nr Pt 0

Dimensions of Consistency in Source Versions and System Compositions

Dewayne E. Perry

Bell Laboratories
Software Production Research Department
600 Mountain Ave
Murray Hill NJ 07974

+1.908.582.2529 (office)
+1.908.582.5809 (fax)
dep@research.bell-labs.com

Abstract

In building systems there are various levels at which we consider the problems reasoning about consistency and it means different things at those various levels. At the version management level, consistency means what it does in databases: no data is lost due to concurrency problems (eg, race conditions). At the composition and substitution (or creation and evolution) levels it means something that is significantly different --- namely, the syntactic and semantic consistency of the various pieces that make up the system.

I first address the issue of what makes a system composition well-formed both syntactically and semantically. I then address the issue of substitution in well-formed system compositions, first in the context of simple substitution and then in the context of compound substitution (that is, the simultaneous substitution of multiple components).

Note: This paper derived and extended from papers: the well-formed system composition paper [9] was published only as a technical report at CMU (though variously used without references or with misleading ones) the version control paper from ICSE9 [16], the extended abstract for SCM3 [19], and the shared dependency paper from SCM6 [20] all of which have been published only in conference or workshop versions. There may be parts of the other Inscape papers (ICSE9 [15], ICSE11 [17], and TAV3 [18]) included as well - all of which have been published only in conference versions.

1. Introduction

In his paper ‘‘Tolerating Inconsistency’’ [2], Balzer stated that ‘‘Instead of treating inconsistency informally (i.e., outside the system) or as hard constraints, formalisms are needed for spotting such violations, treating them as problems, organizing resources to resolve them, recognizing when this has occurred, and limiting access to the inconsistent data by agents not involved in their resolution.’’

Necessary to such a venture is a coherent notion of what it means to be consistent and thus how to recognize when the state of affairs is indeed inconsistent. Moreover, this basis of consistency is needed in providing automated support for inconsistency resolution. To this end, I address the issues of consistency in the context of system compositions and evolution by component substitution.

In building systems there are two important factors relative to consistency that we must address: ensuring that source components and their derivations are consistent with each other, and determining that the components and compositions which comprise a software products are consistent with respect to one another.

Much of the past and current work in version and configuration management has addressed the problem of keeping track of how components are derived and maintaining that level of consistency. We have systems that manage version and configuration histories — for example SCCS [23], RCS [25], NSE [12], 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 [7, etc] or such opportunistic processors as Marvel [10] — 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 [4] for a discussion of the latter issues.

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 component composition and evolution. Moreover, they do not exploit any of the existing relationships that could be used by consistency-specific analysis tools.

It is the purpose of this paper to present what I consider to be the dimensions of consistency for both source and composed versions of components in building software systems. In section 2, I delineate the dimensions of consistency and discuss each of these in turn in sections 3 through 6. Finally, in section 7, I summarize our contribution.

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

---

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

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.

I propose four interdependent dimensions to be considered as basic to reasoning about system composition and evolution (the sources on which this paper is based are listed with each dimension):

- System Composition
  - syntactic 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 [8];
  - semantic well-formedness of compositions — that is, that the intentions of the specifications are properly observed and that system construction has been done properly according to the basic rules of composition [18];
- System Evolution
  - single component substitution, or small-grained semantic consistency — that is, that interfaces of components and compositions are used in a consistent manner [16]; and
  - multiple component substitution or large-grained semantic consistency — that is, that shared dependencies in various forms are resolved in a consistent manner [19, 20].

All three dimensions must be considered to successfully compose a system from components and to successfully evolve via component substitution. In the ensuing discussion, I present the concepts and relationships between components that are needed for reasoning about composition and substitution.

3. Syntactically Well-Formed System Compositions

Good software engineering practices dictate that we design systems by decomposing them in to smaller pieces. In the building of these systems, we then are faced with two basic kinds of components: the basic, atomic components (that is, the modules); and the built-up components (that is, the composed components — for example, subsystems, etc). For purposes of composition I view a component (whether atomic or composed) abstractly as a set of provided facilities and a set of required facilities. Similarly, I consider a composition at the same level of abstraction: a set of components which for which we have a set of required facilities and a set of provided facilities.

I consider first properties of the individual components and then properties of composed components.

In the ensuing discussion I use the following notation in our rules, theorems and discussions.

- A system S is a set of components $C_1 \cdots C_m$ with component indices $M = 1,...,m$
- p_i is the set of facilities provided by component C_i
- $p(S)$ is the set of facilities provided by system S
- r_i is the set of facilities required by component C_i
- $r(S)$ is the set of facilities required by system S
- A composition of a system S, $COMPOS(S)$, is denoted by the set of indices $i_1, i_2, ..., i_k$ where $i_j \in M$

I use the term *system* here in a general sense to mean any coherent collection of components and which may be used to model such typically used terms as subsystems etc as well as *the* system itself. Thus a system may be contain both atomic components and system (or composed) components. I use the term *module* as the term for a basic atomic component.

3.1 Reasoning about Basic System Construction

Modules are atomic units in building systems and hence their provides and requires lists are stated, not derived. The basic property required of a module specification is that it is *free of contradictions* — that is, that a module does not require a facility that it provides.

$$p(module) \cap r(module) = \varnothing \qquad \text{Rule (1.1)}$$

As a system is built from a set of components, the basic property of a system is that it must be *complete* — that is, that the facilities provided by a system must be provided by its components.

$$p(S) \subseteq \bigcup_{i=1}^{m} p_i \qquad \text{Rule (1.2)}$$

The set of required facilities of a system, contrary to those of a module, are derived automatically from the provide and require lists of its components. The required facilities of a system are precisely those required facilities of components that are not supplied by the provisions of the system's components.

$$Let\ \overline{p_j} = \bigcup_{i=1}^{m} (p_i - p_j),\ r(S) = \bigcup_{i=1}^{m} (r_i\ minus\ \overline{p_i}) \qquad \text{Rule (1.3)}$$

From these three rules, I derive the theorem[2] that is the system's analog of Rule (1.1): no facility provided by a system is also required by that system — that is, the system is free of contradictions.

$$p(S) \cap r(S) = \varnothing \qquad \text{Theorem (1.1)}$$

I then derive theorem (1.2): the facilities required by any component are either provided by some other component or are required by the system, but not both.

$$For\ any\ C_i \in S,\ f \in r_i \rightarrow f \in p_j \text{ for } some\ C_j \in S\ \ xor\ \ f \in r(S) \qquad \text{Theorem (1.2)}$$

Note that the relationship between the facilities provided by a system and those provided by its components are not as strict as that relationship between the facilities required by a system and and those required by its components. For example, components may provide facilities that are not provided by the system; they may also provide facilities that are not required by any other component.

