Proceedings of the Fifth European Software Engineering Conference*, number 989 in Lecture Notes in Computer Science, pages 137–153, New York, New York, September 1995. Springer-Verlag.

[17] J.N. Magee, N. Dulay, and J. Kramer. Regis: A Constructive Development Environment for Distributed Systems. *Distributed Systems Engineering Journal*, 1(5):304–312, 1994.

[18] J.N. Magee and J. Kramer. Dynamic Structure in Software Architectures. In *Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, volume 21 of *SIGSOFT Software Engineering Notes*, pages 3–13. Association for Computer Machinery, November 1996.

[19] K. Marzullo and D. Wiebe. A software system modelling facility. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*. ACM SIGSOFT, April 1984.

[20] N. Medvidovic, D.S. Rosenblum, and R.N. Taylor. A language and environment for architecture-based software development and evolution. In *Proceedings of the 1999 International Conference on Software Engineering*, May 1999. To appear.

[21] N. Medvidovic and R.N. Taylor. A framework for classifying and comparing architecture description languages. In *Proceedings of the Sixth European Software Engineering Conference*, number 1301 in Lecture Notes in Computer Science, pages 60–76, New York, New York, September 1997. Springer-Verlag.

[22] P. Oreizy, N. Medvidovic, and R.N. Taylor. Architecture-Based Runtime Software Evolution. In *Proceedings of the 1998 International Conference on Software Engineering*, pages 177–186, Los Alamitos, California, May 1998. IEEE Computer Society Press.

[23] D.E. Perry and C.S. Stieg. Software faults in evolving a large, real-time system: a case study. In *Proceedings of the Fourth European Software Engineering Conference*, September 1993.

[24] D.E. Perry and A.L. Wolf. Foundations for the study of software architecture. *SIGSOFT Software Engineering Notes*, 17(4):40–52, October 1992.

[25] D.J. Richardson and A.L. Wolf. Software testing at the architectural level. In L. Vidal, A. Finkelstein, G. Spanoudakis, and A.L. Wolf, editors, *Joint Proceedings of the SIGSOFT'96 Workshops*, pages 68–71, New York, New York, 1996. ACM Press.

[26] C. Seiwald. Inter-file branching — a practical method for representing variants. In *Proceedings of the Sixth International Workshop on Software Configuration Management*, number 1167 in Lecture Notes in Computer Science, pages 67–75, New York, New York, 1996. Springer-Verlag.

[27] L. Sha, R. Rajkumar, and M. Gagliardi. Evolving Dependable Real Time Systems. In *Component-Based Software Engineering*, pages 125–132, Los Alamitos, California, 1996. IEEE Computer Society Press.

[28] M. Shaw, R. DeLine, D.V. Klein, T.L. Ross, D.M. Young, and G. Zelesnik. Abstractions for software architecture and tools to support them. *IEEE Transactions on Software Engineering*, 21(4):314–335, April 1995.

[29] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, Englewood Cliffs, New Jersey, 1996.

[30] J.A. Stafford, D.J. Richardson, and A.L. Wolf. Aladdin: A tool for architecture-level dependence analysis of software systems. Technical Report CU–CS–858–98, Department of Computer Science, University of Colorado, Boulder, Colorado, April 1998.

[31] R.N. Taylor, N. Medvidovic, K.M. Anderson, E.J. Whitehead, Jr., J.E. Robbins, K.A. Nies, P. Oreizy, and D.L. Dubrow. A component- and message-based architectural style for GUI software. *ACM Transactions on Software Engineering and Methodology*, 22(6):390–406, June 1996.

[32] E. Tryggeseth, B. Gulla, and R. Conradi. Modelling systems with variability using the PROTEUS configuration language. In *Software Configuration Management: ICSE SCM-4 and SCM-5 Workshops Selected Papers*, number 1005 in Lecture Notes in Computer Science, pages 216–240, New York, New York, 1995. Springer-Verlag.

[33] P. van der Hamer and K. Lepoeter. Managing design data: The five dimensions of cad frameworks, configuration management, and product data management. In *Proceedings of the IEEE*, pages 40–56. IEEE, January 1996.

[34] A. van der Hoek, D.M. Heimbigner, and A.L. Wolf. Software architecture, configuration management, and configurable distributed systems: A ménage a trois. Technical Report CU–CS–849–98, Department of Computer Science, University of Colorado, Boulder, Colorado, April 1998.

[35] A. van der Hoek, D.M. Heimbigner, and A.L. Wolf. System modeling resurrected. In *Proceedings of the Eighth International Symposium on System Configuration Management*, number 1439 in Lecture Notes in Computer Science, pages 140–145, New York, New York, 1998. Springer-Verlag.

[36] A. van der Hoek, D.M. Heimbigner, and A.L. Wolf. Versioned software architecture. In *Proceedings of the Third International Software Architecture Workshop*, pages 73–76, November 1998.

[37] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee. The koala component model for product families in consumer electronics software. 1999.