

# Design-Time Product Line Architectures for Any-Time Variability

André van der Hoek  
Department of Informatics  
School of Information and Computer Science  
University of California, Irvine  
Irvine, CA 92697-3425 USA  
andre@ics.uci.edu

## Abstract

*Most solutions for introducing variability in a software system are singular: they support one particular point in the software life cycle at which variability can be resolved to select a specific instance of the system. The presence of significantly increased and dissimilar levels of variability in today's software systems requires a flexible approach that supports selection of a system instance at any point in the life cycle—from statically at design time to dynamically at run time. This paper introduces our approach to supporting any-time variability, an approach based on the ubiquitous use of a product line architecture as the organizing abstraction throughout the lifetime of a software system. The product line architecture specifies the variabilities in a system, both in terms of space (captured as explicit variation points) and time (captured as explicit versions of architectural elements). A system instance can be selected at any point in time by providing a set of desired features, expressed as name-value pairs, to an automated selector tool. We introduce our overall approach, discuss our representation and tools for expressing and managing variability, and demonstrate their use with three representative examples of any-time variability.*

## 1. Introduction

Software variability is defined as “the ability of a software artifact to vary its behavior at some point in the life cycle” [25]. A broad range of mechanisms exist to create variability, including the use of compiler directives, dynamic linking, environment variables, plug-ins, configuration files, configuration management tools [5,8], and software deployment systems [2,13,15]. It is interesting to observe, however, that these solutions are limited in that each supports only a single point in the life cycle at which variability can be resolved to select a specific system instance. Consider, for example, compiler directives such as #IFDEF in C. These directives can only be resolved when the source code is fed through a compiler, not at any other time. As another example, variability managed by a configuration management system can only be resolved when a particular system instance is retrieved from the configuration management system, not at any other time.

In practice, almost all software systems exhibit multiple variabilities that are resolved at different times in the life cycle. As an example, one can download the Apache source code (which resolves a time variability by choosing a particular version), compile it (which resolves variabilities such as the hardware platform and operating system), install it (which resolves variabilities in the functionality that is made available to the users), and edit its configuration files (which resolves variabilities in terms of the directories and permissions used during the execution of Apache). Furthermore, Apache provides a plug-in mechanism with which the behavior of an installed configuration can be changed (resolving invocation time variabilities).

The approach of addressing different variability needs with different variability mechanisms at first seems like a satisfactory solution. In reality, however, it leads to two practical problems:

- When a variability has to move from one point in the software life cycle to another, it requires a change in its implementation mechanism. For instance, suppose a useful plug-in has been developed for enhancing the security of Apache. To increase the availability and use of this plug-in, the developers decide to move its functionality into the core of Apache. This involves changing the implementation of Apache to include the

plug-in, updating the installation mechanism, and changing the configuration files to support resolution of the moved variability. Combined, these activities represent a non-trivial effort.

- When a decision to resolve a variability has to be delayed or moved up, it requires awkward workarounds. For instance, to delay the decision of which version of Apache to install until after compilation of the source code requires separately downloading and compiling all potentially useful versions. Similarly, to move up the decision of which functionality will be available before any compilation takes place would require editing the source code.

These two problems are merely hypothetical examples, but they illustrate the presence of a larger issue regarding variability. In particular, there is a need for *any-time variability* (also identified as timeline variability [11]). We define any-time variability as “the ability of a software artifact to vary its behavior at any point in the life cycle.” Compared to the definition of variability presented earlier, the difference is that the time at which a variability is resolved should not be restricted, but instead should be allowed to be any time after the variability has been defined. Moreover, we postulate that the order in which different variabilities are resolved should not matter. The eventual outcome of resolving the same variabilities in the same manner, irrespective of which order is used, should lead to the selection of precisely the same system instance. Any-time variability, thus, introduces a flexible way of managing variabilities and supports an organization in choosing, on a case-by-case basis, when variabilities should be resolved.

A solution to supporting any-time variability must provide four pieces of functionality. In particular, it needs: (1) a representation to *capture* system variability, (2) a tool to *specify* variability, (3) a tool to *resolve* variability, and (4) tools that at different points in the life cycle *apply* the result of variability resolution. The difference between the tool that resolves variability and the tools that apply it is that the first operates generically at the level of the representation and the second maps the result to a specific task at hand. Whereas the first tool, thus, is reused throughout all phases of the life cycle, the other tools must be uniquely implemented per phase. Clearly, the choice of representation and variability resolution tool influence the design of the other tools; we wish to choose a suitable representation and associated variability resolution tool that allow the specific application tools to be small in their implementation.

In this paper, we introduce results of our explorations in providing and managing any-time variability. Our specific approach resolves around the ubiquitous use of a product line architecture as the organizing abstraction during the lifetime of a software system. The product line architecture, as specified at design-time in XADL 2.0 [9], serves as the representation for variability. Variability is captured in terms of both space (through explicit variation points) and time (through explicit versions of architectural elements). Accompanying the representation are MÉNAGE [12], a tool for specifying, maintaining, and evolving product line architectures, and SELECTORDRIVER, a tool for resolving variabilities. Both MÉNAGE and SELECTORDRIVER operate generically at the level of the product line architecture.

To demonstrate how these tools lay the basis for any-time variability, we have explored different points in the life cycle at which to resolve variability. First, we briefly demonstrate how variability can be resolved early at design-time. Then we demonstrate how our approach supports the construction of simple tools that support resolving variability at the time of invocation of a system and at the time of actual execution of a system. While these examples represent only three points in the life cycle, they represent points at different ends of the spectrum from early resolution to late resolution. Moreover, our experiments show how our generic tools are reused at each stage, and how variability resolution can shift easily from one point in the life cycle to another.

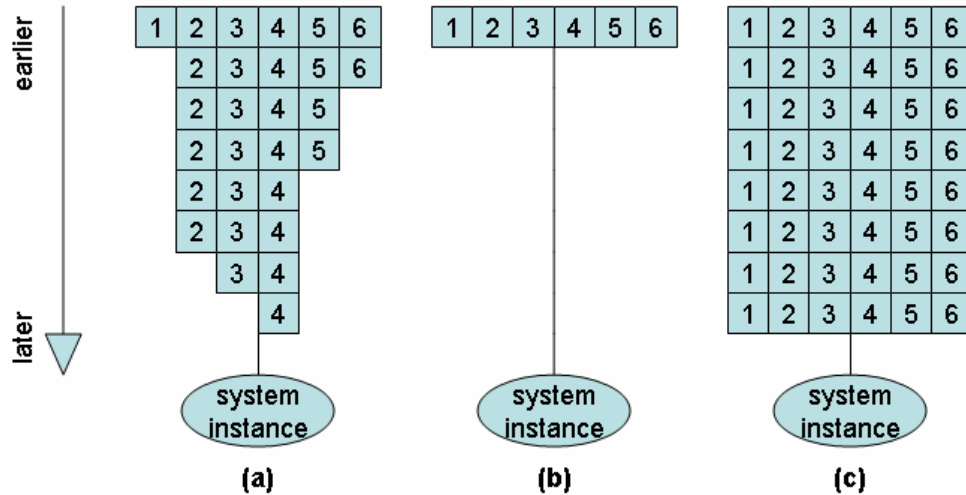
Below, we first discuss background material in the field of variability management to properly set the stage for the ensuing discussion. We introduce our high-level approach in Section 3 and discuss the implementation in Section 4. We show three representative examples of any-time variability in Section 5, present a critical evaluation in Section 6, discuss related work in Section 7, and conclude in Section 8 with an outlook at our future work.

## 2. Background

Many different techniques can be used to introduce and manage software variability. Solutions include implementation-level approaches (e.g., compiler directives, aspect-oriented programming [17], mix-ins [6], dynamic linking, parameterized Makefiles), design-level approaches (product line architectures), tool-based approaches (e.g., configuration management [5,8], software installation [15], software deployment [2,13,20]), run-time approaches (environment variables, configuration files, plug-ins), and numerous others [25]. Clearly, an in-depth treatment of all these approaches is beyond the scope of this paper. Looking at their collective contribution, however, we observe

that they share a common trait: all are currently limited to resolving variability at one particular point in the life cycle.