3.2 Reasoning about Basic System Composition

System compositions specify subsets of the components of a system and as such describe various ways in which the implementation of a system may be realized. The rationale for differing composed versions is as various as the rational for differing module versions. Also, as I mentioned in the previous subsection, I consider composition as a fundamental means of hierarchical construction and that components in a system may be either atomic or composed.

The first rule for compositions is that they are made from components included in the system description.

$$(\ j \in COMPOS(S)\ )\ \ j = k\ \ for\ some\ C_k \in S \qquad \text{Rule (1.4)}$$

Given that a system description often contains multiple versions (either successive or parallel) of the same module, we have to be careful in building compositions to avoid using components that provide the same facilities. This duplication of provided facilities is similar to identical variable names in the same scope. The second rule for compositions requires that they be *conflict-free*.[3]

---

2. The details of the proofs are found in Habermann and Perry [8].

3. We will see in the next section that this rule may seem unnecessarily strict. However, appropriate relabeling provides a straightforward solution. For example, SVCE [9, 11] provides extended naming (by means dot notation with version identifiers) as the means of relabeling.

$$(i, j \in COMPOS(S))\ \ i \neq j \ \rightarrow\ p_i \cap p_j \ =\ \varnothing \qquad \text{Rule (1.5)}$$

Since a composition is a subset of the components in a system, the third rule for compositions requires that the facilities provided by the system are also provided by the composition — that is, that the composition is *self-sufficient.*

$$\begin{array}{c} \textit{A composition is self-sufficient} \ \leftrightarrow \\ p(S) \subseteq p(COMPOS(S)) \end{array} \qquad \text{Rule (1.6)}$$

Analogously, we require that the facilities required by the components of the composition are either satisfied by facilities provided by the those components, or are facilities required by the system — that is, that the composition is *self-contained.*

$$\begin{array}{c} \textit{A composition is self-contained} \ \leftrightarrow \\ r(COMPOS(S)) \ - r(S) \ \subseteq \ P(COMPOS(S)) \end{array} \qquad \text{Rule(1.7)}$$

A composition that is both self-sufficient and self-contained is a *proper* composition. A composition that lacks either of these properties is an *improper* composition.

$$\begin{array}{c} \textit{A composition is proper} \ \leftrightarrow \\ \textit{it is both self-sufficient and self-contained} \end{array} \qquad \text{Rule(1.8)}$$

From rules (1.4) - (1.8), I derive theorem (1.3): the set of facilities required by a proper composition and provided by the system are included in the set of facilities provided by that proper composition and required by the system.

$$\begin{array}{c} \textit{A composition is proper} \ \leftrightarrow \\ r(COMPOS(S)) \cup p(S) \ \subseteq \ p(COMPOS(S)) \cup r(S) \end{array} \qquad \text{Theorem (1.3)}$$

Given the definition of a proper composition, it is important that know that a proper composition exists (theorem (1.4)).

$$\textit{Every System has a proper composition} \qquad \text{Theorem (1.4)}$$

However, this composition in itself may not be very interesting: it may not be even usable as a ‘‘real’’ composition because of conflicting components. However, theorem (1.5) shows that we can construct a conflict-free composition from one that has conflicts.

$$\begin{array}{c} \textit{If a composition CS of system S has a conflict,} \\ \textit{we can derive a system S' from S} \\ \textit{by removing the conflicts from the components so that CS' is conflict-free} \end{array} \qquad \text{Theorem (1.5)}$$

Compositions that are both conflict-free and proper are *well-formed*. A system that has only conflict-free compositions is considered a conflict-free system; a system that has only well-formed compositions is considered a well-formed system.

$$\begin{array}{c} \textit{A composition is well-formed} \ \leftrightarrow \\ \textit{it is both conflict-free and proper} \end{array} \qquad \text{Rule (1.9)}$$

A system that has no superfluous elements — that is, that has no components that neither provide any required facilities of other components or of the system, nor provide any facilities provided by the system — is a *minimal well-formed* composition.

$$\begin{array}{c} \textit{A composition C is minimal-well-formed} \ \leftrightarrow \\ \textit{C is well-formed and } (S \subseteq C)\ C - S \textit{ is not well-formed} \end{array} \qquad \text{Rule (1.10)}$$

3.3 Reasoning about Syntactic Interfaces, Composition and Evolution

Components evolve and in doing so have effects on the various compositions where they are used. Reasoning about the effects of these changes and how they effect the consistency of their use is as important as the basic rules of composition. I delineate several aspects of evolution and their effects on well-formed compositions. Note that these same considerations arise in the substitution of one component

for another (which of course is what we do when we evolve a component).

- 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* [24]).
- 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.

4. Semantically Well-Formed System Compositions

Where in the previous section I considered the problems of dependencies at the level of syntactic requirements and provisions, here I consider those dependencies at the semantic level. The Inscape Environment [15-20] uses formal module interface specifications to describe semantic as well as syntactic interface information. The semantic information is specified by means of user-defined predicates that are either abstractions that hide (possibly complicated) details about the system state (that is, they are considered to be primitive or base terms), basic logical sentences about the system state, or encapsulations of (usually) complex logical formulations about the system state.

There are three sections in an Instress[4] specification: the predicate definition section in which the application specific vocabulary is defined; the data object section in which the types, variables and constants are specified; and the operation section in which the functions and procedures are specified.

A type specification consists of a base type and a set of properties (that is, predicates that provide additional type constraints); a variable specification consists of a defining type, a set of properties (that is, predicates that provide either constraint or relational information), and optionally an initial value with its set of initial properties; and a constant is analogous to a variable specification except that it requires the value specification.

An operation specification is a composite of the type signature, a set of preconditions (predicates that represent the assumptions for the operation) and a set of results, each of which is comprised of a set of postconditions (predicates that indicate the resulting state affected by the operation) and a set of obligations (predicates that represent the dual of preconditions — that is, predicates that must be eventually satisfied).

A module interface then is a composite of data object specifications and operation specifications. For purposes of determining various relationships, I ignore the predicate definition section, as the critical uses of that section are found in the data object and operation sections.

In the sections that follow, I first discuss notation and introduce terms that will be used in the ensuing two sections. I then consider first context-independent relationships and second context-dependent relationships

4. Instress is the module interface specification language for the Inscape Environment.

based on the syntactic and semantic information available in the interface specifications.

