The State-of-the-Art in Concurrent, Distributed Configuration Management

MacKay, Stephen

Publisher's version / Version de l'éditeur:

Proceedings of 5th International Workshop on Software Configuration Management (SCM5), 1995

National Research Council Canada
Conseil national de recherches Canada

Institute for Information Technology
Institut de technologie de l'information

NRC · CNRC

The State of the Art in Concurrent, Distributed Configuration Management

Stephen A. MacKay

Software Engineering

April 1995

Canada

NRC No. 38359

This report also appears in the “Proceedings of the 5th International Workshop on Software Configuration Management (SCM5), Seattle, WA, April 24–25 1995. To be published in the Springer-Verlag Lecture Notes in Computer Science (LNCS) series, 1995.

Additional copies are available free of charge from:

Publication Office
Institute for Information Technology
National Research Council of Canada
Ottawa, Ontario, Canada
K1A 0R6

Des exemplaires supplémentaires peuvent être obtenus gratuitement à l’addresse suivante:

Bureau des publications
Institut de technologie de l’information
Conseil national de recherches du Canada
Ottawa (Ontario) Canada
K1A 0R6

The State of the Art in Concurrent, Distributed Configuration Management

Stephen A. MacKay

Institute for Information Technology—Software Engineering
National Research Council of Canada
Ottawa, Ontario K1A 0R6 Canada

Phone: 613-993-6553, Fax: 613-952-7151
E-Mail: MacKay@iit.nrc.ca
WWWeb: http://wwwsel.iit.nrc.ca/

Introduction

The most widely used definition of software configuration management (SCM) comes from the standards community [IEEE87, IEEE90a, IEEE90b, Buck93]. *Configuration management* (CM) is a discipline that oversees the entire life-cycle of a software product (or family of related products). Specifically, CM requires *identification* of the components to be controlled (configuration items) and the structure of the product, *control over changes* to the items (including documentation), accurate and complete *record keeping*, and a mechanism to *audit* or verify any actions. This definition is not complete. Dart [Dart92] suggests that the definition should be broadened to include *manufacturing issues* (optimally managing the construction of the product), *process management* (ensuring adherence to the defined processes) and *team work* (supporting and controlling the efforts of multiple developers). Tichy [Tich88] provides a definition that is popular in the academic and research communities: software configuration management is a discipline whose goal is to control changes to large software system families, through the functions of: *component identification*, *change tracking*, *version selection and baselining*, *software manufacture*, and *managing simultaneous updates* (team work).

We prefer these definitions because the emphasis is on evolution of and access to software components by teams of developers, rather than control or prevention of access in the standards definition. *Concurrent (or parallel), distributed configuration management* is simply a recognition of the true state of software development in the 1990's—managing the evolution of software produced by geographically distributed teams, working semi-autonomously, but sharing a common software base.

We recently conducted a survey of both commercial and freely available SCM systems for an industrial collaborator and were struck by the lack of support, in most systems, for true concurrent, distributed software development. The purpose of this paper is to examine why distributed and concurrent activities in software development are important, to describe some of the currently popular mechanisms for handling concurrency, and to describe how we handle distributed, concurrent software development in our own research vehicle, the Database and Selectors Cel approach to configuration management (DaSC). This paper ties some of the survey results together with thoughts based on over ten years of research into software configuration management. The survey examined the features and limitations of 33 commercial and free CM systems [MacK94] through data provided by the suppliers of each CM system

NRC Number 38359

(marketing literature, manuals, and conversations with staff). The following categories of data were collected:

- name,
- supplier,
- CM model,
- repository abstraction,
- repository mechanism,
- system/subsystem modelling ability,
- distributed development support,
- concurrent modification support,
- branching support,
- merge support,
- user interface,
- revision numbering,
- handling of non-ASCII files,
- multiple development architectures support,
- handling of historical releases,
- logging facilities,
- directory tracking,
- ability to generate deltas,
- price, and
- other comments.

In a broad survey such as this, there is a tremendous amount of overloading of terminology. In this paper we will use terms that we have been using throughout our research project, but to ensure there is no confusion, we provide the following definitions:

Variants of configuration items are different implementations that remain valid at a given instant in time, created to handle environmental differences (for example, different execution platforms). *Revisions* are the steps a configuration item goes through over time, whether to handle new features, fix bugs or to support permanent changes to the environment (e.g., operating systems upgrades, if the old one is no longer supported). Variants and revisions provide a two-dimensional view into the repository, with variants incrementing along one axis as required and revisions incrementing through time on the other.

Versions of configuration items are understood by the SCM community to be synonymous with either revisions or variants [Tich88]. Therefore a version of a single configuration item denotes an entry in the two-dimensional view of the repository reached from an origin through some path of revisions and variants. In this paper we will use the term, *version-set*[†], to denote a collection (module, product, package, program, system) of configuration item variants taken from specified revision levels.

