An Environment for Managing Evolving Product Line Architectures

Akash Garg, Matt Critchlow, Ping Chen, Christopher Van der Westhuizen, André van der Hoek
School of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425 USA
{agarg,critchlm,pchen,cvanderw,andre}@ics.uci.edu

Abstract

The use of product lines is recognized as beneficial in promoting and structuring both component and architecture reuse throughout an organization. While the business practices of using product lines are well-understood and representations for specifying and capturing the underlying architecture of a product line are coming of age, support environments for managing the evolution of a product line architecture are still lacking. In this paper, we present Ménage, an environment specifically designed to alleviate this problem. Key features of Ménage are its support for: (1) specifying variation points in a product line architecture as optional and/or variant elements, (2) tracking the evolution of a product line architecture and its constituent elements through explicit versioning techniques, and (3) selecting one or more product architectures out of an overall product line architecture by applying user-specified criteria. In this paper, we introduce the approach underlying Ménage, discuss its detailed functionality, and demonstrate its use with a product line architecture for entertainment systems.

1. Introduction

The use of product lines in industrial software development is steadily gaining acceptance, especially since it has been shown that their disciplined use, as backed by strong organizational commitment, can lead to significant advantages in terms of reduced development cost and time [2,4,6]. Organizations such as Nokia [18], Alcatel [22], and Philips [12] have already reported on successfully introducing product lines for some of the software they are developing. Other organizations are not far behind [23].

Instead of focusing on developing a single product at a time (or, at best, multiple, relatively independent products in parallel), the use of a product line carefully coordinates the design, development, and evolution of a set of intimately related products. As compared to component-based software development [15], this entails a paradigm shift from component reuse to architecture reuse. Component-based software development focuses on the creation of reusable component implementations that are subsequently integrated and adapted to form entirely new, often unrelated applications. The use of a product line, on the other hand, is firmly rooted in the development of a standard *product line architecture* that, along with a standard set of components implementing the core of the architecture, forms a reusable basis for the development of new, closely related members in the product line.

The issues involved in creating a development process and business environment tailored to the use of a product line architecture are relatively well understood [4]. Additionally, representations for specifying and storing product line architectures have already been developed [3,10,14,34]. Effective use of a particular product line architecture, however, also requires a support environment to manage its evolving structure—an area of research that has largely been ignored to date.

This paper introduces Ménage, an environment that is specifically designed to fill this void. Ménage builds upon our existing representation for product line architectures, xADL 2.0 [9,10], to provide a software architect with three capabilities that are explicitly geared towards managing an evolving product line architecture. First, Ménage supports the specification of a product line architecture as a set of core architectural elements that is augmented with variation points. These variation points are optional, variant, or optional variant elements, and precisely define the dimensions along which individual product architectures structurally differ from each other.

Second, Ménage uses explicit versioning techniques to track the evolution of all parts of a product line architecture. Every element, ranging from an individual interface type to the overall product line architecture (which potentially can be very large), is explicitly versioned and must be checked out before it can be modified and checked in after the modifications are complete.

Finally, Ménage allows an architect to select one or more product architectures out of an overall product line architecture. Using a user-specified set of criteria (which are expressed as name-value pairs), Ménage creates a reduced version of the original product line architecture. If all variation points are completely resolved, the result is a single product architecture; if one or more variation points remain (partially) unresolved, the result is a smaller product line architecture containing fewer product architectures. Additional selections may be performed to further

reduce the size of the product line architecture and its number of available product architectures.

The remainder of this paper is organized as follows. First, in Section 2, we discuss relevant background material in the field of product line architectures. We detail the problem of managing the evolution of product line architectures in Section 3, and introduce the approach underlying Ménage in Section 4. We describe Ménage in detail in Section 5, and show its application on an entertainment system product line architecture in Section 6. We discuss related work in Section 7 and conclude in Section 8 with an outlook at our future work.

2. Background

Software architectures provide high-level abstractions for representing the structure, behavior, and key properties of a software system [21]. These abstractions typically involve: (1) descriptions of the elements from which systems are built, (2) interactions among those elements, (3) patterns that guide their composition, and (4) constraints on those patterns. In general, a software architecture is defined as a set of *components*, a set of interconnections among those components (*connectors*), and the overall organization of the components and connectors into a single system (*configuration).*

Whereas a "regular" software architecture only defines the structure of a single software system, a *product line architecture* defines the architectural structure for a set of related products [4,6]. As such, a product line architecture consists of a set of closely related *product architectures*, each product architecture defining the software architecture of one, unique product in the product line. To maximize reuse and understanding, a product line architecture distinguishes *core elements* that are present in all product architectures from *variation points* that capture differences among specific product architectures. Three kinds of variation points are used to distinguish different product architectures from each other: (1) *optional* elements, which describe architectural elements that may or may not be present in a particular product architecture, (2) *variant* elements, which define elements that are always present, but can be configured to be one of many alternatives, and (3) *optional variant* elements, which specify variant elements that may or may not exist. A particular product architecture is selected out of a product line architecture by determining, for each optional element, whether or not it is included, and, for each variant element, which variant is incorporated.