In a typical software system, one can find a number of different variability mechanisms that each are resolved at different points in time. This leads to a variability resolution process as characterized by Figure 1a. The figure displays a hypothetical software system that has six variabilities, each represented by a numbered block. Early in the life cycle, all six variabilities are available—none has been resolved yet. As one moves through the stages of the life cycle, variabilities are progressively resolved. The order in which they are resolved (e.g., first variability 1, then variability 6, then variability 5, etc.) is determined by the point in the life cycle at which each variability must be resolved, and typically cannot be changed. Therefore, the shape of the figure remains constant and in essence provides a blueprint for variability resolution.



**Figure 1. Resolving Variabilities throughout the Life Cycle: (a) Traditionally, (b) Early with Any-Time Variability, and (c) Late with Any-Time Variability.**

Any-time variability changes this process dramatically. The increased flexibility provided in terms of when variabilities can be resolved supports a process that can range from that depicted in Figure 1b to that shown in Figure 1c. Figure 1b shows a process that reflects early resolution of all variabilities at once. Throughout the remainder of the life cycle, a single system instance is available. Figure 1c, on the other hand, presents a process in which all variabilities are resolved at the latest possible time. Earlier activities in the life cycle operate upon the system with all of its variabilities in tact.

Of note is that any-time variability supports a process of any shape in between the extremes shown in Figures 1b and 1c. Due to the flexibility in the order and time as to when a variability is resolved, the process can be easily configured to the particular needs that an organization may have. Consider an organization that sells different variants of its highly-configurable software to outside organizations. Some organizations receive an instance in which all variabilities are resolved. Other organizations may receive an instance with some variabilities resolved, but with other variabilities still open for the organization to resolve. Given that a different set of variabilities is left to each different organization, and given that those sets may change over time, a flexible mechanism in which variabilities can be moved to earlier or later times in the life cycle is desired. This leads to a different process “funnel” [30] (as in Figure 1) for each different organization.

### 3. Approach

Whereas traditional variability can be supported with a localized solution that operates during one particular point in the life cycle, any-time variability requires a solution that has a broad impact on the life cycle. In particular, we believe that any solution to any-time variability must provide the following four pieces of functionality:

1. *A representation to capture system variability.* The representation serves as the source for variability, and should be available at all times for examination and resolution of (remaining) variabilities. In effect, an instance of the representation “travels along” with the system as it progresses throughout the life cycle. This is critical to the functioning of any-time variability, but also has wide-ranging impact: each activity must be able to interpret the representation and take proper action based upon its contents.
2. *A tool to specify variability.* The representation by itself is of little use if it has to be maintained by hand. Therefore, a second part of the solution has to be a tool with which users can specify and maintain the variabilities that are present in a system. The tool should provide explicit support for specifying variability both in terms of space (e.g., describing the facilities of the system for use in different contexts) and in terms of time (e.g., capturing the evolution of the system over time).
3. *A generic tool to resolve variability.* Complementing the specification tool should be a tool that can be used to resolve variabilities in the representation. In theory, a combinatorial explosion of potential system instances may result when many of the variabilities are interrelated. In practice, however, variabilities tend to be relatively independent [14]. Nonetheless, resolving variabilities by hand is a tedious and error-prone task. An automated solution must be provided with which variabilities in the representation can be resolved.
4. *Specialized tools that at different point in the life cycle apply the results of variability resolution.* The final part of the solution builds upon the automation resolution tool to contextualize the results of the selection. In particular, a specialized tool must be provided for each activity in the life cycle that maps the results of the generic variability resolution tool to the artifacts pertaining to that activity. For instance, in case one wants to resolve variability at run-time, the generic resolution tool should be invoked to determine the desired system instance, but the result still needs to be interpreted and used to change the currently executing system instance into the newly desired system instance.

Our approach is one particular instantiation of this general solution. Shown in Figure 2, it is rooted in the use of product line architecture as the organizing abstraction throughout the life cycle. Building upon the principle of architecture-based development [19,32], we structure each activity to be performed at the level of one or more architectural elements. This raises the level of abstraction and also brings a unified view to the software life cycle. It also, of course, incurs a cost in that tools must principally be able to operate in terms of architectures rather than “native” artifacts such as source code. The architecture-based development environment upon which we build our approach, ARCHSTUDIO [19], however, has demonstrated that this is a reasonable possibility. We refer to its literature for a more general discussion of architecture-based development and its benefits as compared to traditional software development environments.

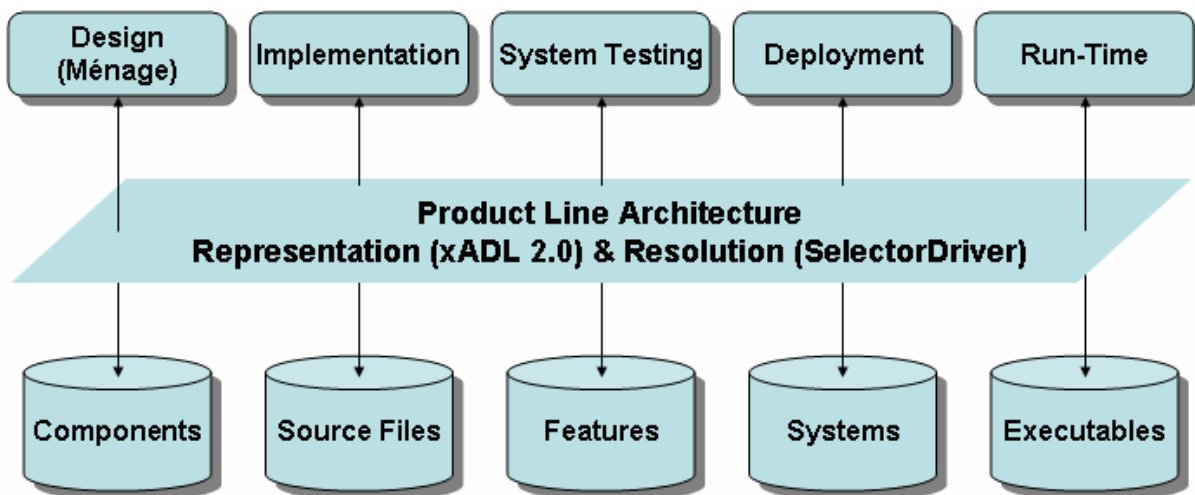


Figure 2. Product Line Architecture as the Organizing Abstraction in the Life Cycle.

Compared to purely architecture-based approaches, our approach leverages the ability of product line architectures to integrally represent variability [30]. In particular, product line architectures extend traditional software architectures with integrated facilities for modeling variation points (capturing the differences among alternative system instances) and versions (capturing the evolution of individual architectural elements and a product line architecture as a whole). Moreover, the component-oriented focus of product line architectures makes it suited for use as the basis for any-time variability, firstly because components already play a critical role in a significant number of software development projects and secondly because components represent a natural granularity for variability.

## 4. Generic Infrastructure Implementation

The generic part of our infrastructure for any-time variability consists of a representation for product line architectures, XADL 2.0; a tool to express variability in that representation, MÉNAGE; and a tool to select a particular system instance out of a product line architecture, SELECTORDRIVER. Common to these three parts is their explicit ability to handle architectures that exhibit variability in space (through variation points) and time (through version management). Below, we discuss each in more detail.

### 4.1 XADL 2.0

XADL 2.0 [9] is an extensible architecture description language that is built as a set of extensible XML schemas. Associated libraries provide a programmatic interface to XADL 2.0 documents, and provide facilities to create, store, and modify XADL 2.0 documents. While XADL 2.0 is centered around the use of XML, the libraries hide all XML details and allow a programmer to manipulate a XADL 2.0 document in terms of components, connectors, interfaces, and other architectural elements.