A *release* is a version-set that has passed some defined quality assurance measures (which, in some cases, are regrettably defined as "none") and is ready for a "customer" (which may be another group within the development organization).

[†] In the DaSC project, we have been using the tighter *version-set* definition of the term *version* since about 1986. We do not refer to versions of individual configuration items because it tends to diminish the concept of variants, which were so long ignored [Mahl94]. To reduce confusion for the reader, we will refer to DaSC versions as version-sets.

Distributed Development

There was a time, not long ago, when distributed development in a software project meant that some programmers could connect their VT100 terminals to the central mainframe through telephone or other slow communication lines. While the computer viewed them as equal participants in the project, using the same tool set as the other developers, the remote developers themselves often saw it differently. Low bandwidth or expensive communication lines (sometimes both), coupled with revision control tools that simply locked them out of particular pieces of source without explanation or indication of when it would be available, made this type of environment virtually unworkable. An alternate strategy was to break the project into completely self-contained pieces that could be worked on in isolation without sharing or on-going communication among the teams. The problem was that development proceeded in isolation without sharing or on-going communication among the teams.

Today, the bulk of software development has moved to the desktop. Workstations and personal computers dominate the workplace and networking is a necessity. Software development has changed greatly, but some things remain the same. The software repository generally resides on one node (or is split across a small number of nodes), so from the point of view of the computers, everyone is still equal, accessing the repository using the same protocols through a standard shared file system (like NFS). Bandwidth is higher and costs have decreased so that even developers continents away have almost immediate access to the source. Most of the current commercial configuration management tools are designed to work in this type of environment.

Intermittently Connected Developer

While today's modern high-speed networks allow remote developers to share code at speeds almost as high as on the local networks, temporary access by the *intermittently connected developer*—someone who, for a variety of reasons, may unpredictably come and go from the network—presents a problem. For example, a software field installer with a notebook computer may need to make minor field customizations or bug fixes with customer 'A' this week, 'B' next week, etc. From the point of view of the field developer, assuming that he is a full-fledged member of the team, it is the software repository that is intermittently connected.

Additionally, large multi-company software projects, typified by military contracts or realtime, embedded system projects, cannot make use of shared software repositories. Joint projects come and go and today's partner may be tomorrow's competitor. Prime contractors depend on subcontractors (likely physically remote) to deliver specific components of the final systems. For reasons of security or physical incompatibility, companies cannot grant each other full network access, leaving the software development effort stuck in the isolation model of the past. This is a recognized area for further SCM research [vand95].

Software Customer

Distributed configuration management not only affects the development organization—the needs of the customers must be considered as well. It is not infrequent that, over time, a customer will obtain a sequence of releases of a piece of software from a developer. Upon receiving each release the customer will need to apply his own changes. It is rare however, for the customer and developer to have the software under common configuration management (particularly when the customer is only one of many for some commercial software product). Another view of the same problem is when a subcontractor delivers a product, which the customer incorporates and changes,

and then the customer goes back to the subcontractor for revision and further development. Questions arise such as, "What should the prime contractor provide to the subcontractor as a base for CM on the revisions?", "What kind of CM hooks should developers supply for the customers?", or "Given that the developer ships with no CM support for the customer, how can the customers do an appropriate merge?"

Commercial Support

Virtually all of SCM systems available today support some amount of distributed development. At the very least, all systems support a single repository available on some kind of network (usually a LAN) that is accessible to a distributed group of users. Remote CVS extends the LAN to a WAN to allow worldwide connectivity. Microsoft Delta, and SPARCworks/TeamWare from SunPro, like our DaSC methodology, allow users to take a copy of all or part of the repository and synchronize any modifications at convenient times so that intermittently connected developers may benefit from full configuration management. Only the CCC family of products from Softool advertise an ability to include vendors' products under configuration control, which is surprising, given the number of organizations that experience the upgrade problem.

Concurrent Modifications

To fully support environments composed of intermittently connected developers or cooperating independent contractors, complete concurrent access to individual software configuration items (read *and* write) is a requirement. However, any notion of developers making concurrent modifications to the same configuration item has traditionally been seen as contrary to the concept of configuration management—if an item could be modified in parallel, then it could not be controlled! Fortunately, concurrent development is increasingly being recognized as an important tool in good software engineering. If developers are allowed to take their time, develop changes carefully, and test extensively, before returning code to the repository (with a high degree of confidence of correctness), then it is inevitable that more than one developer is going to touch the same configuration item. In 1993, support for true concurrent development was found in few commercial CM systems, mainly in very high-end tools, such as CaseWare/CM from CaseWare (now Continuus/CM from Continuus Software), and Expertware's CMVision/CMFacility and in Aide-de-Camp from Software Maintenance and Development Systems. In 1994 various forms of distributed concurrency are appearing in product upgrades or in new, more comprehensive products (not all of which are available for review at the time of this writing)—SPARCworks/TeamWare has CodeManager to coordinate simultaneous development across multiple sites on multiple development platforms, Atria has released ClearCase MultiSite to support parallel development across geographically distributed teams and Adele, from Verilog, has recently added rich workspace support to its branch and merge features [Estu94, Estu95]. New offerings or upgrades are also available from Softool (CCC/Harvest), Continuus (Continuus/CM), IBM (CMVC) and Legent (Endevor/WSX, formerly TeamTools from TeamOne Systems).