4.1 Basic Notation, Terms and Relations

The following abbreviations are use to denote various sets in describing interface relationships. I first introduce the definitions of α-identity, α-equivalence, α-compatibility, and α-incompatibility where α is either a set of properties, preconditions, postconditions or obligations.

- C_i denotes a particular component
- $Impli_i$ or $Impl(C_i)$ denotes a the implementation of C_i
- I_i of $I(C_i)$ denotes a particular interface
- V_i denotes a particular version
- $Prop(O_i)$ denotes the set of properties for data object O_i
- $Pre(O_i)$ denotes the set of preconditions for an operation O_i
- $Res(O_i)$ denotes the set of results for an operation O_i
- $Post(Res_j(O_i))$ denotes the set of postconditions of jth result of operation O_i
- $Obl(Res_j(O_i))$ denotes the set of Obligations of jth result of operation O_i
- $K(P)$ is defined as $\{ S \mid S \in P \text{ or } S \text{ is known from } P \}$ — that is, $K(P)$ is the set of all ''known'' sentences that are either in P or are known from P.[5]

As the reasoning about the semantic content of the interface specifications is done primarily on the basis of sets of predicates, I define the following basic relationships between sets.

- A set S_1 is defined to be *identical* to set S_2 when $S_1 = S_2$
- A set S_1 is defined to be *equivalent* to set S_2 when $K(S_1) \equiv K(S_2)$
- A set S_1 is defined to be *compatible* with set S_2 when $K(S_1) \subseteq K(S_2)$
- A set S_1 is defined to be *incompatible* with set S_2 when $K(S_1) - (K(S_1) \cap K(S_2)) \neq \varnothing$

Postconditions define what is known to be true as a result of the operation's execution. Obligations define what must become true at some time in the future of the computation --- that is, the computation is obliged to fulfill the obligation or it is a semantically incorrect computation. Obligations are generally used to indicate either the relationship of bracketing operations (such as open and close, allocate and deallocate) or the expression of an invariant among components. It is this last purpose that is of particular importance in the sequel.

4.2 Reasoning about Component Semantic Properties

The definition of consistency is straightforward:

$$A \text{ set of predicates } P \text{ is consistent} \leftrightarrow \text{it is not the case that } P \rightarrow false. \qquad \text{Rule (2.1)}$$

In the remaining discussion, the logical notions are those of a standard first order predicate logic.

The consistency of a specification as a whole, then depends on the consistency of the various parts.

---

5. I use the term ''known from'' rather than ''derived from'' because it is a more neutral term. In Inscape, one of the primary research issues is the use of incomplete reasoning rather than full theorem proving. What is known is thus dependent on the reasoning engine; it may or may not be coincident with what is derivable. In the strictest (ie, ideal) sense, ''known from'' means ''derived from'' on the basis of the logic that is used.

An Interface Specification $S = (P, D, O)$ *is consistent* $\leftrightarrow$ Rule (2.2)
the definition of each predicate P_i *in P is consistent* /\
the set of properties defined for *each data object* D_i *in D is consistent* /\
each set of preconditions, *postconditions* and *obligations* for *each operation* O_i *in O is consistent*

4.3 Reasoning Semantically about Compositions

Instress's formal interface specifications are also the basis for reasoning about the constructive composition of these components into implementations. In my paper ''The Logic of Propagation in the Inscape Environment'' [18], I defined the rules of composition for sequence, selection and iteration. On the basis of rules about function invocation and assignment, the rules for sequence, selection and iteration enable one to compose program fragments (and derive their interfaces by the rules of the propagation logic) which can be further composed with other fragments until an implementation sequence has been composed for the desired operation.

One rule that is not covered there concerns assignment: the *referential transparency rule*. Because I want to be able to reason about interfaces and not internal implementations I need to make sure that there are not facts about the implementation that are hidden from the view of external behavior. Assignment can cause that problem to occur (see [21] for a more complete discussion). By limiting visible variables to only one assignment (the one that produces the property visible from the interface behavior) one can guarantee this necessary referential transparency.

It is this notion of a composed sequence that will be of importance in the discussion of reasoning about compositions and single and multiple substitutions. An important aspect of a composed sequence is whether it is *complete* or not --- that is, whether all the preconditions and obligations have been handled properly according to the basic rule in Inscape: all preconditions and obligations in a composed fragment must be either satisfied within that fragment or propagated to the interface of that fragment.

Germane to the definition of the completeness of a program fragment are the notions of *precondition ceilings* and *obligation floors* [18]. In the propagation of preconditions and obligations when constructing program fragments, the preconditions percolate ''upwards'' and the obligations percolate ''downwards'' in search of either satisfying postconditions or the ''edge'' of the implementation (that is, the interface). Preconditions ceilings are logical barriers to that movement of the precondition ''up through'' the implementation to the interface. For example, a postcondition of *not P* forms a ceiling for an unsatisfied precondition *P* in its movement up to the interface. The obligation floor functions similarly for obligation as they move ''down through'' the implementation fragment to the interface, though there is not quite the logical necessity that occurs in the case of preconditions.

An implementation $I = Impl(C_1 \ldots C_N)$ for *a system fragment F is complete* $\leftrightarrow$ Rule (2.3)
Every precondition in I has either been satisfied or is in the interface of F /\
(that is, all precondition ceilings in each C_i (recursively) are empty)
Every obligation in I has either been satisfied or is in the interface of F /\
(that is, all obligation floors in C_i (recursively) are empty)
There are no iteration errors /\
(that is, the preconditions of each iteration in I are consistent with postconditions of their respective iteration bodies)
Each assignment preserves referential transparency
(that is, assignment within operations is used in such a way that we do not have to reason about the implementation, only about the interface behavior)

One further definition is needed to complete the preliminary groundwork: that for a *self-contained* composition.

An implementation I for *a program fragment F is self-contained* $\leftrightarrow$ Rule (2.4)
$Pre(I) = \varnothing$ /\ $Obl(I) = \varnothing$

An operation (that is, a function or procedure) is the basic usable syntactic fragment in most programming

languages. We will see that this rule is important when considering multiple substitutions in evolving a system.

I now extend the notion of a syntactic well-formed system composition to that of a semantically well formed system composition: a syntactically well formed composition is semantically well formed if and only iff each component interface in the composition is consistent and the composition is also semantically complete.

$$\begin{aligned} &\textit{A syntactically well}-\textit{formed composition is semantically well}-\textit{formed} \leftrightarrow \\ &\quad \textit{each component interface is consistent} \ /\backslash \\ &\quad \textit{the implementation of the composition is semantically complete} \end{aligned} \qquad \text{Rule (2.5)}$$

5. System Evolution — Single Substitution Consistency

Using the definitions defined earlier for for set identity, equivalence and compatibility, I consider the notions of interface identity and and equivalence. Since the properties of data objects are sets of predicates, their identity and equivalence depends on those definitions. I concentrate here on interfaces, specifically, interfaces of operations.

