Software Configuration Management Overview

Walter Tichy

Software Configuration Management

Software configuration management (SCM) is the discipline of controlling the evolution of complex software systems. This chapter surveys tools that support or automate aspects of SCM. It proposes a standard terminology, describes the areas that are amenable to automation, discusses a representative set of existing SCM tools, and identifies directions for future research and development. A glossary of terms is included.

1 Introduction

Configuration management (CM) is the discipline of controlling the evolution of complex systems; software configuration management (SCM) is its specialization for computer programs and associated documents. General CM is beneficial for any large system that. Due to its complexity, cannot be made perfect for all the uses to which it will be put. Such a system will be subject to numerous, sometimes conflicting changes during its lifetime, giving rise not to a single system, but to a set of related systems, called a *system family.* A system family consists of a number of components that can be configured to form individual family members. A substantial number of the components must be shared among members to make the family economically viable. Maintaining order in large and expanding system families is the goal of CM.

SCM differs from general CM in the following two ways. First, software is easier to change than hardware, and it therefore changes faster. Even relatively small software systems, developed by a single team, can experience a significant rate of change, and in large systems, such as telecommunications systems, the update activities can totally overwhelm manual configuration management procedures. Second, SCM is potentially more automatable. Because all components of a software system are easily stored on-line. CM for physical systems is hampered by having to handle objects that are not within reach of programmable controls. As CAD/CAM and robotics bring manufacturing processes more and more under computer control physical configuration management will undoubtedly adopt some of the approaches used for software. VLSI already does: Circuit design and circuit processing can be managed like software design and compilation.

Effective software configuration management coordinates programmers working in teams. It improves productivity by reducing or eliminating

some of the confusion caused by interaction among team members. The coordinating functions of configuration management are introduced below, and illustrated with questions or statements familiar to anyone who has worked in software development.

Identification

Identifying the individual components and configurations is a prerequisite for controlling their evolution. Reliable identification helps avoid the following problems:

- "This program worked yesterday. What happened?"
- "I can't reproduce the error in this configuration."
- "I fixed this problem long ago. Why did it reappear?"
- "The online documentation doesn't match the program."
- "Do we have the latest version?"

Change Tracking

Change tracking keeps a record of what was done to which component for what reason, at what time, and by whom. It helps answer the following questions:

- "Has this problem been fixed?"
- "Which bug fixes went into this copy?"
- "This seems like an obvious change. Was it tried before?"
- "Who is responsible for this modification?"
- "Were these independent changes merged?"

Version Selection and Baselining

Selecting the right versions of components and configurations for testing and baselining can be difficult. Machine support for version selection helps with composing consistent configurations and with answering the following questions:

- "How do I configure a test system that contains my temporary fixes to the last baseline, and the released fixes of all other components?"
- "Given a list of fixes and enhancements, how do I configure a system that incorporates them?"
- "This enhancement won't be ready until the next release. How do I configure it out of the current baseline?"
- "How exactly does this version differ from the Baseline?"

Software Manufacture

Putting together a configuration requires numerous steps such as pre- and postprocessing, compiling, linking, formatting, and regression testing. SCM systems must automate that process and at the same time should be open for adding new processing programs. To reduce redundant work, they must manage a cache of recently generated components. Automation avoids the following problems:

- "I just fixed that. Was something not recompiled?"
- "How much recompilation will this change cost?"

- "Did we deliver an up-to-date binary version to the customer?
- "I wonder whether we applied the processing steps in the right order."
- "How exactly was this configuration produced?"
- "Were all regression tests performted on this version?

Managing Simultaneous Update

Simultaneous update of the same component by several programmers cannot always be prevented. The configuration management system must note such situations and supply tools for merging competing changes later. In so doing it helps prevent problems like the following:

- "Why did my change to this module disappear?"
- "What happened to my unfinished modules while I was out of town?"
- "How do I merge these changes into my version?"
- "Do our changes conflict?"

This chapter discusses software tools for automating the functions introduced above. The basis of all tools is representation, so we develop a model for representing multi-version / multiconfiguration systems. Section 2 establishes basic terminology, while Sections 3 and 4 introduces versions. Later sections on version selection, software manufacture and modification requests can be read in any order. Background material and manual CM procedures can be found in References [3, 5, 7].

2 Basic SCM Concepts

This section defines the basic elements of a data base for software configuration management. The data base stores all software objects produced during the life-cycle of a project.

A *software object* is any kind of identifiable, machine-readable document generated during the course of a project. The document must be stored online to be fully controllable by an SCM system. Examples of software objects are requirements documents, design documents, specifications, interface descriptions, program code, test programs, test data, test output, binary code, user manuals, or VLSI designs.

Every software object has a unique identifier and a body containing the actual information. A set of attributes associated with software objects and a facility for linking objects via various relations are also needed. For example, attributes record time of creation and last read access, and relations link objects to their revisions and variants. The set of attributes and relations must be extensible; later sections will introduce a basic set. We also need a facility to describe subclasses or subtypes of the general software object. For instance, the subclass may fix the language in which the body is written, or the structure editor used to compose the body, or whether the object represents an interface or an implementation. The subclass also defines the set of operations available on objects of that class, such as compiling, configuring, printing, etc.

The body of a software object is immutable, that is, once the body has been completed, it can only be read. Any "change" of a body actually creates a new software object with the changed body. Immutability is important for configuration management, because it prevents misidentification: an object identifier is associated with one and only one constant body, and not with several different versions. Most other attributes and relations of software objects remain changeable, however, so new information can be added.

Software objects have two orthogonal refinements, one according to how they were created, the other according to the structure of their body. For creation, we distinguish source and derived objects. For internal structure, we distinguish atomic objects and configurations.

2.1 Creation of Software Objects

A *source object* is a software object that is composed manually, for instance with an interactive editor. Creating a source object requires human action; it cannot be produced automatically.

A *derived object* is generated fully automatically by a program, usually from other software objects. A program that produces derived objects is called a *deriver*. Examples of derivers are compilers, linkers, document formatters, pretty printers, cross referencers, and call graph generators. Normally, derived objects need not be stored, since they can be regenerated, provided both the deriver and the input are available or can be rederived. To reduce the delay caused by regeneration, a smart configuration management system maintains a cache of derived objects that are likely to be reused.

Unlike derived objects, which can be deleted to make room, source objects are "sacred", because deleting them may cause irreparable damage or at least significant delay until they are reconstructed. However, derived objects may also become "sacred", i.e., they must not be deleted merely to make room, if it is impossible or time consuming to reproduce them. For instance, derived objects that are imported from other sites, especially vendor supplied programs, must not be deleted, even though they are derived in most cases. Another example are derived objects for which the original derivers have stopped working (if they have not been ported to new hardware, say), or if the corresponding input objects have been lost.

A special case are derived objects that are modified manually. Examples are automatically generated program skeletons and templates that are fleshed out by hand, or object code that is patched manually. In principle, these manual modifications produce new source objects.[1] However, the SCM system should store a *traceability* link that records the dependency between the two objects. This link can be used for generating a reminder to update the source object if the derived object changes. Traceability links should also be recorded among dependent source objects, for example between a specification and its implementation, or a program and its documentation. In fact, most source objects in an SCM system depend on one or more other objects,

1. *However, a patch that is implanted fully automatically by a program produces a derived object. In that case, the patching program or the input to a general patching program are the corresponding source objects*.

except perhaps the initial requirements specification. Traceability information is extremely valuable for automatically producing update reminders, for reviewing completeness of changes, and for informing maintainers what information they need to consider when preparing a change.