Figure 1 introduces a simple example in the form of a hypothetical product line architecture for a set of related word processors. Solid boxes indicate core components, dashed boxes indicate variation points consisting of optional components, and stacks indicate variation points consisting of variant components. In this case, a product architecture for one particular word processor always incorporates its three core components (USER INTERFACE, LAYOUT ENGINE, and STORAGE), may or may not include the optional component PRINT, and always includes a variant of the spell checking component (ENGLISH SPELL CHECKER, DUTCH SPELL CHECKER, or FRENCH SPELL CHECKER). While the example presents a trivial product line architecture that consists of only a small set of components, one can easily imagine more complicated product line architectures consisting of many components and connectors with many complex and interrelated variation points. The product line architecture for Philips televisions is one example of a real-life product line that exhibits many of these characteristics [12].

Perry [24] outlined the space of possibilities for modeling product line architectures and observed that a product line architecture modeling technique must be both generic enough to encompass all members of the product line architecture and specific enough to provide developers with adequate support for instantiating and implementing individual product architectures. While it is technically possible to reuse architectural styles for this purpose [28], experience with product line architectures has shown a need for higher-level support in terms of explicit facilities for modeling variation points [10,34].

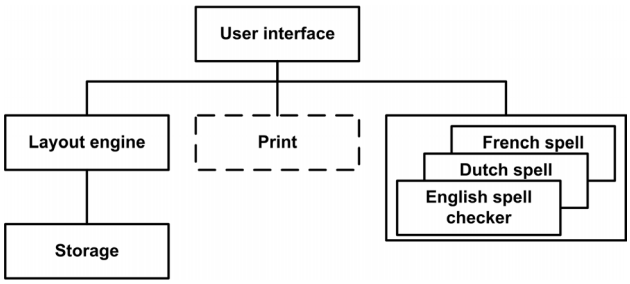

Figure 1. Example Product Line Architecture.

Architecture description languages support architecture-based development [21] by providing formal notations to describe the architecture of a software system. An architecture description language is usually accompanied by various tools for parsing, analysis, simulation, and code generation of a modeled system. Examples of architecture description languages include C2SADEL [20], Darwin [19], Rapide [17], UniCon [27], and Wright [1]. A number of these languages also provide extensive support for modeling *behaviors* and *constraints* on the properties of components and connectors [21]. However, with the exception of xADL 2.0 [10], Koala [34], GenVoca [3] and to some extent Acme [14], existing architecture description languages do not directly support the specification of product line architectures. Given the importance of prod-

uct lines in today's world of software development, we expect this situation to change rapidly.

3. Problem

It is well known that even a simple software architecture typically evolves at least somewhat over its lifetime. By the very nature of a product line architecture, it is no surprise that its overall structure changes far more frequently [5,7]: new product architectures are added, existing product architectures must be modified in response to changing functionality requirements, and defunct product architectures may have to be retired. As a result, a product line architecture finds itself in constant flux as significant parts of the product line architecture change. For instance, optional elements may turn into core elements (and vice versa), new variants may be added to a variation point, a component may be further broken down into subcomponents, wholly new product architectures may be added; or sets of existing product architectures may need to be refactored.

Two fundamental concerns for using a product line architecture, then, are how to: (1) represent and capture the evolution of a product line architecture, and (2) support an architect in managing such evolution. A number of different solutions to the first concern have been proposed, including using a generic configuration management system [34], creating a new architecture description language that explicitly integrates facilities for modeling evolving product line architectures (e.g., variation points, versions) [10], and using feature-oriented domain models [16]. Technically, most of these solutions are able to provide more-or-less equivalent kinds of functionality.

The focus of this paper is on the second concern: how to help an architect in managing the evolution of a product line architecture. Existing environments for architectural design (e.g., ArchStudio 2.0 [20], AcmeStudio [14]) provide little-to-no support for this activity. They, for example, are not equipped to handle different versions of components or connectors, provide no explicit change process, and are focused on the development of a single software architecture rather than a set of related product architectures organized in a product line architecture. The goal of the research presented in this paper is to ameliorate this problem and provide architects with a comprehensive design environment that explicitly supports them in managing evolving product line architectures.

4. Approach

The cornerstone of our approach lies in the observation that a design environment for evolving product line architectures must provide an architect with integrated architectural and configuration management functionality. Consider, for instance, an architect who wants to quickly examine a previous version of their product line architecture. The architect should not have to go to their configuration management system, check out the previous version, and then open it in their design environment. Rather, a support environment for managing evolving product line architectures must allow an architect to simply choose a version to view, with the environment itself taking care of accessing the underlying store to obtain any necessary data (regardless of whether that store is, for example, a configuration management system [34] or an architecture description language that has specific features for modeling product line architectures [10]). Numerous other examples exist of situations in which architects must simultaneously access or manipulate information that is related to both the structure and evolution of a product line architecture. Without an integrated approach, architects will not be able to effectively perform their work.