The two most common mechanisms for handling concurrent modification of software configuration items are branching (usually found in tools implementing the check-out/check-in model of development) and optimistic methods such as copy-modify-merge, workspaces and transactions. We will examine these techniques in more detail as well as looking at how concurrency is handled with tools implementing change sets.

Branching

Branching is a low-level revision control technique, usually found in tools supporting the check-out/check-in model of development. Branching allows a configuration item to follow simultaneously several paths of development and may be employed to accomplish a number of goals. Branching is generally represented as directed acyclic graph, often called a version tree or version graph.

Branches may be created when a variant of a configuration item is needed for a new version of the product (e.g., new software development platform). Such branches are likely to be long lived and merging will rarely take place (Figure 1). A similar situation exists when bugs need to be fixed in released software, perhaps several generations old, usually in large legacy systems (e.g., telephone switches). Merging rarely takes place because the baselines will have changed substantially or the type change required may be quite different, if it is needed at all. These branches usually do not survive as long as in the variant example (Figure 2). When a configuration item is required to fix a bug in the midst of new development, branches usually exist only while the configuration item is locked on the main development path for new enhancements or until a release point when many bug fixes may be incorporated into the main branch for testing. A structure where branching is used solely for bug fixes on the current development path is shown in Figure 3.

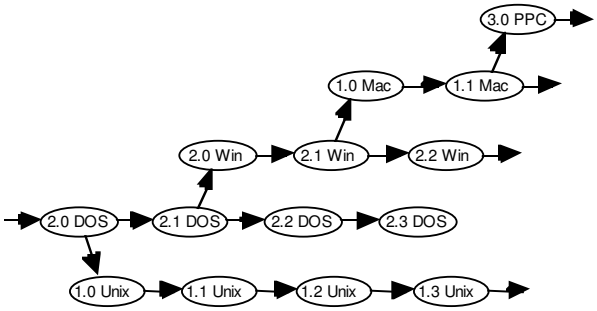

Figure 1. Branching to support new variants. Ovals represent the changing configuration item—horizontal arrows represent revisions, diagonal arrows, variant creation. Terminal ovals indicate that a particular variant is no longer maintained (e.g., hardware no longer available or market sector no longer considered worthy of support).

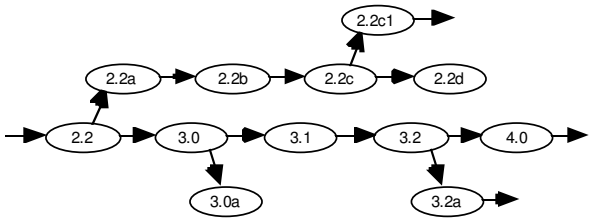

Figure 2. *Branching to support historical revisions.* Ovals represent the changing configuration item—horizontal arrows represent primary revisions, diagonal arrows, creation of another revision that is part of an earlier release. Terminal ovals indicate that a particular version is no longer maintained (e.g., all customers have finally upgraded to a newer release).

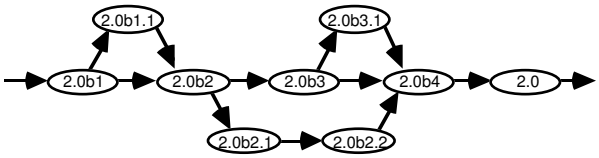

Figure 3. *Branching to support simultaneous bug and feature revisions*. Ovals represent the changing configuration item. A horizontal arrow represents a main development revision and a diagonal arrow leaving an oval represents a bug fix revision. Multiple arrows entering an oval represent a merge.

More commonly, branching is used in all three forms simultaneously, complicating the situation significantly, such that drawing a complete picture, even for an individual configuration item, is difficult to attempt. Branching for full concurrent development is not an effective tool—the model becomes difficult to understand and maintain. Commercial CM systems that support branching, usually discourage its use for full concurrency, even though it is the only mechanism they provide for development to proceed simultaneously on a single configuration item. Some products, including many of those derived from RCS, provide emergency commands to override check-out locks or permit the locks to be turned off, but these features are treated as back-doors and are not recommended by the respective suppliers. Others, such as CMVision/CMFacility, hide RCS or SCCS style branching under a richer interface. With CMVision/CMFacility, virtual views (links to files so that they only appear once in the repository) allow optional concurrent modifications to take place. The line between these systems, and those based on workspaces, often gets a little fuzzy.

