Dimensions of Consistency in Source Versions and System Compositions — An Extended Abstract

Dewayne E. Perry
Software and Systems Research Center
AT&T Bell Laboratories
Murray Hill, NJ 07974

1. Introduction

Among the various issues that must be addressed in the building of software systems, there are two issues that fall within the province of version and configuration management: component derivation and component consistency — that is, recording or determining the derivation of one component from another, and determining whether the components within a particular configuration are consistent with each other.

Much of the past and current work in version and configuration management has addressed the problem of keeping track of how components are derived. We have systems that manage version and configuration histories — for example SCCS [8], RCS [10], NSE [1], etc — by effectively keeping either a tree or a graph representing the derivation history of source versions. We have various tools that provide automated derivation of secondary objects[1] — for example, various forms of Make [3], etc, or such opportunistic processors as Marvel [5] — to help us build executable versions of our systems. In general, we have a fairly deep understanding of the issues in managing the derivation relationship both for manually derived components as well as automatically derived components. For example, see Borrison [2] for a discussion of these latter issues.

1. Secondary only in the sense that we have automated means of deriving them from objects that require manual construction.

The second concern has received much less attention. In general, we use basic system development tools rather than version and configuration-specific tools for purposes of determining consistency and other relationships between components. For example, we use compilers to determine general syntactic consistency, linking loaders to determine the general completeness of a system composition, and testing to determine the fine and large-grained semantic consistency of that composition.

The primary drawback of using these general development tools is that they do not provide a sharp focus on those problems of consistency that are endemic to version and configuration management. Moreover, they do not exploit any of the existing relationships that could be used by domain-specific tools.

It is the purpose of this paper to present what we consider to be the dimensions of consistency for both source and composed versions of components in building software systems.

2. The Dimensions

There are two basic problems that motivate our interest in the consistency of atomic (that is, source modules) and composed components: that of putting them together so that the resulting system is consistent, and that of substituting one component for another in an existing composition in such a way that consistency is preserved.

To accomplish the initial composition and the subsequent substitution, we need to be able to reason about the various aspects of components and compositions. For

this reasoning process, we need to consider relationships that are richer than those we currently use in keeping track of historical derivation or for automatic derivation of secondary objects. The need for these richer relationships has been realized in a rather primitive fashion in that we have overloaded our historical derivation relationships with connotations beyond what the concepts can sustain: we tend to think of successive versions as refinements (more specifically, improvements) of basically equivalent versions and parallel versions as alternative, but equivalent, variants. Neither of these interpretations represents what really happens in a typical derivation history. Some successive versions are not even compatible much less equivalent to the preceding version. Inferences about parallel versions are equally suspect.

We propose three interdependent dimensions to be considered as basic to reasoning about components and compositions:

- well-formedness of compositions — that is, that the provided and required facilities of components and compositions are syntacticly consistent and that compositions are constructed properly;
- small-grained semantic consistency — that is, that interfaces of components and compositions are used in a consistent manner; and
- large-grained semantic consistency — that is, that shared dependencies in various forms are resolved in a consistent manner.

2.1 Well-Formed Compositions

The well-formedness of system compositions from components that meet basic constraints (about the provided and required facilities in the syntactic interface of the component) is the foundation for building syntactically and semantically consistent systems (see Habermann and Perry [4] for a complete discussion of the various aspects of well-formed systems). Within this context, there are syntactic relationships that are important for successful substitution of one component for another:

- At the facilities level (that is, the syntactic objects in a module) *extensions* can be made to the declarations of facilities that preserve *syntactic compatibility* of the new module with the old. These may be either *strict* or *permuted* extensions: strict extensions add new fields or new parameters to structures and operations without altering the order of the existing fields or parameters; permuted extensions add new fields or parameters and do alter the the order of those fields or parameters. Within certain constraints, strict extensions are substitutable without affecting the consistency of the composition or causing recompilation. Permuted extensions to structures are permissible but require recompilation. Permuted extensions to parameter lists require both named parameter support in the language (for example, as in Ada) and recompilation in order to be safely substitutable within the system. Note, however, that in this latter case, existing data may need to be transformed to remain consistent with the permuted extensions.
- At the module level, extensions by means of additional facilities are always substitutable as long as these extensions do not require additional facilities (see Tichy's definition of *upward compatibility* [9]).
- At the system level, we can relax the rule about no new required facilities for module extensions and define a notion of *system compatibility* in which a module is allowed additional required facilities as long as they are already required by the system or are provided internally by the system.