2.2 Structure of Software Objects

The body of a software object is either atomic or structured. An *atomic object*, or *atom,* has a body that is not decomposable for SCM; its body is an opaque data structure with a set of generic operations such as copying, deletion, renaming, and editing. An atomic object may consist of a program written in some language, a syntax tree produced by a structure editor, a data structure generated by a WYSIWYG word processor, or an object code module produced by a compiler.

A *configuration* has a body that consists of sub-objects, which may themselves have subobjects, and so on. Configurations have two subclasses: composites and sequences. A *composite object,* or simply *composite,* is a record structure comprised of fields. Each field consists of a field identifier and s field value. A field value is either an object identifier or a version group identifier. An example of a composite object is a software package consisting of a program, a users manual, and an installation procedure. Another example is a regression test object, consisting of a test program, input data, expected output data, and a comparator for comparing expected and actual output. Thus, fields may contain data as well as operations.

A *sequence* is a list of object and version group identifiers. Sequences represent ordered multisets of objects. They are used for combining sub-objects that are of the same class, or when the number of sub-objects is indeterminate. In contrast to composites, the individual elements of a sequence fulfill identical roles and are treated in the same way for SCM purposes, such as the list of object code modules constituting a library.

Note that the above definitions permit version group identifiers in composites and sequences. A version group is a set of related source or derived objects that can replace each other under certain assumptions (see Sections 3 and 4 for details). The purpose of version groups here is to permit compact representations of multiple software objects with the same structure. By using a version group identifier instead of an object identifier, configurations need not be updated if new versions are added to the groups. On the other hand, a version selection process must decide which versions to choose when processing such configurations.

Because of the need to distinguish between "precise" and "loose" configurations, we introduce the following terms. A *generic composite* is a composite with at least one field value that is either a version group identifier or a generic configuration (i.e., a generic composite or a generic sequence). The opposite of a generic composite is a *baseline composite,* which is a composite whose field values are atomic objects, baseline composites, or baseline sequences. The subclasses g*eneric sequences* and *baseline sequence* are defined analogously. Finally, a *generic configuration*, also called a *system model*, is a generic composite or a generic sequence. A *baseline configuration*, or simply *baseline,* is a baseline composite or baseline sequence.[1]

We follow Clemm [9] in stipulating that derivers which produce several outputs must package them into a single, derived configuration. For example, a compiler which produces object code, a list of warnings, and a symbol table would store all three of these derived objects into one composite. This convention simplifies the bookkeeping involved in managing derived objects.

In both composites and sequences, source and derived objects may be freely intermixed. However, including derived objects presents a problem: Since the derived objects may not yet exist, there may be no known identifiers for them. Instead, we must represent a derived object with a descriptor that will cause the object to be generated when it is needed. This descriptor must specify not only the derivers, but all parameter settings for the derivers as well. If some of the parameter settings are under-specified, then the version selection process must choose and record them (see Section 5).

For clarity, we should point out some uses of the above definitions. Suppose a software house delivers a single, binary program to a customer. This program is a single, derived object. It most cases, this object was generated from a baseline configuration recorded at the software house. The purpose of the baseline is to guarantee that the derived object can be reproduced when needed. The software house may also deliver a configuration, perhaps a composite that consists of one or more binaries and a manual. The delivered configuration may also contain source programs, because the programs will be interpreted, or because the customer wishes to compile source locally. The customer may also need to adapt the source code to local needs. Thus, depending on how much the customer expects to do, a more or less complete SCM system must be available at the customer site to take over portions of the software house's SCM functions.

3 Source Versions

SCM systems have to cope with constant change. Corrective, adaptive, and perfective maintenance activities produce a steady stream of updates. Since most changes are incremental, they are best viewed as producing related versions of objects rather than separate, unrelated objects. This section deals with versions of source objects; versions produced by derivers will be treated in Section 4.

3.1 Source Version Groups

An important concept for dealing with multiple versions is the *source version group*. A source version group is a set of source objects that are connected via the relations *revision-of, variant-of,* and their subtypes. These relations are defined below. Note, however, that source version groups may contain atoms, composites, sequences, and even mixtures of those.

- **y revision-of x:** This relation holds if and only if *x* and *y* are source objects and *y* was produced by changing a copy of *x*. Thus, *revision-of* records the development history of source objects. The subtypes of this

1. The term "parametric" is sometimes used as synonym for "generic".

relation, *correction-of, adaptation-of,* and *enhancement-of,* capture the nature of the change. It is possible for several of these subtypes to hold simultaneously between a pair of objects.

The relation *revision-of* and its subtypes are transitive, antisymmetric, and irreflexive. Objects of a version group that are transitively related by *revision-of* etc. are simply called *revisions.*

- **y variant-of x:** This relation holds if and only if *x* and *y* are source objects that are indistinguishable under a given abstraction. An abstraction defines relevant properties while ignoring (irrelevant) details. It permits variation by **not** prescribing certain properties o behaviors. The intent is to define abstractions in such a way that variants can replace each other in a software systems without requiring changes in their client programs.

  Variant-of is actually a ternary relation, since it must identify a common abstraction. Few programming environments permit the specification of such an abstraction. One approach is to introduce subsets of interfaces, called *views.* Another, more promising approach is to represent abstractions explicitly as superclasses in object-oriented programming languages

  A commonly used abstraction is the functional specification. The functional specification ignores space and time efficiencies, so two variants under this abstraction may differ in internal algorithms and data structures. Similarly, one may decide to ignore the choice of programming language, target machine, target operating system, or target user group. As long as the functional specifications of variants are identical, client programs depending of only the functional specification do not have to be rewritten if a different variant is chosen. Thus, software systems can be reconfigured by merely replacing individual objects. The interested reader is referred to Parnas' work [30, 31] for criteria on how to design software systems in such a way that likely changes can be hidden behind invariant interfaces.

  For some abstractions, it is possible that details even in the functional specification are irrelevant. For example, sorting programs can be classified as stable or unstable. A stable sort guarantees to leave elements with the same sorting key in the original order. Under some abstraction, stableness of sorting procedures may be irrelevant. Thus, the common property of two variants can be a subset of their functional specifications. A common manifestation of this aspect is that only a subset of the interface made available by a program is used by clients.

  For a given abstraction, the relation *variant-o*f is an equivalence relation because it is transitive, symmetric, and reflexive. Objects in a version group that are transitively related by *variant-of* are simply called *variants.* Subclasses of *variant-of* describe the abstraction under which the variants are indistinguishable. Characterizing variants under system-defined and user-defined abstractions is a topic of current research in SCM.

The relation *variant-of* may or may not parallel *revision-of,* depending on whether the variant was produced by changing an existing object. Variants must usually be maintained in parallel, and in practice their number should be kept small.

The relations *revision-of, variant-of* and their subtypes apply to configurations. This is in contrast to SCM systems like SCCS [34], RCS [39], CMS [1], and DSEE [25], where versions of configurations appear to be an afterthought. For example, with RCS, one would have to collect descriptions of all configurations and subconfigurations into a single, atomic object called a *Makefile,* and allow versions of the entire set only. Versioning of configurations at this level is on too coarse a grain for effective SCM. The Gandalf project [16, 41] was among the first to experiment with versions of configurations (called *compositions)* as well as variants (called *implementations* or *realizations).*

3.2 Structure of Source Version Groups

As defined previously, a source version group is a set of source objects that are related via *revision-of* and *variant-of* For simplicity, we assume that version groups are closed with respect to these relations. In other words, no *revision-of* and *variant-of* link may cross version group boundaries[1].