Change Sets

The change set method of configuration management focuses on logical changes to the product, not on revisions of the individual configuration items. A bottom-up (sometimes called *Chinese menu*) view of the change set method tracks individual revisions, but collects them into logical groups, called change sets. New versions of the product are then created by applying relevant change sets to a previously baselined version. Different change sets define alternate versions. The change set model would appear to be a natural way of dealing with parallel modifications, however concurrency control is outside this view of the change set abstraction and is handled separately. For example, Peter Miller's aegis, a free program that implements the change set model, maintains changes at the file level, but sits on top some other revision controls system (another example of branching hidden below the user level). Aide-de-Camp, the only commercial example of the change set model, maintains changes in a database, at the line-of-code level, with an optimistic copy-modify-merge scheme to handle concurrent modifications. A top-down view of the change set method, where the change sets themselves are treated as first-class objects, is analogous to the workspace model of SCM, discussed below.

Optimistic Concurrency

Optimistic concurrency is not a single method but a different way of thinking about doing software development. It is a recognition that the software development organization encourages multiple development threads to happen simultaneously, widely distributed or at a single site, but that at some well-defined points in the development cycle some coordination and synchronization of the different streams will have to hap-

pen. For optimistic methods to be effective, there is an underlying assumption that management practices will insure that having multiple developers working on a single configuration item is not the norm, but when it is required, there will be no penalty.

One of the most effective ways of managing concurrency and minimizing overlap is to ensure that the granularity of the configuration items is as small as appropriate. This is not to say that developers must continually work with fine-grained fragments that do not present enough context, but the CM tools must take care of the mapping between what the developer needs to see and what is stored in the repository.

The most commonly implemented mechanism for optimistic concurrency among the current generation of CM tools can be categorized as copy-modify-merge. Upon request, a developer is provided with a copy of the requested software configuration item—a direct copy from a file-based repository, or generated from delta-based repository or from a database repository. The developer then has an unlocked copy of the item and the time to modify it properly, but no one else is blocked from working with the same item. When the modification is complete and fully tested, the item is returned to the repository. Because other modifications may have been made while the local copy existed, the procedure for returning the configuration item must check for potential clashes. Depending on the implementation, a merge may be done on each write to the repository, or more intelligently, the parallel changes are managed separately until a human authorizes and supervises the merge.

Workspaces and Transactions

Dart presented a set of 14 categories that describe the functionality of CM systems [Dart90, Dart92]. Of these, three are particularly suited to model concurrent software development: workspaces, transparent views and transactions. The workspace model is a more general form of copy-modify-merge. A *workspace* is an area where a user can take configuration items from the repository and modify them independently, without disturbing or being disturbed by other developers. Any changes committed to the repository by other users after the files are placed in the workspace are not automatically visible, however developers can choose to see what has changed in the repository at convenient milestones. Dart refers to this as insulated, not isolated [Dart94]. When a certain level of satisfaction is reached (such as completion of a major feature or approval by the CM administrator), the items in the workspace are returned to the central repository. There is an implication that the workspace itself is under revision control (possibly private), rather than simply a working directory (i.e., a local history is maintained). A workspace can be used to implement a top-down view of a first-class change set. The *transparent view* extends the workspace model by providing a view into the central repository, but only the variants and revisions of interest are visible. Configuration items may exist in either the workspace or the repository. All those requiring a particular view may share the workspace. Continuus/CM, CMVision/CMFacility, and DRTS from ILSI implement various flavours of workspaces. *Transactions* further enhance the workspace and transparency categories with a set of commands or protocols that coordinate and synchronize the workspaces with each other and with the repository. Feiler, in his attempt to classify CM systems [Feil91], provides more detail on transactions. Often referred to as “long transactions” to differentiate them from simple database transactions, the configuration items can remain out of the repository for weeks or even months. In the extreme case, the workspaces and transactions last indefinitely and become the repository. The focus is on revisions of the configuration and concurrency control. Endevor/WSX from

Legent (formerly TeamTools from TeamOne Systems) is a good example of transaction-based CM.

DaSC Approach

The Database and Selectors Cel (DaSC) approach to software configuration management, developed in our laboratories at the National Research Council of Canada, includes top-down change set, workspace, transparency and transaction features. The concepts of DaSC have evolved over 10 years of practical software development experience, but from the beginning support for concurrent, distributed development was fundamental. The first published description of DaSC and early results from our research into software configuration management appear in [Gent89], a discussion of DaSC and the importance of a good visual metaphor for configuration management can be found in [Wein92], and the evolution of our DaSC model and of the supporting tools will be discussed in a forthcoming paper [Wein95].