2.2 Semantics of Interfaces

The syntactic level of reasoning does not account for objects (or versions of objects) that may have identical syntactic interfaces but entirely different semantics. It is for this reason that we must also consider fine-grained semantic consistency issues. One example of this type of approach is found in the interface specifications and static analysis supported by the Inscape Environment [7]. Formally defined predicates provide the fine-grained semantic dependencies and interconnections with which we can formally define semantic relationships and reason

about them in both constructing compositions and in substituting one component for another in a composition. Perry's ''Version Control in the Inscape Environment'' [6] provides an example of this approach.

On the basis of the formal interface specifications, we can formally define the context-independent notions of *semantic equivalence*, *semantic compatibility* and *semantic incompatibility* for the various facilities in a module interface. We then extend these notions to module equivalence, compatibility and incompatibility.

Analogous to syntactic extensions to facilities, we define the notion of semantic extensions and several flavors of compatibility. One aspect that is important with respect to substitution is the *semantic independence* of the extensions.

On the basis of semantic equivalence, compatibility, and independence, we then define various flavors of context-dependent *implementation compatibility* that capture various kinds of side-effects that result when one component is substituted for another.

2.3 Semantics of Shared Dependencies

The semantics of interfaces considers only aspects of semantic relationships of two modules in relation to each other or of a single module in the context of a composition. It is increasingly common that our software systems have multiple dimensions of organization, particularly when considered from the standpoint of version management. For example, we have the notion of features in telephone switching systems that often are orthogonal to the decomposition or design structure [11] — that is, multiple components cooperate in implementing some particular behavior; this kind organizational complexity is further compounded by such things optioning and portability.

The simplest manifestation of these large-grained semantic dependencies is the sharing of a particular structure between several components. As the structure evolves in arbitrary ways, the sharing components may have to evolve as well. Mixing versions of these components that have this form of shared dependency often results in a composition that is syntactically consistent but semantically inconsistent.

A more complicated form of shared dependency is that of several components sharing in a cooperative implementation (of, say, an algorithm, procedure, or feature). Here the details of the implementation and how they are apportioned among the components are not at all obvious from even the semantics of the interfaces.

In either case, it is exceedingly important that these shared dependencies be made explicit and considered in building consistent compositions and in substituting components within those compositions.

3. Summary

Building large, complex systems from components is a task that requires the consideration of a large number of factors. We need a rich variety of relationships between components in order to reason about various aspects of composition and substitution. All three dimensions presented above must be considered to successfully compose a system from components and to successfully substitute one component for another.

References

[1] Evan W. Adams, Masahiro Honda and Terrance C. Miller. ''Object Management in a CASE Environment'', *The Proceedings of the Eleventh International Conference on Software Engineering*, May 1989, Pittsburgh, PA. pp 154-163.

[2] Ellen Borrison. ''A Model of Software Manufacture'' *Advanced programming Environments*, Trondheim, Norway, June 1986. pp 197-220. Lecture Notes in Computer Science 244, Springer-Verlag, 1986.

[3] Stuart I. Feldman. ''Make — a Program for Maintaining Computer Programs'', *Software — Practice & Experience* 9:4 (April 1979). pp 255-265.

[4] A. Nico Habermann and Dewayne E. Perry. *Well Formed System Composition.* Carnegie-Mellon

University, Technical Report CMU-CS-80-117. March 1980.

[5] Gail E. Kaiser. ‘‘Intelligent Assistance for Software Development and Maintenance’’, *IEEE Software* 5:3 (May 1988). pp 40-49.

[6] 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.

[7] Dewayne E. Perry. ‘‘The Inscape Environment’’. *The Proceedings of the Eleventh International Conference on Software Engineering*, May 1989, Pittsburgh, PA.

[8] M. J. Rochkind. ‘‘The Source Code Control System’’, *IEEE Transactions on Software Engineering*, SE-1:4 (December 1975). pp 364-370.

[9] Walter F. Tichy. *Software Development Control Based on System Structure Descriptions*. Ph.D. Thesis, Computer Science Department, Carnegie-Mellon University, January 1980.

[10] Walter F. Tichy. ‘‘RCS — A System for Version Control’’, *Software — Practice & Experience*, 15:7 (July 1985). pp 637-654.

[11] P. A. Tuscany. ‘‘Software Development Environment for Large Switching Projects’’, *Proceedings of Software Engineering for Telecommunications Switching Systems Conference*, 1987