Maintaining Consistent, Minimal Configurations

Dewayne E. Perry

Software Production Research, Bell Laboratories
600 Mountain Avenue, Murray Hill, NJ 07974 USA
dep@research.bell-labs.com

Abstract. An important ingredient in meeting today's market demands is the ability to respond quickly to a customer's changing needs. One way of responding to this pressure is to capitalize ones' software assets by means of generic, domain—specific, or product line architectures. Within that context, an extremely important aspect of meeting those changing needs is the ability to dynamically reconfigure a system rather than resorting to the current individual customization. Given cost pressures, it is useful to be able to do this reconfiguration as efficiently as possible, loading and keeping only what is needed. We provide algorithms for adding only the minimal and removing the maximal sets of components. We also note that in addition to semantic consistency of configurations, we must now also consider the architectural consistency of the evolving configurations.

1 Introduction and Context

One of the major trends in building modern software–based systems is that of domain–specific, generic, or product–line architectures [1]. The underlying reason for this trend is the need to capitalize on both hardware and software assets. Defining a generic, domain–specific, or product–line architecture enables the developers to define both the range of products buildable from that architectural style [10] and the assets needed for their implementations.

These assets are divided into three kinds: those which are unique to each architectural instance, those for which the implementation is shared across all architectural instances and those for which the shape is shared across all architectural instances. In the last case, it is the definition of the component's interface and behavior that is important even though its implementations may vary from instance to instance. It is this set of components with which we build the various architectural instances.

In this paper we look specifically at aspects of a generic architecture and consider the range of the parameters that are needed within the applicable range of products. Currently each product is built individually according to customer specifications. When the customer's needs change and the specific products need to be changed to reflect those changes in requirements, a new product has to be built for those altered specifications.

The goal, then, is to build a generic set of products each of which may be individualized according to customer requirements and altered dynamically as customer needs change. From the hardware point of view, the various interchangeable components must be standardized and plug compatible. From the software point of view, there are two interdependent things needed to achieve this goal: dynamic (re–)configuration and a distribution independent generic software architecture.

Underlying both of these is the need for an object request broker (ORB) [6]. An ORB provides the necessary substrate for dynamically changing the software configuration: the new components can be dynamically registered to replace those components that are no longer used. The ORB also provides us with the means of making our architecture distribution independent since it hides the locations of the various components and acts as the intermediary in the communication among the various software components.

An important consideration in implementing dynamic (re–)configuration is the system cost. To keep these costs as competitive as possible the hardware components will be those that are just sufficient to support the software driving the products. This means that, among other things, that system storage space is likely to be limited to just that needed for each class of instances in the generic architecture.

Because of this storage limitation we cannot afford to accumulate software components that have been replaced in reconfiguring the system and thus are no longer used. We need a way to maintain a configuration of only those components that are needed for the current configuration. Moreover, to minimize the time needed for dynamic reconfiguration, we want to load only those components that are needed but not already available in the system.

In the remainder of the paper, we present the algorithms for removing components no longer needed and for loading only those components that are not already present in the current configuration. In short, we present a way of maintaining the minimal configuration for any defined set of architectural components. We also note that in addition to the necessity of maintaining the semantic consistency of the new configuration we must maintain the architectural consistency. Finally, we summarize the results of our work.

2 Minimal Configurations

The basis for both the addition and removal of reconfigured components is a dependency graph. We first discuss the dependencies that we exploit and then present the minimal addition of components for reconfiguration and the maximal removal of components not longer in the new configuration.

2.1 Component Dependencies

Let the set *Comp* be the set of all components that can possibly be used to construct an architectural instance of our generic architecture. The set *Dep* is a set of tuples (a, b) where $a \in Compandb \in Comp$ — that is, a is dependent on b. Thus, Dep is the subset of $Comp \times Comp$ that represents the dependencies of each element in $Comp$. For example, in a system built using **C**, the dependencies are defined by the **#include** statements.

Let $TC(c_i)$ where $c_i \in Comp$ be the transitive closure of c_i in Dep — that is, $TC(c_i)$ is the transitive closure of dependencies defined in Dep beginning with component c_i.

For convenience, let us consider the set of basic architectural components, $AComp$, as bases for defining configurations. Since they are in $Comp$ and their dependencies are defined in Dep, we suffer no loss of generality in the ensuing discussion.