XADL 2.0 consists of nine XML schemas that incrementally define its full functionality. Of interest to this paper are the six “lowest” schemas, which provide facilities for modeling evolving product line architectures. The cornerstone of XADL 2.0 is formed by its *Structure and Types* schema, which defines the modeling constructs for capturing a single architecture. Specifically, the schema allows the definition of the basic structure of one particular architecture as a set of components and connectors. Both components and connectors have interfaces, which are the elements that are linked together to form an architectural configuration (e.g., two components can be “hooked up” via a connector by placing links in between the interfaces on the components and the interfaces on the connector).

In addition to supporting the definition of the structure of an architecture, the *Structure and Types* schema provides a typing mechanism through which components, connectors, and interfaces can be assigned specific types. This supports the use of external tools for type checking purposes, and, for the usual reasons, enhances the understandability and usability of the architecture [23].

The *Options*, *Variants*, and *Boolean Guard* schemas extend the modeling facilities of the *Structure and Types* schema with facilities for modeling variability in space through explicit variation points in the architecture. The *Options* schema supports the definition of optional architectural elements. For example, an architecture may define an authentication component that, if included, enhances the security of the system. Because not everyone using the system needs the additional security, it should be defined as a variation point using an optional component.

The *Variants* schema supports the definition of a second kind of variation point, namely variant elements. Variant elements are always included in an architecture, but are configured to be one of a set of alternative types. Variant elements are well-suited for expressing differences in, for instance, computing platform. As an example, an architect may design a system with a user interface (which must always be present) that can be configured for either the Windows or Unix platform. To do so, the user interface should be defined as a variation point using a variant component. Variant elements may be optional. For instance, an architect may define an optional security component that uses one of multiple available security protocols.

The *Boolean Guard* schema defines the kinds of Boolean expressions with which optional and variant elements are guarded. In particular, each optional element is guarded by one Boolean expression, and each set of variant elements is guarded by a series of Boolean expressions, one per variant. Guards, thus, form the basis for determining the inclusion of optional elements and the selection of particular variants.

The representation created by using the *Structure and Types*, *Options*, *Variants*, and *Boolean Guard* schemas is that of a product line architecture. In effect, these schemas are equivalent in their ability to express variability as reported in many approaches to modeling product line architectures [29]. For our purposes, however, we also need to provide facilities for modeling variability in time. The *Versions* schema builds on top of the other schemas to support

the evolution of the architecture at large as well as the individual elements that make up the architecture. It extends each type with a version identifier and relates those versions in a version graph that can be traversed. As a result, XADL 2.0 provides support for modeling evolving product line architectures,

XADL 2.0 and its associated libraries provide three important benefits for the purposes of supporting any-time variability:

1. *The core of the language supports variability in both space and time.* The language was carefully defined to support the modeling of evolving product line architectures. As such, variabilities are a natural and integral part of the language, and the language is “complete” for the purposes of representing the essence of any-time variability.
2. *The language can be extended.* Any representation for any-time variability must be extensible. In particular, individual activities in the life cycle must be able to attach additional information. For instance, during deployment, the deployment tool may want to attach information that is later used when the system is executed. Because XADL 2.0 is extensible, and in fact itself is built as a series of extensions on top of a common core, it can easily accommodate additional information.
3. *The library provides a generic interface to easily access XADL 2.0 documents.* This supports the rapid construction of new tools supporting any-time variability. These tools do not have to worry about the intrinsic difficulties of processing XML, but can simply be created by programming naturally architectural objects.

## 4.2 MÉNAGE

MÉNAGE is the design environment that an architect uses to initially specify and then maintain variabilities. Displayed in Figure 3, the graphical user interface is partitioned into three separate panels. The panel on the left side lists component types, connector types, and interface types that have been previously defined. The top panel shows the version graph of the type that is currently displayed in the main panel. Finally, the main panel is where actual design of a product line architecture and its variabilities takes place. A discussion of all the features of MÉNAGE is beyond the scope of this paper, and provided elsewhere [12,27]. Our discussion here focuses on how MÉNAGE supports the specification of space variabilities and time variabilities.

### 4.2.1 Space Variabilities

Ménage supports the specification of all three kinds of variation points defined by XADL 2.0: optional elements, variant types, and optional variant elements. Optional elements are added just as regular elements, but require an architect to additionally provide a Boolean guard that determines inclusion or exclusion of the optional element. The Boolean guard has to adhere to the following BNF that is determined by the *Boolean Guard* schema of XADL 2.0:

```

<BooleanGuard> ::= <BooleanExp>
<BooleanExp> ::= <And> | <Or> | <Not> | <GreaterThan> | <GreaterThanOrEquals> | <LessThan> |
  <LessThanOrEquals> | <Equals> | <NotEquals> | <InSet> | <InRange> | <Bool> | <Paren>
<And> ::= <BooleanExp> && <BooleanExp>
<Or> ::= <BooleanExp> || <BooleanExp>
<Not> ::= !<BooleanExp>
<GreaterThan> ::= <LeftOperand> > <RightOperand>
<GreaterThanOrEquals> ::= <LeftOperand> >= <RightOperand>
<LessThan> ::= <LeftOperand> < <RightOperand>
<LessThanOrEquals> ::= <LeftOperand> <= <RightOperand>
<Equals> ::= <LeftOperand> == <RightOperand>
<NotEquals> ::= <LeftOperand> != <RightOperand>
<InSet> ::= <LeftOperand> @ { <Set> }
<InRange> ::= <LeftOperand> @ [ <RightOperand>, <RightOperand> ]
<Paren> ::= ( <BooleanExp> )
<Set> ::= <RightOperand> | <RightOperand>, <Set>
<LeftOperand> ::= Variable
<RightOperand> ::= Variable | Value

```

<Bool> ::= true | false

This BNF supports a wide range of expressions, including set and range operations and the use of variables, and thus far has been adequate to express the conditions for the variabilities in the architectures we have developed. Boolean guards are typically of a rather trivial nature, and often rely on equality or simple true/false decisions. The availability of a rich language, however, allows architects to establish intricate relationships among variation points. For instance, one can model that selection of a particular variant in one variant type should lead to the selection of a specific variant in another variant type by carefully matching the Boolean guards on the two variants.