Assumptions

The initial target for DaSC was the realtime and embedded systems community, however we have found it applicable to a wide range of applications. As yet, the only environments where DaSC gave us little advantage were those where the configuration items were extremely large-grained and, due to project requirements, we were not able to sub-divide them into smaller entities. The chief assumptions behind DaSC are:

- *Small companies or small teams*—groups of cooperating individuals, numbering in the tens.
- *Distributed development* (within *and* across teams *and* companies)—the intermittently connected developer model is assumed.
- *Software components*—the basic building blocks are fined-grained entities.
- *Cost-sensitive*—no expensive, highly specialized development tools or environments are assumed.
- *Not device independent*—the software structure must be such that alternate implementations are easily integrated.
- *Not just a temporal evolution*—a family of programs sharing common components and evolving together (not necessarily in lock-step) over a long period of time (typically decades).
- *Two-box world*—separate host development environment and specialized target hardware is the norm (also assumed that development may occur simultaneously on a variety of hosts, using a variety of file systems and cross-compilation tools).

Source Code Database

DaSC is based on the principles of managing multi-version software through a software database, and handling concurrent, distributed evolution through a multi-layered approach (analogous to the cel used by film animators). DaSC represents a methodology for configuration management and may be implemented in a number of valid ways. Our examples frequently will refer to source code, but the DaSC model can be applied to any other configuration items, such as documentation and other binary files, as long as there is some form of fine-grained database representation and an appropriate method of selection.

Figure 4 shows a simple example of a DaSC database. Five configuration items are represented by rectangles and two sets of database selectors are represented by ovals.

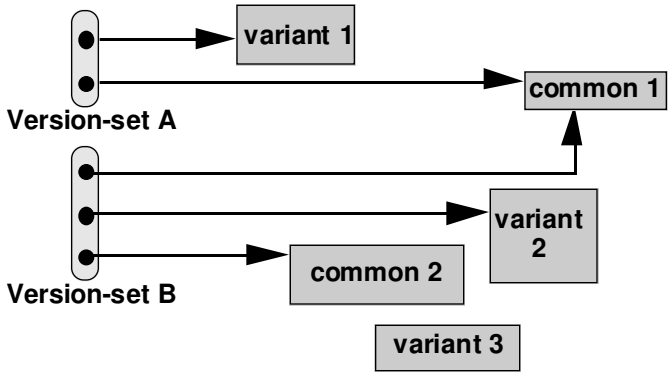

Figure 4. *Database and selectors.*

Individual configuration items may be characterized as *common* or *variant*. Common items, common 1 and common 2 in Figure 4, are those that are (or are expected to be) referenced by a number of selector sets. Variants variant 1, variant 2, and variant 3 provide the same functionality and are selected based on the context. For example, they may represent code appropriate for Unix, Macintosh, or DOS file systems, or for three different compilers, GNU, Borland, or MPW. A DaSC version-set (or selector set) is simply a collection of selectors into the database of common and variant configuration items. In Figure 4, Version-set A selects common 1 and variant 1, and Version-set B selects common 1, common 2, and variant 2.

In order to achieve our goals of portability of host environments and to provide small companies with a low cost entry point, our initial implementation of the DaSC software repository used the file system tree as the database and an inclusion file technique (e.g., files that consist solely of a number of C language #include statements) for database selectors. In addition to providing an effective selection tool, inclusion files also isolate all local file system dependent names in a single substructure.

In DaSC, revisions are represented by new cels or layers. Figure 5 shows an example of a revision layer that might be placed on top of the baseline in Figure 4. Additions are simply shown by new rectangles or ovals, depending on whether a version-set or a database item is being added (e.g., Version-set C, common 3, and variant 4). Changes to an item are represented by a newer copy appearing at the same location on the new layer (e.g., Version-set A and variant 2). A special marker that "paints over" the original item below it is used to denote deletions—the item is not physically removed (e.g., the cross-hatches over the position of variant 1).

Revisions therefore extend the concept of version-sets. A version-set can select common code or a variant from any valid revision layer. In Figure 5, note that the selectors on the revision layer always point to the appropriate spot on the baseline. We provide a tool in DaSC, called derive, which given a version-set and a list of revision layers to be considered, will generate a version-set description that points to the correct configuration items. Figure 6 shows what the database would look like if we looked through the revision layer of Figure 5 onto the baseline of Figure 4.

In the initial implementation of DaSC, cels were realized as parallel file system trees. A set of scripts and tools free the user from concerns about the mapping between the file system and the layered model.

