AT&T Bell Laboratories
Murray Hill, New Jersey 07974

Computer Technology Research Laboratory
Technical Report

Version Control in the Inscape Environment

Dewayne E. Perry

published in

Proceedings of the 9th International Conference on Software Engineering.
March 30 - April 2, 1987, Monterey CA.
pages 142-149.

Version Control in the Inscape Environment

Dewayne E. Perry
AT&T Bell Laboratories
Murray Hill, NJ 07974
(201) 582-2529

Abstract

We present the important issues to be considered in version control mechanisms and characterize and compare the kinds of version control systems extant in current programming environments. We then characterize Inscape's version control mechanism, *Invariant*, and show that it makes several significant advances in the state of the art. Using *Instress* (Inscape's module interface specification language) specifications, Invariant provides a better understanding of the notion of parallel versions, a more comprehensive notion of version consistency, and a more flexible method of system composition than current mechanisms. In particular, Invariant provides a formalization of the notions of version equivalence and compatibility that correspond closely with our intuitive (and practical) notions of version equivalence and compatibility. These various forms of version compatibility provide the system builder with the concept of *plug-compatibility* — an extremely useful facility in composing systems from component parts.

1. Introduction

An extremely important problem in building large software systems is that of *programming-in-the-large* [1] — that is, building a large system out of components built by a number of programmers. There are four interrelated aspects to the programming-in-the-large process:

- describing the interfaces for the individual components,
- managing the variants of these interfaces and their implementations as the systems evolve,
- modeling, or configuring, the system from its components, and
- generating a system from its model or configuration specification.*

The design of each of these aspects affects the functionality that is possible in the others — each cannot be considered in isolation, but must be considered as part of the whole. In this paper, we consider the implications that the kinds of interface specifications found in the Inscape Environment [8] have on the problems of managing system and component variants and of configuring a system from its components.

We first discuss the background and present the issues found in what is generally referred to as version control. Second, we characterize and compare the kinds of version control systems extant in current programming environments. We then characterize Inscape's version control mechanism, *Invariant*, and describe its facilities. Finally, we summarize our contributions and compare them to current version control mechanisms.

2. Background and Issues

Multiple software versions arise in the building and evolving of software systems for a variety of reasons: error corrections, enhancements, alternate implementations for identical interfaces, divergent functionality for different uses, and differing system configurations. The result is an exceedingly difficult problem managing multiple software versions that arise during system evolution.

We distinguish three different kinds of versions: successive, parallel, and composed versions.** *Successive* versions are those that result from the small corrections and improvements that occur in the development process. They represent refinements to previous versions. *Parallel* versions are those that result from providing alternate implementations or divergent functionality. In the first case, the implementations usually represent different space-time trade-offs in the implementation of a single interface specification; in the second case, various uses of a component require a large degree of common functionality but with differences that are significant for each individual use. *Composed* versions are those that result from constructing a component from separate pieces, such as subsystems from modules, or modules from

* I distinguish the modeling from the generation of systems because the system generation mechanism may use more information than is provided by the system model and because other components in the programming environment may use the modeling information.

** Whenever we use the term *version* in the remainder of the paper, we mean *software version*.

syntactic objects.

A number of important questions arise from these distinctions. When does a revision generate a new parallel version instead of merely another successive version? When does a parallel version become a different version altogether? How does one determine the correct version to use when building a system? How will different versions interact — will they be consistent?

In answering such questions, we must consider the following important issues: version identification,* equivalence, compatibility, and consistency. Each of the mechanisms discussed below will provide explicit meaning to each of these terms. We give intuitive definitions here to provide a standard against which to evaluate these mechanisms.

Version Identification	Version identification enables us to distinguish one version from another.
Version Equivalence	If two versions are equivalent, we ought to be able to use them interchangeably without any unexpected effects.
Version Compatibility	Compatibility is not as strict as equivalence. For example, we have an intuitive understanding of upward compatibility: an upward compatible version preserves the functionality of the previous version but makes some extensions; however, previous uses should remain unaffected by the changes.
Version Consistency	There are two forms of version consistency: syntactic consistency and semantic consistency. A version is *syntactically consistent* with a set of components in a composed version when it does not cause any syntax or type-checking errors to occur. A version is *semantically consistent* with a set of components in a composed version when it does not cause semantic (or functionality) errors to occur — that is, it provides the behavior that is required by the system.

3. Current Version Control Mechanisms

There are three general approaches to the problems of version control and system configuration that exist in current software development environments: no control at all, basic version control, and strongly-typed version control. We describe and evaluate each of these both in terms of the kinds of versions they support and in terms of their solutions to the basic issues in version control.