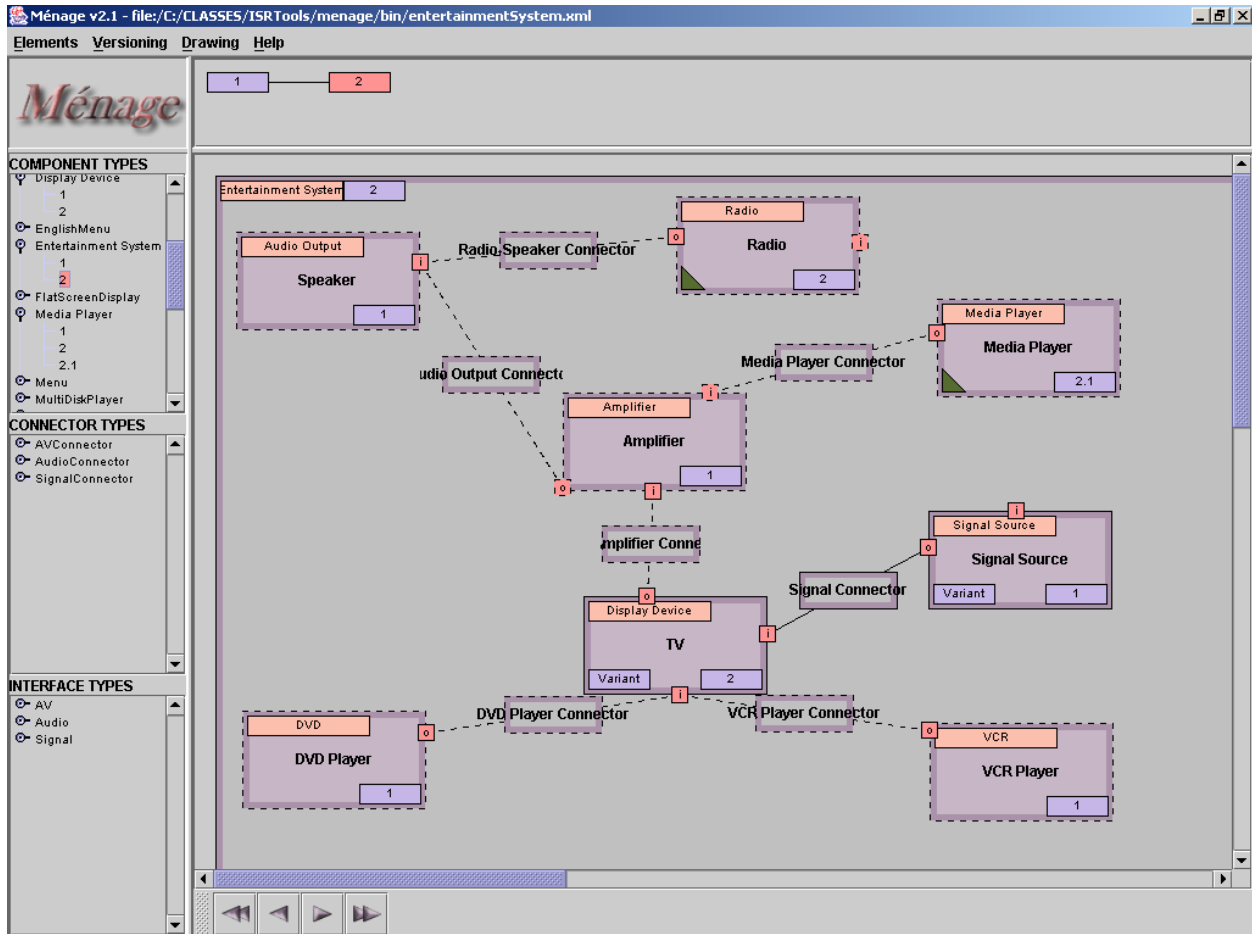


Figure 3. Specifying and Evolving an Entertainment System Product Line Architecture in Ménége.

Graphically, optional elements are shown using dashed lines. Because the entertainment system product line architecture shown in Figure 3 is highly configurable, many of its components and connectors are optional. Note that links connecting optional components or connectors are automatically optional as well, inheriting the guards of the optional elements they connect. This preserves the integrity of the architectures that are eventually the result of variability resolution: no dangling links will be present in those architectures.

Small rectangular “variant” tags identify variant elements in the product line architecture. For instance, the component *TV* is a variant component. Double clicking on the element leads to the specification shown in Figure 4. The television is either a plasma screen or a flat screen, depending on the value of the variable *displayType*. The guards are mutually exclusive, guaranteeing that only a single variant is chosen at a time.

Note the presence of the small triangle in the lower left corner of the flat screen variant in Figure 4. This indicates the presence of substructure; the internal structure of the flat screen television consists of other components.

Of note is also that, in the case of the example in Figure 4, the interfaces on the variants are exactly the same as the interfaces on the overarching variant type (2 input interfaces and 1 output interface). The general rule that is fol-

lowed in MÉNAGE is that interfaces may differ among variants, but that optionality should be used on the encompassing variant type to ensure compliance. Suppose, for instance, that the flat screen television component has an additional interface for adjusting its picture quality. Such an interface should be declared optional at the level of the variant type (*Display Device* version 2), since the plasma television does not provide this interface. Principled use of this approach guarantees compatibility within the remainder of the product line architecture, irrespective of which variant is eventually selected.

Optionality and variability can be mixed to create optional variant elements. In such cases, a specific variant will only be chosen if the variant element is included per its optional Boolean guard. The Boolean guards, thus, are layered and interpreted in order.

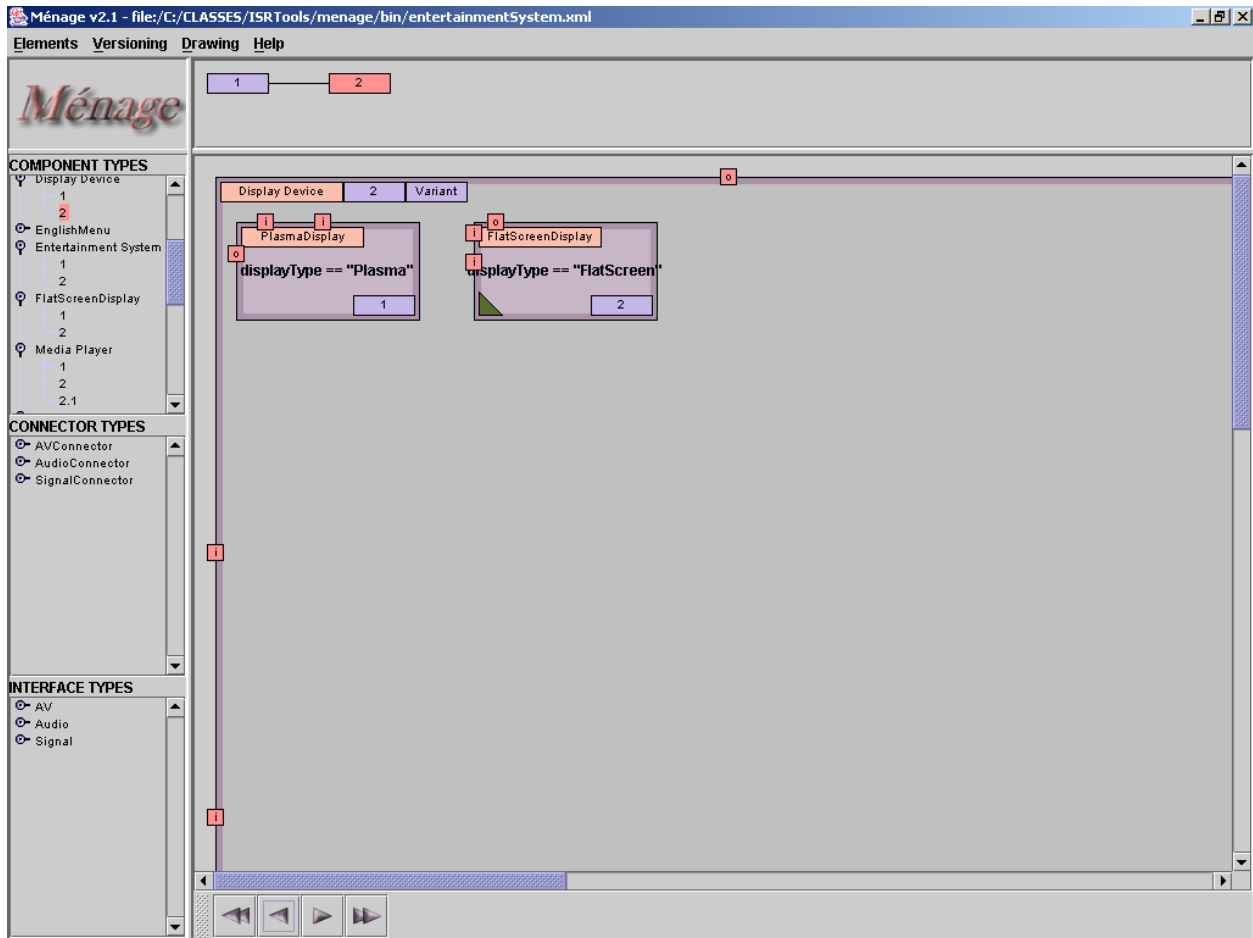


Figure 4. Specifying Two Television Variants in Ménége.

