System Compositions and Shared Dependencies

Dewayne E. Perry

Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974 USA

Abstract. Much of the work in configuration management has addressed the problems of version history and derivation. Little has been done to address the problems of reasoning about the consistency of composed components or the effects of substituting one version for another. In my paper, "Version Control in the Inscape Environment" [13], I defined a number of concepts to be used in reasoning about substituting one component for another. In this paper, I discuss the problem of shared dependencies (that is, substituting one of more interdependent components in a context), propose an approach for specifying such dependencies, and show how this approach can be used to reason about the substitution in the context of interdependent components in a configuration.

1 Introduction

In building software systems from components, there are two important concerns that we must address: keeping track of how components in a system are derived, and determining that the components comprising a system are consistent with each other.

Much of the past and current work in version and configuration management has been focused on determining and keeping track of how components are derived. We have systems, such as SCCS [16], RCS [18], and NSE [1], to help us manage version and configuration histories. Typically, these tools use trees or graphs to represent the derivation of source versions. We have tools, such as various forms of Make [6] and such opportunistic processors as Marvel [9], to automatically derive objects we need to build executable versions of our systems. Typically, these tools use (unit or syntactic) dependency [12] descriptions as the basis for their automation. In general, we have a fairly deep understanding of how to manage the derivation relationships both for manually derived and for automatically derived components [3].

The problem of component consistency, however, has received much less attention by researchers in configuration management. Instead of using tools specific to programming-in-the-large, we manage consistency checking with basic component-building and systems-building tools, such as compilers and link-loaders, and with laborious and (necessarily) incomplete testing.

There are two basic problems that motivate our interest in the consistency of atomic 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.

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 [7] for a complete discussion of the various aspects of syntactically well-formed systems.

Once we have basic syntactic consistency, we must then worry about semantic consistency. It is for this reason that such approaches as found in the Inscape Environment [14] and Narayanaswamy and Scacchi [11] are important. The primary concern of Inscape has been exploring the constructive use of interface specifications in sequential programs. Thus, Inscape provides us a way of constructing the *semantic interconnections* [12] among components that provides the basis for reasoning about the initial consistency of system compositions and the preservation of consistency when substituting one component for another.

In my paper on version control in Inscape [13], I addressed the problem of reasoning about independent substitutions in system composition.s In this paper, I address the problem of interdependent substitutions — that is, the substitution of components that participate in shared dependencies. I first explain shared dependencies in section 2, the current state of the art in section 3, and then discuss the necessary groundwork in section 4. I discuss my approach to reasoning about shared dependencies in section 5. I first introduce the form of shared-dependency specifications and the method for constructing them in section 5.1. I next present the rules for determining the well-formedness of those specifications in section 5.2, and the rules for determining their satisfaction by compositions in section 5.3. I summarize the results of the paper in section 6.

2 Shared Dependencies

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 [20] — 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. We note in passing that the occurrence of multiple dimensions of organization is a general problem, not one endemic to switching systems.

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.

3 Current State of the Art and Research

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 [5] and Workshop [4]. Both provide facilities at what I call the *unit interconnection* 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 [8] 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 [10] 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 [2] 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 [15].

4 Preliminary Groundwork

The approach I present in the next section is based on the approach that I have previously taken in the Inscape Environment: specifying module interfaces in Instress [14], reasoning about the composition of components in the construction and evolution of systems [17], and reasoning about the relationships among component interfaces [13].

4.1 Interface Specifications

Instress module specifications contain three components: predicate definitions, data specifications, and operation specifications. Predicates may be defined as primitive (that is, uninterpreted), or in terms of other defined predicates and/or boolean predicates. Data specifications consist of a declaration with additional

properties to further constrain the values of the type or object. Operation specifications consists of a function or procedure declaration, a set of preconditions (a conjunct of predicates), and a set of results. Each result consist of a set of postconditions and a set of obligations (both are conjuncts of predicates) — see the example of an operation specification in Instress.

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.

The definition of consistency is straightforward:

> A predicate or set of predicates P is *consistent* if and only if it is not the case that P $\vdash$ false [1].

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

> An Interface Specification S = (P, D, O) is consistent if and only if
> - 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, and
> - each set of preconditions, postconditions and obligations for each operation O_i in O is consistent.

4.2 Interface Relationships

These formal descriptions of interfaces are the basis for reasoning about the relationships among various components and versions of components. Because of the semantic interconnections established during construction of the software, we can reason about the substitution of one version for another both dependent on the context of that construction and independent of it.