* Note that we do not discuss the problem of one version being *identical* to another. It is sufficient for our discussion here to be able to distinguish different versions by some means of identification and then talk about equivalent versions, instead of identical versions. Thus we sidestep the philosophical and legal problems of what it means for two versions to be identical.

3.1 No Version Control

It is surprising how many systems are still built without a version control mechanism as part of the development environment. At best, developers may get some notion of successive versions from the file system (if the file system incorporates some form of backup files); at worst, there is no notion at all of successive versions, only the current version. There may be some vague notion of parallel versions where the current version gets frozen for a particular release. Composed versions are whatever was thrown together at one particular time with no concept of a configuration specification.

The identification, equivalence, and compatibility of versions are determined by programmer fiat: versions are identified by the programmer; versions are equivalent or compatible because they are said to be so by the programmer. Version consistency, on the other hand, is determined by a cycle of compile, link, execute, and test.

In summary, there is no control at all. System building is a very error-prone process because there is no means of determining clearly what needs to be incorporated into the system or how it is to be done. Furthermore, it is extremely difficult to reconstruct previous versions should the current system prove to be unusable.

3.2 Basic Version Control

Systems such as SCCS [10], RCS [11], Cedar's System Modeler [5], and Apollo's DSEE [6] provide basic version control.* The three kinds of versions (successive, parallel and composed versions) are supported within the version control system. Versions are distinguished by their version identifiers which in their simplest form are a pair of numbers separated by dots — for example, 1.4, 4.17, etc. The first number indicates the basic version and the second indicates the revision (i.e., the successive revision). Successive and parallel versions are distinguished from each other (by convention) by the form of their version identifiers: to create a new parallel version, append another "n.1" to the current version number where n is the nth parallel version spawned from that version; to create a new successive version, increment the rightmost number of the version identifier. For example, 3.5.1.1 and 3.5.2.1 are two new parallel versions created from version 3.5; 3.6 is the new successive version created from version 3.5.

Composed versions are denoted by source lists defined by the user. These source lists define the unit components in the composition and specify the appropriate versions that are to be used. For example, *{CC 1.4, Grep 3.2.1.1, ... , Stdio 1.17}* and *{ * *.4.1 }* might be two such compositions. The first lists each version explicitly; the second might designate all of the initial versions in release 4.

This basic approach to version control provides a workable solution to the problem of version identity. Modules are *reserved* for making changes and then *deposited* back into the version control system when the changes are completed. At the time of reservation, a new version identifier is automatically generated and assigned to the modified component. Unfortunately, there is no system notion of version equivalence or

* We describe here a generalization of SCCS and RCS. System Modeler and DSEE, while richer in functionality and different in details, still support only this basic form of version control.

compatibility. Consistency, as in the case where there is no version control mechanism, is determined by the cycle of compile, link, execute, and test.

While we have support for the required kinds of versions, there is no system-determined difference between successive and parallel versions — that is, there is no internal rationale that distinguishes them; as in systems with no control, they are determined by programmer fiat. Note, however, there is often an administrative rationale provided for creating new parallel versions — for example, each release may create new parallel versions.

Compositions in SCCS and RCS must explicitly designate each version of each component that is to be included in the system. There is no notion of default version. Furthermore, the actual dependencies between components in the system configuration are implicit in the code and not explicit in the descriptions of the versions.

3.3 Strongly Typed Version Control

Where basic version control mechanisms provide version control at the level of files or modules, strongly typed version control mechanisms such as Gandalf's SVCE (see [2] and [4]) provide a version control mechanism at the level of syntactic objects — that is, the granularity of versions is smaller than files or modules; it is at the level of procedures, functions, data structures, etc. In SVCE, not only are successive, parallel and composed versions supported by the mechanism, they are syntactic objects in SVCE's programming-in-the-large language. These different kinds of versions are understood as part of the language to describe versions and systems. Successive revisions are produced automatically as part of the SVCE environment's management of the change process. Parallel versions, on the other hand, are still determined by programmer choice as in the previous mechanism.

In SVCE, compositions refer to syntactic objects (such as procedures, modules, data structures, etc.) and may refer to them in varying degrees of specificity. For example, a composition may refer to a specific version, the latest version, or the current default version.* Where dependencies in the basic mechanisms were implicit, SVCE provides facilities for explicitly defining these dependencies. Using these facilities, the builder of components can determine clearly what is required in order to configure a specific portion of a system.