#### 4.2.2 Time Variabilities

Time variabilities are introduced through the check out/check in configuration management policy of MÉNAGE. Before changes can be made, an architect must check out the set of architectural elements they will be modifying. Once all desired changes have been made, the architect checks in the modified parts of the product line architecture. In response, Ménége automatically creates a new version of each element and, in the process, creates a history of changes that can be revisited over time. Once a version has been checked in, that version becomes immutable. It can no longer be modified in order to protect any other parts of the product line architecture that depend on the immutable element. This guarantees incremental stability as a product line architecture is designed, and during maintenance guarantees the integrity of the old versions of the product line architecture. Branching is supported should it nonetheless become necessary to change an old version.

Only a single time variability is automatically supported by the check out/check in mechanism, namely that of the overall product line architecture. Since it is specified in terms of specific versions of specific elements (i.e., see the version tags in both Figure 3 and Figure 4), no time variability for the lower level elements is supported. To circumvent this fact and introduce an explicit time variability (e.g., an architect wishes to express that either version 2 or 3 of a subsystem can be used), the architect should create a variant element with versions 2 and 3 of the subsystem as the variant elements. Although requiring extra work, the benefit is that the time variability is explicitly recorded and maintained in the product line architecture, thereby circumventing many problems that exist with seemingly easier automated default mechanisms (e.g., automatically creating time variabilities or automatically adding new versions to a time variability).

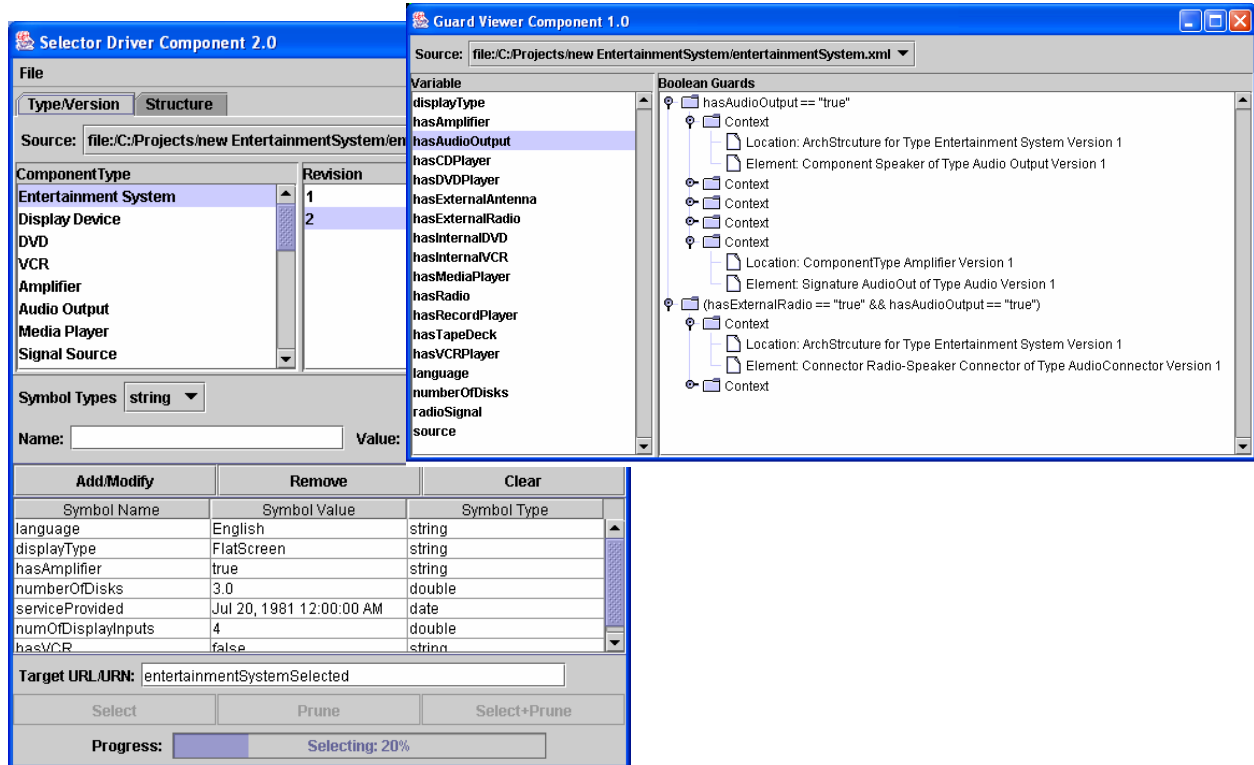


Figure 5. Selecting a System Instance.

### 4.3 SELECTORDRIVER

At any point after the product line architecture has been specified and the variabilities have been introduced, it is possible to choose a specific system instance by providing a set of desired features to the SELECTORDRIVER. Shown in Figure 5, the SELECTORDRIVER takes as input a set of typed name-value pairs and creates as output a new XADL 2.0 document in which it has resolved as many and as much of the variabilities as possible. Any variability that can be fully resolved is fully resolved in the new document: optional elements are either included or excluded and if a specific variant can be chosen it is selected. It cannot be guaranteed, however, that all variabilities can be resolved completely. Some may only be partially resolved and some may not be resolved at all. In those cases the new document will still contain variabilities, although they will typically be fewer in number and smaller in size (because the Boolean guards are evaluated and resolved as much as possible).

Variability resolution, thus, is incremental and an architect may resolve different variabilities at different times. To assist an architect in knowing which variabilities may still be left, our toolset includes an integrated component called the GUARDVIEWER. The GUARDVIEWER, shown on the right in Figure 5, inspects the current product line ar-

chitecture, lists the names of all variables that are governing the remaining variabilities, and shows per variable in which Boolean guards it is present and where those guards appear in the architecture.

Of note is that the SELECTORDRIVER is a front end to a generic SELECTOR component. Therefore, it is possible to perform architectural selection without the need for the graphical user interface. This is important when selection has to be performed automatically and embedded in an application (see Section 5.3 for an example).

## 5. Three Examples

To demonstrate the principle of any-time variability, we show how our tools can be used to resolve variability at three different points in the life cycle. All assume that variability is specified at design time at the architecture level, but resolve the specified variabilities at a different time, namely design time, invocation time, and run time.