Despite the need for an integrated approach, the primary focus of any environment for product line architectures should remain on design. The primary task of architects, after all, is to design, precisely specify, and maintain product line architectures.

Overall, then, our approach is rooted in the following overarching objectives:

- An architect should be able to design a product line architecture much like they design a "regular" architecture. In particular, the familiar approach of simply combining components and connectors must be preserved.
- Variation points should be explicit within a product line architecture, yet seamlessly integrated in the design process. For instance, the difference between adding a core component and an optional component should be minimal.
- Evolution should be managed with an explicit change management process. In particular, it is important that a meaningful history of changes is created when an architect modifies a product line architecture. The change management process should be non-obtrusive to allow an architect to focus on their task at hand.
- The environment should automate as much support as possible. For instance, selection of a particular product architecture or subset of product architectures (a "smaller" product line architecture) should not require manual interpretation of variation points.

Together, these objectives create an environment for managing evolving product line architectures that is familiar and easy to use, and that provides an architect with automated and extensive support in all aspects of managing a product line architecture—ranging from initial inception, throughout many changes, to eventual selection of individual product architectures.

5. Implementation

Figure 2 presents the overall architecture of Ménage, as consisting of three components. At the lowest level, Ménage uses the xADL 2.0 libraries, which provide a programmatic interface to load and store (parts of) particular product line architectures [10]. Two components use those libraries: a design environment and a selector. The design environment component provides an architect with facilities to graphically create, inspect, and modify product line architectures. The selector component complements the design environment by providing an architect the ability to select a subset of one or more product architectures out of a product line architecture. Architecturally, we separated the design environment from the selector, since the selector by itself provides functionality that can be employed at times when the full design environment is not needed (for instance, during product selection at a customer site). Below, we discuss the details of each of the components.

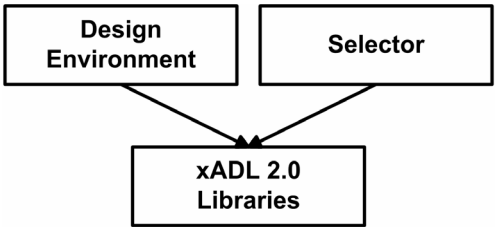

Figure 2. Ménage Architecture.

5.1 xADL 2.0 Libraries

The xADL 2.0 libraries [10] provide a programmatic interface to xADL 2.0 documents containing descriptions of product line architectures. Specifically, the libraries provide facilities to create, load, store, and modify xADL 2.0 documents. While xADL 2.0 is built as a set of extensible XML schemas, the libraries hide all XML details and allow other components (e.g., the design environment and selector) to manipulate xADL 2.0 documents in terms of architectural elements such as components, connectors, and interfaces.

The full functionality and the degree of extensibility offered by xADL 2.0, as well as its benefits as compared to other languages such as Acme [14] or Koala [34], are beyond the scope of this paper and described elsewhere [10]. Of importance here, however, are the features that it provides for modeling product line architectures. We describe these features briefly.

The core of xADL 2.0 is formed by its STRUCTURE AND TYPES schema, which defines modeling constructs for capturing a product architecture at design-time. Specifically, the schema allows the definition of the structure of one particular product architecture in terms of a set of components and connectors. Both components and connectors exhibit interfaces, which are the elements that are linked together to form the overall structure of the product architecture (e.g., two components can be "hooked up" via a connector by placing links in between interfaces on the components and interfaces on the connector). All elements are typed, and the STRUCTURE AND TYPES schema supports the specification of subarchitectures to address scalability in architectural specification.

The xADL 2.0 OPTIONS and VARIANTS schemas extend the STRUCTURE AND TYPES schema with variation points, thereby enhancing modeling support in xADL 2.0 from individual product architectures to multiple product architectures as related in a product line architecture. The OPTIONS schema allows for the definition of architectural elements that are optional in a product line architecture. Optionality is governed by a Boolean guard that determines the conditions under which the optional element should be included in a particular product architecture.

Boolean guards also form the core of the VARIANTS schema. In particular, a component (connector) type may be a "variant type", which means that it is a placeholder for a set of other component (connector) types. Mutually exclusive Boolean guards determine which type is eventually used in a selected product architecture. Of note is that optionality is dealt with at the structural level (e.g., an element may or may not be part of the structure of a product line architecture) and variability at the type level (e.g., the type of a component or connector is one of many available types). Therefore, combined optional variant elements are naturally supported by xADL 2.0.

Finally, the VERSIONS schema allows the modeling of the evolution of a product line architecture. Each type is versioned and different versions of a type are organized in a version graph. An architect, thus, can keep track of the evolution of both individual elements and the structure of the overall product line architecture.