FIGURE 1

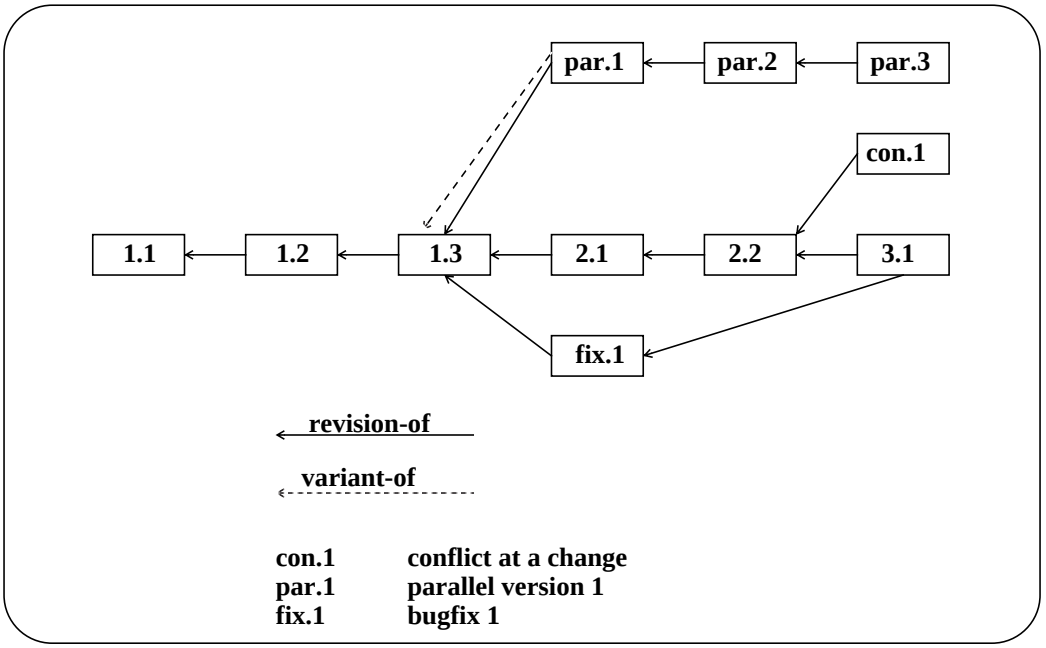

Figure 1. A source version group with revisions and variants

Revision-of forms a directed, acyclic graph reflecting the development history, whereas *variant-of* identifies the starting points of parallel lines of devel-

1. *Closure is not strictly necessary. Sometimes it is convenient to make some revision in a group the initial revision in another.*

opment. For a young source version group without variants, the graph structure is simple: It consists of a single list linked via *revision-of* that begins with the most recent object and ends with the oldest. At least initially, this list represents the main line of development and is often called the main branch or trunk. As the version group ages, side branches may form. Some of these side branches may wither, others may later be merged with the main branch. Side branches are needed for accommodating parallel development, conflicting updates, and temporary fixes.

Consider FIGURE 1 as an example of a source version group illustrating various types of branches. The revisions numbered 1.1,1.2,...,3.1 represent the main branch. The revisions *par.*l, *par.*2, and *par.*3 constitute a parallel line of development. Note that *par.*l is both a variant and a revision of 1.3. A special case of parallel development is distributed development, in which customers modify released software themselves. The modifications can be relayed back to the development organization for merging into the next release, or must be merged into future releases by the customers locally. See Reference [39] for an example of how to set up version groups for distributed development.

Revisions 3.1 and *con.*l illustrate conflicting updates. This situation arises when two programmers wish to update the same revision (here: 2.2) simultaneously, and neither can wait for the other to finish. This situation is undesirable, yet cannot always be prevented in practice. SCM should warn programmers in this case, but allow work to proceed by forming a temporary side branch for later merging. Note that such conflicts can only occur at branch tips. Reference [39] discusses a range of strategies for dealing with these conflicts.

Revision *fix.*1 illustrates the handling of temporary fixes. Suppose the need to correct revision 1.3 arises after 2.1 and 2.2 have been completed. To reflect the actual development history, SCM places the correction on a side branch starting at revision 1.3. The correction is later merged with 2.2, resulting in 3.1.

3.3 Operations on Source Version Groups

Virtually all SCM systems in use today use some form of a check-out / edit / check-in cycle for adding revisions to source version groups. The *check-out* operation creates a copy of the revision to be modified and reserves it for the user. Check-out also links the new copy to its original with the *revision-of* relation. The user can then update the copy with an arbitrary editor. As long as the copy remains checked out, it remains inaccessible to others. Any subsequent check-out of the same original revision causes a branch to form, with a warning stating that a merge operation will be necessary later. The *check-in* operation signals the completion of the changes. This operation makes the (modified) copy visible to other users. Before a revision is checked in, it should satisfy some quality control criterion, such as a successful test, to make sure it is usable by other team members.

In the period between check-out and check-in, a revision may actually go through several successive edit cycles, until the change is acceptable. Whenever the editor writes out an object, a new revision is created. All of

these revisions, except for the latest one, are called *minor revisions*. Minor revisions are deleted upon check-in of the latest revision. They are needed for short-term backup purposes, in case of machine crashes or inadvertent, disastrous deletes during editing. Most programming environments limit the number of minor revisions to one or two. For instance, EMACS [36] saves one minor revision and periodically writes a checkpoint as another.

Three-way revision merging is important for combining parallel lines of development. A three way merge first identifies the commonalities among a base version and two of its parallel revisions, and then integrates the changes. The merge process also detects conflicting changes. These must be resolved manually. In practice, the merging process works well, provided changed segments are well separated from each other by unchanged ones. Examples of three-way text mergers are *diff3* and *rcsmerge* [39]. These programs are based on the algorithms that compute deltas, i.e., the differences among revisions (see Section 3.4). Recently, Reps *et al* [33] have made some progress towards improved merge conflict resolution using data flow information.

A consistent revision numbering scheme is important for version selection. Most SCM systems use a Dewey decimal notation, with revisions on the main branch numbered by a pair of the form *(release-number, level number)*. Some systems extend this notation to branches in such a way that the structure of the revision graph is reflected by the numbering. Unfortunately, this notation becomes clumsy as the number of branches increases. A better approach is to simply select a unique, symbolic identifier for each branch and to number revisions on each branch with a single number or a pair. The relation *revision-of* can be consulted to determine the lineage of a revision.

While revision numbers together with attributes such as check-in date, author, and state are sufficient for selecting revisions, additional, descriptive attributes are needed for differentiating and selecting variants. An adequate approach is to let variant attributes take on subsets of values from enumerated types. For instance, one may wish to provide an attribute that indicates the target operating systems on which a certain variant can run. This attribute would have as value a subset of an enumerated type listing all relevant operating systems. All revisions of a variant would have the same variant attributes; changing them creates a new variant. Clearly, the attributes and types for describing variants must be user-definable.

