Capturing Product Line Architectures

André van der Hoek
Institute for Software Research
Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425 USA

andre@ics.uci.edu

Abstract

Although product line architectures serve an increasingly important role in the software development process, no representation currently exists in which they can be precisely captured. This position paper presents our work to date in addressing this problem. Specifically, we introduce a novel representation that can be used to specify the details of a product line architecture. The representation not only captures the variability that exists within a product line architecture, but also captures the evolution of the interfaces, components, and connections that constitute a product line architecture.

Introduction

In moving from the development of a single product at a time to a model in which a family of products is developed in a coordinated fashion, software development organizations have recognized the increased importance of maintaining a product line architecture. Benefits of doing so have already been demonstrated in a number of settings, and the amount of research and development dedicated to the topic illustrates that it is not just a passing fad.

It is surprising, however, that one of the major issues in the use of product line architectures remains virtually untouched: how do we accurately describe and capture a product line architecture? With the exception of Koala [5], none of the architecture description languages developed to date has any special provisions to accommodate product line architectures. Precisely capturing the details of a product line architecture remains a problem.

This position paper introduces Ménage, a project that addresses this problem. Ménage consists of two parts. The first part is a definition of a representation for product line architectures. The definition not only addresses the variability that may exist in a product line architecture, but also addresses the evolution of the interfaces, components, and connections that constitute a product line architecture. The second part of Ménage is an environment that facilitates the specification—in the defined representation—of product line architectures. The environment allows graphical input and contains several analysis tools that can verify the soundness of a product line architecture.

The remainder of this paper is organized as follows. The definition of the representation is introduced first, followed by a brief discussion of the Ménage environment. After that, related work is discussed and a brief outlook on future work concludes the paper.

Representation

Figure 1 presents our definition of a representation for product line architectures. The representation is based on the traditional representations employed by many architecture description languages (an overview of which is presented in [2]) and includes the concepts of interfaces (ports or roles), connections (connectors), and components. To manage product line architectures, however, the set of concepts in the representation also incorporates variants, revisions, and options. These concepts, borrowed from the field of configuration management, suffice to be able to express the variability and evolution that is typically present in a product line architecture.

ComponentType	VariantComponentType
name revision {interface [, optionalPropName, optionalPropValue]}* {component [, optionalPropName, optionalPropValue]}* {connection [, optionalPropName, optionalPropValue]}* behavior constraints representation {propName, propValue}* ascendant {descendant}*	name revision {interface [, optionalPropName, optionalPropValue]}* variantPropName {component, variantPropValue}* representation {propName, propValue}* ascendant {descendant}*
ConnectionType	VariantConnectionType
componentType	variantComponentType
InterfaceType	
name revision representation ascendant {descendant}*	
Component	Connection
name componentType	variantComponentType	name {sourceInterface [, myDestinationInterface]}* {mySourceInterface [, destinationInterface]}* connectionType	variantConnectionType
Interface	
name direction interfaceType	

Figure 1: Representation for Product Line Architectures.

Although the full details of the definition are beyond the scope of this paper ([4] contains a full description), several important observations are in place.

- **The representation is based on types and instances.** All elements (e.g., interfaces, connections, and components) are defined by types. Instances of these types are used, in turn, to create higher-level types. In addition to facilitating the inclusion of duplicate instances in a single product line architecture, this mechanism facilitates the construction of a product line architecture in a hierarchical fashion.
- **Variability is captured in special variant types.** These variant types, only existing for connections and components since interfaces are used as elementary building blocks, consist of sets of related instances. Each set shares the same external definition (e.g., all instances can be used interchangeably), but each instance has a different internal implementation (e.g., all instances achieve their behavior in a different manner). Based on a property matching algorithm that is guided by the property values set in higher-level elements in the product line architecture, only one of the instances is chosen as the variant to be actually incorporated in a certain configuration.
- **All types are versioned.** Since variability is not the only way in which a product line architecture evolves [1], each of the types in the definition of Figure 1 has an associated revision number. The resulting set of revision numbers are related in a version tree that represents the evolution of a type. As a result, each instance of a type is in fact an instance of a specific revision of that type. Note that variant types are versioned as well, since variants can evolve when, for example, new alternatives are added or old alternatives are removed.
- **Types may have optional parts.** As observed by the authors of Koala [5], one of the crucial aspects of a product line architecture is the presence of optional parts. Depending on the particular set of instances in which a product line architecture is configured, certain interfaces, connections, or components may or may not be present. Treating optionality in the same way as we treat variants, instances are guarded by a property value that determines whether or not the instance is part of a chosen configuration.

Two other important characteristics of the representation defined in this paper are that it is language independent and orthogonal. Language independence is achieved by an opaque representation of such details as the behavior and constraints of components or connections. These can be specified in any of the architecture description languages developed to date. Given a desired set of properties, a selection algorithm can then generate a complete architectural configuration that represents an instance of the product line architecture in the language that was used to specify the behavior and constraints.

Orthogonality exhibits itself in the fact that all concepts are treated in an independent manner. Specifically, composition, revisions, variants, and options are all orthogonal in that a designer of a product line architecture has to worry about only one of these concepts at a time.