5.2 Design Environment

The design environment is the component that an architect uses to initially specify and then maintain an evolving product line architecture. Shown 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. Instances of these types can be used to construct other types. The top panel shows the version graph of the type that is currently displayed in the main panel. Simply clicking on one of the version nodes brings up the structure for that version. Finally, the main panel is where actual design of a product line architecture takes place. Ménage provides a large number of different edit operations in support of this activity, ranging from adding components and connectors, to connecting two components via their interfaces, to creating and using subarchitectures,

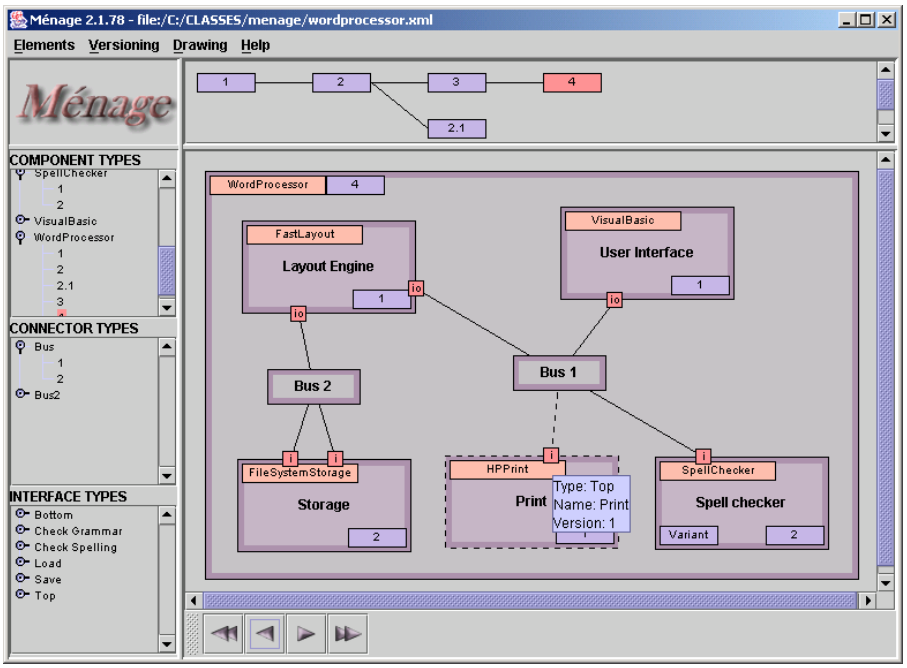

Figure 3. Specifying a New Version of a Component Type in Ménage.

and many other kinds of useful functionalities that are customary in architectural design environments.

An important aspect of Ménage is that, during editing, it always displays the type of every architectural element, both in terms of its type name and type version. Rather than relying on a default versioning model such as always using a latest version, use of specific versions of architectural elements allows an architect to precisely control the evolution of a product line architecture in terms of which versions are used, where those versions are used, and when the versions are changed. In the example of Figure 3, for instance, one can quickly discern that the architect is currently editing WORDPROCESSOR component type version 4, and that it in turn consists of instances of versions of other component and connector types (e.g., a USER INTERFACE component of type VISUALBASIC version 1, a STORAGE COMPONENT of type FILESYSTEMSTORAGE version 2, etc.). Because connectors and interfaces are visually too small to contain the same level of information, tool tips are used to provide their relevant data (as shown for the interface PRINT of interface type TOP version 1).

5.2.1 Change Process

Before any changes can be made, Ménage requires an architect to check out the set of architectural elements they will be modifying. After that, the architect is free to manipulate those elements in order to change the product line architecture. Once all desired changes have been made, the architect checks in the modified parts of the product line architecture. In response, Ménage automatically creates a new version of each element and, in the process, creates a history of changes that can be revisited over time. This history is critical in managing the evolution of a product line architecture: it captures all the changes over time, relates those changes to each other, and allows an architect to revisit previous versions to understand the nature of past changes.

During the change process, it may happen that an architect loses track of which elements they currently have checked out. Ménage, therefore, provides a mode in which it highlights those elements in a different color. Moreover, it supports an architect in checking in either a single element, an element and the hierarchically contained elements that are currently checked out, or all checked out elements. The latter two options allow an architect to check in related changes as a group.

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.

If an old version must be changed nonetheless, proper procedure requires that it is checked out again, thereby creating a branch. Version 2.1 of the WORDPROCESSOR component type is an example of such a branch. Currently, Ménage provides no support for merging branches, but we are in the process of adapting our architectural differencing and merging algorithms [32] to be able to operate on product line architectures.

