Technical Report

CMU/SEI-91-TR-7
ESD-9-TR-7

Configuration Management Models in Commercial Environments

Peter H. Feiler

March 1991

Technical Report

CMU/SEI-91-TR-7
ESD-91-TR-7
March 1991

Configuration Management Models in Commercial Environments

Peter H. Feiler

Software Development Environments Project

Approved for public release.
JPO approval signature on file.

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

Configuration Management Models in Commercial Environments

Abstract: A number of advances can be observed in recent commercial software development environments in support of configuration management (CM). These advances include: configurations as managed objects; transparent access to repositories; and, in addition to the familiar *checkout/checkin model*, three CM models based on configurations. These CM models are the *composition model*, the *long transaction model*, and the *change set model*. Typically, one or two of the models can be found in a system.

This report analyzes the models with respect to their potential impact on the software development process, resulting in several observations.[1] Some of the models exist in a number of variations, each impacting the software process differently. CM capabilities can be found not only in CM tools and environment frameworks, but also in development tools. Integration of such tools into environments raises the need for different CM models to interoperate. Therefore, it is desirable to evolve to a unified CM model that encompasses the full range of CM concepts and can be adapted to different process needs.

1. Introduction

Configuration management (CM) is a key element of the software process. It is a discipline that provides stability to the production of a software system by controlling the product evolution, i.e., continued and concurrent change. As a management discipline, CM controls the evolution of a product through identification of product components and changes, through initiation, evaluation, authorization, and control of change, and by recording and reporting the history and status of the product and its changes. As a development support function, CM maintains the actual components of a product, records the history of the components as well as of the whole product, provides a stable working context for changing the product, supports the manufacture of the product from its components, and coordinates concurrent changes.

CM has received increased attention in the last several years, both from a management perspective, and from a tool and environment support perspective. In the former case, CM is an important component of several levels in the software process maturity model, [12] and CM related standards have been developed and improved, e.g., by IEEE. [14] [13] In the latter case, there is both research activity and commercial development. The research is reflected in papers in many software engineering and environments conferences as well as specialized workshops such as the International Software Configuration Management Workshop series. [25] [26] [27]

[1]This report corresponds to contract deliverable "Special Report (Advances in Configuration Management Support)" of SEI TO&P #2-114.

At the same time, commercial systems are offering new CM functionality. This functionality is a step beyond the *checkout/checkin model*, which has been made popular with UNIX Revision Control System (RCS) and its ancestor, Source Code Control System (SCCS). Examples of these systems are: standalone CM tools, e.g., Softool Change and Configuration Control (CCC), Software Maintenance & Development Systems Aide-De-Camp, and CaseWare Amplify Control; environment frameworks with CM capabilities, such as Apollo Domain Software Engineering Environment (DSEE), and Sun Network Software Environment (NSE); and CASE tools with multi-user support, e.g., Interactive Development Environments Software Through Pictures, Procase SmartSystem, and the Rational Ada Environment. Unfortunately, many of the systems are described in terms of their operations, reflecting their particular implementation of a concept and using their own terminology. We have examined a number of these systems, in many cases through hands-on experimentation, extracted relevant CM concepts to highlight similarities and differences, and observed advances in CM capabilities.

The advances we have observed can be summarized as follows:

- Smart processing techniques have evolved to improve the cost of rebuilding after changes to a product.
- Tool versions are being tracked in order to provide consistent application of tools in the build process.
- Version management of components has been complemented with the ability to operate on configurations.
- Repository access and version selection can be made transparent to both users and tools.
- New CM concepts have been introduced; these concepts can be summarized in three CM models, in addition to the checkout/checkin model: the *composition model*, the *long transaction model*, and the *change set model*.

The four CM models can be characterized as follows. The *checkout/checkin model* offers version management of individual system components. The *composition model* focuses on improving the construction of system configurations through selection of alternative versions. The *long transaction model* emphasizes the evolution of systems as a series of configuration versions and the coordination of concurrent team activity. The *change set model* promotes a view of configuration management focused on logical changes.

In this report we focus on the concepts represented in the four CM models. Dart [5] provides an overview of the spectrum of functionality in CM systems that we have observed. Surveys of particular CM products can be found in publications such as CASE outlook. [10]

Many of the systems support only one or two of the models. For that reason, we examine each of the models separately. For each of the models, first the concepts introduced by the model are discussed. Then, the actual use of a CM system based primarily on the model is summarized in the context of typical software development process scenarios. They include maintenance of version history, system construction, evolution of a family of systems, coordination of team development, controlled promotion of new system versions, and system evolution based on logical changes.

The report concludes with several observations about the future of CM support. Some of the models exist in a number of variations, each impacting the software process differently. CM capabilities can be found not only in CM tools and environment frameworks, but also in development tools. Integration of such tools into environments raises the need for different CM models to interoperate. Therefore, it is desirable to evolve to a unified CM model that encompasses the full range of CM concepts and can be adapted to different software process needs.

2. The Checkout/Checkin Model

The checkout/checkin model characterizes CM support as it is exemplified by UNIX *SCCS* [24] or *RCS* [29] and *make* [9]. Such CM systems consist of two relatively independent tools: a repository tool, and a build tool. The *repository tool* stores versions of files and provides mechanisms for controlling the creation of new versions. The *build tool*, given a description of the components that make up a product, automates the generation of *derived files*, such as object code and linked executables, through application of tools to source files. (In this report the terms component and file are used interchangeably. Many commercial CM systems are still file based, though they may present files to the user as logical components.)

In this section, we first describe the concepts that are embedded in systems with the checkout/checkin model and point out potential restrictions they may place on the user. Then, we discuss how the checkout/checkin model is used to support various development scenarios. Notice that over time users have developed conventions of usage that more closely reflect their needs. Those conventions may be embedded in command scripts layered on top of the CM system, but, as we will see in this report, the other three CM models introduce concepts that more directly support users' needs.

2.1. Concepts in the Checkout/Checkin Model

The checkout/checkin model focuses on version support for individual files. Figure 2-1 shows how a CM system supporting the checkout/checkin model presents itself to a user. The user works with a repository and with the file system. Files are versioned and stored in the repository. Creation of new versions is controlled by the repository tool. Files are, however, not directly accessible in the repository. Users have to retrieve, i.e., *check out*, a version of a file into the file system in order to access its content. Files can be retrieved for read access, e.g., for the user to examine a design document or for the compiler to access a file that has been included by another file. Files can also be retrieved for write access. In that case, concurrency control mechanisms of the repository coordinate multiple retrieval for modification. Modified files can be stored back into the repository, i.e., *checked in*, resulting in a new version of the file.

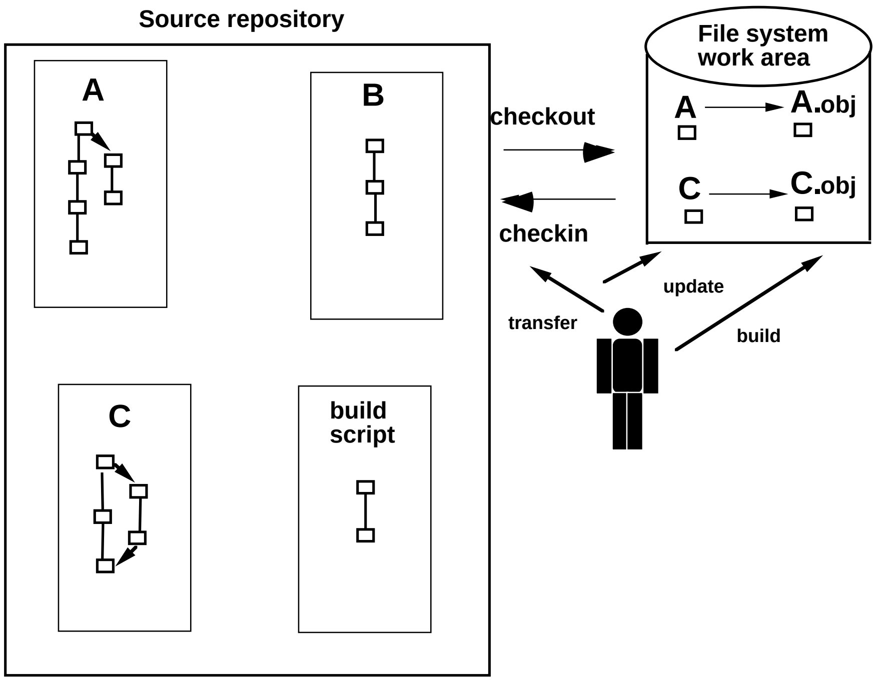

Figure 2-1: Operational Model for Checkout/Checkin

The file system is the actual work area for the user and tools. The repository tool does not provide support for managing the work area and relies on the file system to provide exclusive write access to the user who checked out the file for modification. The user operates tools, including the build tool, in the usual manner. The tools are not aware of versioning of files and selection of the desired version. Directory hierarchies can be used to structure the files, i.e., the components, that make up a system. Build tools may simply be command scripts or tools that interpret a description of the files in the file system that make up a system. These descriptions or command scripts can be stored and versioned in the repository. They represent a description of the configuration of a system without being concerned about versioning.