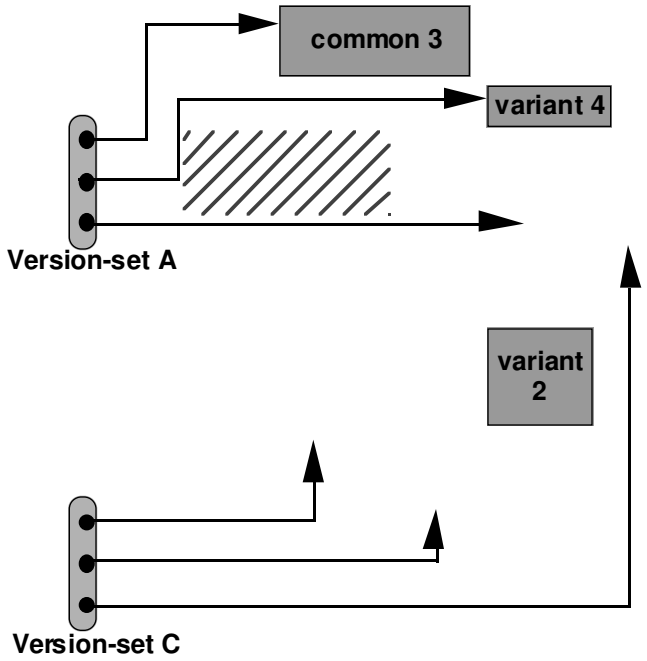

Figure 5. *A revision layer.*

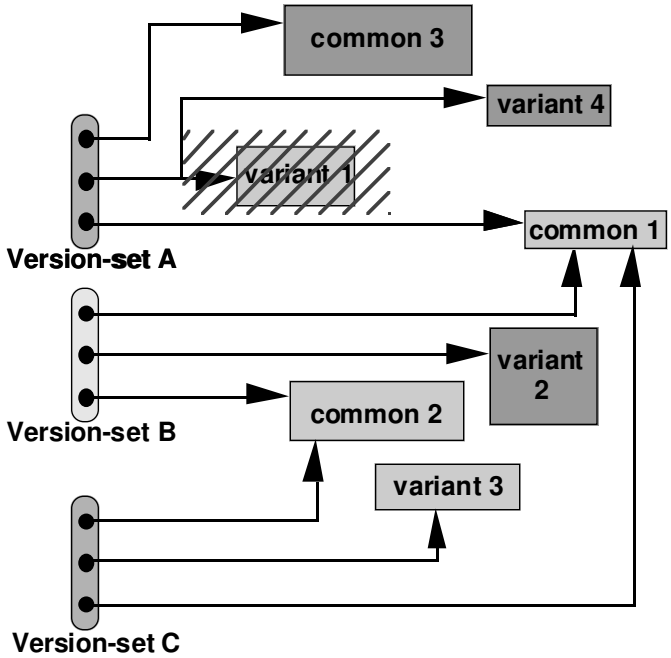

Figure 6. *Figure 5 overlaid on Figure 4.*

DaSC for Concurrent, Distributed Development

The layered approach to software evolution is ideally suited to software development by a group of developers working in parallel in a distributed environment. Cels provide a representation of change without violating the integrity of the original source—first-class change sets. Evolution in time (revision) is supported by adding subsequent layers to a master repository. Layers may be exchanged among developers to synchronize their activities.

Layers exist for as long as is necessary. Layers can exist for short periods of time, to allow a developer to experiment temporarily with a creative idea (later expanded to a full feature or thrown away), for quick bug fixes (later combined with other layers), or even as a sketch for a feature that may eventually be implemented, but is not part of the current product. As in the transaction model of configuration management, individual layers can last "forever" and become the repository of software history. Usually a new layer is created to facilitate a single logical change to the software. It is a logically complete entity—applying the new layer to the layer (or layers) below it will result in stable and complete rendition of the software with the change applied to it. Thus long developer assignments are encouraged. Unlike the check-out/check-in model of development where individuals are urged to return the configuration item to the repository as soon as possible so that others can have access to it, DaSC allows for fully distributed development in the workspace model. The developer is given the time necessary to implement and test changes because the layer is invisible to other developers. They continue to access and modify items in their own layers, taken from the same baselined, master repository. Note that because there are never any check-out locks, intermittently connected developers can take a local copy of all or part of the repository as required.

As development proceeds, many layers are created and they are either considered stacked or adjacent, depending on their relationship with the other layers. Conceptually, a new layer may be *stacked* on top of previous layers if it has been developed with prior knowledge of the layers below it (e.g., an individual developer making a series of modifications). Any configuration item appearing on an upper layer will always appear to overwrite the identically specified item on any lower layer (as shown in Figure 6). Layers are *adjacent* to each other if each layer was developed independently of the other—most commonly by separate developers working in parallel. At convenient times, the layers from individual developers can be combined for integration testing and release, in a process we call *consolidation*.