On the basis of Inscape's interface formalism, I defined the concepts of identity, equivalence, strict compatibility, upward compatibility, and various forms of implementation-dependent compatibility [13]. I separated the notion of compatibility into two distinct concepts: dependency preserving compatibility (strict) and functionality preserving compatibility (upward).

Of the two forms of context-independent upward compatibility, the more useful in reasoning about single substitution was strict compatibility. That utility is due precisely to its focus on dependency preservation.

> Operation O2 is a *strictly* compatible version of O1 if and only if

[1] In the reamining discussion, the logical notions are those of a standard first order predicate logic.

OpenFile(<in> filename fn, <out> fileptr fp)

returns unsigned int status

Synopsis:

If a file exists with the file name in fn, then the file is opened for I/O and a pointer to the file is returned for all further I/O operations.

Preconditions:

[V] LegalFileName(fn)

[V] FileExists(FileNamedBy(fn))

{Successful result}

Determined by:

status' == 0

Synopsis:

The file named in fn is open for i/o and fp references that file.

Postconditions:

LegalFileName(fn)

FileExists(FileNamedBy(fn))

FileOpen(*fp')

ValidFilePtr(fp')

FileNamedBy(fn) == *fp'

Obligations:

FileClosed(*fp')

{Exception Result: IllegalFileName}

Determined By:

status' == 1

Synopsis:

The file name is not a legal one.

Precondition failure:

LegalFileName(fn)

Postconditions:

not: LegalFileName(fn)

Obligations:

<none>

Recovery Method:

Correct the file name.

{Exception Result: NonExistantFile}

Determined By:

status' == 3

Synopsis:

The file named by fn does not exists.

Precondition failure:

FileExists(FileNamedBy(fn))

Postconditions:

LegalFileName(fn)

not: FileExists(FileNamedBy(fn))

Obligations:

<none>

Recovery Method:

Choose a name for a file that exists.

- PRE(O1) $\vdash$ PRE(O2) and
- POST(O2) $\vdash$ POST(O1) and
- OBL(O2) $\vdash$ OBL(O1) and OBL(O1) $\vdash$ OBL(O2).

That is, O2 requires no more than O1, produces no less and obligates equally. We shall see below that the more useful concept in reasoning about shared dependencies is upward compatibility, precisely for its emphasis on functionality (or if you will, behavioral) preservation.

The definition used in the rest of the paper for upward compatibility is that an upwardly compatible version preserves the original functionality or behavior while extending it.

Operation O2 is an *upwardly compatible* version of O1 if and only if
- PRE(O2) $\vdash$ PRE(O1) and
- POST(O2) $\vdash$ POST(O1) and
- OBL(O2) $\vdash$ OBL(O1).

Formally, the base component interface is derivable from the upwardly compatible component interface — that is, the preconditions, postconditions and obligations of the base component are derivable from the preconditions, postconditions and obligations of the upwardly compatible component.

4.3 Composition

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" [15], 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.

It is this notion of a composed sequence that will be of importance in the discussion of reasoning about shared dependencies. 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* [15]. 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 = sequence S = $S_1 \ldots S_N$ for a program fragment F is *complete* if and only if
- Every precondition in S has either been satisfied or is in the interface of F — that is, all precondition ceilings in S (recursively) are empty
- Every obligation in S has either been satisfied or is in the interface of F — that is, all obligation floors in S (recursively) are empty.
- There are no iteration errors — that is, the preconditions of each iteration are consistent with postconditions of their respective iteration bodies.

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* if and only if
- PRE(I) = $\emptyset$, and
- OBL(I) = $\emptyset$

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

5 Shared Dependency Specifications

We now have the basis 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.

5.1 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.

A Shared Dependency Specification SDS =
({ Partial Specifications }, { Partial Instantiations })