The capabilities of the checkout/checkin model focus on maintaining a version history of individual files, and on controlling their concurrent modification. The remainder of this section will discuss these two aspects in more detail.

2.1.1. Version Branching and Merging

The checkout/checkin model provides the following conceptual primitives for versioning of files: evolution of a sequential version history (referred to as *revisions*); creation of *version branches*, i.e., version sequences that have as their starting point a particular version in an existing branch, but evolve independently; and *merging* of two versions from different branches into a new version in one of the branches. The result is a version history for files that has a graph structure. This structure is referred to as a *version graph* and is illustrated in Figure 2-2.

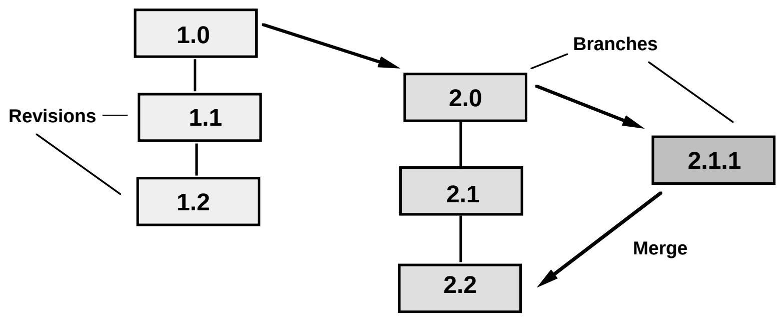

Figure 2-2: Branches and Merges in Version Graphs

A version branch is used for several purposes:

- To represent an independent path of development, i.e., the maintenance of a component as part of a field release vs. its further development.
- To represent different variants of the component. Variants may reflect different time- or space-efficient implementations, ports to different platforms, interfacing with different window systems, etc.
- To represent experimental development, which may be abandoned or folded into the primary development at a later stage.
- To accomodate for the fact that two developers were required to concurrently make changes to a component and thus, serial update of the component would have been too restrictive. In this case the branch exists only temporarily and is merged as soon as both modifications are completed.

These interpretations of branches are based on usage conventions that users place on the CM system rather than on usage patterns that the CM system directly supports.

A merge combines the file modifications that have independently occurred in two version branches into a new version in one of the branches. A merge has two aspects: first, the actual merging of the content of the two file versions; and second, the reflection of the merge in the version graph as part of the version history.

Many CM systems include tools for merging ASCII text files. Such tools determine text line changes to both file versions with respect to their common ancestor in the version graph, and check for line changes common to both versions. For lines that have changed in both versions, the user is asked to edit the line to produce a merged version. Some tools automatically include lines changed in only one of the two versions, while other tools ask users to confirm the inclusion of the change, allowing them to check for side effects on the other version.

This technique does not work well for ASCII files that are external representations of development tool-specific structures, e.g., formatted documents or design graphics. Merging of non-ASCII files is generally not supported. There is ongoing research in structural and semantics-based merging. [30] [23]

Merges can be recorded as part of the version history by reflecting them in the version graph. The semantics of the merge operation differs depending on the set of version pairs it can be applied to. The differing semantics result in different limitations in the ability of a CM system to support various development patterns. In the following, we discuss several merge semantics starting with a restrictive form of merge and progressively removing restrictions.

- A merge can be limited to merging the latest version of a branch (offspring branch) with the branch it originated from (ancestor branch), resulting in a new version in that branch. Furthermore, the offspring may be terminated, i.e., new versions cannot be added after the merge. This form of merge supports branches for experimental development, i.e., work that is intended to be merged back to where it originated from or to be discarded, as well as for managing concurrent modification of a component well, but is too restrictive for other purposes.
- The merge semantics can allow offspring branches to continue to evolve after a merge. The removal of the termination restriction allows propagation of change to be supported. Continuing work in an offspring branch, e.g., representing bug fixes to a release, can also repeatedly be included in the ancestor branch representing further development. Notice that the restriction to merge with the ancestor branch results in the ability to support propagation of change in only one direction—up the hierarchy of branches.
- The merge semantics may allow not only merging from, but also merging to an offspring branch. The effect of this capability is that changes can be propagated in both directions of the branch hierarchy. It allows the CM system to directly support scenarios such as the following. Assume that an application is maintained on multiple platforms. The ancestor branch represents the linear evolution of the platform-independent implementation. The port to different platforms and their maintenance is managed in offspring branches. Platform-independent changes can be carried out in the ancestor branch and propagated to the platform-specific branches independently. Similarly, some changes performed in a platform-specific branch may turn out to be applicable to all platforms, thus, can be propagated to the ancestor branch, and from there to the other branches.

- The merge semantics may allow merging between any two branches in the same version graph. Merging is not restricted by the time order in which branches have been created, but the branching hierarchy is still used to determine the common ancestor version for merging. Merging arbitrary branches allows a CM system to support selective propagation of a change that was originally performed in one branch to other branches and to maintain a record of that propagation.
- The merge semantics may allow merging of branch versions other than the latest. In effect, a consecutive subset of changes in a branch is propagated. This property supports propagation of changes at a later time, i.e., even after additional versions have been added to a branch. Either one of the branches participating in the merge may be the recipient of the merged version, i.e., its latest version gets extended with the (sub)set of changes from the other branch, or neither of the branch versions is the latest and the resulting version is the initial version of a new branch. Finally, merging of selected changes may be allowed. Selected changes may not necessarily have to be consecutive as long as they do not depend on each other. These refinements of the merge semantics improve the ability of a CM system to support change management, i.e., the management of logical changes as reflected in change requests, and their inclusion in different configurations. The change set model, discussed in Section 5 focuses on improving change-oriented configuration management.

Not all CM systems opt for the same merge semantics on the version graph. It is important to understand the limitations of the merge semantics in a particular CM system, especially if the CM system is primarily based on the checkout/checkin model. Since the checkout/checkin model focuses on versioning of individual files, the impact of some of the restrictions may not have been immediately obvious. However, usage conventions of such systems, expressed in terms of higher-level concepts such as those found in the other three CM models, make the limitations more apparent.

2.1.2. Concurrency Control

Concurrency control is provided by controlling the retrieval of modifiable copies of files from the repository into the file system. The latest version of a branch can be retrieved for modification and the branch is locked. Branch locking guarantees that only one person at a time is in the process of creating a new version for a particular branch. When the modified copy is checked in, a new version is added to the branch and the lock is released.

Alternatively, a version can be retrieved as the initial version of a separate branch. This may be desirable when one developer makes a major change, locking the original branch over a long time, and a second developer is prevented from making a quick bug fix until the first change is completed. Once both modifications are completed and checked in, the two changes can be merged.

While a retrieved file resides in the file system, it is outside the control of the CM system. Access control and write locking mechanisms of the file system are relied upon to guarantee exclusive access for modifications. In addition, a CM system may support access control on the repository. An access control list may determine whether a particular person is permitted to check out a file for modification. Restriction of checkout to a single person has the effect of a lock.

2.2. The Checkout/Checkin Model in Use

This section discusses how developers use the model in evolving families of systems. First, we examine how a system consisting of a collection of components, i.e., files, is managed. Then, we discuss the developer work area in the file system, and follow with a discussion of change in context and concurrent change to systems. The section concludes with ways of managing variants of systems and providing scopes of visibility in the repository.

2.2.1. System Structure

In the file system, the directory hierarchy is typically used to organize a particular system's files into a system structure. Directories represent subsystems, and files contained in it are components of the subsystem. Some of the repositories support only a flat name space of files, e.g., SCCS and RCS. In such cases, the system structure cannot be reflected in the repository. Instead, separate repositories are placed at each of the directory levels that are part of the system structure.

Build scripts provide a further refinement of the system structure and add *use* dependencies among components. These use dependencies are usually already recorded in the program, e.g., as include or import statements, and must be replicated by the user. The scripts may include not only rules for building a system out of its components, but also information for retrieving a particular version for each of the components. The file containing the build script is maintained by the developer and can be stored in the repository just as any other file. The result is versioning of a file that, in effect, represents a configuration description of the system.

Some repositories support user-defined labels to be attached to branches and individual versions, e.g., RCS. In such a case, version selection information of a configuration can be included in the version graph of all components in the following way: the version of each component that belongs to a particular configuration can be labeled with the configuration name. This permits retrieval of the correct component version by the configuration name label. This scheme still requires a separate record to be kept of the names of components making up the system.

2.2.2. Developer Work Area

As discussed earlier, in the checkout/checkin model the developer does not actually modify files in the repository. Instead, the working version resides in a directory in the file system—outside the control of the CM system. Conventions are required to organize work areas for different developers and different work areas for one developer. Different work areas may be represented by different directories. File system access rights can be used to control access to a work area.