Version identification in SVCE is better (and intuitively easier to comprehend) than that provided by SCCS-style mechanisms. There are distinct means of identification for parallel and successive versions. To obtain a particular version, the builder specifies the particular successive version of a particular parallel version. For example, a complete version identifier consists of the parallel version identifier concatenated with the appropriate successive version identifier. This latter identifier, as in the previous mechanism, is generated automatically by the system at the time of deposit after a sequence of changes.

* System Modeler and DSEE provide similar facilities but for the file and module level of version specifications.

Version equivalence in this type of version control mechanism is defined in terms of syntactic equivalence: data objects are equivalent if their types or structures are equivalent; operations and modules are equivalent if their signatures are equivalent. (Signatures of operations are equivalent if the operation names are identical and the number, order and types of the arguments are identical. Modules are equivalent if their names are identical and their components are equivalent.) However, there is no explicit notion of version compatibility except in the sense that what is not equivalent is incompatible.

As this is a mechanism that is built upon the foundations of strong typing, it is not surprising that the syntactic consistency is guaranteed by the version control mechanism. The composition specifications can be type-checked by the system independent of the system generation cycle. However, the semantic consistency of composed versions must still be determined by the generate and test cycle.

While there are significant gains in functionality and in system support with these kinds of version control mechanisms, there are still some important problems that have not been solved well. The notion of version equivalence does not correspond to our intuitive notion of version equivalence. On the one hand, the strongly typed definition is too broad. Versions are considered equivalent that clearly are not equivalent in our intuitive sense of equivalence. For example, two versions of an operation may have the same signature but have radically different functionality. This case violates our intuitive sense of equivalence where we ought to be able to substitute the one for the other without any difference in the system execution. On the other hand, the strongly typed definition is too narrow. Versions are not considered as equivalent where intuitively we would consider them to be so. Consider the case where operations *A* and *B* provide exactly the same functionality but have different signatures (which in this mechanism guarantees their incompatibility and, hence, their non-equivalence). Furthermore, the notion of compatibility is too strict. For example, consider the operations *P(int)* and *P(int, bool)*. According to their parameter types, they are incompatible, but in an intuitive sense, the second is the upward compatible version of the first — the second preserves the functionality of the first and extends it in a small way by means of the second parameter.

4. Invariant, Inscape's Version Control Mechanism

The Inscape Environment is an environment for constructing and evolving large software systems and is based on the constructive use of formal module interface specifications [7]. These interface specifications describe the semantics of data and operations and are used by the environment* to enforce automatically their consistent use and to form the basis of a symbiotic relationship between the developer and the environment in the construction and evolution of large systems.

We present an overview of *Instress,* Inscape's module interface specification language, in order to give the reader an intuition for the specifications used in Inscape. We then discuss the notions of versions and Invariant's solutions to version control issues, describing in detail the precise definitions of version

* See [9] for a discussion of the semantic interconnection model that provides the basis for Inscape and its ability to use specifications symbiotically with the programmer.

equivalence and compatibility that result from our specification approach. Finally, we evaluate our mechanism and compare it with the preceding mechanisms.

4.1 Inscape's Module Interface Specifications

```
ReadRecord   ( <in> fileptr FP; <in> int R;
               <out> int L; <out> buffer B)
      Preconditions:
            ValidFilePtr(FP)
            FileOpen(FP)
            LegalRecordNr(R)
            RecordExists(R)
            RecordReadable(R)
            RecordConsistent(R)
      Postconditions:
            ValidFilePtr(FP)
            FileOpen(FP)
            LegalRecordNr(R)
            RecordExists(R)
            was RecordReadable(R)
            was RecordConsistent(R)
            Allocated(B)
            RecordIn(B)
            BufferSizeSufficient(B, L)
      Obligations:
            Deallocated(B)
```

Example 1

Instress (the module interface specification language) extends Hoare's input/output predicates [3] in order to describe the properties of data and the behavior of operations. In addition to Hoare's preconditions and postconditions, we introduce the notion of *obligations* to extend the specification of an operation's results. Postconditions by themselves are not sufficient to capture all the side-effects of an operation; they describe what is known to be true after the execution of an operation, but do not specify what the programmer is obliged to satisfy eventually as a result of using the operation (e.g., closing files, deallocating buffers, making data consistent, etc.). *Example 1* illustrates what a specification might look like for the successful execution of reading a record from a file. Further, we provide the notion of multiple results so that exceptions can be precisely and exactly specified in terms of what they mean (i.e., their postconditions) and what is minimally required to handle them (i.e., their obligations). Pragmatic information is also included with the exception specifications to indicate recommended recovery techniques or specific recovery operations. *Example 2* illustrates what an exception specification might look like for the *ReadRecord* operation where the data is readable but not consistent.