An architecture configuration AC is a set $\{ac_1 ... ac_n\}$ where $ac_i \in AComp$ — that is, a configuration is a set of architectural components. A build configuration BC is the union of the transitive closures of each element in an architectural configuration — $BC(AC) = \bigcup_{i=1}^{n} TC(ac_i)$ where $ac_i \in AC$

2.2 Minimal Addition

Given an architectural configuration $Current$ and a new architectural configuration New to be added to configuration $Current$, the following is the algorithm for determining the minimal set of components to dynamically add to the existing system. Note that the addition of a single component reduces to an architectural configuration with one element.

$$MinimumAdd = BC(New) - BC(Current)$$

$$Current = Current \cup New$$

Thus, $MinimumAdd$ is the set of components that are needed in the new build configuration determined by the architectural configuration New that are not already in the build configuration determined by $Current$.

2.3 Maximal Removal

Given an architectural configuration $Current$ and an architectural configuration $Remove$ to be removed from the configuration $Current$, the following is the algorithm for determining the maximal set of components to dynamically remove from the existing system. Note that the removal of a single component reduces to an architectural configuration with one element.

$$MaximalRemove = BC(Remove) - BC(Current - Remove)$$

$$Current = Current - Remove$$

Thus, $MaximalRemove$ is the set of components that are no longer needed in the build configuration determined by the architectural configuration $Current - Remove$.

2.4 Replacement

For replacing one architectural configuration for another, simply do the addition first to get the minimal set of build components to add to the system and the removal second. This ordering of operations is needed to avoid removing and adding the same component.

3 Architectural Consistency

In previous work, we concentrated on the semantic consistency of building and evolving configurations from components [7] [8] [9]. In addition to having to reason about the semantic consistency of the dynamic reconfigurations, we must also reason about the architectural consistency of the instances with respect to the specified generic architecture.

Perry and Wolf [10] proposed a model for software architecture that we use to consider the notion of consistency between a generic architecture and its various instances.

$$\text{Software Architecture} = \{ \text{ Elements, Form, Rationale } \}$$

Architectural elements are either processing, data or connecting elements; the form specifies the properties and relationships of and among these elements; and the rationale provides the justification for both the set of elements and the form of the architecture.

We consider a generic architecture to be an architectural style — that is it is an abstraction from a set of specific architectures. A typical approach in defining an architectural style abstracts various kinds of elements by defining their properties and relationships.

We note the following different kinds of formal aspects that may be used to define an architectural style:

- properties of individual elements
- relationships among elements
- constraints on properties
- constraints on relationships

It is these properties, relationships and constraints that must be maintained by each instance of the desired generic architecture. Some examples of work that utilizes constraints for various forms of reasoning about architectures are [5] [2] [3] [4].

4 Summary

We have provided an approach that minimizes the components that must be added and maximizes the components to be removed in evolving a configuration dynamically. The current implementation consists of the use of shell scripts and the sort and join operations. This approach is not very effecient but for the time being works quite satisfactorily. Should a more efficient approach be needed, we have the design for an approach in which we generate a set of programs that are then compiled and executed.

References

1. David Garlan and Dewayne E. Perry. “Introduction to the Special Issue on Software Architecture”, *IEEE Transactions on Software Engineering*, 21:4 (April 1995).
2. Jeff Magee and Jeff Kramer. “Dynamic Structure in Software Architectures”, *Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, San Francisco, CA, October 1996. pp. 3-14
3. Daniel Le Metayer. “Software Architecture Styles as Graph Grammars”, *Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, San Francisco, CA, October 1996. pp. 15-23
4. Nenad Medvidovic, Payman Oreizy, Jason E. Roberts and Richard N. Taylor. “Using Object–Oriented Typing to Support Architectural Design in the C2 Style”, *Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, San Francisco, CA, October 1996. pp. 24-32.
5. Naftaly H. Minsky. “Law–governed systems”, *The IEE Software Engineering Journal*, September 1991.
6. Object Management Group. “The Common Object Request Broker: Architecture and Specification”, Revision 2.0, July 1995, available from http:www.omg.orgcorbask.htm.
7. Dewayne E. Perry. “Software Interconnection Models”, *Proceedings of the 9th International Conference on Software Engineering*, Monterey, CA, March 1987. pp 61-69.
8. Dewayne E. Perry. “Version Control in the Inscape Environment”, *Proceedings of the 9th International Conference on Software Engineering*, March 30 - April 2, 1987, Monterey CA.
9. Dewayne E. Perry, “System Compositions and Shared Dependencies”, *Proceedings of the ICSE’96 SCM–6 Workshop*, Berlin Germany, March 1996.
10. Dewayne E. Perry and Alexander L. Wolf. “Foundations for the Study of Software Architecture”, *ACM SIGSOFT Software Engineering Notes*, 17:4 (October 1992).

This article was processed using the LaTeX macro package with LLNCS style