The work area holds copies of files retrieved from the repository, both for write and read access. In case of write access, the copy in the file system is the actual working copy. In case of read access, the file system copy can be viewed as a cache copy of the file version contained in the repository. There are two common working patterns for retrieving files for read purposes from the repository and for maintaining the work area up to date. Each of the two patterns results in a different degree of stability of the work area with respect to changes to the same file by other developers. However, the choice of one pattern over the other is sometimes driven by performance characteristics of the CM system rather than by the desired stability of the work area.

In the first working pattern the repository is the primary storage medium. The file system representing the work area is treated as temporary storage through which tools can directly access the component. Components are retrieved from the repository when read access is required. When reading is completed the file system copy is deleted in order to conserve disk storage. Retrieval is repeated every time the file needs to be read. This repetition guarantees that the file being read is up to date with the one in the repository. In particular, if the retrieval defaults to the latest version, new versions in the repository are automatically made visible to the work area as soon as they are released by other developers.

The cost of repeated retrieval of files for read access has resulted in a second working pattern. In this case, the work area contains all components of a configuration, potentially increasing storage consumption due to multiple copies of the same file in different work areas. In this working pattern the developer focuses primarily on the work area. Interactions with the repository occur when the developer releases modifications into the repository, and when the work area needs to be brought up to date with changes released by others.

2.2.3. Change in Context

Modifications to individual files are usually not made in isolation, but rather in the context of other files. A logical change to the system may require several files to be modified together. Such a change is performed in one work area. Unfortunately, repositories record only who has retrieved a file for modification, not the work area into which the copy has been retrieved. In such a case, the repository itself does not have enough information to identify what modifications make up logical changes. Instead, as part of a checkin file, versions making up a change can be, by convention, labeled with the same user-defined change label, e.g., the change request number. In addition, the comment added to the history log of each of the checked in files may contain an indication that the modifications make up one logical change. These records, however, are typically not interpreted by the repository.

The work area also represents the working context for system builds. Files are compiled in the context of particular versions of other files. Files are linked together into aggregates and data dictionaries containing information from multiple files are generated. As files are released into the repository, this build context will have to be preserved as well. Otherwise, the validity of the derived files, i.e., the compilations and generated data dictionaries is not guaranteed, and they have to be regenerated.

2.2.4. Concurrent Logical Changes

Modifications between multiple developers is coordinated by the repository allowing only one person at a time to check out a file for modification to produce the next sequential version. A logical change may require modifications to several files. Two people making different logical changes may require modification access to some of the same files. Since modification rights are granted on a file-by-file basis, a deadlock situation is possible. For example, both developers have to modify files A and B. The first developer checks out A and later attempts to retrieve B for modification, while the second developer checks out B and later attempts retrieval of A. This situation can be overcome by gaining modification access in a predefined order (a common technique for resource allocation in operating systems), effectively serializing the changes. This indicates that concurrency control has to be addressed at the level of a working context.

Locking of the whole working context, i.e., the complete system configuration, is too conservative. Alternatively, an *optimistic* concurrency scheme can be considered. In this scheme, concurrent modification is permitted by creating a version branch. At the time a change is released to the repository conflicting modifications will have to be merged. Optimistic concurrency schemes for configurations are discussed in more detail in Section 4.1.3.

2.2.5. Version and Variant Selection

A system family is supported by representing different variants as different branches in the version graph of the components. In many cases, only a subset of the components differs between two variants. As a result, the version graphs of different components can vary considerably. Version and variant selection for components becomes a challenging task that the user has to accomplish through consistent naming conventions.

CM systems supporting primarily the checkout/checkin model offer very limited direct version selection support. In many cases, CM systems allow symbolic labels to be attached not only to versions, but also to branches. This labeling allows users to indicate the variant and a version within the variant, e.g., the latest version. It is the user's responsibility to introduce a variant label even to components that do not have a particular variant branch, or to emulate a version selection process that searches for a version label based on a search path. In essence, users are emulating version selection schemes found in the composition model (see Section 3.1.2 for a discussion of selection schemes).

2.2.6. Scopes of Visibility

The basic mode of operation of the checkout/checkin model offers two scopes of visibility for changes: the work area for files checked out for modification, and the repository. However, multiple scopes of visibility may be desirable. Individual developers may want to make snapshots of their work, i.e., create versions not visible to others. Similarly, a team of developers may want to keep changes local to the team before releasing a stable version to others, e.g., to the quality assurance group, who in turn place tested versions into a public release area.

A common way of providing multiple scopes is to represent them as separate repositories. Access control on the repository will place limitations on who can add new versions and who can retrieve versions for reading. For example, an individual developer may have a repository for versions while working on a change. When completed, the change is checked in to the team repository. This approach results in replication of components in different repositories. It also may require reprocessing of moved components in the new repository, e.g., recompilation of the files in order to ensure that the object code in that repository is up to date.

Some CM systems support access control to branches in a repository. In such cases, different branches can represent different scopes of visibility. Control of a particular scope is achieved by limiting access to versions in a branch to a particular group. The scope of a version is changed by propagating it to, i.e., merging it with, the branch representing the target scope. This merging is illustrated in Figure 2-3. Changes are made in the development branch. When appropriate, they are promoted to the test and the release branch. In essence, a particular software process has been encoded by imposing a structure on the version graph and by restricting its evolution.

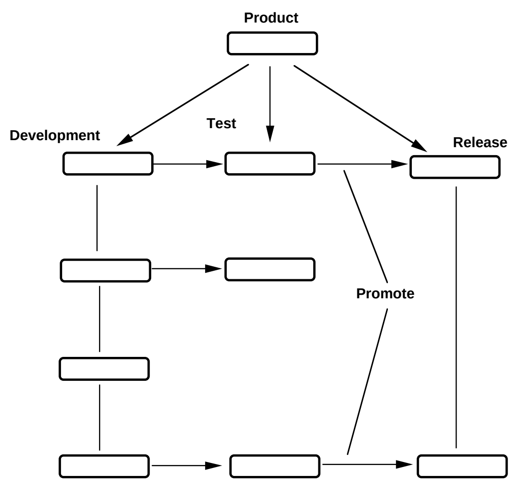

Figure 2-3: Promotion of Changes

2.3. Summary of the Checkout/Checkin Model

In summary, the checkout/checkin model focuses on versioning of product components. The operational concepts of checkout, checkin, branch, and merge are low-level primitives for which users have to develop usage conventions to better address their CM support needs. Examples are conventions for the use of branches, for maintenance of configuration information, and for supporting scopes of visibility of changes. CM systems primarily supporting this model focus on managing the repository. Support of users in their work areas is limited to component branch locking in the repository as a means of coordinating modifications. In practice, users of such CM systems have evolved conventions whose patterns emulate some of the concepts found in the other CM models.

3. The Composition Model

The composition model, a natural outgrowth of the checkout/checkin model, relies on the notions of repository and work area, as well as concurrency control through component locking, while it builds on the properties of a component version graph. It focuses on improving support for creating configurations, for managing their history, and for using them as working contexts under the auspices of a CM system. In this model, configurations are entities understood by the CM system. Developers operate with configurations by repeatedly composing a system from its components and by selecting the desired version for each component. We proceed by first discussing the concepts introduced specifically in support of composition, and then discussing the effectiveness of CM systems that primarily support the composition model in typical development scenarios.

3.1. Concepts in the Composition Model

A configuration in this model consists of a *system model* and *version selection rules*. A system model lists all the components that make up a system. Version selection rules indicate which version is to be chosen for each component to make up a configuration. The selection rules are applied to a system model, selecting a component version, i.e., *binding* a component to a version. The mode of operation in this model is that developers define the system in terms of its components and, in a separate step, select an appropriate version for each component. This mode of operation for dealing with versions is illustrated in Figure 3-1. In this figure and the remaining figures components are shown as boxes with square corners, while configurations are shown as boxes with rounded corners.

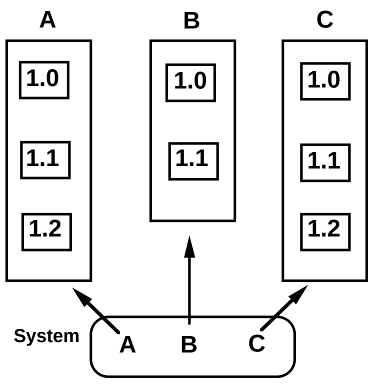

Figure 3-1: Component Version Selection

This two step process of composition and selection can be graphically visualized as an AND/OR graph [28]. One step is the aggregation of components into a composite (an AND node). The other step is the selection of an appropriate version for each of the composite elements (an OR node). In this general form, the selection step can occur at any level of the system composition, thus allowing a CM system to provide flexible support for management of system *variants*, i.e., system families at different levels of granularity.

A configuration can act as the working context. The CM system uses the versions indicated by a particular configuration as the default version when components are retrieved by developers. The section proceeds by discussing system models, version selection, and working contexts in turn.