To support change tracking, every object in a source version group carries a state attribute and a log entry. The state attribute indicates the status of a revision. For example, check-out and check-in set the attribute to 'in-preparation' and 'experimental', respectively. A revision can later be promoted to a higher state, for example 'stable' or 'released'. The set of states should be extensible. To allow for effective tracing, the attribute should not just show the current value, but actually log all state changes with date and person responsible for the change.

The log entry is extremely important for change tracking. It stores a commentary requested during check-in, describing the changes completed. Browsing the log messages helps determine what happened to a software

object over time, and sometimes prevents attempting changes that had earlier been abandoned as unsuccessful. Because of the usefulness of the log entry, the Crystal SCM system [2] actually requests a log message during check-out. For recording the programmer's intentions. A check-out log helps determine what changes are in preparation. Check-in returns this message to the user, who can then edit it into the final, permanent log entry.

3.4 Implementation of Source Version Groups

Source version Groups and the objects in them must be represented as persistent objects in an object base. The object base has traditionally been implemented with hierarchical file systems, by either placing the objects and relations in separate files in a special directory, or by encoding this information in a single file. These implementations provide sufficient reliability, but recovery, consistency control, access synchronization, and authorization are realized in an *ad hoc* manner.

Building the object base on top of a full-fledged data base management system seems to be an attractive alternative, because a DBMS would provide high reliability and systematic mechanisms for handling recovery, consistency control, access synchronization, and authorization. However, commercial DBMSs are optimized towards business applications, i.e., for processing of large quantities of rather sml records. SGM presents exactly the opposite requirement, namely moderate quantities of large objects with complex internal structure. Using a business-oriented DBMS for SCM therefore results in an "impedance mismatch", characterized by awkward data modeling and poor performance [27]. Current developments in engineering databases, such as DAMOKLES [11] or object-oriented data bases, should lead to more appropriate data models and adequate efficiency.

In both file and data base implementations, accessing a particular source object usually requires a special regeneration process that reconstitutes the object from deltas. Deltas are used to conserve space (see below). First generation SCM systems such as SCCS, CMS, and RCS provided separate operations to rebuild a desired version in a temporary file, which could then be opened for reading or writing. Second generation systems such as DSEE integrate the management of source version groups into the file system. Opening a source object of version group for reading regenerates it from delta storage; opening for writing does the same but includes the semantics of the check-out operation. Besides being easier to program, integrating versioning into the file system or DBMS has the effect of better protection: Users are prevented from destroying the data structures of a version group by accidentally or intentionally tampering with them using inappropriate tools such as text editors.

Delta storage is important for conserving space. A delta is a script of edit commands that generates one object from another. The space saving achievable with delta storage for atomic objects are significant. A simple, line-based delta consumes between 9 and 16 per cent of its cleartext representation [34, 39]. Leblang reports that delta storage combined with blank compression reduces that space to 1-2 per cent of clear-text [25].[1] Clearly, delta storage makes the luxury of saving multiple versions of atomic objects af-

fordable. It could also be applied to configurations, but may not produce dramatic savings because of the small size of those objects.

There are several important design parameters affecting the speed with which an object can be regenerated from deltas, and how deltas are computed. First, deltas can be stored in forward or reverse direction. The reverse direction is preferred, since this method keeps the youngest, most recent revision on s branch in clear-text, while the others have to be regenerated since younger revisions are more likely to be needed than older ones, reverse deltas save overall regeneration time.

Second, deltas can either be interleaved or separate. In interleaved deltas, the lines of all versions are sorted into a linear data structure, such that a single pass over that data structure can collect all lines for a desired version in the correct order. This data structure ha the property that regeneration slows down as the number of versions increases. For this reason, reverse deltas are best stored separately. For a thorough discussion of delta storage techniques see Reference [39].

Finally, computing the deltas themselves is an important problem. Deltas can be generated by a special program that compares pairs of objects, or they can be produced incrementally by the editors in the programming environment. Relying exclusively on editors to produce the deltas is risky, because that decision would require that every editor in a programming environment keep track of differences. There exist only a few of those editors, and they are often functionally limited. Examples are Kruskal's P-edit [24] and Fraser's EH [15], which are both line-oriented. Another drawback of relying on delta editors is that all other programs that modify source objects, such as pretty printers, would have to record their changes, too. Updates received from the field presents another problem: The updates might not have been produced with delta editors, or, worse yet, the deltas might be relative to old or inaccessible versions. Thus, a separate program that computes deltas in batch mode is necessary. The right time for this process is during check-in, when changes are complete and the user has reached a point of closure when a short wait is tolerable.

There are two efficient algorithms for computing deltas in batch mode. One is based on isolating a longest common subsequence [18], the other one on identifying block moves [42]. A delta based on a longest common substring is not necessarily minirnal, because it cannot detect crossing block moves. Crossing block moves arise if two or more segments (e.g., procedures) appear in a different order in two revisions. An edit script derived from a longest common substring first deletes the shorter of the two segments, and then reinserts it. Tichy's block move algorithm [42] detects such permutations and is guaranteed to produce a minimal delta.

Most deltas used in practice are line-based, i.e., the unit for comparison is the line. Two lines are considered different if they differ by a single character. Clearly, a byte- or word-based delta would be smaller, but computing it would require many more comparisons and therefore much more time.

---

1. Blank compression saves space if a significant fraction of an object's size is due to indentation.

Obst [29] reports that with special heuristics, a character-based block-move algorithm runs in the same time as a line-based one, and produces deltas that are on average 30 per cent smaller. The heuristic is specifically oriented towards block moves and does not seem applicable to longest common substrings.

For objects that consist of a representation other than text, the existing delta algorithms are easily adapted by choosing an appropriate unit for comparison and converting the representation into a linear sequence. For example, the difference between two syntax trees can be computed by comparing prefix representations of the trees at the level of individual nodes.

4 Derived Versions

Handling derived versions is much simpler than handling source versions, since they are computed fully automatically and no human actions need be observed or supported. A *derived version group* is a set of derived objects that were generated from the same set of software objects by varying derivation parameters or derivers. For example, a compiler may be able to produce code for different target machines, optimized code, non-optimized code, code with runtime checks, code with debugging hooks, etc. There may also be several compiler versions available. Conditional compilation falls in this class also. The term *derived variants* is used for those objects in a derived version group that offer identical functional specifications to their client programs.

Derivers may also be able to produce information quite different from intermediate or binary code. There exist derivers to generate call graphs, pretty-printed listings, cross reference tables, or indexes. These transformations are not called variants, because they do not preserve the semantic content as compilers do. However, both these transformations and the derived variants are collected into a derived version group, as long as they were generated from the same input.

The relations *revision-of*, *variant-of* and their subtypes are defined on source objects, but extend naturally to derived objects. For example, if two source objects are revisions of each other, then so are their derived objects, provided the derived objects were produced with the same deriver and parameters. By definition, these two derived objects would be in different derived version groups. A minor difficulty here is that derived objects are often generated from several source objects. When stating that two derived objects are variants or revisions of each other, it is therefore useful to qualify this statement with respect to the source object(s) involved.

Section 6 discusses the details of how to generate and keep track of derived objects.