We use this Hoare-like approach in *Instress* rather than an algebraic approach such as [Goguen 82] and [Guttag 85] because it seems better suited to specifying exceptions and obligations that may occur and what they mean when they do occur.

Exception:
RecordInconsistent(R)
Postconditions:
ValidFilePtr(FP)
FileOpen(FP)
LegalRecordNr(R)
RecordExists(R)
was RecordReadable(R)
not RecordConsistent(R)
Allocated(B)
RecordIn(B)
BufferSizeSufficient(B, L)
Obligations:
Deallocated(B)
Recovery:
ReconstructRecord(L, B)

Example 2

4.2 Versions and Issues

The notions of versions in Invariant are quite similar to those found in SVCE. We extend SVCE to incorporate Inscape's knowledge about the specifications of the semantics of module interfaces. Successive versions are the same as in SVCE and compositions are similar to, but more flexible than, those in SVCE. It is in parallel versions that Invariant provides a significant difference over the preceding version control mechanisms: Invariant can distinguish different parallel versions to the extent that they exhibit similar but slightly different behavior. The extent to which this can be accomplished will be made clear in the discussion below.

Basic version control and strongly typed version control mechanisms provide us with workable notions of version identity. These mechanisms are deficient, however, in providing notions of version equivalence and compatibility that correspond to our intuitive understanding of these concepts. Given the behavioral specifications provided by Instress, Invariant provides precise definitions of these concepts that do correspond very closely to our intuition. Moreover, we introduce some further distinctions about compatibility that are very useful in the composition of systems within the Inscape Environment and that provide the system builder with considerably more flexibility in this building process than any other version control mechanism.

As we have extended the notions of equivalence and compatibility on the basis of the semantic specifications, so we extend in Invariant the amount of consistency checking than can be performed on system compositions. As in SVCE, we guarantee the syntactic consistency among the objects in the composition, but in a looser sense than found in strongly typed systems. There is some relaxation possible concerning the signatures in syntax checking. The discussions in the next two sections will clarify how this is done. Furthermore, semantic consistency among the components is guaranteed (within the limits of the strength of consistency checking provided by the Inscape Environment — see [8]).

4.3 Version Identity and Equivalence

For purposes of simplifying the discussion, we limit the scope of the definitions to a single successful result of an operation — the extension to multiple results (i.e., exceptional results) incorporates the same considerations that we discuss in the single successful case and the extension to data specifications can be made by analogy. There should be no loss of generality by making this simplification.

Let us 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 definitions. Thus, the problem whether interfaces, that differ only in their choice of parameter names, are identical disappears since it is their instantiations with arguments that are used to determine identity.

An operation interface I2 is identical to I1 if and only if

PRE(I1)	*= PRE(I2) and*
POST(I1)	*= POST(I2) and*
OBL(I1)	*= OBL(I2).*

Where *PRE(I) is the set of preconditions for operation I,*
POST(I) is the set of postconditions for operation I, and
OBL(I) is the set of obligations for operation I.

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

A version V2 is identical to version V1 if and only if

the interface of V2 is identical to the interface of V1 and
their implementations are identical.

More important is the notion of version equivalence.

A Version V2 is equivalent to version V1 if and only if

their interfaces are identical.

4.4 Version Compatibility

We present four different kinds of compatibility: strict, upward, implementation, and system compatibility. The first two are forms of our intuitive notion of upward compatibility: the first captures the notion of substitutability and the second captures the notion of extended functionality. The first notion is, in fact, the more useful of the two as far as building systems is concerned.

Version V2 is a strictly compatible version of V1 if and only if

$$\begin{aligned} &PRE(V1) &&\supseteq PRE(V2) \text{ and} \\ &POST(V1) &&\subseteq POST(V2) \text{ and} \\ &OBL(V1) &&= OBL(V2). \end{aligned}$$

That is, version V2 is a strictly compatible version of V1 if and only if it assumes no more than V1, guarantees no less than V1, and obliges the same as V1.* This form of compatibility guarantees that a strictly compatible version may be substituted for any occurrence of a version it is compatible with for the following reasons:

- since its preconditions are a subset of the original, they will be either be satisfied or propagated in exactly the same way as the original occurrence;
- since its postconditions are a superset of the originals, all dependencies provided by the original occurrence will still be satisfied by this version (at worst, there will be a beneficial side effect in that some preconditions may be satisfied that were not satisfied before); and
- as the obligations are identical, there will be no change in the interconnections as a result of the substitution.