3.1.1. System Models and Consistent Configurations

System models represent the structure of a system and provide a list of all components. Components may be grouped together, and the aggregate may in turn be part of another group, resulting in a hierarchical structure. This hierarchical structure may be augmented with use dependencies—intended use as expressed by import or include statements, or actual use in form of a procedure call. This use dependency permits a minimal configuration to be determined, i.e., a configuration containing only components that actually make up a system. Notice that system models provide a link between configurations, system build, and language systems.

The information represented in system models may be contained in a number of different places. The system structure may be reflected in the directory hierarchy in the file system as well as in the configuration hierarchy in the repository. Build scripts typically contain both system structure and dependency information. The components themselves may have both structural and dependency information embedded, e.g., in form of language constructs used in software. Language in this context is not limited to programming or textual language. Though, having system model information in common, configuration management support, system build, and language systems differ in their responsibilities regarding configuration consistency.

The concern of the CM support is to maintain a version history of a system and its components, and to select component versions that result in a consistent configuration. A configuration is considered version consistent if the selected version of a component is consistent with the selected version of other components. [21]

The responsibility of a system build facility is to maintain a set of derived components up to date with their sources. Those components may be derived from one or more sources, e.g., a compilation processing the source file and several include files, and may represent the aggregation of several components, e.g., linked code or a data dictionary.

The language system performs checks between components to validate the consistency of the composition from the perspective of the language semantics. This support may be as simple as checking the number and types of parameters, or as complex as full module interconnection language (MIL) support [22]. Some language systems include full system build capabilities.

CM systems, system build tools, and language systems must cooperate in order to avoid putting the responsibility of consistently maintaining redundant system model information on the user. Unfortunately, there is a wide range of CM systems, system build tools, and language systems with varying approaches to system model support, making their integration into a development environment a nightmare. For example, CM systems and system build tools can coexist: by the CM system providing a built-in build tool (Apollo DSEE [18]), by the CM system cooperating with a generic build tool (Sun NSE [4]), or by the CM system providing primitives based on which build capabilities can be implemented (Softool CCC [3]). Similarly, several approaches can be found for inclusion of language systems.

For the discussions in the remainder of this report it is sufficient for the reader to be aware of the interaction between consistent configurations, consistently updated derived components, and semantically consistent systems.

3.1.2. Version Selection Rules

CM systems supporting composition allow the user to indicate *version selection rules* rather than to manually pick component versions when retrieving components from the repository. The selection rules may uniquely identify the version of each component, i.e., repeated application of the selection rule will result in the same component versions. In this case, the configuration described by the system model and the selection rules is referred to as a *bound* configuration. Otherwise, a configuration is called *partially bound* or a *configuration template*. In this case, application of the selection rule at different times may result in a different bound configuration, e.g., selecting the latest version.

Selection rules can be characterized in two ways. One way is to view selection rules as search paths into a version graph with labeled versions and branches. In such a search path, the user indicates a sequence of selection options according to which the version of the components should be selected. Selection options can include the user's work area, other developers' work areas, labeled branches representing particular development paths (e.g., field release or internal development) and system variants (e.g., the Sun3 variant), and uniquely labeled versions (e.g., baseline configurations). Selection options may apply to all components or to selected subsets. Specifying the same selection option for all components results in consistently choosing the same alternative, e.g., system variant, throughout the system. A different system variant can be selected by changing the value of one selection option. Restriction of a selection option to a subset of components allows developers to use different criteria for selecting different subsystems, e.g., the tested version of one subsystem and the experimental version of another. Thus, different selection rules result in configurations with different characteristics. Apollo DSEE [18] is a good example of this view of composition through selection rules.

A second way of characterizing selection rules is based on a world of objects and attributes. Components and all their versions are treated as objects with attributes. Attributes include: version identifier, modification date, object status, and variant identification such as operating system, window system, etc. Selection rules are predicates in first order logic. Component versions that satisfy the predicate are part of a configuration selection. An example predicate is:

WindowSystem = X11 *and* **WSRelease > R3 & (status = tested** *or* **reserved = me)**

A configuration is partially bound if some components have multiple versions that satisfy the predicate. An example is a selection rule that does not choose between two variants. Such selection rules effectively represent configuration templates. In such a case, the predicate must be strengthened in order to result in a fully bound configuration. At the same time a predicate may be overspecified so that it cannot be satisfied by some components. The Adele system [6] is an example illustrating this view of composition.

The first approach to selection rules relies on labeling version graphs. Labeling conventions must be carefully chosen. If variants need to be distinguished according to several properties, e.g., distinguish support for different operating systems as well as window systems, the two dimensions must be encoded in the naming scheme. Furthermore, the version graph structure explicitly represents whether a version is a revision or a variant. This means a version must be categorized as variant or revision when created, since the version graph cannot be modified—only extended. The second approach of the composition model overcomes this shortcoming by allowing variant characterization to be expressed as attributes, independent of the evolution history in the version graph. In short, the second approach provides a more general solution, but requires a more sophisticated implementation.

3.1.3. Configurations as Working Context

In this CM model, configurations have version history. They may be versioned by versioning the system model and the selection rules, and by giving bound configurations version names. Developers can choose a particular configuration to be their *working context*. Working context means that as developers are accessing different components of a system, whether directly or through tools, by default the version indicated by the configuration description is used. The version may be selected when the component is retrieved from the repository or when it is accessed transparently. For a discussion of transparent repository access see Section 4.1.2.

A bound configuration provides a more stable working context than does a partially bound configuration. A partially bound configuration may be explicitly applied by the developer to retrieve components, thus effectively creating a bound configuration in the work area. This configuration will not change until the developer updates the work area by applying the partially bound configuration again. In other words, the developer decides when to upgrade the work area through an explicit command.

Alternatively, a partially bound configuration may be rebound every time a build is performed. In this situation, developers may instead choose fully bound configurations in order for the work area not to change constantly. In other words, developers control the content of their work area by choosing different selection rules (possibly defining new ones) or combinations of selection rules and system models to be their current working context.

The use of a configuration as a working context has important implications on the management of derived files. The build tool produces derived components from source component versions that the CM system is aware of. For a bound configuration, the component versions are constant. Thus, the build tool has to be concerned only with changes to components that are accessible for modification. For partially bound configurations, the build tool and the CM system must interact to enable the build tool to guarantee consistent update of derived components.

Through cooperation with the build tool, the CM system can become responsible for storing derived components in the repository in the context of a configuration. Similarly, new versions of components can be passed to a new scope of visibility, e.g., from the development team repository to a test team repository, as part of a configuration. The derived components of this configuration will retain their validity—eliminating unnecessary recompilations that may be encountered with CM systems primarily using the checkout/checkin model (see Section 2.2.6).

3.2. The Composition Model in Use

The composition model accommodates management of a system family. Typically, variants are represented by branches and are identified by branch labels. Selection of alternative members of a family becomes a matter of indicating alternative variant choices in the selection rule. This is done for the complete system configuration or for sub-configurations (subsystems). The latter is accomplished by limiting the selection option to a subset of the components—either by listing them individually or by referring to an aggregate in the system structure. Thus, developers can indicate selection of alternative family members at any level of the system structure in a natural way.

In the remainder of this section, we examine how the composition model deals with other aspects of configuration management. First we examine versioning of configurations in terms of system models and selection rules. Then, we discuss the evolution of configurations in terms of changing selection rules and the ability to cooperate by sharing work areas. Finally, we address support for managing logical changes.

3.2.1. Versioning of Configurations

CM systems, whose primary model is the composition model, still emphasize evolving a system by versioning individual components. This is due to the fact that the composition model is an outgrowth of the checkout/checkin model. As described in Section 2.2.5, the component version graphs are decorated with user-defined labels to record variant and configuration information.

In addition, configurations are represented as a combination of system model and selection rule, and by named bound configurations. System models and selection rules can be treated as components and stored as versions in the repository. For bound configurations developers resort to naming conventions to reflect the version history of configurations.

The difficulty lies in relating the version graphs of the system model and the selection rules to the version history of the configuration. The system model may change over time and has its own version history. However, this history is only a partial history of the system, since a system may evolve without changing its structure. Selection rules exist in variants representing configurations with different characteristics. When applied to system models at different points in time, different bound configurations may result. Though selection rules may change over time, they do not capture the system configuration history. It is the combination of system models and selection rules and their instantiations in time that result in configuration instances from their version history.

3.2.2. Evolution and Selection Rules

With CM systems based on the composition concept developers evolve a system as follows. A system model and a selection rule identify a configuration in the repository that is the starting point for change. A developer defines a new configuration consisting of the system model and a selection rule that includes the work area into which modified components are checked out. Once the work is completed, the developer preserves the configuration in the repository as a new system version by checking in all modified components as new versions. A new selection rule, which does not include the work area, but refers to the new component versions, must be created to enable the developer to identify the same system configuration after it is released from the work area.

If a CM system supports naming of a bound configuration separate from the system model/selection rule pair, i.e., by labeling the configuration, developers can rely on the labeling scheme to retrieve bound configurations from the repository. Developers can use partially bound configuration descriptions, referring to the latest released configuration in the repository to generate the appropriate bound configuration of the released system version.