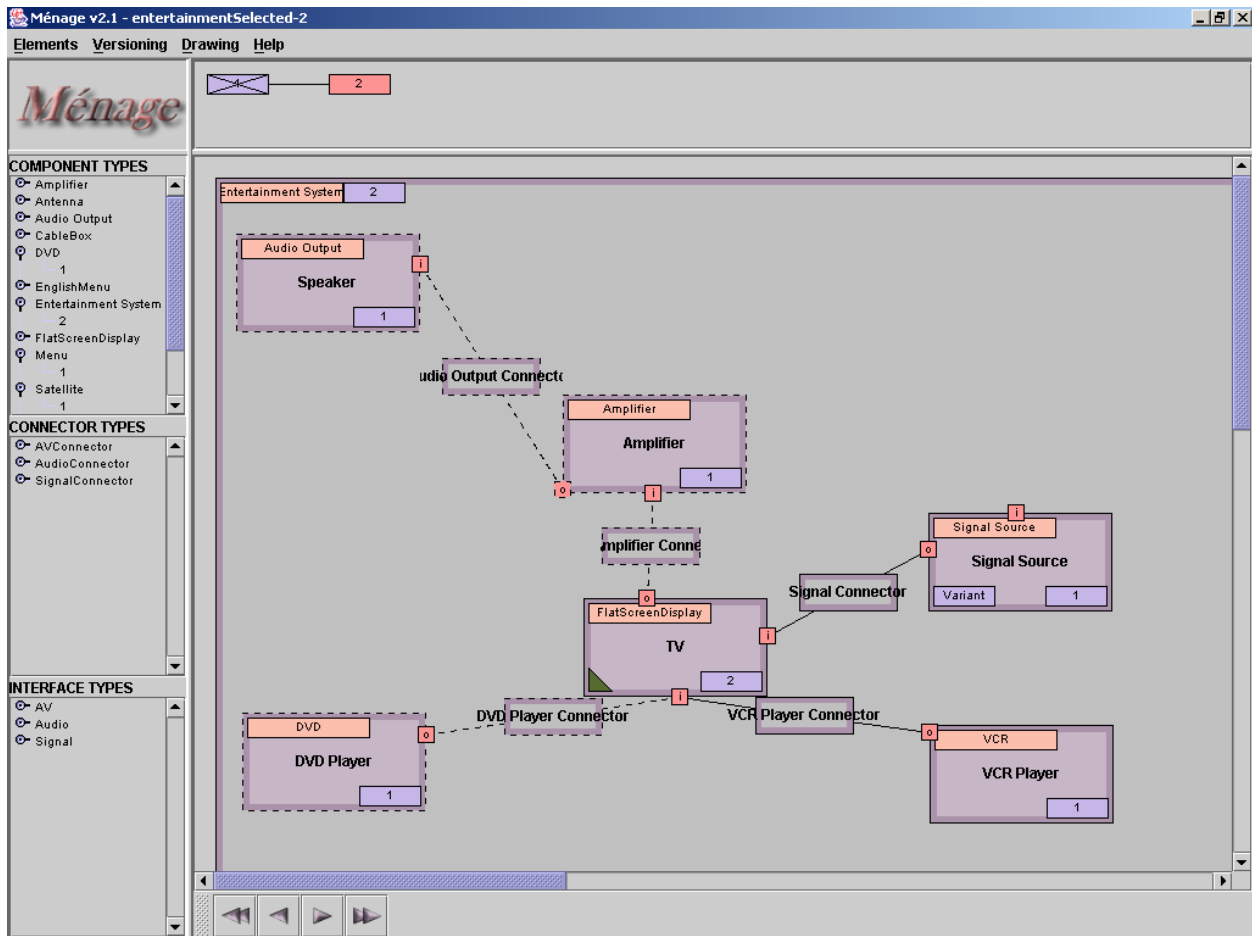


Figure 6. The Entertainment System Product Line Architecture after Partial Resolution at Design Time.

### 5.1 Design-Time Variability

As a first experiment in any-time variability, we examined its use at the earliest possible time: during design. In particular, we applied the SELECTORDRIVER to the entertainment system product line architecture shown in Figure 3, resolving some of the space variabilities and some of the time variabilities. The result is shown in Figure 6. As compared to the original, fully-defined product line architecture in Figure 3, we note the following differences in terms of variabilities in space:

- The optional *Radio* and *Media Player* components are no longer part of the architecture, and neither are the connectors and links that were used to hook them up to the remainder of the architecture.
- The optional *VCR Player* turned into a regular component and is no longer optional.
- The variant *TV* was resolved to be a flat screen television.

Additionally, a variability in time was resolved. As indicated by the crossed-out version tag, version 1 of the entertainment system is no longer available for use.

While a rather trivial example, this is also a very important example. It demonstrates that variabilities can be resolved at the earliest convenient time, that the SELECTORDRIVER can be used unchanged to do so, and that MÉNAGE supports viewing and even further editing of partially resolved product line architectures. During and right after design, thus, it is already possible to limit which variabilities will be available for use in later activities in the life cycle.

## 5.2 Invocation-Time Variability

Our second example of any-time variability applies the SELECTORDRIVER at invocation time. Rather than simply executing a system, as in traditional invocation of a software program, invocation in the face of our approach to any-time variability must be performed in two steps. The desired system instance must first be selected by resolving all variabilities and only then can it be instantiated.

To support this process, we had to add two capabilities to the generic infrastructure described in Section 4. The first extension is a mapping from design-level components to executable components; the second an architecture-based invocation system. The mapping is needed to realize the architecture; without it, the architecture remains a design-level artifact with little value. The invocation system is needed to start the application based on a particular architectural configuration.

We defined the mapping as a XADL 2.0 schema that maps architectural components and connectors to sets of Java class files. Each component (connector) can have one or more associated Java class files, one of which is tagged as the main class file that should be invoked to start the component (connector). When presented with an architecture description of a system to invoke, the invocation system leverages this information to instantiate the system and one-by-one start the components and connectors. Both the mappings and invocation system were already built as part of an unrelated effort in self-healing software [10], and we could utilize them unchanged for the purpose of any-time variability.

As a demonstration of invocation-time variability, we created a XADL 2.0 document for a notional AWACS aircraft system [9]. The system consisted of about 15 inter-connected components and connectors, and through its variabilities could be configured into about 20 different system instances. The invocation system (AEM DRIVER) and one particular instance of the notional AWACS aircraft system are shown in Figure 7.

The demonstration of invocation-time variability on the AWACS example highlights the following benefits of our approach to any-time variability:

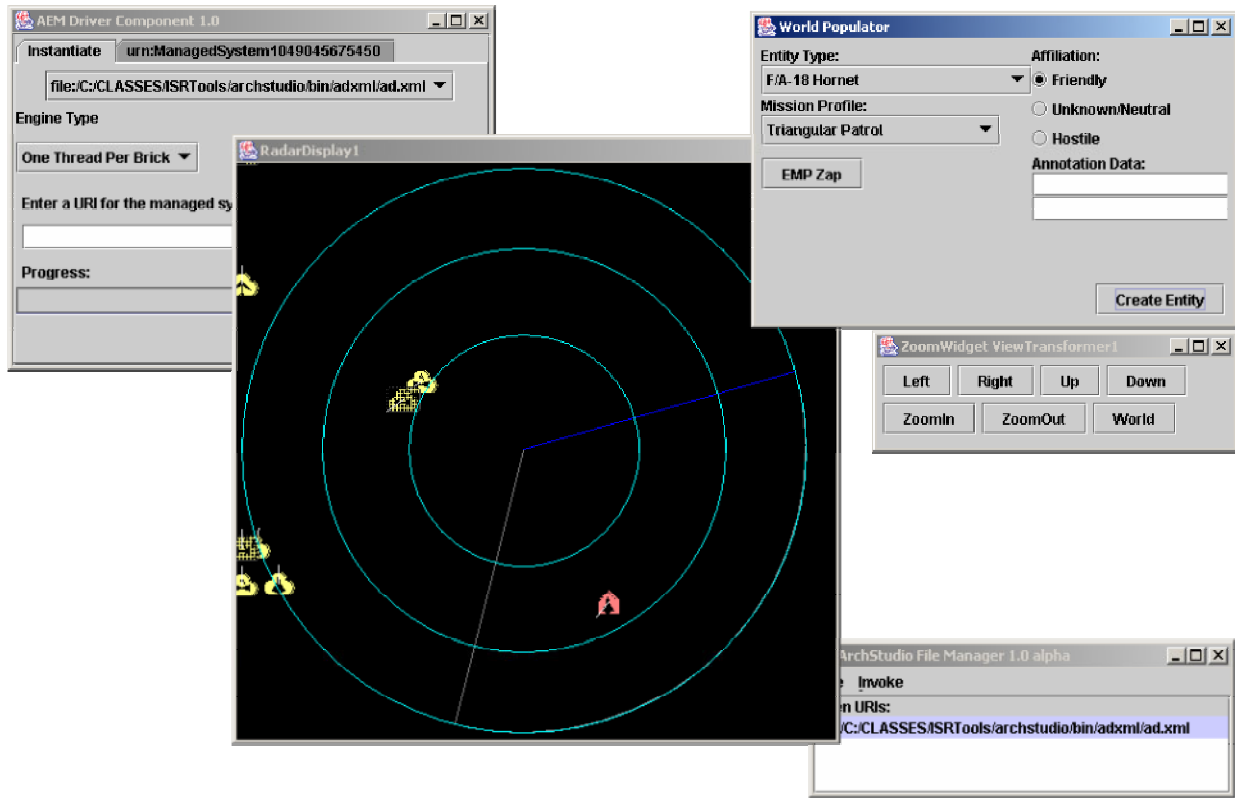
- The SELECTORDRIVER could, once again, be used unchanged. As described in Section 4.1, the extensibility of XADL 2.0 supports attaching information to core architectural elements without influencing the behavior of the tools that operate on the core elements. In this case, the mappings are attached to components and connectors, and are automatically included in any selection.
- Variabilities specified at design time using MÉNAGE could indeed be resolved at a much later time.
- When combined with design-time variability as discussed in the previous section, it is possible to move the resolution of some or all of the variabilities from design time to invocation time and back. In effect, we were able to demonstrate approaches ranging from resolving all variabilities at design time to resolving all variabilities at invocation time, without having to change our tools, methods, or application implementation.