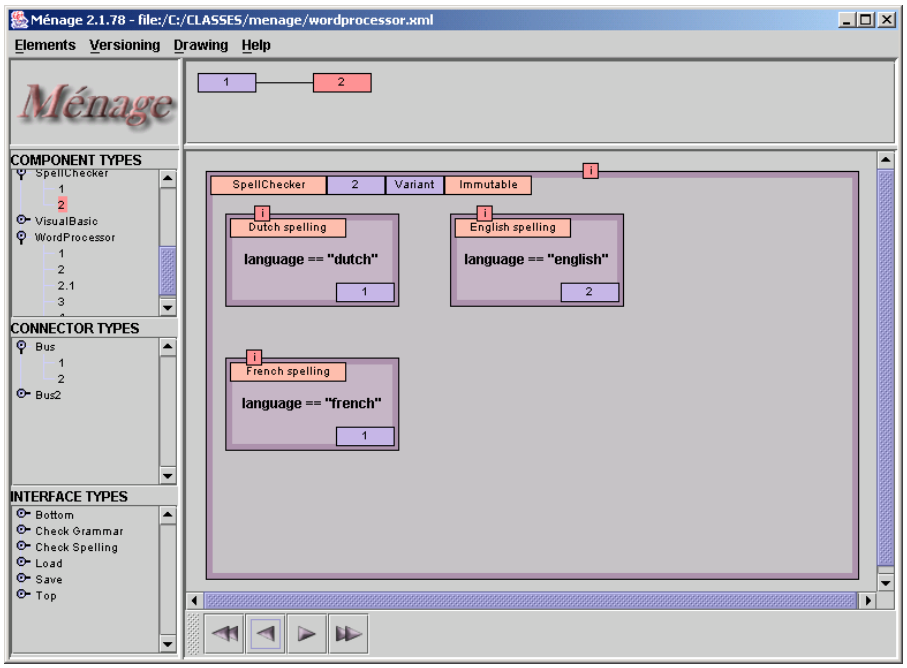

Figure 4. Viewing a Variant Component Type.

5.2.2 Variation Points

Ménage supports the specification of all three kinds of variation points: optional elements, variant types, and optional variant elements. Optional elements are added just as regular elements, simply by providing a Boolean guard at the time of creation of the element. The Boolean guard has to adhere to the following BNF:

```
<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
```

Most Boolean guards will be of a rather trivial nature. 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 other variant in another variant type by carefully matching the Boolean guards on the variants.

Graphically, optional elements are shown using dashed lines. The component PRINT in Figure 3, for instance, is an optional component. Note that, because the PRINT component is optional, its link to the connector BUS1 is automatically optional as well. If the PRINT component is included in a particular product architecture, the link is included as well; otherwise, it is left out.

Ménage treats variant types in a special way. Instead of containing a subarchitecture of components and connectors, a variant type only contains references to other types. As shown in Figure 4, references are guarded with mutually exclusive Boolean expressions to ensure that only one type can be selected at a time. The guards are used to ensure that only a single spelling checker component can be selected covering one particular language. Of note is that, in the case of the example, the interfaces on the variants are exactly the same to the interfaces on the overarching variant type. The general rule that is followed in Ménage is that interfaces may differ, but that optionality should be used to ensure compliance. For instance, suppose that the Dutch spell checker also has an interface for thesaurus functionality. Such an interface should be declared as optional at the level of the variant type, since not all variants provide this interface. This guarantees compatibility within the remainder of the product line architecture, irrespective of which variant is eventually selected.

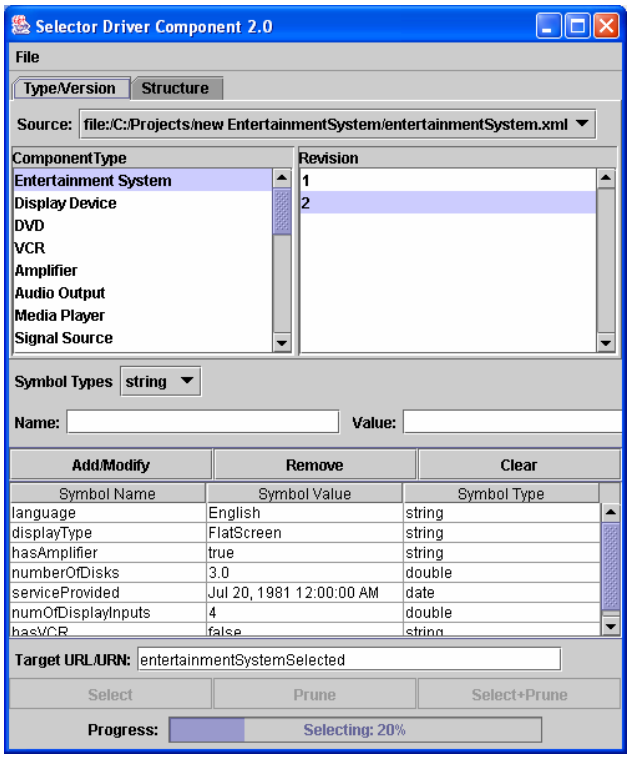

Figure 5. Selecting a Product Architecture.

When an instance of a variant component or connector type is used in a product line architecture, Ménage highlights that component or connector with a variant tag. This makes it easier for an architect to locate variation points (see, for example, the annotation of the SPELL CHECKER component in Figure 3).

Of note is that, because optionality is expressed at the level of the structure of the product line architecture and because variability is expressed using types, the two seamlessly combine to create optional variant elements. To do so, an architect adds a new instance of a variant type and annotates it with a Boolean guard that determines its inclusion. Given that individual variants may have subarchitectures, an architect should carefully establish the layers of variation points that are introduced within the product line architecture—large and highly variable hierarchies of elements may be established.