FIGURE 2

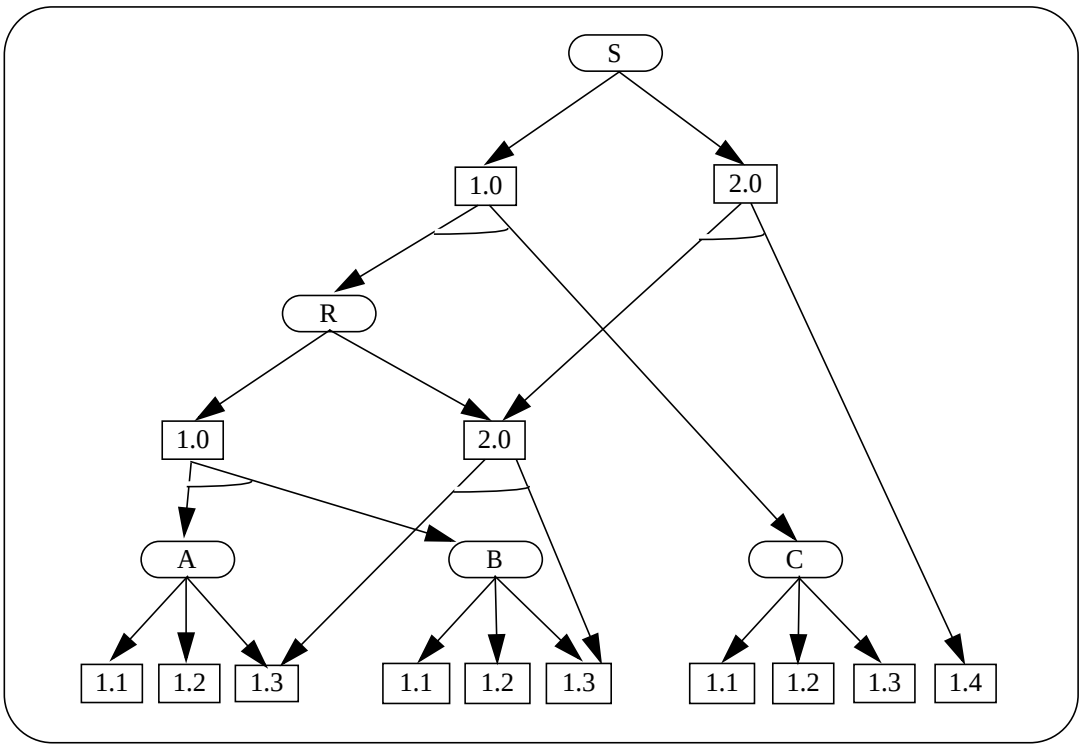

Figure 2. An AND/OR graph representing a system family

5 Version Selection and Baselining

Generic configurations may represent a large number of baselines. For example, a medium-sized software system could easily consist of 100 source version groups. Assuming each version group has merely two versions, a generic configuration containing all 100 groups represents $2^{100} \approx 10^{30}$ separate baselines — an impossibly large number. In practice, few of those baselines will actually work. The *selection problem* of software configuration management is finding viable configurations without exhaustive search.

A simple, structural model that clarifies the selection problem is the AND / OR graph model introduced in Reference [38]. This model represents atomic objects as leaf nodes, configurations as AND-nodes, and source or derived version groups as OR-nodes. The successors of an OR-node are the objects in the version group. (We ignore the relations *revision-of* and *variant-of* in case of a source version group for now.) An OA-node implies a choice among its successors, while an AND-node implies an integration. The model permits not just a tree, but a general, acyclic graph, because objects may be shared among several configurations. The selection problem in this model is formulated as searching the graph from a given start node, and making choices at each OR-node such that the nodes selected form a viable configuration.

An example of an AND / OR graph appears in FIGURE 2. Nodes *A*, *B*, and *C* are version groups of atomic objects, while *S* and *R* are version groups containing configurations. AND-nodes are depicted graphically by arcs connecting their offsprings. Labels on the out-arcs of AND-nodes distinguish composites from sequences. For example, version 1.0 of R is a composite. Note that by searching the graph starting with version 2.0 of node 5, we reach no OR-nodes. Such a start node identifies a baseline, because it unambiguously specifies a set of nodes making up a configuration. Establishing a baseline is important at release time. In a large project, where multiple changes are carried out concurrently, a baseline is an important point of reference. Updates usually are relative to a baseline. A *private baseline* is created whenever an actual system instance is generated. It may contain revisions that are not yet checked in. It is handled like a minor revision in that only a few of them are stored per user. A *public baseline* must not contain checked-out revisions, is itself checked into a version group, and should satisfy established quality control criteria. Quality control is a subject beyond the scope of this survey. For more information on baselining, see Reference [5].

An AND-node that leads to one or more OR-nodes represents a generic configuration, since some selection will be necessary when constructing an actual system instance. Generic configurations are important for compactly representing a large set of possible baselines, without having to enumerate all combinations. Without generic configurations. SCM requires the maintenance of bulky configuration tables. The problem with these tables is that they are difficult to keep up to date in a large project. For instance, the addition of an upward compatible version of a pervasively used module may cause such tables to double in size because the new version can be used wherever the old one was permitted.

Version selection is currently an active research area within SCM. The general approach is to associate constraints with generic configurations. The constraints are conditions on attributes of software objects that select appropriate variants, revisions, and derived versions. Attributes usable for revision selection are revision number and state, creation date, author, and the relation *revision-of* with its subclasses. With these attributes it is possible to express the following example constraint:

> For all version groups where the invoker has a revision checked out, select that revision; otherwise use the most recent revision that is checked in and has state 'stable'.

Constraints of this sort are called "configuration threads" in DSEE [25]. By adding a cut-off constraint for the creation date (a maximum date), a configuration can be regenerated as it would have been produced at a certain date.

Variants should be selected based upon the relation *variant-of* and user-defined variant attributes, as described in Section 3.2. For example, one may want to choose a variant on the basis of the hardware processors on which it can run. Note that a variant attribute may be single-valued or set-valued. Using the previous example, a variant may actually run on several proces-

sors. Single-valued attributes for differentiating variants were used in INTERCOL and RCS [41, 38]. The Adele and Nomade configuration managers [12, 13] use sophisticated constraints on attributes, including negation and conditionals. The latter can be used to specify preferences, that is, if a certain constraint cannot be met, then some secondary choice may be acceptable. A similar approach to preferences, based on a relational database for describing generic configurations, is due to Bernard *et al* [4]. Winkler [44] discusses set-valued attributes and introduces constraints expressed as functions over attribute values.

Additional selection criteria can be based on modification requests (see Section 7). For instance, a constraint of the following sort would configure a new release:

> Select the previous baseline. Let *O* be the set of objects in this baseline that have modification requests to be addressed in the current release, and have a corrected revision for each request. Replace the elements in *0* with the corrected revisions.

Parameters for the derivers finally select derived versions. An additional degree of freedom is available here: If a certain parameter is left unspecified, the SCM system can make its own choice. For instance, if the user does not care whether certain subconfigurations have been compiled with optimization on or off, SCM can choose whatever is available and save derivation time that way.

If constraint-based version selection is available, it is straightforward to provide an automatic function for constructing baselines. This function simply runs the selection process and records the outcome. Recording the outcome involves creating new revisions in the visited configuration groups. For example, revision 2.0 of *S* and *R* in Figure 2 could have been generated automatically. It is convenient to store the constraints used to produce a baseline along with it, in order to document the intent behind the baseline. Saving the constraints permits a similar selection to be repeated at the next release time.