We observe, however, that in case we had not developed the mappings and the invocation system as part of our previous work, introducing invocation-time variability would have required us to build some serious infrastructure. We also note that mappings currently have to be specified by hand, which is a drawback that we intend to overcome in the near future.

## 5.3 Run-Time Variability

Our last demonstration of any-time variability moves the point of variability resolution even later than invocation time, namely into the executing system itself. Rather than letting an outside installation or configuration tool statically resolve the variabilities to determine the exact system instance to be executed, we have examined utilizing the principle of any-time variability dynamically at run-time to let a system itself determine its particular instance based on the context in which it executes. In fact, we have extended this approach such that the system can reconfigure itself during its execution should its context change.

We have built an example collaborative scheduling system based on this approach. Particular variabilities include the use of a secure or non-secure communication protocol depending on whether the application is connected via a wireless or wired network, use of a shared database if more than one instance of the system is executing, and a difference in presentation mechanism that depends on whether the application executes on a handheld device or on a fixed computer.

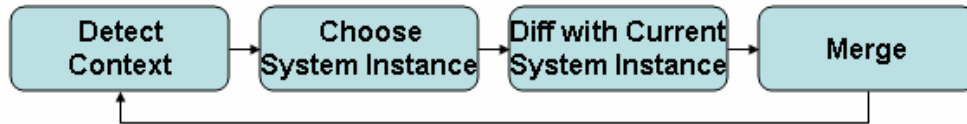


**Figure 7. One Configuration of the Notional AWACS System after Invocation using AEM Driver.**

In this particular case, details of the scheduling functionality of the application are not as important as how it actually chooses and configures into a suitable system instance. The process that it follows is shown in Figure 8. The system includes a standard *Context Detection* component that, in a bootstrap fashion, is first instantiated. This component determines the current context, and then uses the SELECTORDRIVER to select a particular system configuration suitable for that context (e.g., the context is translated into attribute values that are fed into the SELECTORDRIVER). Once the system configuration is selected, it is instantiated using a diff/merge process. The desired system instance is compared to the currently active system instance (null in the case of first-time instantiation), resulting in an architectural patch that is subsequently merged with the running instance. As compared to a wholesale reinstantiation of the application each time the context changes, the diff/merge process has minimal impact on the running application. Only those parts that need to change are changed while the remainder of the application continues executing.

We were able to reuse the differencing and merging algorithms from a previous effort [28], but did need to develop the layered architecture in which the *Context Detection*, *SELECTORDRIVER*, *DIFF*, and *MERGE* components are an integral part of the application architecture. Once again, the introduction of any-time variability reflects a change that can be rather major in nature. In this case, applications must be developed using our layered architecture in order to accommodate resolving any-time variabilities at run-time.

As a final note, we observe that the example application adheres to the principle of any-time variability and supports resolution of variabilities from early to late in the life cycle. If certain variabilities are already resolved, for instance during design time or invocation time, the application can only run in a limited set of contexts.



**Figure 8. Run-time Reconfiguration Process in the Presence of Run-time Variabilities.**

## 6. Critical Evaluation

Our work thus far has merely demonstrated the beginnings of support for any-time variability. Based on a design-time product line architecture that contains both space and time variabilities, we have demonstrated how one can resolve those variabilities from early and statically (at design-time) to late and dynamically (at invocation time and run time) using the same infrastructure. Clearly, mappings must be created at each point in the life cycle where it is desired to resolve variability. We have illustrated some of these mappings, but clearly need to perform further experimentation in other phases of the life cycle. In particular, we want to develop a configuration management system and a software deployment system that are based on the principle of any-time variability. By introducing those two tools, we believe we can provide a more complete solution to any-time variability that has automated support for creating and maintaining the mappings onto source code and executable class files, mappings that are essential but that we currently must maintain by hand.

While we believe our approach demonstrates merit in terms of the abilities garnered, our explorations thus far also raise some questions. Perhaps the first and foremost question pertains to why we chose product line architectures as the central abstraction for organizing the activities in the software life cycle. Another viable choice, for instance, is the abstraction of features and the resulting approach of feature engineering [26]. We have no concrete evidence whether one or the other is better, but found product line architectures to be a useful abstraction since it serves as a bridge between features (identified in the requirements document) and their realization (codified in the implementation, configurations, libraries, binaries, etc.). Moreover, elements of a product line architecture can easily be represented and mapped onto different kinds of artifacts, something that is more difficult with features. Finally, we observe that features still play a role in product line architectures. Codified into attributes in guards, they are the determining factors for choosing particular system instances.

We further observe that any-time variability as introduced in this paper does not cover all of the functionality provided by existing variability mechanisms. For instance, it does not and cannot provide support for handling compiler directives. This is a conscious choice: to gain the ability of flexibly moving variability resolution to different times in the life cycle, it is necessary to choose one particular abstraction for representing variabilities. Clearly, not all variabilities can be represented in such an abstraction, and attempting to do so would be a mistake. Our choice of product line architecture as the central abstraction represents a tradeoff between being able to handle a large portion of typical variabilities and ignoring more specialized variabilities at single points in the life cycle.

Finally, our experiences show that our general infrastructure is currently missing one critical piece of functionality. It has to be able to guarantee the consistency of a selected system instance. At present, we rely on the architect to create flawless product line architectures. Given the potentially complicated nature of product line architectures, and given the intricacies involved in precisely specifying Boolean guards, we ran into situations in which the result of a selection was inconsistent. As part of our previous work we developed an infrastructure for critics [10]. We intend to leverage that infrastructure by building consistency critics that verify the result of a system selection.

## 7. Related Work

Variability has been an integral part of most software development projects to date. As discussed in Section 2, a plethora of techniques is available to introduce variability in a software system. Our work, in seeking any-time variabilities rather than variability at one point in the life cycle, takes a rather radical departure from most of these technologies. Nonetheless, our work is based on several existing approaches. Work in product line architectures is at the heart of our solution. The greatest influence has been Koala [31], which provides an excellent example of how principled use of product line architectures can shape and improve the software development process. Additionally, our XADL 2.0 representation is roughly equivalent in expressiveness to the Koala representation. Due to a need for optimized code, a significant difference between Koala and our approach is that Koala resolves all variability at compile time.

Many other approaches to representing variability in product line architectures exist [1,7,16,22,24,25,29]. Our approach is equivalent to those approaches in expressiveness, but provides an integrated design environment for expressing variability and an integrated tool for resolving those variabilities. We are aware of only two other approaches that provide these generic tools: CONSUL [3] and GEARS [18]. CONSUL provides a Prolog-based tool for modeling features and mappings of features onto classes. It follows much of the outline of our solution, but limits itself to resolving variability to right before compilation time. GEARS also provides a feature-based approach with tools for specifying and resolving variabilities. Once again, however, GEARS limits itself to resolution of variabilities at product compilation time. Both CONSUL and GEARS, thus, are limited to a single point in the life cycle at which variabilities can be resolved and, despite their powerful capabilities, fall short of providing any-time variability.

Perhaps closest to our work is feature engineering [26]. In this work, features, rather than product line architectures and components, are posed as the central abstraction in the software life cycle. Feature engineering shares the goal of making variability pervasive throughout the life cycle, but uses feature specifications to do so. An example feature-based configuration management system is provided, but it is unclear how the approach can be propagated to other phases in the life cycle without the availability of an intermediate representation such as our product line architecture representation.