We have built a tool, consolidate, that manages the process. Figure 7 shows a typical layer diagram and the steps toward consolidation. In step (a), the consolidate tool collapses stacked layers B, C, and D downwards to create a new temporary layer, T^1. Likewise, layers E and F are simultaneously collapsed onto T^2, layers H, I, J, and K onto T^3, and M, N, and O onto T^4. In step (b), adjacent layers T^3, L, and T^4 are reconciled for clashes by comparing each layer with the others to find identically specified configuration items. If any clashes are detected among the three layers, some minor manual intervention will be needed before consolidating sideways to create layer T^5 (if no clashes are found the layers are simply merged to create T^5). In step (c), T^5 is collapsed with G to create T^6. Layers T^1, T^2, and T^6 are then reconciled in step (d) and the resulting temporary layer, T^7, is collapsed, in step (e), with A to create layer Z, the result of the consolidation. While any consolidation of this size, results in some clashes being detected, it is extremely rare that the same line of code was touched, so correction is

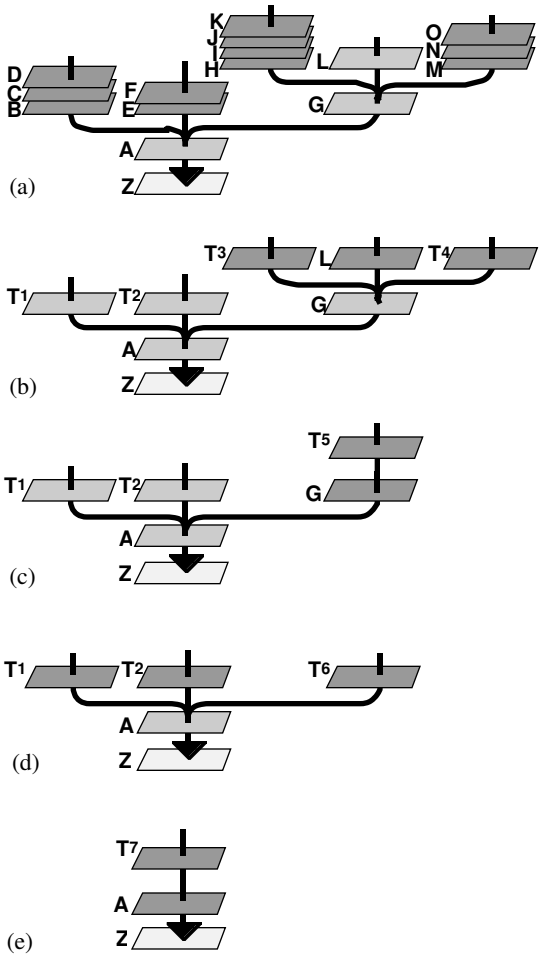

Figure 7. *Typical consolidation in 5 steps.*

usually trivial. Planned improvements to the consolidate tool will allow it process situations, like the one shown in Figure 7, with even less manual intervention [Nodd94].

The layer resulting from a consolidation can then be re-tested and considered part of the master repository. The new layer can be sent to customers as an upgrade or it can be distributed to other developers who may not have been involved in the consolidation (for example, if they are working on more long term development paths). They can then consolidate their code with the new layer at a time that is convenient, such as when they reach a milestone—they are not forced to co-exist with possibly incompatible code until they reach a point when it is reasonable to consider all the ramifications of the new layer and make the appropriate changes, if any are required, to their own code.

Conclusions and Further Research

Modern software development is a team effort, and in today's global marketplace, we cannot assume that the teams are in the same geographic location. We also cannot assume that it is possible to provide continuous shared access to the same software repository. As a result, concurrent development is becoming the norm and vendors of commercial CM systems are reacting with improved features for full optimistic concurrency. The SCM research community needs to investigate a unified representation for concurrent development [vand95]—the branching model is no longer adequate and workspace/change set concepts are too general. The DaSC layer model (or similar models, such as Tandem's Fully Populated Paths [Schw95]) could serve as the basis for such a representation.

In surveying the currently available CM tools, we were struck by the lack of support for dealing with configuration items that were not represented in ASCII text, including word processor produced documentation, databases (e.g., test cases), the project files for advance graphical user interface generators (e.g., XVT), and "source code" for non-textual languages (e.g., Prograph CPX). Most tools will allow the entire "binary" entity to be placed under configuration control, but they do not have the ability to answer such basic questions as, "What has changed in this non-textual configuration item?", "What are the differences between these version-sets?", "How can we revise item B in a manner similar to the revision already applied to A?" We believe that DaSC is ideally suited to work in this environment and plan to extend the implementation fill the void.