Module interconnection languages (MILs) take a different approach to version selection. They concentrate on the interfaces among software modules. Type checking the interfaces assures that only type-safe configurations are constructed. DeRemer and Kron [10] originated the concept of a MIL, as a language separate from the programming language. Prieto-Diaz [32] gives an extensive survey of the MILs developed since then. Most MILs suffer from not treating interfaces as firstclass software objects. Thus, it is difficult to represent versions of interfaces. This is a serious limitation, even though versions of interfaces do not arise as frequently as versions of the implementing programs. Exceptions are the programming languages Mesa and Cedar [28, 37]. Both provide a common sub-language for describing configurations, called C-Mesa. A key aspect is the distinction between interfaces and implementations. An interface contains the types, variables, subprogram headers, etc., visible to clients of the interface, whereas an implementation of an interface provides the subprogram bodies and data structures invisible to clients. C-Mesa programs represent not only configurations, but

also record the relations *has-implementation* and *has-client*. The first relation holds between interfaces and corresponding implementations; the second between interfaces and their clients. Both can be viewed as subtypes of the general traceability relation, because the change of an interface must trigger changes in affected implementations and clients. A serious limitation of C-Mesa is that its version scheme distinguishes only two revisions, the current one and its predecessor.

Ada and Modula [19, 45] also separate interfaces and implementations and the relations *has-implementation* and *has-client* make dependencies traceable. Ada and Modula do not provide a separate configuration language. The implicit configurations and unnecessarily strict recompilation rules in both languages make treatment of versions difficult.

6 Software Manufacture

Software manufacture is the process of generating derived objects. Using the AND/OR graph, software manufacture operates on a baseline and produces a mirror image of that baseline containing only derived objects. The nodes in that minor image are connected to the corresponding nodes in the input baseline, showing the derivation history. In Figure 2, consider what must be produced by compiling and linking revision 2.0 of *S*.

To speed up the derivation process, an SGM system must manage a cache containing derived objects which are likely to be reused. Make [14] is a widely used program that uses a simple form of such a cache. It is based on a time-stamp mechanism for deciding when to update the cache: If a derived object is older than its input objects, then rederivation is necessary. Make also uses simple rules to process objects based on their types. One such rule describes how to produce machine code from C source code. Make can be combined with SCCS or RCS to provide a limited versioning capability.

Despite its popularity, Make has a number of serious drawbacks for large-scale SGM. The timestamping mechanism is inappropriate for determining whether a derived object can be reused. When there are multiple versions, a time stamp is insufficient for deciding from which versions of input objects a derived object was generated [38]. Another problem is that Make does not record the parameter settings on derivers. For example, it is impossible to decide whether a given machine code module was produced with optimization turned on or off. Make also handles derivation processes with intermediate objects inefficiently, because it always rederives a target object if its intermediate objects have been deleted, regardless of whether the target is up-to-date. Finally, Make provides derivation rules for atomic objects only; processing of configurations must be programmed explicitly.

DSEE's handling of derived objects is more reliable [25]. Each derived object carries a *history attribute* that describes precisely how the object was produced, including version identifiers and parameter setting. For high speed processing, DSEE performs parallel manufacture on idle workstations [26]. A remaining drawback is that DSEE provides no general rule for processing

configurations; the individual steps have to be programmed explicitly for every configuration.

Smile [22] and Marvel [21) provide *opportunistic manufacture.* In this processing mode, derivers start up automatically as soon as new source object versions are created. By running derivations in parallel with the programmer's activities, opportunistic manufacture attempts to have derived objects ready ahead of time. This approach reduces programmer idle time. However, a problem is limiting the combinatorial explosion of derivations caused by multiple versions. Without a specific target configuration, almost all of the derivation runs after a change could be useless.

Odin [8,9] is a flexible system for managing derived objects. Similar to Make, it uses an extensible set of rules that form a derivation graph for object types. Unlike Make, Odin's rule language covers derived configurations as well as atoms, and distinguishes sequences and composites. (Make only has sequences.) Users need only indicate the objects to be combined in configurations, and Odin determines how to process them, based on their types. For instance, it is not necessary to always redescribe how configurations are linked, or how documents consisting of several parts are processed. Furthermore, composites handle derivation processes with more than one output cleanly. Odin also provides facilities for including quality control tests, such as regression tests, as part of the derivation. In its cache of derived objects, Odin stores a full history attribute, including the parameters used during derivation. Unfortunately, support for versioning is poor.

Automatic system manufacture guarantees that the correct derived objects are produced when necessary. However, the cost of the processing involved may be too high. In large system families, changing a single line in an object with shared declarations may trigger massive recompilations. Many of these recompilations may be redundant, because the change may actually affect only a small fraction of the compilation units. *Selective recompilation* mechanisms, such as smart recompilation [40], reduce the number of redundant derivations. These mechanisms analyzes changes for their effect and prevent redundant compilations when, for example, an unused declaration is deleted, a new declaration is added, or a comment is changed.

Hood *et al* [17] generalize smart recompilation to recursive interface dependencies. Smarter recompilation [35] reduces the number of recompilations further by allowing harmless inconsistencies to remain. As an example, consider a type declaration *T* used in a set *S* of source objects. Assume we change *T* into *T'*, and update a few source objects to be compatible with *T'*. Suppose furthermore we can partition *S* into a subset *S1* in which only *T* is used, and a subset *S2* in which only *T'* is used. If there are no interactions among *S1* and *S2* that depend in any way on *T* or *T'*, then recompilation of S1 is not necessary and smarter recompilation will suppress it. More important than saving the recompilations is perhaps the fact that programmers can delay the work of making the source objects in *S1* compatible with *T'*. Thus, programmers can test their changes without having to wait for others to bring their modules up-to-date. Without a mechanism for managing inconsistencies in this manner, programmers have to resort to the unsafe

practice of subverting the type checking and manufacturing system to get their work done.

7 Modification Requests

A *modification request* (MR) is a change proposal. General configuration management is MR-driven, that is, every change is initiated by one or more MRs. Tracking of modification requests makes it possible to answer questions about past, current, and future capabilities of a system family, as well as providing important management data about project status. There is no reason why SCM should not be MR-driven as well, yet few tools for managing software modification requests exist. The author is aware of only two published tools: MRCS [23], a control system running on Unix, and Crystal [2], an SCM system that integrates version control, MR tracking, and project management.

Modification requests propose to correct errors, to modify existing system capabilities, or to extend or contract capabilities. An MR may address any set of source objects in the software lifecycle: requirements documents, design documents, interfaces, program code, test data, documentation, etc. An MR should be machine-readable and is itself a source object.[1]

MRs can be processed into derived objects, for instance into formatted objects or summary reports. Versions of MRs do not seem necessary, but each MR has an attribute that reflects its state. A useful set of states is *submitted, rejected, accepted, delayed, in progress,* and *completed*. When an MR is first entered, it has state *submitted.* A review decides whether to accept or reject the MR. A rejected MR is not discarded, but filed with a note describing the reason for rejection. A third alternative is to delay an MR, which means that it will assure the state *submitted* again at a later time for reconsideration. Once the work involved in an MR is assigned to a person, then the MR assumes state *in progress.* State *completed* indicates the modifications required by the MR have been performed and tested. To allow for effective tracing of an MR, the state attribute should not just have the current value, but should actually log all previous states, including the times when the state changes occurred. That way, it is easy to determine the history of MRs and to find MRs that have fallen behind schedule.