5.3 Selector

Once a number of variation points have been introduced in a product line architecture, it becomes necessary to be able to resolve those variation points in order to select one or more product architectures out of the overall product line architecture. Selection by hand can turn into an arduous task given that a product line architecture may have many variation points that each may have one or more complex Boolean expressions as guards. Therefore, Ménage includes a SELECTOR component to automate the process.

Given a set of desired properties, which are expressed as typed name-value pairs, and given a starting point in the product line architecture (e.g., the "top-level" component type from which selection should begin), Ménage iterates over the product line architecture and attempts to resolve each of the Boolean guards that it encounters. If it can fully resolve a Boolean guard to TRUE, the respective element is included. If it can fully resolve a Boolean guard to FALSE, the respective element is removed. If a Boolean guard can only be partially resolved, the element is included with the reduced Boolean guard attached. While a single selection may only result in a smaller product line architecture, iterative use of the SELECTOR will eventually result in the selection of a single product architecture.

Shown in Figure 5, the selector can operate in three different modes. In the first ("Select"), it only attempts to resolve variation points, but it does not remove any unused types or versions. In the second ("Prune"), it removes unused types and versions from a product line architecture to clean up the specification. In the third ("Select+Prune"), it combines the two in one step to minimize manual involvement. Depending on their purpose, an architect would choose a preferred mode of operation.

6. Evaluation

To evaluate Ménage, we used it to create and evolve an example product line architecture. Often, actual product line architectures are considered important organizational assets that cannot be shared. Based on limited information available on an existing product line architecture for consumer electronics [33], we attempted to create a representative but hypothetical example of a software product line architecture for a highly customizable entertainment system. The result of our efforts is shown in Figure 6. The product line architecture consists of 25 component types, 3 connector types, and 3 interface types, all available in a number of different versions. The top level element, the ENTERTAINMENTSYSTEM, is hierarchically constructed out of many other components, some of which exhibit further subarchitectures (as indicated by the small triangles in the lower left corner). Numerous variation points exist in the product line architecture, guarded by a number of different Boolean guards.

Our evaluation focused on how well Ménage achieves the four objectives listed in Section 4. We first examined whether we were able to create a product line architecture much like one creates an architecture in an environment such as ArchStudio [20] or AcmeStudio [14]. For simple architectures, Ménage operates exactly like those environments. Only when an architect must capture evolution or specify a variation point, Ménage incurs overhead for

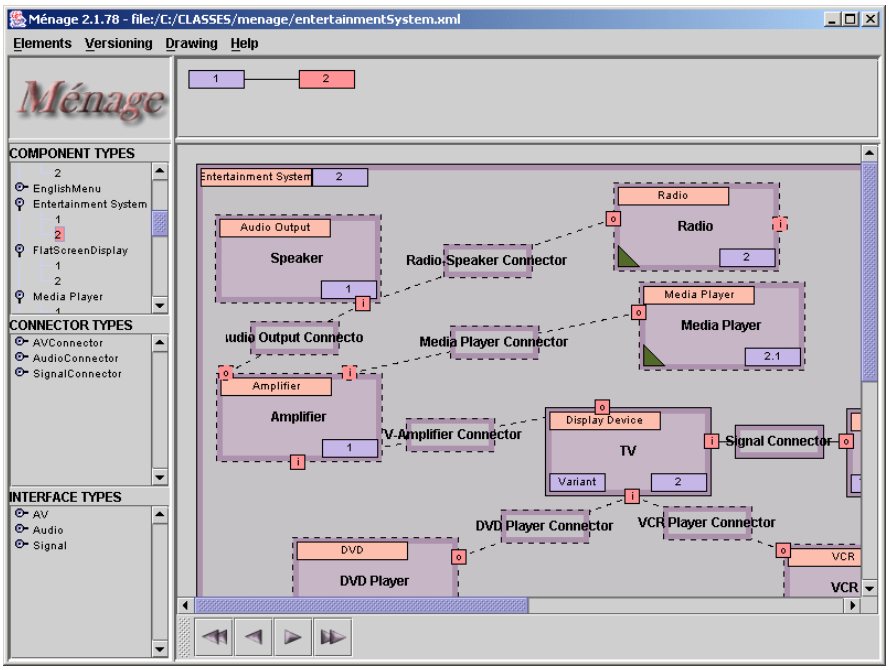

Figure 6. Ménage Applied to the Entertainment System Example.

the architect. Overhead is limited to a few actions, except in the case of check out: an architect currently must manually check out, one by one, all the elements they intend to modify. This clearly is cumbersome, and will be addressed in an upcoming version of Ménage (see below).

The second objective states that variation points should be explicit within a product line architecture, yet seamlessly integrated in the design process. Based on creating the ENTERTAINMENTSYSTEM product line architecture, we believe we have succeeded in achieving this goal: optional, variant, and optional variant elements are clearly identified in a product line architecture, yet easily incorporated in much the same way regular components and connectors are specified.