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

Version V2 is an upwardly compatible version of V1 if and only if

$$\begin{aligned} &PRE(V1) &&\subseteq PRE(V2) \text{ and} \\ &POST(V1) &&\subseteq POST(V2) \text{ and} \\ &OBL(V1) &&\subseteq OBL(V2). \end{aligned}$$

While this form is 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.

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

* Note that if V1's obligations are included in V2's, then the source may have to be modified to cover the extra obligations incurred by V2. Similarly, if V2's obligations are included in V1's, then it may be the case that too much is done in the implementation if V2 is substituted for V1 — that is, obligations will be met that are non-existent.

as a replacement component in a composition even though it is neither strictly nor upwardly compatible with the original version.** To this end we 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** of that operation.

A version V2 is exactly implementation compatible with V1 if and only if

PI{..., V1, ...} = PI{..., V2, ...}

Where PI{...} is the propagated interface of the implementation and
{..., V, ...} is the implementation with version V.

Clearly, any version that is equivalent is also exactly implementation compatible. A version that is 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.

We 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.

A version V2 is strongly implementation compatible with V1 if and only if

PI{..., V2, ...} is a strictly compatible version of PI{..., V1, ...}

Clearly, a strictly compatible version will be strongly implementation compatible. Here, as in the more restrictive form, it is possible for an operation not to be strictly or upwardly compatible and still be strongly implementation compatible.

** 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.

** In [8], we 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.

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. 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.

We 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.

A version V2 is α *system compatible with V1 if and only if*

V2 is an α *implementation compatible version of V1*
for all occurrences of V1 in the system.

where α *is either "exactly", "strongly" or "weakly".*

5. Summary

The use of Inscape's module interface specification language Instress as a programming-in-the-large language enables us to make several significant advances in the state of the art of version control. With respect to the kinds of versions that are supported in version control mechanisms, Invariant provides the following advances.

- The environment has a better understanding about the notion of parallel versions — one can talk about parallel versions either in terms of version equivalence or, possibly, upward compatibility. The problems concerning questions about parallel versions are by no means all solved, but Invariant's facilities for determining version equivalence and compatibility provide a start towards solving them.
- Invariant provides a more liberal, flexible method of composition wherein pieces of the system can be built from other components than is available in other version control mechanisms. Compositions are freed from the adverse restrictions of strong typing and instead are guided by the notions of version equivalence and compatibility that incorporate the more useful aspects of strong typing with the semantic specifications of interfaces.

It is with respect to the issues of version equivalence, compatibility and consistency, that Invariant's formalization offers the most significant contributions.

- Inscape's consistency checking, used in the process of constructing and evolving systems, is used by Invariant to determine the consistency of different versions in the process of building systems out of components.

- The definition of version equivalence provided by Invariant matches our intuitive notion that versions are equivalent when they have identical behavior.
- The various forms of compatibility correspond very closely to our intuitive notions about compatibility among versions. Moreover, these forms of compatibility provide a unique notion of *version plug-compatibility* that enables the system builder a high degree of freedom and flexibility in composing systems. The builder may substitute for the original components either equivalent versions (with no ill effects) or non-equivalent but compatible versions (and use Invariant to determine the effects of these substitutions).

References

[1] Frank DeRemer and Hans H. Kron. "Programming-in-the-Large Versus Programming-in-the-Small", *IEEE Transactions on Software Engineering*, SE-2 (June 1976). pp. 80-86.

[2] 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.

[3] C. A. R. Hoare. "An Axiomatic Approach to Computer Programming", *CACM* 12:10 (October 1969). pp. 576-580, 583.

[4] 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.

[5] Butler W. Lampson and Eric E. Schmidt. "Organizing Software in a Distributed Environment", *Proceedings of the SIGPLAN '83 Symposium on Programming Language Issues in Software Systems*, *SIGPLAN Notices*, 18:6 (June 1983).

[6] 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.

[7] Dewayne E. Perry. "Position Paper: The Constructive Use of Module Interface Specifications", *Third International Workshop on Software Specification and Design*. IEEE Computer Society, August 26-27, 1985, London, England.

[8] Dewayne E. Perry. *The Inscape Program Construction and Evolution Environment*. Technical Report. Computer Technology Research Lab, AT&T Bell Laboratories, August 1986.

[9] Dewayne E. Perry. "Software Interconnection Models", This proceedings, *Proceedings of the 9th International Conference on Software Engineering*, March 30 — April 2, 1987, Monterey CA.

[10] M. J. Rochkind. "The source code control system", *IEEE Transactions on Software Engineering*, SE-1 (1975). pp. 364-370.

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