Finally, any-time variability was first introduced as timeline variability [11]. Our solution differs from their proposed solution. First, we do not see a need to explicitly model the time at which a variability may be resolved, since our representation provides an implicit model in the form of any remaining variabilities. Second, their proposed approach relies on an advanced configuration management system. That still limits any-time variability to a few phases in the life cycle and prohibits, for instance, run-time variability. Our approach, thus, requires less modeling work but offers greater flexibility in terms of when variabilities are resolved.

## 8. Conclusions

While most approaches to variability focus on a single point in the life cycle, we introduce a novel approach that flexibly supports variability throughout the life cycle. To address any-time variability, we have developed and demonstrated an infrastructure with which variabilities can be specified at design time, but resolved at any time thereafter. The generic part of the infrastructure consists of a representation for expressing variability, a tool for specifying variability, and a tool for resolving variability. The specific parts of the infrastructure each support one activity in the software life cycle and bridge the functionality provided by our generic infrastructure to the specific artifacts of that activity.

Any-time variability represents a serious departure from existing approaches, and requires rethinking of the role and nature of variability, as well as of the software engineering environments that currently support the development of software systems. We recognize the extent to which our approach requires changes to those software engineering environments and practices (as witnessed by Section 5), but are hopeful that recent activities and reports of successful product line architecture-based approaches will lay a foundation upon which we can build. In particular, the trend of treating software as a product line [1,4,7,21] and the support tools that are being developed to support this approach [3,18,31] make us believe that any-time variability will become an integral part of these efforts.

Clearly, we are only at the beginnings of our explorations. While we successfully demonstrated the possibilities of our approach to any-time variability with the three examples discussed in Section 5, much work remains to be done. At the forefront of our efforts is the development of a configuration management system and a software deployment system that are centered on the principle of any-time variability. These two tools represent two critical

phases in the life cycle (implementation and deployment), and must be supported for an approach to any-time variability to be successful and comprehensive throughout the life cycle.

In parallel, we intend to explore “widening” of variabilities. While much of our current approach focuses on narrowing the width of the variability funnel (shown in Section 2) as one progresses through the life cycle, introduction of new variabilities at design time (whether space or time) requires the ability to propagate those variabilities forward in the life cycle. This amounts to widening the funnel of available variabilities at later stages in the life cycle, and must be supported with automated tools. We believe the differencing and merging algorithms that we briefly introduced in Section 5.3 will form the basis of our efforts in this domain and are currently extending those algorithms to address this problem .

## Acknowledgments

The author wishes to thank the following individuals who have made great contributions to the design and implementation of the presented materials: Ping Chen, Matt Critchlow, Eric Dashofy, Rob Egelink, Akash Garg, Maulik Oza, and Chris Van der Westhuizen. Without their dedication, none of the approach would have ever materialized.

Effort sponsored by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement numbers F30602-00-2-0599 and F30602-00-2-0608. Effort also partially funded by the National Science Foundation under grant numbers CCR-0093489 and IIS-0205724. 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 (DARPA), the Air Force Laboratory, or the U.S. Government.

## References

- [1] C. Atkinson, et al., *Component-based Product Line Engineering with UML*. Addison-Wesley, New York, New York, 2002.
- [2] E.C. Bailey, *Maximum RPM*. Red Hat Software Inc., 1997.
- [3] D. Beuche, H. Papajewski, and W. Schröder-Preikschat. *Variability Management with Feature Models*. Proceedings of the Workshop on Software Variability Management, 2003: p. 72-83.
- [4] J. Bosch, *Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach*. Addison Wesley, 2000.
- [5] C. Burrows and I. Wesley, *Ovum Evaluates Configuration Management*. Ovum Ltd., Burlington, Massachusetts, 1998.
- [6] R. Cardone, et al. *Using Mixins to Build Flexible Widgets*. Proceedings of the First International Conference on Aspect-Oriented Development, 2002: p. 76-85.
- [7] P. Clements and L.M. Northrop, *Software Product Lines: Practices and Patterns*. Addison-Wesley, New York, New York, 2002.
- [8] R. Conradi and B. Westfechtel, *Version Models for Software Configuration Management*. ACM Computing Surveys, 1998. 30(2): p. 232-282.
- [9] E.M. Dashofy, A. van der Hoek, and R.N. Taylor. *An Infrastructure for the Rapid Development of XML-Based Architecture Description Languages*. Proceedings of the 24th International Conference on Software Engineering, 2002: p. 266-276.
- [10] E.M. Dashofy, A. van der Hoek, and R.N. Taylor. *Towards Architecture-Based Self-Healing Systems*. Proceedings of the First ACM SIGSOFT Workshop on Self-Healing Systems, 2002: p. 21-26.
- [11] E. Dolstra, G. Florijn, and E. Visser. *Timeline Variability: The Variability of Binding Time of Variation Points*. Proceedings of the Workshop on Software Variability Management, 2003: p. 119-122.
- [12] A. Garg, et al. *An Environment for Managing Evolving Product Line Architectures*. Proceedings of the, 2003 (in submission).
- [13] R.S. Hall, D.M. Heimbigner, and A.L. Wolf. *A Cooperative Approach to Support Software Deployment Using the Software Dock*. Proceedings of the 1999 International Conference on Software Engineering, 1999: p. 174-183.

- [14] D.M. Heimbigner, R.S. Hall, and A.L. Wolf. *A Framework for Analyzing Configurations of Deployable Software Systems*. Proceedings of the Fifth IEEE International Conference on Engineering of Complex Computer Systems, 1999.
- [15] InstallShield, <http://www.installshield.com/>, 2001.
- [16] K. Kang, et al. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Software Engineering Institute, 1990.
- [17] I. Kiselev, *Aspect-Oriented Programming with AspectJ*. Sams, 2002.
- [18] C.W. Krüeger. *Variation Management for Software Production Lines*. Proceedings of the Second International Software Product Line Conference, 2002: p. 37-48.
- [19] 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, 1999: p. 44-53.
- [20] NetDeploy, <http://www.netdeploy.com/>, 2001.
- [21] L.M. Northrop. *Reuse That Pays: ICSE Keynote Presentation*. Proceedings of the 23rd International Conference on Software Engineering, 2001.
- [22] D.E. Perry. *Generic Descriptions for Product Line Architectures*. Proceedings of the Second International Workshop on Development and Evolution of Software Architectures for Product Families, 1998.
- [23] M. Shaw and D. Garlan, eds. *Software Architecture: Perspectives on an Emerging Discipline*. 1996, Prentice-Hall.
- [24] A. Speck, E. Pulvermüller, and M. Clauss. *Versioning in Software Modeling*. Proceedings of the Sixth International Conference on Integrated Design and Process Technology, 2002.
- [25] M. Svahnberg, J. van Gurp, and J. Bosch, *A Taxonomy of Variability Realization Techniques*. 2002 (in submission).
- [26] C.R. Turner, et al., *A Conceptual Basis for Feature Engineering*. Journal of Systems and Software, 1999. 49(1): p. 3-15.
- [27] A. van der Hoek, et al. *Taming Architectural Evolution*. Proceedings of the Sixth European Software Engineering Conference and the Ninth ACM SIGSOFT Symposium on the Foundations of Software Engineering, 2001: p. 1-10.
- [28] C. Van der Westhuizen and A. van der Hoek. *Understanding and Propagating Architectural Changes*. Proceedings of the Working IFIP Conference on Software Architecture (to appear), 2002.
- [29] J. van Gurp and J. Bosch, eds. *Proceedings of the Workshop on Software Variability Management*. 2003.
- [30] J. van Gurp, J. Bosch, and M. Svahnberg. *On the Notion of Variability in Software Product Lines*. Proceedings of the Working International Conference on Software Architecture, 2001: p. 45-54.
- [31] R. van Ommering. *Building Product Populations with Software Components*. Proceedings of the Twenty-fourth International Conference on Software Engineering, 2002: p. 255-265.
- [32] R. van Ommering, et al., *The Koala Component Model for Consumer Electronics Software*. Computer, 2000. 33(3): p. 78-85.