I begin with the following simple and straightforward definition of the *interface identity* of an operation. In this definition, we are concerned with the uses of these interfaces, not their implementations. Thus, the problem whether interfaces, that differ only in their choice of parameter names, are identical disappears since the emphasis here is on the properties of the interfaces.

Operation interfaces are identical if and only if their set of preconditions are identical and their results are identical — that is, for each result the postconditions and obligations are identical.

$$\begin{aligned} &\textit{An operation interface } I_2 = I_1 \leftrightarrow \\ &\quad Pre(I_1) = Pre(I_2) \ /\backslash \\ &\quad \forall\ Res_i(I_1), Res_i(I_2) \\ &\qquad Post(Res_i(I_1)) = Post(Res_i(I_2)) \ /\backslash \ Obl(Res_i(I_1)) = Obl(Res_i(I_2)) \end{aligned} \qquad \text{Rule (3.1)}$$

While the notion of version identity is not needed in the subsequent discussion, I can define it in a general way if I ignore the philosophical and legal problems in the definition of the notion of two implementations being identical.

$$\begin{aligned} &\textit{A version } V_2 = V_1 \leftrightarrow \\ &I_{V_2} = I_{V_1} \ /\backslash \ \textit{their implementations are identical} \end{aligned} \qquad \text{Rule (3.2)}$$

More important is the notion of version equivalence: two versions are equivalent if and only if their individual components are equivalent. I ignore the fact that the results may be ordered differently in equivalent interfaces and assume that the individual results are in comparable order.

$$\begin{aligned} &\textit{A version } V_2 \equiv V_1 \leftrightarrow \\ &\quad K(Pre(V_2)) \equiv Pre(V_1) \ /\backslash \\ &\quad \forall\ Res_i(V_2), Res_i(V_1) \\ &\qquad K(Post(Res_i(V_2))) \equiv K(Post(Res_i(V_1))) \ /\backslash \\ &\qquad K(Obl(Res_i(V_2))) \equiv K(Obl(Res_i(V_1))) \end{aligned} \qquad \text{Rule (3.3)}$$

Obviously, if the interfaces are either identical or equivalent, then they will preserve the property of semantic well-formedness. However, it is not always the case that substituted components have either identical or equivalent interfaces. They are often different in some meaningful way because of system evolution. There are still some cases that we can reason about in a systematic way such that we do not need to begin the system analysis from scratch.

To this end, I define five different kinds of compatibility: exact, strict, upward, implementation, and system compatibility. The first three are forms of our intuitive notion of upward compatibility: the first two capture the notion of substitutability and the second captures the notion of extended functionality. The first two are, in fact, the more useful of the two as far as single substitution evolution is concerned. We shall see in

the next section that the third notion has its uses as well.

$$V_2 \text{ is a exactly compatible version of } V_1 \leftrightarrow \quad \text{Rule (3.4)}$$
$$K(Pre(V_1)) \supseteq K(Pre(V_2)) \ /\backslash$$
$$\forall\, Res_i(V_2), Res_i(V_1)$$
$$K(Post(Res_i(V_2))) \equiv K(Post(Res_i(V_1))) \ /\backslash$$
$$K(Obl(Res_i(V_2))) \equiv K(Obl(Res_i(V_1)))$$

The only difference between an equivalent version and *exactly* compatible one is that the exactly compatible one may make fewer assumptions. Clearly the substitution of an exactly compatible version will have no effect on the semantic well-formedness of a composition. This form of compatibility leads to theorem 3.1

$$If\ COMPOS(S) \text{ is semantically well-formed} \quad \text{Theorem (3.1)}$$
$$/\backslash\ C_j \text{ is a exactly compatible version of } C_i \text{ for some } i \in COMPOS(S) \rightarrow$$
$$COMPOS(S)' \text{ is semantically well-formed with } C_j \text{ sustituted for } C_i$$

One of several things can happen if the substituted component makes fewer assumptions than the component substituted for. Let P be such an assumption — that is, a precondition. If P is not in $Pre(C_j)$ but was satisfied in the composition using C_i, the there will be no effect caused by the substitution. If P was not satisfied but was propagated to the interface of the component where it was used, then the result of the substitution will be that P is no longer propagated to the interface, and the resulting encompassing components interface will then be *exactly compatible* to that prior to the substitution. One further case needs to be considered: P caused a precondition *not P* to be ceilinged in the original composition. For that composition to be considered semantically well-formed, it must have been satisfied subsequent to that use and hence will not suddenly become propagated to the interface no that it is no longer blocked by P. Hence at best the composition interface will remain the same or it will be exactly compatible. In either case, the substitution will not effect the semantic well-formedness of the composition.

$$V_2 \text{ is a stricly compatible version of } V_1 \leftrightarrow \quad \text{Rule (3.5)}$$
$$K(Pre(V_1)) \supseteq K(Pre(V_2)) \ /\backslash$$
$$\forall\, Res_i(V_2), Res_i(V_1)$$
$$K(Post(Res_i(V_2))) \subseteq K(Post(Res_i(V_1))) \ /\backslash$$
$$K(Obl(Res_i(V_2))) \equiv K(Obl(Res_i(V_1)))$$

That is, version V_2 is a strictly compatible version of V_1 if and only if it assumes no more than V_1, guarantees no less than V_1, and obliges the same as V_1.[6] It would be desirable to be able to claim the same kind of substitability for strict compatibility as I did for exact compatibility. Unfortunately, there are two possibilities that may occur in the substitution that preclude this theorem. First, a new postcondition might ceiling a previously propagated precondition, thereby making the substituted composition incomplete semantically. Second, a new postcondition might satisfy a previously propagated obligation thereby causing the obligation not to be propagated to the interface whee it had been previously satisfied or further propagated. There are some cases where obligation satisfaction is idempotent. If that is the case, then there is no semantic problem. Causing a certain state condition to be set is an example of an idempotent satisfaction — for example, X = high. However, there are many such obligation satisfactions which are not idempotent, such as releasing a buffer where doing it twice may cause buffering problems or an unexpected exception.

Whether a strictly compatible component may be substituted without ill effects will depend on whether the extra postconditions are already known at the point of use. If they are, then all preconditions that can be ceilinged will already have been ceilinged and all obligations which could be satisfied will already have been satisfied.