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 )
  }
)
```

5.2 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 we can address.

The basic intuition, given that we 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. We note, that for the sake of simplifying the presentation, we 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* if and only if
> - 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
>   - $\Gamma_k \vdash P_i$
> - For each Obligation O_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
>   - $\Gamma_k \vdash O_i$

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* if and only if

- 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* if and only if
- SD is consistent, and
- SD is strongly composable.

5.3 Sets of Shared Dependency Specifications

We mentioned in the discussion on shared dependencies that components often share in multiple dependencies. One has the choice of specifying these interrelated 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 })

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.

5.4 Satisfying Shared Dependency Specifications

We first consider the problem of a component satisfying a shared dependency specification, first for simple satisfaction and then for aggregate satisfaction. We then consider the problem of a composition satisfying a shared dependency specification. Finally, we 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 we 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, we 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 if and only if
> - C and SC are both predicates, C $\vdash$ SC and SC $\vdash$ C, or
> - C and SC are both type definitions (or they are both object declarations) and their types are equivalent, or
> - 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 A = ($SC_1, \ldots, SC_n$) if and only if

- PI is the propagated interface of a complete sequential composition of the components of A, and
- 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 from Composition Components $C_1 \ldots C_N$ to Shared Dependency Components $SC_1 \ldots SC_M$ is *well-formed* if and only if
- 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, and
- For each SC_j, SC_j appears in the range of only one composition component, and
- 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 of components $C_1 \ldots C_N$ and Map M *satisfy* a Shared Dependency Specification SD of components $SC_1 \ldots SC_M$ if and only if
- Map M is well-formed, and
- 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.

We 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 we 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.

6 Summary

In my prior work I was concerned about the syntactic well-formedness of system compositions, the semantic consistency of interface specifications, the semantic

interconnections created in system construction, and the semantic relationships between individual interfaces.

Here, I have presented a structure and method for specifying interrelated components that share some semantic aspect of a computation. My claim is that only with this extra information can one properly evolve the components that we use on constructing systems. Not only do we need to know the dependencies, but we also need to know the context of those dependencies. Shared dependency specifications provide a means of describing these contexts. Moreover, it is this sort of shared dependency information that enables us to systematically and effectively evolve our software systems: they provide us with the means of identifying some of the implications of changes to the system and reasoning about those changes prior to testing them.

This gain in understanding of system evolution comes at a cost: the shared dependency specifications tend to be somewhat large and cumbersome. Ameliorating this cumbersomeness is the fact that these specifications could easily be built as an adjunct of the construction and evolution process supported by environments such as Inscape with the environment providing much of the mundane work in capturing these shared dependencies.

It is my claim that the concepts introduced here enable one to effectively reason about both single and multiple component substitution and the effects of those substitutions. Satisfying the shared dependencies, however, is not sufficient in itself to guarantee consistency of a composition. Something like the semantic interconnections provided by Inscape during the construction and evolution of the system are also needed.

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. 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.
3. 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.
4. 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.
5. J. Estublier and R. Casallas. The Adele Configuration Manager. In [19], pp 73-97.
6. Stuart I. Feldman. Make — a Program for Maintaining Computer Programs. Software — Practice & Experience 9:4 (April 1979). pp 255-265.
7. A. Nico Habermann and Dewayne E. Perry. Well Formed System Composition. Carnegie-Mellon University, Technical Report CMU-CS-80-117. March 1980.

8. A. Nico Habermann and Dewayne E. Perry. System Composition and Version Control for Ada. Software Engineering Environments. H. Huenke, editor. North-Holland, 1981. pp. 331-343.
9. Gail E. Kaiser. Intelligent Assistance for Software Development and Maintenance. IEEE Software 5:3 (May 1988). pp 40-49.
10. Axel Mahler. Variants: Keeping Things Together and Telling Them Apart. In [19], pp 73-97.
11. K. Narayanaswamy and W. Scacchi. Maintaining Configurations of Evolving Software Systems. IEEE Transactions of Software Engineering, SE13:3 (March 1987), pp 324-334.
12. Dewayne E. Perry. Models of Software Interconnections. Proceedings of the 9th International Conference on Software Engineering, March 30 - April 2, 1987, Monterey CA. pp 61-69.
13. 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.
14. Dewayne E. Perry. The Inscape Environment. The Proceedings of the Eleventh International Conference on Software Engineering, May 1989, Pittsburgh, PA.
15. Dewayne E. Perry. The Logic of Propagation in the Inscape Environment. Proceedings of the ACM SIGSOFT '89 Third Symposium on Software Testing, Analysis, and Verification (TAV3), 13-15 December 1989, Key West, FL. Software Engineering Notes 14:8 (December 1989), pp 114-121.
16. M. J. Rochkind. The Source Code Control System. IEEE Transactions on Software Engineering, SE-1:4 (December 1975). pp 364-370.
17. Walter F. Tichy. Software Development Control Based on System Structure Descriptions. Ph.D. Thesis, Computer Science Department, Carnegie-Mellon University, January 1980.
18. Walter F. Tichy. RCS — A System for Version Control. Software — Practice & Experience, 15:7 (July 1985). pp 637-654.
19. Walter F. Tichy, editor. Configuration Management. Trends in Software, Volume 2. Chichester: John Wiley & Sons, 1994.
20. P. A. Tuscany. Software Development Environment for Large Switching Projects. Proceedings of Software Engineering for Telecommunications Switching Systems Conference, 1987.

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