The third objective pertains to managing evolution: it should be governed by an explicit change management process. Ménage provides such a process with its check out and check in mechanism. Use of these two simple operations creates a historical archive of all previous versions of all architectural elements, regardless of whether the element is a simple interface type or the complete product line architecture.

The last objective is that Ménage should automate as much of its support as possible. Our experience in modeling the example product line architecture shows that we have achieved that. The selector component is perhaps the chief example: based on simple input from an architect, it automatically selects the desired subset of product architectures. As mentioned above, the check out operation is an exception: to reduce the manual effort of checking out each and every element to be modified, we will develop a version of Ménage that automatically and saliently checks out an element when an architect starts changing it. This should alleviate much of the burden imposed by the current change process.

7. Related Work

The work presented in this paper draws from a number of research areas. Within the domain of software architecture, perhaps the two most closely related technologies are Koala and Acme. Koala [33,34] is an architecture description language specifically designed for modeling product line architectures and, as such, shares many of its features with Ménage. Compared to Ménage, however, Koala does not include a versioning mechanism to capture the evolution of a product line architecture. Instead, Koala relies on an external configuration management system to version its architectural descriptions. While a viable alternative, this strategy prevents the incorporation of multiple versions of a single component in a single product architecture. An additional drawback of Koala is that its variability is largely code-based and resolved at compile-time of a particular product; our Selector component provides this capability at the level of product line architectures.

Acme [14], as supported by AcmeStudio environment, is based on a rather different mechanism to capture product line architectures. Instead of providing specific language features, Acme is based on the use of constraints to model all sorts of concepts, including styles, component and connector types, and product lines. While this provides the advantage of an architect having to know only a

few language constructs, it has the distinct disadvantage that it becomes difficult to conceptually separate logically different parts of an actual product line architecture specification. Especially when the system to be modeled is large, this rapidly becomes a serious problem.

UML [26] is a powerful modeling language that sometimes is proposed as a vehicle for modeling software architectures. Unfortunately, support for versioning individual UML elements (or even whole UML diagrams) and for expressing variant elements are still in their infancy. These limitations often result in clumsy endeavors relying on external tools. Perhaps even more problematic is that UML is a less than optimal solution for modeling software architectures (and thus product line architectures). Its features, even when extended specifically for modeling software architectures, have been demonstrated to prevent the accurate modeling of some architectural concepts [25].

Feature-oriented domain analysis (FODA) is an area of research that has produced models that are very similar to product line architectures [16]. Instead of representing architectural elements, however, FODA models represent features that may or may not be present in a software system. Not surprisingly, FODA models include support for the various types of variation points. FODA, however, still seems to be in the phase of finding proper languages to represent features and the authors are not aware of any extensive support environment for specifying particular FODA models, nor are they aware of any FODA-based approaches that account for the presence of multiple versions—a key feature underlying Ménage.

Finally, our work is related to many contributions in the field of configuration management [8]. In particular, configuration management system models such as Adele [13] and Proteus PCL [29] provide similar mechanisms for modeling variation points within software configurations. While borrowing concepts from these system models, our approach is oriented at product line architectures and, as such, is rooted in architectural concepts that are not addressed by the field of configuration management.

8. Conclusions

This paper has presented Ménage, an environment for managing the evolution of product line architectures. Ménage is unique in being a graphical environment that provides an architect with the ability to specify and evolve a product line architecture as new product architectures are added, existing product architectures are modified, and obsolete product architectures are removed. Key to the functionality of Ménage is its tight integration of architectural design functionality (to manage the structure of a product line architecture) with configuration management functionality (to specify variation points and manage the evolution of a product line architecture).

We have already embarked on three research directions in efforts to further enhance the functionality of Ménage. First, we are examining the role that architectural differencing and merging may play in propagating changes across multiple product architectures as well as branches. Currently, an architect has to manually restructure a product line architecture to do so, but we intend to adapt our existing architectural differencing and merging algorithms (which only operate on single architectures [32]) to be able to operate on product line architectures.

Our second research effort aims to support an architect in understanding the structure of a product line architecture. After many changes, the overall structure generally has disintegrated and the “clean” design picture that once existed has deteriorated. In addition to exploring how design critics [11] and analysis techniques help in maintaining a consistent product line architecture [31], we are investigating how metrics that calculate the utilization of the functionalities provided by components in a product line architecture [30] can provide an architect with graphical visualizations that highlight potential structural problems in the product line architecture. Typically, these problems indicate a need for refactoring of elements, for instance splitting a particular variant in a “smaller” variant and an optional element containing the rest of the functionality.

Finally, we observe that a realization of the full power of product line engineering requires a careful mapping from the product line architecture to actual source code (components). Maintaining such a mapping is a difficult endeavor due to architectural erosion. We intend to develop a product line architecture-aware configuration management system to aid in maintaining such a mapping.

Availability

Ménage can be downloaded from http://www.isr.uci.edu/projects/menage/.