---

6. Note that if V_1's obligations are included in V_2's, then the source may have to be modified to cover the extra obligations incurred by V_2. Similarly, if V_2's obligations are included in V_1's, then it may be the case that too much is done in the implementation if V_2 is substituted for V_1 — that is, obligations will be met that are non-existent.

These considerations force us to wait until we consider implementation compatibility for further remarks.

The third form of compatibility I call *upward compatibility* because the original functionality is preserved while it is extended.

$$V_2 \text{ is an upwardly compatible version of } V_1 \leftrightarrow \quad \text{Rule (3.6)}$$
$$K(Pre(V_1)) \subseteq K(Pre(V_2)) \ /\backslash$$
$$\forall\, Res_i(V_2),\ Res_i(V_1)$$
$$K(Post(Res_i(V_1))) \subseteq K(Post(Res_i(V_2))) \ /\backslash$$
$$K(Obl(Res_i(V_1))) \subseteq K(Obl(Res_i(V_2)))$$

The utility of upward compatible versions like that of strictly compatible versions depends on the implementation context. Thus, while these last two forms are useful in determining, for example, when a new version is still a parallel version of the previous version, it is not as useful as the first in determining substitutability in composing or generating new components.

Exact, strict and upward compatibility place restrictions on what might be suitable as a substituted component in a system composition. There are situations where we might find these restrictions too constraining. For example, we often use only a part of the functionality of an operation rather than its entire functionality. If another operation provides that bit of functionality that we use in the original, we might want to consider it as a replacement component in a composition even though it is neither strictly nor upwardly compatible with the original version.[7] To this end I introduce several forms of the notion of *implementation compatibility*: exact, strong, and weak implementation compatibility. These different forms represent degrees of relaxation of the constraints on the extent of the effects that we are willing to accept in the substitution of one version for another.

A version is *exactly implementation compatible* with another in the implementation of an operation if it has no effect on the propagated interface[8] and the resulting substitution preserves the completeness of the implementation of that operation.

I use the notation $COMPOS(C_k)_{i \to j}$ to mean that C_j is substituted for C_i in that composition with the assumption that C_j is not already in $COMPOS(C_k)$.

$$C_j \text{ is exactly implementation compatible with } C_i \text{ in } COMPOS(C_k) \leftrightarrow \quad \text{Rule (3.7)}$$
$$I(COMPOS(C_k) \equiv I(COMPOS(C_k)_{i \to j}) \ /\backslash \ COMPOS(C_k)_{i \to j} \text{ is semantically complete}$$

Given this formulation, I can show that a semantically well-formed composition in which a substitution is made of an exactly implementation compatible component will remain semantically well-formed.

$$COMPOS(C_k) \text{ is semantically well-formed} \ /\backslash \quad \text{Theorem (3.2)}$$
$$C_j \text{ is an exactly implementation compatible with } C_i \text{ in } COMPOS(C_k) \rightarrow$$
$$COMPOS(C_k)_{i \to j} \text{ is semantically well-formed}$$

The fact that the substitution causes no change in the interface of $COMPOS(C_k)_{i-j}$ by the definition of being exactly implementation compatible the interface of $COMPOS(C_k)_{i \to j}$ will remain consistent. The fact that $COMPOS(C_k) i \to j$ is also complete also by the definition then guarantees that $COMPOS(C_k)_{i \to j}$ remains semantically well-formed.

---

7. Note that there are certain facilities that we need to make this practical in terms of the programming language support. Facilities like Ada's default values for parameters, C's ability to have parameter lists of arbitrary length, and Prolog's 'don't care' argument are the kinds of features that would expedite this approach.

8. In [18], I discuss the construction of components on the basis of the Instress interface specifications and describe how an interface is automatically constructed from its implementation. This automatically constructed interface represents the *propagatable* interface — that is, it represents all requirements and functionality that result from the implementation. While some requirements and results must be propagated, there are some that may be optionally propagated depending on how they arise in the implementation, thus reducing the strength of the results to be guaranteed by the interface. This user-selected interface is the *propagated* interface.

Clearly, any version that is equivalent is also exactly implementation compatible. A version that is exactly or strictly compatible may be exactly implementation compatible, depending upon the characteristics of the interface and the implementation. For example, a strictly compatible version may be exactly implementation compatible in one occurrence but not in another. It is also possible for a weaker version (that is, one that guarantees less functionality) to be exactly implementation compatible if those results of the original version not covered by the weaker version are duplicated elsewhere in the implementation.

I relax that constraint on effects on the propagated interface slightly and consider a version to be *strongly implementation compatible* with another in the implementation of an operation if it has only what we intuitively consider to be acceptable, or benign, effects.

$$C_j \text{ is strongly implementation compatible with } C_i \text{ in } COMPOS(C_k) \leftrightarrow COMPOS(C_k)_{i \to j} \text{ is semantically complete} \qquad \text{Rule (3.8)}$$

Clearly, if C_j is an exactly compatible version of C_i it will be strongly implementation compatible because at most it will cause an exactly compatible propagated interface. Here, as in the more restrictive form, it is possible for an operation not to be exactly, strictly or upwardly compatible and still be strongly implementation compatible. It all depends on the implementation context, the dependencies established on the basis of the original component and the other interfaces and their interdependencies.

A much weaker form that allows immediately unacceptable effects that eventually become acceptable — like the ripples resulting from a pebble thrown into a calm pond that eventually subside — is that in which one version is *weakly implementation compatible* with another. For this rule we need recursion: the base step is that the component in question is exactly implementation compatible. The recursive step is that the component is part of a composition and that, as a result of the substitution, it is a weakly compatible version of the component without the substitution.

$$C_j \text{ is weakly implementation compatible with } C_i \text{ in } COMPOS(C_k) \leftrightarrow C_j \text{ is exactly implementation compatible with } C_i \text{ in } COMPOS(C_k) \;\backslash/\; (\exists C_l \text{ such that } k \in COMPOS(C_l) \;/\backslash\; C_{k_{i \to j}} \text{ is weakly implementation compatible with } C_k \text{ in } COMPOS(C_l)) \qquad \text{Rule (3.9)}$$

Virtually any version is a candidate for this form of compatibility. The only requirement is that at some point, the effects of the substitution of one version for another eventually cease to have an effect — that is, in propagating the resulting changes throughout the implementation of the system, at some point, there are no longer any effects, or at worst, there are only benign effects.