Usually, each programmer is responsible for a set of related MRs. This set can be represented naturally by a configuration. Configurations of MRs are often called *tasks,* and are associated with a workspace for managing temporary objects.

Additional useful data items associated with an MR are the relations *has-MR* and *has-change.* The first links an object with its MRs, the second an MR with the updates it caused. These relations support MR-based selection, as illustrated in the second query in Section 5. The submitter or reviewer of an MR establishes *has-MR,* while *has-change* is entered during check-in. To simplify entry and prevent errors, check-in should allow selection from a menu

1. *Modification requests for modification requests appear useless*.

of relevant MRs. This set can easily be derived from the MR configurations in the work space.

Crystal [2] implements the above relations. A simple experiment with bug reports on a medium-sized system showed that software engineers can attach their MRs to the affected objects in an unfamiliar system with high accuracy, provided the overall system architecture is explained with a few sentences per object. Crystal therefore presents the submitter of an MR with a sophisticated browser for locating relevant objects. This browser shows system configurations graphically and lets the user read documentation as well as the existing MRs (to avoid duplication). As a heuristic to speed up the search process, the browser even highlights "suspect" components, i.e., those that changed relative to the last baseline. Once the relation has-MR has been entered, it opens several possibilities for project management support. The history of the objects can be inspected to identify competent programmers for carrying out the changes. The history can also yield a rough estimate for the time required for the change, by averaging past periods between check-out and check-in. In Crystal, this information is used to update a PERT-chart of maintenance activities.

8 Conclusions

Software configuration management is a discipline whose goal is to control changes to large software systems. The present state of the art is that managing and tracking the update of atomic objects via the check-in / edit / check-out cycle is well understood. Reliably selecting versions and software manufacture are also well developed. The remaining paragraphs enumerate areas in need of further research.

Progress in accommodating and managing unavoidable inconsistencies in very large systems is still needed, as discussed by Schwanke and Kaiser [35]. An area presently under active investigation is version selection based on constraints and preferences. An area that is virtually untapped is representing and manipulating traceability relations, for instance the relations between specifications, their implementations, the associated documentation, and the tests. MR-driven SCM and its integration with project management is also an underdeveloped area.

With the spread of workstations, the problem of manufacturing distributed applications has gained in importance. The difficulties in these applications involve interfacing multiple programming languages and operating systems, distributing configurations over a network of (possibly heterogeneous) computers, and initializing the processes and the communication paths. Two representative approaches are Matchmaker [20] and Agora[6].

Semantic modelling of SCM with semantic networks [2, 43] or object-oriented programming languages is a topic worthy of further exploration. A semantic model of SCM would associate the various software object types with appropriate operations and organize the types into a class hierarchy with inheritance. The model would have to be extensible, for instance with new programming languages or configuration types. Another requirement

is to accommodate versions of operations, for example versions of compilers. The benefits of semantic modelling are greater conceptual clarity, direct representation of the model for machine interpretation, more sophisticated operations and queries, and simplified implementation.

Finally, an interesting topic is building a *maintainer's assistant*, *i.e.*, a program that helps with carrying out changes in complex software systems. The maintainer initiates a change, while the assistant provides decision support and takes over the task of bringing the system back into a consistent state. For example, the assistant detects all places that are affected by a given change and present them to the programmer for update. It proposes corrections and perhaps even derives corrections by observing the programmer. This approach is, of course, not limited to programs; it is just as applicable to updating specifications or other formal representations consistently. Intensive research in smart editing systems will be needed to achieve the goal of automating consistency maintenance.

9 A Glossary

AND/OR graph model: a model for describing system families with multiple versions and configurations.

Atom, atomic object: a software object whose body is not decomposable for SCM.

Baseline, baseline configuration: a baseline composite or baseline sequence.

Baseline composite : a composite whose body's field values are atomic objects or baselines.

Baseline sequence : a sequence whose body's elements are atomic objects or baselines.

Check-in: a command applied to an object reserved with check-out; it removes the reservation and makes the object publicly visible.

Check-out: a command that creates a private copy of an existing source object and reserves it for editing by the invoker.

Composite, composite object: a software object whose body consists of named fields with software object identifiers or version group identifiers as field values.

Configuration: a software object with a structured body; either a composite or a sequence. Delta: a record of the difference between two software objects, suitable for generating one from the other.

Derived object: a software object that is produced by a program from some other software object(s).

Derived variant: a derived object that has the same abstract specification as the software objects from which it was derived.

Derived version: a member of a derived version group.

Derived version group: a set of derived objects generated from the same software objects, but with different derivers or different parameter choices for the derivers.

Deriver: a program that generates derived objects.

Generic configuration: a generic composite or generic sequence.

Generic composite: a composite whose body has at least one field value that is (i) a version group or (ii) a generic configuration.

Generic sequence: a sequence whose body has at least one element that is (i) a version group or (ii) a generic configuration.

History attribute: a data structure that describes exactly how a derived object was produced.

Minor revisions: revisions of a source object created between check-out and check-in.

Modification request: a source object that proposes a change.

Opportunistic manufacture: Starting derivers as soon as a new source object is completed.

Parametric: synonym for generic (as in *parametric configuration).*

Private baseline: a temporary baseline managed as a minor revision. May contain checked-out revisions.

Public baseline: a baseline that is checked into a version group. Must not contain checked-out revisions.

Revision: a source object linked to another via the relation *revision-of.*

Revision-of: a relation linking two source objects if one was produced by changing the other.

Selective recompilation: a mechanism for saving redundant recompilations after changes.

Sequence: a software object whose body consists of a list (ordered multiset) of identifiers of software objects or version groups.

Software manufacture: the process of producing derived objects.

Software object: a machine-readable document.

Source object: a software object entered manually.

Source version: a member of a source version group.

Source version group: a set of source objects related via *revision-of* and *variant-of*

System family: a set of related systems, sharing common objects.

System model: see generic configuration.

Traceability link: a relation connecting a source object to those (source or derived) objects, whose contents was used when the source object was composed.

Variant: a source object linked to another one via the relation *variant-of*.

Variant-of: a relation linking two source objects that are indistinguishable under a given abstraction.

Version: a member of a version group.

Version group: a source or derived version group.

10 References

10.1 Main reference

Walter F. Tichy: "Tools for Software Configuration Management", In *Proceedings of the International Workshop on Software Version and Configuration Control,* Grassau, Germany Jan 27-29, 1988. Also In Ed. Jürgen Winkler: *Software Version and Configuration Control,* Band 30, G.G: Teubner, Stuttgart, 1988.

10.2 Other references

[1] *Code Management System*. Digital Equipment Corporation, 1982. Document No. EA-23134-82.

[2] Lori B. Alperin and Beverly I. Kedzierski. AI-based software maintenance. *In IEEE AI Applications Conference, February 1987.*

[3] Wayne A. Babich. *Software Configuration Management*. Addison-Wesley, 1986.

[4] Y. Bernard, M. Lacroix, P. Lavency, and M. Vanhoedenaghe. Configuration management in an open environment. In *Proceedings of the First European Software Engineering Conference,* pages 37-45, AFCET, Springer Verlag, September 1987.

[5] Edward H. Bersoff, Vilas D. Henderson, and Stan G. Siegel. Software configuration management: a tutorial. *IEEE Computer,* 12(1):6-14, January 1979.

[6] Roberto Bisiani, F. Alleva, F. Correrini, A. Florin, F. Lecouat, and R. Lerner. *The Agora Programming Environment.* Technical Report CMU-CS-87-113, Carnegie-Mellon University, Department of Computer Science, March 1987.