3.2.3. Cooperation Through Shared Work Areas

Concurrency control of multiple developers relies on the locking and branching mechanisms on components in the checkout/checkin model (see Section 2.2.4). Despite the locking scheme, developers can cooperate by sharing modified components before those components are released to others. They can do so by sharing a work area. The configuration includes all modified files. Appropriate access rights prevent developers from changing each other’s components. Notice that several developers share the build process and the management of the derived components. Alternatively, different developers can maintain their own work area in which they perform builds independently, but still share modifications. This is accomplished by including more than one work area in the selection rules. Modifications can be shared selectively by limiting the appropriate selection option to the desired components.

3.2.4. Managing Logical Changes

CM systems based on the composition model support identification and propagation of a logical change as follows. Using a configuration as a working context, a developer is assumed to make the modifications to a collection of components in a work area. In addition, the developer may indicate the name of a formal change request or task description to be part of the working context. In such a case, the CM system can identify new component versions to be part of a logical change even though the developer checks in the components independently. The version created by the checkin can be labeled with a change name, e.g., the identification of the change request. [3] This permits the retrieval by name of all component versions involved in a logical change. Similarly, the names and newly created versions of modified components can be recorded in a log attached to a change request or task description. [17] Typically, such logs are not interpreted by CM systems.

Support for propagation of change is somewhat limited. The capabilities of a CM system supporting the composition model through labeled version graphs are inadequate for determining whether a logical change is contained in a particular configuration. Similarly, propagation of a logical change, i.e., its inclusion in another configuration, is not a simple matter. On one hand, the change may be part of the same branch. In this case, the selection rule can be updated to refer to the desired version. The result is the inclusion of all logical changes in the branch preceding the desired one, even though some may be independent of each other. On the other hand, the change may be in a different branch. In this case, the set of components modified in either of the two configurations must be determined, before the files can be merged.

3.3. Summary of the Composition Model

The composition model operates on configurations by composing aggregates from components and selecting appropriate versions of each. The system structure is captured in a system model. The system model provides the link between configuration support, system build tools, and language systems. This link permits the CM system supporting the composition model to include management of derived objects and checking of interfaces between components as well as between aggregates, i.e., subsystems.

Selection rules provides guidelines for the CM system to perform version selection. This allows the developer to express selection of alternatives in a natural way. Support for evolution must be expressed in terms of composition with changing selection rules. Support for change migration is limited to the capabilities of change merging at the component level and record keeping by the developer.

Version selection by selection rules can be modeled in two ways: by search path on labeled version graphs, and by predicates on attributed version objects. The search path view of selection rules is more intuitive for users to express desired configurations. Its limitation is that variant and logical change information have to be encoded in naming conventions. The predicate view of selection rules provides more flexibility and extensibility to adapt to different modeling requirements. The underlying theories permit validation of consistent configurations to be expressed in the same conceptual framework. Its disadvantages are that developers may not want to deal with first order logic and that its implementation is more complex.

4. The Long Transaction Model

The long transaction model focuses on supporting the evolution of whole systems as a series of apparently atomic changes, and on coordinating the change of systems by teams of developers. Developers operate primarily with configurations rather than with individual components. A change is performed in a transaction. A particular configuration is chosen as the starting point for changes. The modifications are not visible outside the transaction until the transaction is committed. Multiple transactions are coordinated through concurrency control schemes in order to guarantee no loss of changes. The result of a transaction commit is a new configuration version. The result of a series of sequential changes is a sequence of configuration versions, referred to as *development path*. Configuration versions can branch from an existing development path, resulting in a new, *independent* development path. It is called independent because both paths can evolve independently. This interpretation of the version graph of a configuration is illustrated in Figure 4-1.

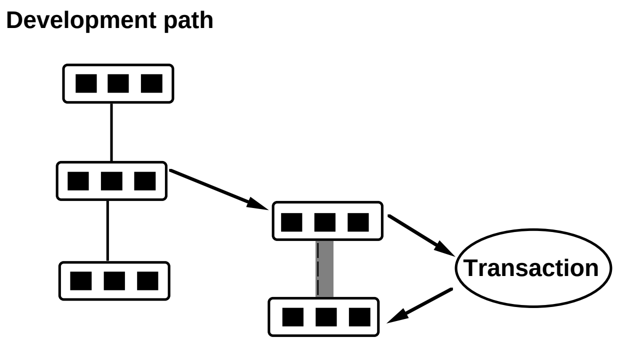

Figure 4-1: Version History of a Configuration

The long transaction model differs from a traditional database transaction. The database transaction performs an update operation on the database, while a long transaction creates a new version of modified data elements. The long transaction also differs in that it is persistent. This means that the long transaction may have a duration of hours, days, or months, and it must exist beyond the duration of a developer's login session as well as potentially beyond the uptime of a developer's workstation. Finally, a long transaction may represent the work of a group of developers with coordination among them through nested transactions, while a database transaction represents one activity and nesting is primarily used to decompose the update into finer grain operations. An extensive discussion of database transactions, their limitations in support of long transactions, and recent research on extensions to better address long transactions is found in Barghouti. [1] In this section, we first discuss the concepts specifically introduced by this CM model, and then elaborate on the effectiveness of CM systems primarily supporting the long transaction model.

4.1. Concepts in the Long Transaction Model

In the long transaction model, developers primarily work with versions of configurations. Developers first select the version of the system configuration, then focus on the system structure. The version of the components has implicitly been determined by the configuration. This operational view of version selection is illustrated in Figure 4-2. It is contrary to that of the composition model, where the developer is first concerned with the system structure, and then focuses on the selection of component versions (see Figure 3-1).

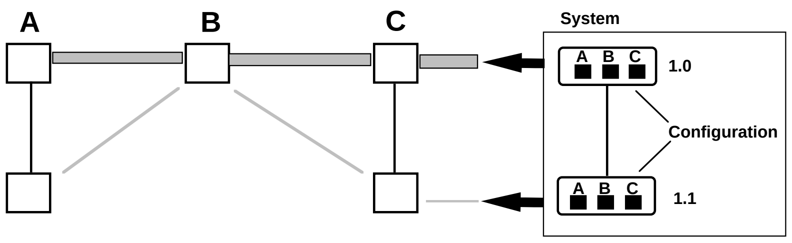

Figure 4-2: Configuration Version Selection

A long transaction consists of two concepts: a *workspace*, and a *concurrency control scheme*. A workspace represents the working context and provides local memory, i.e., data storage visible only within the scope of the workspace. As such, the workspace replaces the use of the file system as the work area, allowing the CM system to not only manage the repository, but to also support developers while they change the system in their work areas. A concurrency control scheme is a policy for coordinating concurrent change. Typical examples of such a policy for database transactions are various protocols for serialization of transactions. A number of different policies can be found in actual CM systems, each with a different degree of concurrency and cooperation among developers. The section proceeds by first discussing the concept of workspace, and the implications of workspace management on a CM system, and then elaborating on the range of concurrency control schemes possible in the context of long transactions.

4.1.1. Workspaces

The workspace concept provides local history of changes, commitment of changes separate from their preservation, stability of workspace under developer control, nesting of workspaces, and the ability to control access. These are discussed in turn in this section.

The basic elements of a workspace are illustrated in Figure 4-3. A workspace *originates* from a bound configuration in the repository or an enclosing workspace. A workspace consists of a *working* configuration and a series of *preserved* configurations. The working configuration represents the configuration in which components and the system structure can actively be modified. Initially, the working configuration starts with the originating configuration. Modifications are made in the working configuration. A modifiable version of components and the system structure is created local to the workspace, either explicitly by the developer issuing a command, or automatically when a modification is attempted. A history of changes is maintained local to the workspace by preserving them in a sequence of immutable configurations. Preserving a configuration effectively requires the creation of an immutable version of all modified components and the addition of a bound configuration referring to these component versions to the version sequence of preserved configurations. Modifications continue to occur in the working configuration. The sequence of preserved configurations provides checkpoints to which developers can *revert* to. Reverting to the originating configuration effectively discards the work, i.e., aborts the transaction.