Environment

It cannot be expected that developers specify product line architectures by hand. Therefore, we have constructed an environment that supports the graphical specification of product line architectures. Shown in Figure 2, this environment separates out the various concepts much like the representation itself. The left side of the environment displays all revisions of all available types. In the top pane the environment shows the version tree of the interface, connection, or component type that is currently being manipulated in the main pane of the environment. As an example, the figure presents the construction of revision 2 of variant component type Optimizer. This revision is being constructed out of instances, called slowOpt and fastOpt, of specific revisions of the component types SlowOptimizer and FastOptimizer, respectively. The variant property is speed, which should be set to the
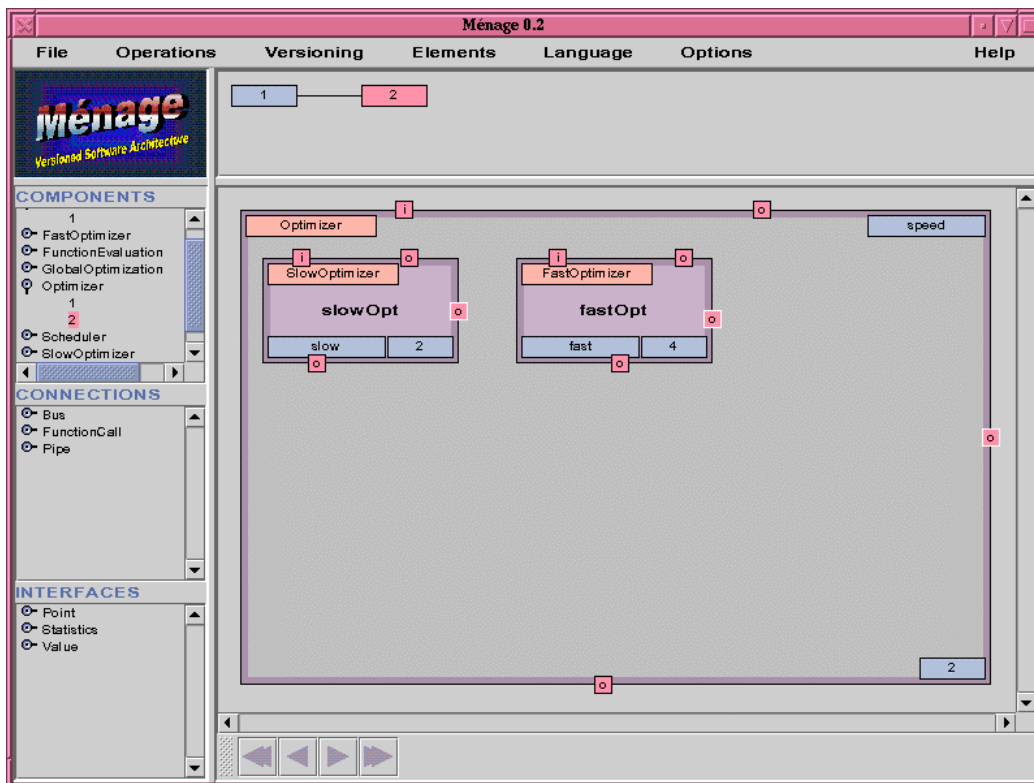

Figure 2. Specifying Variability in Ménage.

value slow for the instance slowOpt to be selected and to fast for the instance fastOpt to be selected. Note that, because a variant component type is being defined, the interfaces of Optimizer, slowOpt, and FastOpt are exactly the same.

An important part of the environment is the analyses that are provided. In particular, it is possible to verify a particular product line architecture for undefined properties, for properties that conflict between elements that are descendants of each other, and for properties that conflict between elements that reside in different parts of the hierarchy.

Related Work

With the exception of Koala [5], our work is unique in specifically targeting product line architectures. Two important differences exist between our representation and the one used by Koala. First, our representation captures the evolution of entities in the product line architecture. Linear evolution and branching are supported, both of which are aspects of product line architectures that Koala ignores. Second, our representation recognizes variability as a separate type of connection or component, whereas Koala "hides" variability inside the definitions of components. Although the net effect to be achieved is the same, the separation of concerns provided in our representation allows variant components to evolve much like regular components.

Conclusions

In this position paper we have briefly introduced Ménage, a project geared towards creating a representation and environment for product line architectures. Much work still remains to be done. Two issues are at the forefront. First, we intend to integrate Ménage with a dynamic architecture framework such as C2 [3] in order to allow reconfiguration—within product line boundaries—in the field. Second, it seems that Ménage is ideally suited to formally capture the architecture of multi-version systems, a capability that current architecture description languages lack.

Acknowledgements

References

[1]   J. Kuusela. Architectural evolution. In *Proceedings of the First Working IFIP Conference on Software Architecture*, pages 471—478, Boston, Massachusetts, February 1999. Kluwer Academic.

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

[3]   P. Oreizy, N. Medvidovic, and R.N. Taylor. Architecture-based runtime software evolution. In *Proceedings of the 1998 International Conference on Software Engineering*, pages 177—186, Los Alamitos, California, May 1998. IEEE Computer Society Press.

[4]   A. van der Hoek, D.M. Heimbigner, and A.L. Wolf. Capturing architectural configurability: variants, options, and evolution. Technical Report CU—CS—895—99, Department of Computer Science, University of Colorado, Boulder, Colorado, December 1999.

[5]   R. van Ommering, F. van der Linden, J. Kramer, and J. Magee. The Koala component model for product families in consumer electronics software. *Computer*, 33(2):78—85, March 2000.