I extend the notion of implementation compatibility to that of system compatibility — if each occurrence of the substitution of the one version for another has the same form of implementation compatibility, then it has that form of system compatibility.

$$V_2 \text{ is } \alpha \text{ system compatible with } V_1 \leftrightarrow V_2 \text{ is an } \alpha \text{ implementation compatible version of } V_1 \text{ for all occurrences of } V_1 \text{ in the system where } \alpha = \text{exactly } \backslash/ \text{ strongly } \backslash/ \text{ weakly} \qquad \text{Rule (3.10)}$$

6. System Evolution — Multiple Substitutions

Shared dependencies among components arise naturally in the way we build systems and are not necessarily the result of having built them badly. Because of our desire to separate concerns, encapsulate and abstract, we break up our complex systems into distinct components that cannot, of necessity, be completely independent.

It is also increasingly common that our software systems have multiple dimensions of organizations, particularly large and complex systems. For example, we have the notion of features in telephone switching systems that are often orthogonal to the design structure [26] --- that is, the implementation of a feature is to be found distributed among design components that also share in the implementation of other features. This kind of organizational complexity is further compounded by such considerations as

specialization, optioning and portability. I note in passing that the occurrence of multiple dimensions of organization is a general problem, not one endemic to switching systems.

In a study about parallel changes in a subsystem of 5ESS [22], Perry, Siy and Votta found that 83% of the features implemented in that subsystem required changes to more than one file. Indeed, 25% of the features required changes to 2 to 5 files, 25% of the features required changes to 6 to 20 files, and 33% of the files required changes to more than 20 files with a maximum of 900 files for one feature. This is a significant amount of shared dependencies.

At a smaller grained level of parallelism, features are divided into one or more Initial Modification Requests (IMRs) that represent problems to solve. 49% of those IMRs require more than one file to solve the associated problem, 34% require 2-5 files, 12% require 6-20 files, and 3% 21 or more files to a maximum of 400 files. Again, this represents a significant amount of shared dependencies, even at this finer level of resolution.

A common form of shared dependency occurs where several components share data structures. These dependencies are implicit in the assumptions about the state of the shared structures that each component makes when using those shared structures. The shared use of devices is another example of this form of shared dependency.

We find that a similar but more complicated form of sharing occurs when several components share in the implementation of a complex algorithm. This form is similar to the previous one because the distributed processing is usually glued together by means of a shared data structure, or set of data structures. Not only is the assumed state important to the processing by each component, but there is an invariant, or set of invariants, that must be maintained for the shared structure or structures.

Producers and consumers interacting and communicating by sharing a queue is a simple example of the first form of shared dependencies. A slightly more complicated example is that where one component opens a file, another components reads and processes some of the contents, another makes use of that information, and yet another closes the file. In each case, the components have assumptions about the state of the shared structures.

Two problems arise from these shared dependencies. First, one must treat the components together in context and not in isolation. In evolving any one of these components, one must often change other components participating in the shared dependency as well. Second, this problem of context is compounded by the fact that it is not unusual for a component to participate in several shared dependencies. This is particularly true in large complex systems where there are multiple dimensions of organization. In both cases, substitution in a system composition is not a simple consideration. Because shared dependencies involving a single component often extend in several different directions simultaneously, integration of individual component changes is complex and error prone.

6.1 Current Technology for Managing Shared Dependencies

The current state of the art in handling shared dependencies is represented by two different kinds of approaches: attribute-based configuration management systems and language-based programming-in-the-large facilities.

Two such CM systems are Adelle [6] and Workshop [5]. Both provide facilities at what I call the *unit interconnection* [15] level --- that is, dependencies are expressed between rather large-grained units (files, procedures, etc.). In Adelle, objects have attributes that may be used to indicate shared traits. For example, attributes may be used to indicate that certain versions are for a particular machine or for particular options. In Workshop, attributes are attached by the system to all objects edited in a particular workshop session. These attributes then indicate related sets of changes and can be used in a relatively coarse-grained way to indicate shared dependencies.

Two programming languages that offer some help with shared dependencies are ML and Ada. Both enable one to pass objects to modules and thus explicitly specify when objects are being shared between several modules. They provide what I call *syntactic interconnections* --- that is, dependencies between syntactic entities in the languages. In ML, one can specify the sharing of data structures by means of functors. In Ada, one can specify the sharing of data structures as parameters to generic instantiations of modules.

The attribute-based approach has the advantage of indicating which components share a particular dependency. It does not however indicate what the dependent data structures are nor what the actual semantic dependencies are between those components. The language-based approach has the advantage of making explicit what the dependent data structures are. Unfortunately, that is all that it does indicate. It only indirectly indicates what components are involved, but does not offer much assistance when multiple data structures are involved. Neither does it provide any information about the actual semantic dependencies among the components.

SVCE [9, 11] provides programming-in-the-large (again, *syntactic interconnections* facilities for both encapsulation and system composition. Both the encapsulation facilities and the system composition facilities enable one to group collections of related components together. Thus, one can indicate what is being shared and bound the scope of that sharing by either of these means. However, these facilities only work where any of the components only participate in a single shared dependency. SVCE also suffers from the disadvantage of not expressing the semantic dependencies between components.

To our knowledge, these approaches represent not only the current state of the art, but the current state of research as well. Mahler in his article about Shape [13] mentions the problems of multiple variances and the problems of semantic consistency in the presence of building compositions where components share in such multiple variances, but does not address them in that paper.

Batory and Geraci [3] come to grips with some of these problems in the context of their domain-specific system generators, adjusting the choice of some of the components dependent on other component choices to generate a consistent domain-specific system. Their mechanisms for doing this consistent generation are analogous to my approach in Inscape, but using only primitive predicates. Their rules of composition are similar to those of Inscape [18].

6.2 Reasoning about Shared Dependency Specifications

In previous sections I laid the groundwork for reasoning about shared dependencies: the definition of what it means to be a consistent interface specification, the definition of what it means for a component to be an upwardly compatible version of another, and the definitions of what it means for an implementation to be complete and self-contained.

In the next subsection, I introduce the structure of a shared dependency specification and propose the method for describing these dependencies. I then define what it means for a shared dependency to be well-formed. Finally, I discuss various ways of satisfying these shared dependencies.

6.2.1 Form and Method

A shared dependency is a set of partial predicate, data and operation specifications together with a set of partially instantiated interface specifications.

$$\textit{A Shared Dependency Specification SDS} = (\textit{Partial Specifications}, \textit{Partial Instantiations}) \qquad \text{Rule (4.1)}$$

The specifications and instantiations are *partial* because they may not contain all the type, parameter, or

behavioral information that would be found in a full specification and its use.