Acknowledgements

The authors thank Eric Dashofy for his valuable contributions to the development of Ménage and Rob Egelink for the implementation of many concepts in Ménage.

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] R. Allen and D. Garlan, *A Formal Basis for Architectural Connection*. ACM Transactions on Software Engineering and Methodology, 1997. 6(3): p. 213-249.

[2] C. Atkinson, et al., *Component-based Product Line Engineering with UML*. Addison-Wesley, New York, New York, 2002.

[3] D. Batory and B.J. Geraci, *Composition Validation and Subjectivity in GenVoca Generators.* IEEE Transactions on Software Engineering, 1997. 23(2): p. 67-82.

[4] J. Bosch, *Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach*. Addison-Wesley, New York, New York, 2000.

[5] J. Bosch, et al. *Variability Issues in Software Product Lines*. Proceedings of the Product Family Architecture Workshop, 2001: p. 13-21.

[6] P. Clements and L.M. Northrop, *Software Product Lines: Practices and Patterns*. Addison-Wesley, New York, New York, 2002.

[7] P.C. Clements and N. Weiderman. *Report on the Second International Workshop on Development and Evolution of Software Architectures for Product Families*. Software Engineering Institute, 1998.

[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. *A Highly-Extensible, XML-Based Architecture Description Language*. Proceedings of the Working IEEE/IFIP Conference on Software Architecture, 2001.

[10] 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.

[11] 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.

[12] F. de Lange and T. Jansen. *The Philips-OpenTV Product Family Architecture for Interactive Set-Top Boxes*. Proceedings of the Product Family Architecture Workshop, 2001: p. 177-190.

[13] J. Estublier and R. Casalles, *The Adele Configuration Manager*, in Configuration Management, W.F. Tichy, Editor. 1994: p. 99-134.

[14] D. Garlan, R. Monroe, and D. Wile, *ACME: An Architecture Description Interchange Language*, in Proceedings of CASCON'97. 1997.

[15] G.T. Heineman and W.T. Councill, eds. *Component-Based Software Engineering: Putting the Pieces Together*. 2001, Addison-Wesley: Reading, Massachusetts.

[16] K. Kang, et al. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Software Engineering Institute, 1990.

[17] D.C. Luckham and J. Vera, *An Event-Based Architecture Definition Language.* IEEE Transactions on Software Engineering, 1995. 21(9): p. 717-734.

[18] A. Maccari and C. Riva. *Architectural Evolution of Legacy Product Families*. Proceedings of the Product Family Architecture Workshop, 2001.

[19] J. Magee and J. Kramer. *Dynamic Structure in Software Architectures*. Proceedings of the Fourth Symposium on the Foundations of Software Engineering, 1996: p. 3-14.

[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. 1999: p. 44-53.

[21] N. Medvidovic and R.N. Taylor, *A Classification and Comparison Framework for Software Architecture Description Languages.* IEEE Transactions on Software Engineering, 2000. 26(1): p. 70-93.

[22] J. Mellado and J.C. Duenas. *Automated Validation Environment for a Product Line of Railway Traffic Control Systems*. Proceedings of the Product Family Architecture Workshop, 2001: p. 389-397.

[23] L.M. Northrop. *Reuse That Pays: ICSE Keynote Presentation*. Proceedings of the 23rd International Conference on Software Engineering, 2001: p. 667.

[24] 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: p. 51-56.

[25] J.E. Robbins, et al. *Integrating Architecture Description Languages with a Standard Design Method*. Proceedings of the 20th International Conference on Software Engineering, 1998: p. 209-218.

[26] J. Rumbaugh, I. Jacobson, and G. Booch, *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1998.

[27] M. Shaw, et al., *Abstractions for Software Architecture and Tools to Support Them.* IEEE Transactions on Software Engineering, 1995. 21(4): p. 314-335.

[28] M. Shaw and D. Garlan, eds. *Software Architecture: Perspectives on an Emerging Discipline*. 1996, Prentice-Hall.

[29] E. Tryggeseth, B. Gulla, and R. Conradi. *Modelling Systems with Variability Using the PROTEUS Configuration Language*. Proceedings of the International Workshop on Software Configuration Management: ICSE SCM-4 and SCM-5 Workshops Selected Papers, 1995: p. 216-240.

[30] A. van der Hoek, E. Dincel, and N. Medvidovic. *Using Service Utilization Metrics to Assess the Structure of Product Line Architectures*. Proceedings of the Ninth International Software Metrics Symposium, 2003 (to appear).

[31] 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.

[32] C. Van der Westhuizen and A. van der Hoek. *Understanding and Propagating Architectural Changes*. Proceedings of the Working IFIP Conference on Software Architecture, 2002: p. 95-109.

[33] R. van Ommering. *Building Product Populations with Software Components.* Proceedings of the Twenty-fourth International Conference on Software Engineering, 2002: p. 255-265.

[34] R. van Ommering, et al., *The Koala Component Model for Consumer Electronics Software.* Computer, 2000. 33(3): p. 78-85.