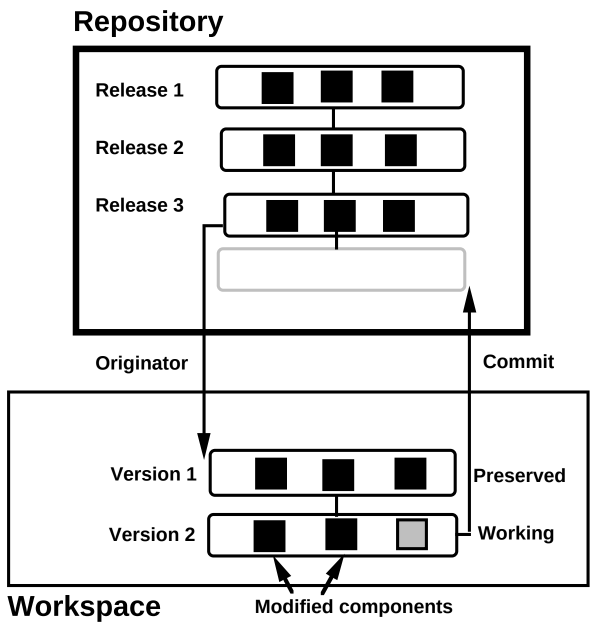

Figure 4-3: Workspace With Local History

Changes are made visible outside the workspace by *committing* a configuration from the workspace to the originating context, i.e., the repository or enclosing workspace. The committed configuration can be either the working configuration, in which case it is first preserved, or a selected preserved configuration. Committing a configuration from the workspace adds this configuration as a successor configuration version to the version history of the originating configuration version. After a configuration is committed, the workspace may be deleted, i.e., the transaction is terminated, or the workspace may be used for further changes. In the latter case, the effect is a *split transaction*, [15] making available partial changes before the transaction is completed.

While operating in a workspace, the developer is isolated from changes in other workspaces and from changes committed by others to the repository. Propagation of changes both out of and into the workspace are explicit operations and under the control of the developer. This is due to the fact that changes are made relative to the originating configuration and this configuration is an immutable bound configuration. The workspace is not up to date if a configuration version newer than the originating version exists. The user of a workspace can find out about this fact and ask for the workspace to be updated, resulting in a new originating configuration. Depending on the concurrency control scheme deployed, conflicts between changes existing in the workspace and those in the updated originating configuration have to be resolved (see also Section 4.1.3).

Workspaces can be *nested*. Instead of originating from the repository, a workspace can originate from a preserved configuration of another workspace. The result is a hierarchy of workspaces corresponding to a hierarchy of nested transactions, as illustrated in Figure 4-4. Each workspace represents a separate scope of visibility for changes. The bottom workspaces represent the work area of individual developers. The next level workspace represents the work area of a development team, while the third level workspace represents a testing area. Individual developers commit their changes to the team workspace, making them available for other developers to include in their workspaces. From the team workspace, configurations representing consolidated changes of team members are made available to the testing team by committing those configurations to the testing workspace. Finally, configurations passing a test suite may be released into the repository.

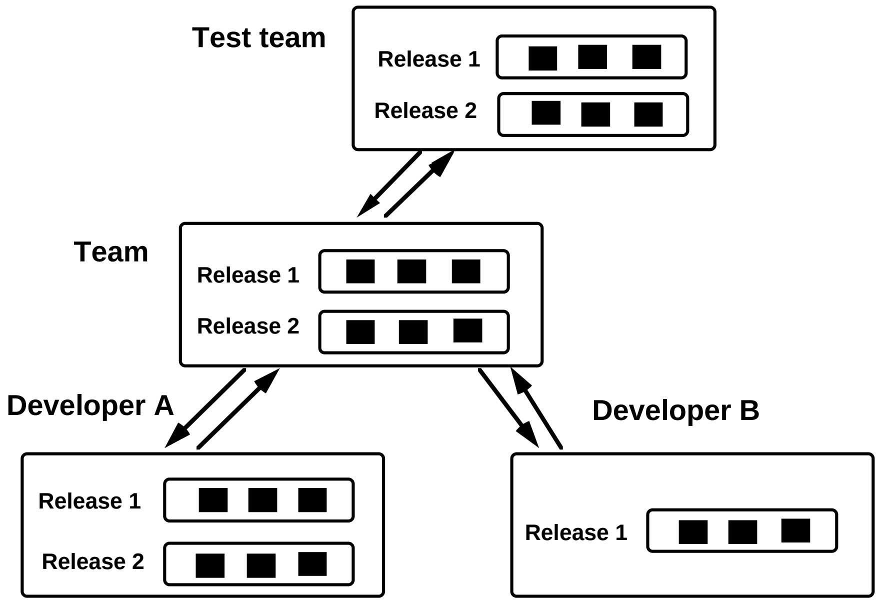

Figure 4-4: Transactions as Tasks

Access control on workspaces guarantees that read and modification rights as well as rights to propagate changes to another scope are restricted to appropriate project personnel. The access list of a workspace and its offspring workspace may be non-overlapping, e.g., the testing team and the development team are different sets of people, or the offspring access list may be a subset, e.g., the person listed on the individual workspace must be included in the team list. Offspring workspaces may not only restrict access lists, but also limit access to a subset of the system. Usually, this partitioning follows the system structure as reflected in the system model, i.e., the workspace is restricted to operate on a subsystem. In this case, the ancestor workspace represents the integrated system, while the offspring workspace represents work on each subsystem.

4.1.2. Workspace Management

CM systems supporting the long transaction model manage the workspaces. This has several implications on the services a CM system offers. The CM system provides transparent repository access via workspaces, applies space-efficient repository storage techniques to workspaces, and manages versions of derived components. These are discussed in turn in this section.

The developers enter a workspace instead of explicitly retrieving components from the repository into the file system. Two alternatives for entering a workspace can be found.

In the first alternative, the workspaces and their configurations appear as directories, and developers navigate to the appropriate one. The selection of the workspace and desired configuration version is embedded in the pathname and is apparent to the developer. Two configurations can be accessed simultaneously by a single process using two paths. This scheme can be found in the Rational Environment. [20] [7]

In the second alternative, the developer asks the CM system to map the desired configuration to a mountpoint in the file system. Access to the configuration as directories and files, which actually reside in a repository area managed by the CM system, is provided in a manner similar to the Network File system (NFS). However, in contrast to the NFS, the configuration is mounted on a per-process basis, i.e., it is accessible only to the process issuing the mount and to its offsprings (see Figure 4-5). Thus, different processes can access different configurations at the same mountpoint. Once a configuration is mapped, selection of the configuration version and the versions of its components is transparent, i.e., the version is not embedded in pathnames. A single process is limited to accessing one configuration version at a time using paths. This alternative can be found in Sun NSE.

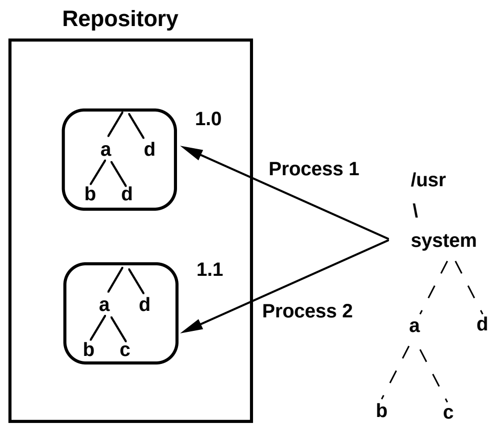

Figure 4-5: Transparent Configuration Access

By owning the workspaces, the CM system can employ a combination of techniques for reducing space consumption both in the repository and in workspaces. First, configurations in the workspace may be *virtual copies*, i.e., only copies of modified components exist. Other components are (transparently) shared between configurations, e.g., in Sun NSE. Second, component versions may be stored in compressed form and transparently reconstituted on demand, as found in Apollo DSEE.

As discussed in Section 3.1.3, a bound configuration provides the working context for builds, i.e., the generation of derived files. Thus, for each configuration in a workspace, the derived files are managed by the CM system as well. When a configuration is preserved, both the source files and derived files consistent with the sources are included. The CM system requires the system build to succeed before the configuration is preserved, or the system automatically deletes out of date derived files. This requires cooperation between the workspace management and the build capability. When a configuration is committed to the repository or to an enclosing workspace, both the sources and the derived files can be propagated.

Once source and derived files are distinguished, multiple variants of derived files can be attached to the same configuration of source files. Different variants of derived files for the same source can represent use of different tools or tool versions, e.g., compilers for different machine architectures, or different parameters to tools in the build process, such as different conditional compilation flags. In this case, entering a workspace involves selecting both a configuration and a derived file variant. The transparent access mechanisms discussed earlier can also support transparent selection of the desired derived file variant. Thus, management of these variants does not have to be embedded in build scripts.

4.1.3. Concurrency Control Schemes

In this section we discuss the range of possible concurrency control alternatives that have been found in different CM systems. Support for concurrent development falls into three categories: concurrency within one workspace; concurrency between workspaces requiring coordination; and concurrent, independent development. The first two categories address concurrent changes to one system configuration, while the third category assumes that a system evolves in independent development paths. These changes do not require coordination during their creation. These independent changes can later be serialized through propagation and merging, if desired.

Concurrency within one workspace occurs when multiple developers are active in the same workspace or when one developer is active in a workspace multiple times, .e.g., through separate windows. Since these simultaneously occurring activities share the data of the workspace, modifications must be synchronized through mutual exclusion. The following concurrency control schemes have been encountered for controlling concurrent modifications within a workspace:

- Limiting access to a workspace to one person through access control. In this case workspaces are used exclusively by individual developers, and concurrent work by different developers is coordinated through different workspaces.