The method for defining such shared dependencies is as follows:

- Define the predicates needed for the partial object and operation specifications.
- Declare only those types and objects necessary for defining the constraints on sharing.
- Specify only that part of the semantics (the preconditions, postconditions and obligations) of the operations needed to define the sharing of dependencies.
- Instantiate only the arguments needed to define the relationships between the objects and the operations (use ‘‘_’’ for those arguments that do not participate in the dependency).

A simple example should suffice to illustrate both the method and the specification form. The example shared dependency specification illustrates two operations sharing the use of a particular data structure Q of type Queue, such that operation O1 depends on the state of the shared object Q to be P(Q) and operation O2 provides this state. Only the predicate P, the type Queue, the object Q, and the operations O1 and O2 need to be declared. The operations O1 and O2 are then partially instantiated with the shared object Q.

```
shareddependency Eg1 = (
    declarations {
        P ( queue q ) :: . . . ;
        type . . . queue ;
        var queue Q ;
        O1 ( queue x, . . . )
            pre: P ( x )
        O2 ( . . . , Queue y )
            post: P ( y )
    }
    instantiations {
        O1 ( Q, _, . . . )
            pre: P ( Q )
        O2 ( _, . . . , Q )
            post: P ( Q )
    }
)
```

6.2.2 Well-formedness of Dependency Specifications

There are two important questions to ask of any specification: whether it is well-formed and whether it accurately represents the intent of the designer. The second question is one that all specifiers must wrestle with in the same way that implementors wrestle with the question of whether the code accurately represents the intent of the design. The first question, however, is one that I can address.

The basic intuition, given that I want to concentrate only on those aspects germane to the specific dependency, is that all of the specifications are consistent and that semantic interconnections ought to be ‘‘matched up’’ with just the information available in the shared dependency specification.

Basic consistency is the first consideration for the partial specifications in just the same way that it is the first concern in full specifications. Moreover, the definition remains the same for partial specifications as for full specifications. I note, that for the sake of simplifying the presentation, I consider only the semantics of operations in the discussion below.

There are two ways by which one might ‘‘match up’’ the semantic dependencies. The first way, I call *weak composability* and the second way I call *strong composability*. The difference is in the way that the

semantic interconnections are established --- that is, in the way in which the semantic dependencies are satisfied.

In weak composability, it is sufficient for each precondition and obligation to be satisfied in some way by the postconditions found in the partial instantiations. That is,

A Shared Dependency Specification SD is weakly composable $\leftrightarrow$ Rule (4.2)

- *For each Precondition* P_i *of each Instantiated Interface* I_j,
  - *there is a set* γ_k *such that* γ_k *is included in the set Post of all postconditions of all the Instantiated Interfaces except* I_j, *and*
  - $\gamma_k \rightarrow P_i$
- *For each Obligation* O_i *of each instantiated interface Isubj*
  - *there is a set* γ_k *such that* γ_k *is included in the set Post of all postconditions of all the Instantiated Interfaces except* I_j, *and*
  - $\gamma_k \rightarrow O_i$

The disadvantage of this form of composability is that it only guarantees that it is possible to satisfy the preconditions and obligations. It does not guarantee that there is any composable sequence that satisfies all of the specified constraints.

The intent of strong composability, however, is precisely to provide that guarantee: there is a self-sufficient sequence in which all the preconditions and obligations are satisfied.

A Shared Dependency Specification SD is strongly composable $\leftrightarrow$ Rule (4.3)

- *there exists a sequential composition C including all of the Instantiated Interfaces* $I_1 \ \ldots \ I_N$ *such that*
  - *C is complete, and*
  - *C is self-contained.*

The definition of a well-formed shared dependency specification then matches our basic intuition, using strong composability as the means of ‘‘matching up’’ the semantic dependencies.

A Shared Dependency Specification SD is well – formed $\leftrightarrow$ *Rule (4.4)*
SD is consistent /\ *SD is strongly composable*

6.2.3 Sets of Shared Dependency Specifications

I mentioned in the discussion on shared dependencies that components often share in multiple dependencies. One has the choice of specifying these inter-related dependencies as either independent or as integrated specifications. Given that these interdependencies represent system design aspects, perhaps even architectural aspects of the system, the preferred method of specification is to specify them independently and then to combine them.

A combined shared dependency specification is a set of equations and a set of shared dependencies and has the following form.

A Combined Shared Dependency Specification CSDS = (Equations, SD Specifications) *Rule (4.5)*

The set of component equations specify which components in the different shared dependency specifications are to be considered the same components. Applying the set of equations to the set of shared dependencies results in a shared dependency specification in which each set of equated components is merged into a single component. Names are kept distinct in all cases by using the standard dot qualified names in which the name of the specification is prepended to each component name. Merged components are renamed by arbitrarily using one of the equated names. For example,

Eg3 = { Eg1.O1 == Eg2.O3 } applied to { Eg1, Eg2 }

results in Eg3 containing the components of Eg1 and Eg2 that were independent of the equation, and the merged version of Eg1.O1 and Eg2.O3 called (arbitrarily) Eg3.O1.

Having a well-formed shared dependency specification as a result of combining well-formed shared dependency specifications would be a very nice resulting property. However, in merging two separate partial specifications it is all too possible to inadvertently create an inconsistent set of predicates. Moreover, it is very easy to create a non-composable set of operations as a result of the merging.

The best that we can guarantee is that the results of combining shared dependencies will be weakly composable if the original shared dependencies were at least weakly composable.

There is a second reason for combining shared dependencies: creating higher level dependency relationships by aggregating existing shared dependency relationships. Typically, this approach combines independent relationships together. So, for example

Eg3 = { } applied to { Eg1, Eg2 }

yields a shared dependency that has two independent components as parts of the shared dependency Eg3. In this case, we do have a well-formed result returned from applying the empty set of equations to the well-formed two shared dependencies. Both specifications remain consistent, and both remain strongly composed.

6.2.4 Satisfying Shared Dependency Specifications

I first consider the problem of a component satisfying a shared dependency specification, first for simple satisfaction and then for aggregate satisfaction. I then consider the problem of a composition satisfying a shared dependency specification. Finally, I note that the problems of a composition satisfying a set of shared dependency specifications reduces to the single specification cases.

How a component satisfies a shared dependency specification depends on what the component is. For types I here choose a simple solution: type equivalence (leaving the question of whether it is name or structural equivalence to be answered by the implementation language). Alternatively, one might want to explore the possibility of using type compatibility instead. For predicates, I again choose a simple solution: equivalence of the definitions. For operations, the component must be an upwardly compatible version of the shared dependency component that it is satisfying.