[7] William Bryan, Christopher Chadbourne, and Stan Siegel. *Tutorial: Software Configuration Management*. IEEE Computer Society Press, 1980.

[8] Geoffrey M. Clemm. The Odin specification language. In *Proceedings of the International Workshop on Software Version and Configuration Control,* Teubner Verlag, Stuttgart, FRG, January 1988.

[9] Geoffrey M. Clemm. *The Odin System: An Object Manager for Extensible Software Environments.* PhD thesis, University of Colorado-Boulder, Department of Computer Science, 1986.

[10] Frank DeRemer and Hans H. Kron. Programming-in-the-large vs. programming-in-the-small. *IEEE Transactions on Software Engineering*, SE-2(2):80-86, June 1976.

[11] Klaus R. Dittrich, Willi Gotthard, and Peter C. Loekemann. Damokles - a database system for software engineering environments. In *Proceedings of the Workshop on Advanced Programming Environments, IFIP,* Springer Verlag, LNCS Vol. 244, June 1986.

[12] Jacky Estublier. A configuration manager: the adele data base of programs. In *Proceedings of the Workshop on Software Engineering Environments for Programming-in-the-Large,* pages 14014T, Harwichport, MA, June 1985.

[13] Jacky Estublier. Configuration management: the notation and the tools. In *Proceedings of the International Workshop on Software Version and Configuration Control,* Teubner Verlag, Stuttgart, FRG, January 1988.

[14] Stuart I. Feldman. Make - a program for maintaining computer programs. *Software, Practice and Experience*, 9(3):255-265, March 1979.

[15] Christopher W. Fraser and Eugene W. Meyers. An editor for revision control. *ACM Transactions on Programming Languages and Systems,* 9(2):277-29S, April 1987.

[16] A. Nico Habermann and David Notkin. Gandalf: software development environments. *IEEE Transactions on Software Engineering*, SE-12(12):1117-1127, December 1986.

[17] Robert Hood, Ken Kennedy, and Hausi A. Mueller. Efficient recompilation of module interfaces in a software development environment. *ACM SIGPLAN Notices*, 22(1):180-189, January 1987. Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments.

[18] James W. Hunt and Thomas G. Szymanski. A fast algorithm for computing longest common subsequences. *Communications of the ACM*, 20(5):350-353, May 1977.

[19] Jean D. Ichbiah. *Ada Programming Language,* Military Standard. United States Department of Defense, January 1983.

[20] Michael B. Jones, Richard F. Rashid, and Mary R. Thompson. Matchmaker: an interface specification language for distributed processing. In *12th Annual Symposium on the Principles of Programming Languages,* pages 225-235, ACM, January 1985.

[21] Gail E. Kaiser and Peter H. Feiler. An architecture for intelligent assistance in software development. In *Proceedings of the Ninth International Conference on Software Engineering,* pages 180-188, IEEE, March 1987.

[22] Gail E. Kaiser and Peter H. Feiler. Intelligent assistance without artificial intelligence. In *Proceedings of the Thirty-Second IEEE Computer Society International Conference (COM PCON),* pages 236-241, IEEE, February 1987.

[23] D. B. Knudsen, A. Barofsky, and L. R. Satz. A modification request control system. In *Proceedings of the 2nd International Conference on Software Engineering,* pages 187--192, IEEE and ACM, 1977.

[24] Vincent Kruskal. Managing multi-version programs with an editor. *IBM Journal of Research and Development*, 28(1):74-81, January 1984.

[25] David B. Leblang Jr. and Robert P. Chase. Computer-aided software engineering in a distributed workstation environment. *ACM SIGPLAN Notices,*19(5):104-112, May 1984. *Proceedings of the ACM SIGSOFT/SIGPLAN* Software Engineering Symposium on Practical Software Development Environments.

[26] David B. Leblang Jr. and Robert P. Chase. Parallel software configuration management in a network environment. *IEEE Software*, 4(6):28-35, November 1987.

[27] Mark A. Linton. Implementing relational views of programs. *ACM SIGPLAN Notices*, 19(5):14-21, May 1984.

[28] James G. Mitchell, William Maybury, and Richard Sweet. *Mesa Language Manual*. Technical Report, Technical Report, Xerox Palo Alto Research Center, Feb. 1978.

[29] Wolfgang Obst. Delta technique and string-to-string correction. In *Proceedings of the First European Software Engineering Conference,* pages 69-73, AFCET, Springer Verlag, September 198T.

[30] David L. Parnas. Designing software for ease of extension and contraction. *IEEE Transactions on Software Engineering,* SE-5(2):128-138, March 1979.

[31] David L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM,* 15(2):lOS3-1058, December 1972.

[32] Ruben Prieto-Diaz and James M. Neighbors. Module interconnection languages. *The Journal of Systems and Software*, 6:307-334, 1986.

[33] Thomas Reps, Susan Horwitz, and Jan Prins. Support for integrating program variants in an environment for programming in the large. In *Proceedings of the International Workshop on Software Version and Configuration Control,* Teubner Verlag, Stuttgart, FRG, January 1988.

[34] Marc J. Roehkind. The source code control system. IEEE Transactions *on Software Engineering,* SE-1(4):364-370, December 1975.

[35] Robert W. Schwanke and Gail E. Kaiser. Living with inconsistency in large systems. In *Proceedings of the International Workshop on Software Version and Configuration Control,* Teubner Verlag, Stuttgart, FRG, January 1988.

[36] Richard M. Stallman. EMACS: the extensible, customizable, self-documenting display editor. In *Interactive Programming Environments,* pages 300-325, MacGraw-Hill, 1986.

[37] Daniel C. Swinehart, Polle T. Zellweger, Richard J. Beach, and Robert B. Hagman. A structural view of the Cedar programming environment. *ACM Transactions on Programming Languages and Systems,*8(4):419-490, October 1986.

[38] Walter F. Tichy. A data model for programming support environments and its application. In Hans-Jochen Schneider and Anthony I. Wasserman, editors, *Automated Tools for Information System Design and Development,* pages 31-48, North-Holland Publishing Co, Amsterdam, 1982. Reprinted in *Trends in Information Systems,* B. Langefors et al, editors, North-Holland Publishing Co, 1986.

[39] Walter F. Tichy. RCS — a system for version control. *Software-Practice and Experience*, 15(7):637-654, July 1985.

[40] Walter F. Tichy. Smart recompilation. *ACM Transactions on Programming Languages and Systems*, 8(3):273-291, July 1986.

[41] Walter F. Tichy. Software development control based on module interconnection. In Anthony I. Wasserman, editor, *Software Development Environments,* pages 272-284, IEEE Computer Society Press, 1981. Originally published in Proceedings of the Fourth International Conference on Software Engineering, September 1979, IEEE.

[42] Walter F. Tichy. The string-to-string correction problem with block moves. *ACM Transactions on Computer Systems,* 2(4):309-321, November 1984.

[43] Walter F. Tichy. What can software engineers learn from AI? *IEEE Computer,* 20(11), November 1987.

[44] Jürgen F.H. Winkler. Version control in families of large programs. In *Proceedings of the Ninth International Conference on Software Engineering,* pages 150-161, IEEE, March 1987.

[45] Niklaus Wirth. *Programming in Modula-2.* Springer Verlag, 1985.