- Allowing only one person at a time to be active in a workspace, even though several persons are included in the access list. This is accomplished by locking the workspace upon entry. Workspace entry for read access and modification can be distinguished. Multiple persons may be allowed to access preserved configurations, which are immutable, while locking is applied to the working configuration. This scheme can be used on a team workspace allowing one developer at a time to do integration work in the team area.
- Allowing simultaneous entry of a workspace, while coordinating modification of components, e.g., when two developers intentionally want to share a workspace in order to work cooperatively on a subsystem. Modifications by different people are serialized through locking of the component. The lock is set explicitly by a developer or it is set automatically when the first modification is attempted. The lock is released explicitly or it is released for all locked components automatically when the configuration is preserved.
- Either preventing a developer from entering the same workspace multiple times with modification rights, or relying on the tools used for modifying components to use a separate locking scheme to coordinate multiple edit sessions.

Concurrency requiring coordination between workspaces occurs when different developers work in separate stable workspaces and the collection of their changes evolves a system. Schemes for controlling concurrency in this situation fall into two categories: *conservative* and *optimistic* schemes. Conservative schemes require a priori locking across workspaces. The optimistic scheme allows modifications to occur concurrently, and conflicts are detected at the time it is committed.

The following conservative schemes have been found:

- Locking of the version branch of the system configuration in the ancestor workspace to limit the number of offspring workspaces to one. The lock is set in the enclosing entity, i.e., the ancestor workspace or the repository. Notice that a workspace contains a version sequence, i.e., a single version branch. The effect is a conservative transaction that assumes the complete system to be part of the modification set. This scheme is employed when a team workspace is created, guaranteeing that a single team will work on evolving the system to its next version.
- Locking of a sub-configuration version branch in the ancestor workspace, utilizing the ability to limit access of a workspace to a subset of the system. This scheme partitions the system according to the system structure with one workspace per subsystem. This scheme is employed when changes to a system can be partitioned according to the system structure. Each subsystem workspace itself may represent the work of a team. It may have several offspring workspaces with a different concurrency scheme for concurrent work by individuals.
- Locking of component versions in the ancestor workspace, allowing a system configuration to be modified concurrently, but no component in two workspaces simultaneously. This scheme takes advantage of the assumption that only a small subset of a system’s components is actually modified in a workspace, and the subsets rarely overlap. Component locking guarantees that no such overlap occurs.

The conservative scheme relying on component locking can be found in three variants, differing in the way they support commitment of changes:

- The first variant commits all modified components. The locks are released as part of the commit, resulting in the termination of the transaction. Some systems support commitment of changes without release of the locks, resulting in a split transaction. This scheme supports commitment of logical changes, i.e., logical changes are promoted and handed to other developers. This variant assumes that each committed logical change has no side effect on the other changes, and therefore permits their combination in the enclosing workspace. The language system can revalidate the semantic consistency of the resulting configuration in the enclosing workspace.
- The second variant supports commitment of individual components. Modifications to individual components are made available without their context. This allows developers to hand individual modified components to each other via the enclosing workspace. This variant can be used as an alternative to the sharing of a single workspace by multiple people.
- The third variant combines locking of components with the optimistic concurrency scheme on the configuration (see below). This scheme requires logical changes committed by sibling workspaces to be brought into a workspace before it can commit its changes. Although no modification conflicts exist due to component locking, this scheme encourages a workspace to validate a logical change in the context of other logical changes before committing it. The responsibility of validating the combination of logical changes lies with the developer workspace rather than the team workspace.

The optimistic concurrency scheme permits two workspaces to modify the same component in different workspaces as part of concurrent logical changes. This is illustrated in Figure 4-6. Conflicts are recognized when committing the change is attempted. A conflict is considered to occur when a component has been modified in more than one workspace. The first commitment succeeds, while other commits are aborted if a conflict occurs. When a conflict occurs, the workspace is updated with changes in the enclosing workspace, and the developer is asked to merge them with the modifications in the workspace. Effectively, the transaction is aborted and replayed. Once the merge is completed, the commit can be attempted again. In order to avoid repeated race conditions with other concurrent transactions, system can resort to locking the configuration in the enclosing workspace (see for example Sun NSE). The effect of this scheme is to force integration of concurrent changes before they can be made available to others. Concurrent workspaces committing later are responsible for resolving conflicts. The enclosing workspace or repository will always contain merged sets of changes. This means that in a hierarchy of optimistic transactions, changes can be propagated down easily, but propagation up is restricted.

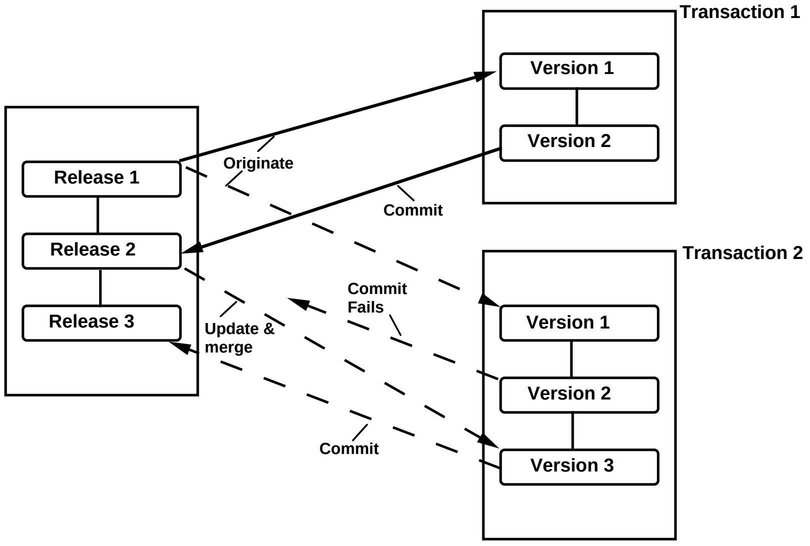

Figure 4-6: Optimistic Concurrency of Two Transactions

Three observations can be made regarding the above list. The first observation is that transactions can be performed at different levels of granularity, for individual components, and for composites (i.e., configurations) at different levels of the system structure. The combination of component and composite transactions with different concurrency control schemes at various nesting levels results in a range of supported concurrency behavior. The second observation is that many CM systems support a subset of the possible schemes, in many cases with little support for adaptation. Locking schemes offered at the level of the checkout/checkin model require the developer to work out higher level usage conventions and coordination schemes. CM systems supporting the long transaction model with a subset of concurrency control schemes, e.g., Sun NSE's choice of optimistic concurrency on configuration, pose potential limitations for certain development scenarios. [8] The third observation is that different concurrency control schemes are good matches for different team support scenarios. Therefore, it is desirable for a CM system to support a range of schemes and to permit the adaptation to the specific need.

4.2. The Long Transaction Model in Use

The previous section has illustrated that CM systems supporting the long transaction model are not focused on the repository, but extend their support to developers during active work. This section discusses the use of CM systems with long transactions as their primary CM model. It examines long transactions in the role of a repository, use of long transactions to manage promotion of changes and potential restrictions in the propagation of changes, long transactions and system composition, and management of variants.

4.2.1. Transactions as Repository

If the primary model supported by a CM system is the long transaction model, the transaction mechanism is also used to provide repository functionality [8]. As we have seen in the previous section, the workspace concept provides a version history at the configuration level. It maintains a linear development path, i.e., a single sequence of configuration versions without branches. Offspring workspaces represent branches, i.e., independent development paths. This is illustrated in Figure 4-7. Not terminating a long transaction results in its workspace being retained. Thus, the collection of workspaces of non-terminating long transactions acts as the repository.

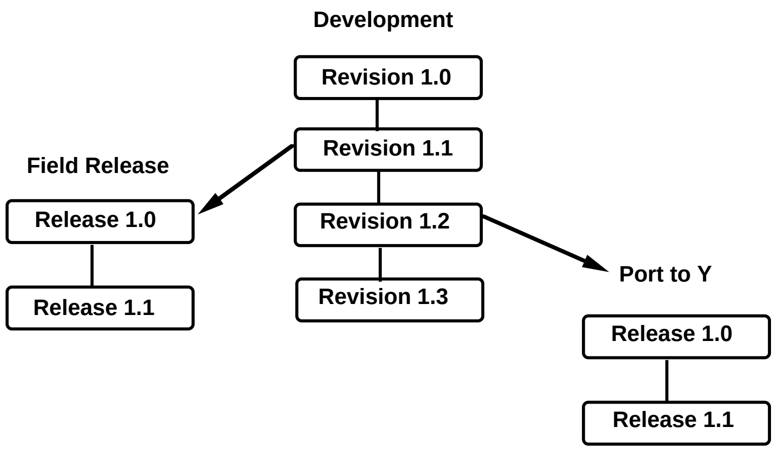

Figure 4-7: Transactions as Development Paths

4.2.2. Promotion and Propagation of Changes

Versions of configurations or changes may be in one of several states, e.g., in development, tested, or released. The discussion on nested workspaces in Section 4.1.1 illustrates such a scenario. The state reflects that the version satisfies certain conditions. The actual set of state values and transitions is determined by the particulars of the development process being captured by the CM system. Support for scopes of visibility in the long transaction model can be used to reflect such states. A separate workspace can be set up for each state to contain the appropriate versions. A state transition occurs when a configuration is committed to the enclosing workspace which represents the new state. Thus, configuration versions migrate up the workspace hierarchy as their state is promoted.