Acknowledgments

Thanks are due to the entire DaSC team for their efforts in this research project: Morven Gentleman, Charles Gauthier, Darlene Stewart, Marceli Wein and Anatol Kark. I would also like to thank the many readers of the early drafts of this paper and of the CM survey for their helpful suggestions and references to additional CM systems.

References and Bibliography

[Berl92] H. Ronald Berlack. *Software configuration management***.** John Wiley and Sons, New York, NY, USA. 1992. 330 pages.

[Buck93] Fletcher J. Buckley. *Implementing configuration management: hardware, software, and firmware*. IEEE Press, New York, NY, USA. 1993. 249 pages.

[Dart90] Susan Dart. Spectrum of functionality in configuration management systems. Carnegie Mellon University, Software Engineering Institute Technical Report: *CMU/SEI-90-TR-11*, December 1990. 38 pages.

[Dart92] Susan Dart. The past, present, and future of configuration management. Carnegie Mellon University, Software Engineering Institute Technical Report: *CMU/SEI-92-TR-8*, July 1992. 28 pages.

[Dart94] Susan Dart. Configuration Management: the KEY to Process Improvement. Talk to the Groupe d'amélioration des processus de génie logiciel, Centre de Recherche Informatique de Montréal, April 20, 1994.

[Estu94] Jacky Estublier and Rubby Casallas. The Adele Configuration Manager. *Configuration Management* (Walter F. Tichy, ed.). John Wiley and Sons, Chichester, England. 1994. pp. 99–133.

[Estu95] Jacky Estublier. Work Space Management in Software Engineering Environments. Private communication.

[Feil91] Peter Feiler. Configuration management models in commercial environments. Carnegie Mellon University, Software Engineering Institute Technical Report: *CMU/SEI-91-TR-7*, March 1991. 54 pages.

[Gent89] W.M. Gentleman, S.A. MacKay, D.A. Stewart, and M. Wein. Commercial realtime software needs different configuration management. *Proceedings of 2nd International Workshop on Software Configuration Management (SCM)*, Princeton, NJ. October 24-27, 1989. Published as *Software Eng. Notes*, 17(7): 152-161; 1989. NRC 30695.

[IEEE87] IEEE/ANSI. IEEE Guide to software configuration management. *ANSI/IEEE Std 1042-1987*. IEEE Press, New York, NY, USA. 1987. 92 pages.

[IEEE90a] IEEE/ANSI. IEEE Standard glossary of software engineering terminology. *IEEE Std 610.12-1990* (revision and redesignation of *IEEE Std 729-1983*). IEEE Press, New York, NY, USA. 1990. 83 pages.

[IEEE90b] IEEE/ANSI. IEEE Standard for software configuration management plans. *IEEE Std 828-1990*. IEEE Press, New York, NY, USA. 1990. 16 pages.

[MacK94] Stephen A. MacKay. An Evaluation of Configuration Management Systems and Tools. In preparation.

[Mahl94] Axel Mahler. Variants: Keeping Things Together and Telling Them Apart. *Configuration Management* (Walter F. Tichy, ed.). John Wiley and Sons, Chichester, England. 1994. pp. 73–97.

[Nodd94] K.E. Noddin. Derive and Consolidate in the DaSC Configuration Management Model. National Research Council of Canada, Institute for Information Technology Technical Report. In preparation.

[Schw95] Bill Schweitzer. Fully Populated Paths: A Conservative, Simple Model for Parallel Development. *Proceedings of 5th International Workshop on Software Configuration Management (SCM-5)*, Seattle, WA. April 24-25, 1995. (These Proceedings).

[Tich88] Walter F. Tichy. Tools for Software Configuration Management. *Proceedings of the International Workshop on Software Version and Configuration Control*, Grassau, FRG. January 27–29, 1988. pp. 1–20.

[vand95] André van der Hoek, Dennis Heimbigner, and Alexander Wolf. Does Configuration Management Research Have a Future? *Proceedings of 5th International Workshop on Software Configuration Management (SCM-5)*, Seattle, WA. April 24-25, 1995. (These Proceedings).

[vend94] Various papers, manuals, advertising brochures, electronic mail messages, etc. supplied by the CM vendors.

[Wein92] M. Wein, Wm. Cowan, and W.M. Gentleman. Visual Support for Version Management. *Proceedings of the 1992 ACM/SIGAPP Symposium on Applied Computing (SAC)*, Kansas City. March 1–3, 1992. pp. 1712–1723. NRC 33170.

[Wein95] M. Wein, S.A. MacKay, W.M. Gentleman, D.A. Stewart and C.-A. Gauthier. Evolution is Essential for Software Tool Development. *Proceedings of the Eighth International Workshop on Computer-Aided Software Engineering (CASE '95)*, Toronto. July 9-14, 1995.