A component C satisfies a Shared Dependency Specification component SC $\leftrightarrow$ Rule (4.6)

- *C and SC are both predicates, C* $\rightarrow$ *SC and SC* $\rightarrow$ *C* \/
- *C and SC are both type definitions (or they are both object declarations) and their types are equivalent* \/
- *C and SC are both operation specifications and C is an upwardly compatible version of SC.*

This definition enables us to satisfy components in a specification in a simple, one-to-one fashion. We may have an operation that combines several of the specification operations into a single component. For this case, we need a slightly richer definition of operation satisfaction.

A operation O satisfies$_2$ an aggregate of Shared Dependency Components Rule (4.7)
$A = (SC_{1,} \ldots , SC_n) \leftrightarrow$

- *PI is the propagated interface of a complete sequential composition of the components of A* /\
- *O is an upwardly compatible version of PI.*

Just as in the combining of shared dependency specifications, we required extra information to determine how various parts were related to each other, so we need an equivalent structure here to relate components in the composition to those in the specifications. This required structure is a map from composition components to specification components.

A Map M Composition Components in Compos$(C_{1,} \ldots , C_N)$ Rule (4.8)

Shared Dependency Components $SC_{1,} \ldots , SC_M$ *is well – formed* $\leftrightarrow$

- *For all* C_i, *M(*C_i*) are distinct (that is, no two composition components are mapped to the same shared dependency component or set of components* /\
- *For each* SC_j, SC_j *appears in the range of only one composition component* /\
- *All shared dependency components are in the range of M*

Thus a composition is a set of source components (in the required specification form) and a mapping from those source components to shared dependency components. The composition satisfies a shared dependency specification when all of the source components satisfy all of the specification components.

A Composition $C = Compos(C_{1,} \ldots , C_N)$ /\ Rule (4.9)
Map M satisfy a well – formed Shared Dependency Specification SD of components $SC_1, \ldots , SC_M \leftrightarrow$
Map M is well – formed /\ *For each* $C_{i,}$ C_i *satisfies* $M(C_i)$

The well-formedness of the map guarantees that all the components in the shared dependency will be satisfied either by simple satisfaction or by aggregate satisfaction by the source components in the composition.

I note that we can form a single shared dependency specification from a set of such specifications by applying the empty set of equations to those sets. As I argued, this reduces the set of a single specification with independently related sets of components. We can then apply one large composition to the entire set. Alternatively, we could combine sets of compositions that have been independently applied to their respective shared dependency specifications. To achieve the same results as the first alternative, we would need the additional constraint that all the compositions be disjoint.

7. Conclusions

I have brought together various means of reasoning about the consistency of component composition and substitution in a variety of its forms. In particular, I have laid the groundwork for reasoning about both the syntactic an semantic well-formedness of system compositions, reasoning about the substitution of one component for another in compositions, and reasoning about the simultaneous substitution of multiple components to satisfy the constraints of shared dependencies.

It is only in the context of such a basis of consistency as that presented here that one may reason adequately about inconsistency and how to manage it.

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] Robert Balzer. ‘Tolerating Insconsistency’’, *13th International Conference on Software Engineering*, May 1991, Austin Tx.

[3] Don Batory and Bart J. Geraci. ‘‘Validating Component Compositions in Software System Generators’’, Technical Report TR-95-03, Department of Computer Sciences, University of Texas at Austin, February 1995. Updated August 1995.

[4] 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.

[5] Geoffrey M. Clemm. ‘‘The Workshop System --- A Practical Knowledge-Based Software Environment’’, *ACM SIGSOFT’88: Third Symposium on Software Development Environments (SDE3)*, Cambridge MA, November 1988. In *ACM SIGSOFT Software Engineering Notes* 13:5 (November 1988). pp 55-64.

[6] J. Estublier and R. Casallas. ‘‘The Adele Configuration Manager’’, In [26] , pp 73-97.

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

[8] A. Nico Habermann and Dewayne E. Perry. *Well Formed System Composition.* Carnegie-Mellon University, Technical Report CMU-CS-80-117. March 1980.

[9] A. Nico Habermann and Dewayne E. Perry. ‘‘System Composition and Version Control for Ada’’. Symposium on Software Engineering Environments. Bonn, West Germany. June 16-20, 1980. Published in *Software Engineering Environments*, edited by H. Huenke, North Holland, l98l, pp. 331-343.

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

[11] Gail E. Kaiser and A. Nico Habermann. "An Environment for System Version Control", *Digest of Papers Spring CompCon ’83*, IEEE Computer Society Press, February 1983. pp. 415-420.

[12] David B. Leblang and Gordon D. McLean, Jr. "Computer-Aided Software Engineering in a Distributed Workstation Environment", *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, SIGPLAN Notices*, 19:5 (May 1984). pp. 104-112.

[13] Axel Mahler. ‘‘Variants: Keeping Things Together and Telling Them Apart’’, In [26], pp 73-97.

[14] K. Narayanaswamy and W. Scacchi. ‘‘Maintaining Configurations of Evolving Software Systems’’, *IEEE Transactions of Software Engineering*, SE13:3 (March 1987), pp 324-334.

[15] Dewayne E. Perry. ‘‘Software Interconnection Models’’, *Proceedings of the 9th International Conference on Software Engineering*, Monterey, CA, March 1987. pp 61-69. Best Paper Award.

[16] 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.

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

[18] Dewayne E. Perry. ‘‘The Logic of Propagation in The Inscape Environment’’. *Proceedings of SIGSOFT ’89: Testing, Analysis and Verification Symposium*, Key West FL, December 1989.

[19] Dewayne E. Perry. ‘‘Dimensions of Consistency in Source Versions and System Compositions — A Extended Abstract’’, *Proceedings of the 3rd Workshop on Software Configuration Management.* Trondheim, Norway, June 1991.

[20] Dewayne E. Perry. ‘‘Dimensions of Software Evolution’’ Invited Keynote Paper, *International Conference on Software Maintenance 1994*, Victoria BC, September 1994 .

[21] Dewayne E. Perry. ‘‘System Compositions and Shared Dependencies’’, 6th Workshop on Software Configuration Management, ICSE18, Berlin Germany, March 1996. Springer-Verlag, 1996.

[22] Dewayne E. Perry, Harvey P. Siy and Lawrence G. Votta. ‘‘Parallel Changes in Large Scale Software Development: An Observational Case Study’’, submitted for publication.

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

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

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

[26] Walter F. Tichy, editor. *Configuration Management*. Trends in Software, Volume 2. Chichester: John Wiley & Sons, 1994.

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