This form of modeling configuration state has several potential limitations:

- The nesting supports a linear progression of state transitions, which is satisfactory for many development scenarios.
- A committed configuration can be found in both the offspring and the enclosing workspace, although under different version identifications (unless the offspring workspace terminates or the version is explicitly deleted). Thus, it is assumed that promotion to a new state does not invalidate satisfaction of the previous state.
- This setup is primarily geared to retrieving configurations in a given state. Other queries, such as what state a particular configuration is in or tracking of the progress of a configuration through a progression of states require additional work.

When long transactions are the primary model of a CM system, the concurrency control schemes not only govern concurrent activity of a team of developers, but also place restrictions on the information flow in the repository. In the former case, one of the objectives of concurrency control is to serialize change. This forces merging of concurrent changes. The result is that, in a hierarchy of nested transactions, change can flow down the hierarchy relatively freely, while flowing up the hierarchy requires merging with changes from the enclosed transaction by first propagating those changes down. Propagation from one offspring transaction to another must follow the hierarchy. Under certain circumstances, these restrictions are too strict for information flow in the repository. Developers may want to propagate changes to other development paths more freely than the hierarchy of branches implies. For example, developers may want to selectively include changes made to one variant of a system in other variants (see also discussion on merge restrictions in Section 2.1.1). Furthermore, changes may want to be propagated to other development paths without requiring the originating development path to be brought up to date. For example, propagation of a bug fix from a field release path to an internal development path is not possible without picking up development changes, if the field release is an offspring path.

4.2.3. Transactions and Composition

Long transactions operate on configurations. As mentioned in Section 4.1.1, transactions can be applied to the whole configuration or to a subsystem. This means that transactions interact with the system structure. It is assumed that the system structure exists, and that the scopes of the nested transactions are to be partitioned according to the system structure. Workspaces do not have to be limited to decomposition. A workspace may be allowed to have more than one enclosing workspace, each representing a different subsystem. In this case, the offspring workspace represents a composition of subsystems. The subsystem versions are selected by indicating an appropriate version for the originating subsystem configuration. Such a capability, however, violates the semantics of nested transactions, and is typically not found as part of the long transaction model.

4.2.4. Variant Management

CM systems primarily supporting the long transaction model accomodate for maintenance of system families in two ways. First, sequential versions of a system are kept in a workspace, i.e., represent a development path. Different variants of the system are represented as separate offspring workspaces, allowing for independent evolution of the variants. Changes are propagated between the variants through the enclosing workspace. Variants are selected by choosing between development paths. Second, some CM systems support variants within one workspace. For example, Sun NSE supports multiple derived file variants for each version of a source configuration. In this case, the developer is led toward evolving all variants in lockstep before preserving or committing a new configuration version.

Typically, workspace names and version identification are user definable. Appropriate naming conventions will have to be employed by the developer to appropriately encode the attributes of each family member. Similarly, naming conventions distinguish between roles of transactions as development paths in the repository, and as work areas for change.

A long transaction represents a working context. The local memory of its workspace contains all modified components, i.e., a record of all modifications making up a logical change. When a configuration is preserved as a new version in the workspace, a version history log entry similar to a checkin log entry can be attached to both the component and configuration version. When a configuration version is committed to the enclosing workspace, the collection of log entries since the previous commit can be attached to the committed version in the enclosing workspace. This record of logical changes is usually not interpreted by the CM system and has limited capabilities for querying by the developer.

4.3. Summary of the Long Transaction Model

The long transaction model supports evolution at the configuration, i.e., composite level in a natural way. It provides stable workspaces with control over isolation from external change, scopes of visibility for changes, and coordination of concurrent change activity at various granularities of the system structure. By managing workspaces, CM systems can support developers during active development. A range of concurrency control schemes can be attached to transactions. Since each has a different impact on cooperative team support, it is desirable for a CM system to support the adaptation of schemes to different process needs.

When used as the primary CM model in a CM system, long transactions play the role of development paths in the repository as well. However, information flow restrictions due to concurrency control schemes are often too restrictive for change propagation in the repository.

Long transactions do not directly support composition, but support evolution of subsystems based on a decomposition according to the system structure. System families are supported as independent development paths treated as system variants, or through explicit variant support within a workspace.

Long transactions represent a working context for logical changes. The modifications recorded as part of the transaction log represent a logical change. Although such information is the basis of the change set model discussed in the next section, change propagation restrictions and limitations in change query capabilities are common.

5. The Change Set Model

The change set model focuses on supporting management of logical changes to system configurations. The *change set* concept, introduced by this model, represents the set of modifications to different components making up a logical change. It is the record of a logical change that persists after the activity creating the change has completed. Users of a CM system supporting this model can directly operate with change sets. In this model, configurations can be described as consisting of a baseline and a set of change sets. Changes are propagated to other configurations by including the respective change set. Different integration strategies can be pursued by developers for inclusion of a collection of logical changes into a new system release. Developers can track logical changes and determine whether these changes are part of a particular configuration. This view of configuration management, referred to as *change-oriented* configuration management due to its focus on logical changes, [19] differs from the *version-oriented* view of configuration management present in the other three CM models, which focuses on versioning of components and configurations.

The change set concept provides a natural link to *change requests*. Change requests are management tools for controlling the initiation, evaluation, authorization, and approval of changes. They are a record of the status of a change in the software process. While change requests contain information about a change, change sets represent the actual modifications.

An example of the change set concept are patches applied to system releases, e.g., mainframe operating systems. Some sets of patches may represent revisions to the I/O subsystem, while others may be fixes to the process subsystem. There may be constraints between patches. Some patches depend on other patches having been applied, while other patches may be incompatible with each other. Belady [2] discusses an approach for creating consistent configurations by validating legal combinations of patches.

5.1. The Change Set Concept

In the context of a single component, the change set is the set of differences (usually referred to as the *delta*) between two file versions. Applied to configurations, the change set is the set of differences between two configuration versions. This set of differences is the collection of deltas of those components that have been modified between the two configuration versions, i.e., the deltas of the set of changed components. This is illustrated in Figure 5-1.

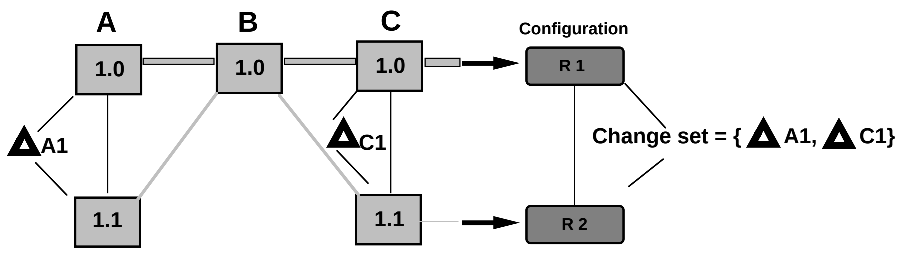

Figure 5-1: A Change Set

Developers may want to not only name individual logical changes, but also be able to refer to groups of changes. The Aide-De-Camp product [11] is an example of a CM system that supports repeated grouping of change sets into named entities. They can be used the same way as the original change sets. This facility for structuring change sets supports management of large numbers of change sets.

In the previously discussed CM models, change sets are not available as named entities. Instead, developers refer to versions of components and configurations by name. The difference between the ability to name change sets and to name versions seems subtle, but it has impact on the capabilities a CM system can provide. This section proceeds by first discussing this difference in naming, then elaborating on additional capabilities due to change sets as a recognized concept, and finally addressing consistent combination of change sets into new configurations.

5.1.1. Version Naming Versus Change Set Naming

The change set model focuses on change-oriented configuration management. Change sets are used to track logical changes and to define new configurations in terms of logical changes. A number of CM systems offer some aspects of the change set model, usually in the form of versions labeled with change names, and history logs of checked in components with respect to working contexts, i.e., logs attached to configurations, e.g., Sun NSE, [4] or to task descriptions, e.g., Apollo DSEE. [18] This section first summarizes use of version naming to record logical changes and then contrasts it with the ability to name logical changes directly.

The other CM models typically do not support change sets as explicitly named entities that can be operated on. Instead, developers refer to versions of components or configurations that include the change. User-defined labels on versions allow tagging of the component or configuration with a name for the change, e.g., the identification of the change request. Developers can then retrieve the appropriate version by that name and ask that the differences to an earlier version of each component be generated.

Paradoxically, many CM systems internally store component versions as a baseline version and a partially ordered set of deltas. These deltas are not directly retrievable, though. Similarly, some CM systems internally maintain sequences of configuration versions as a baseline and collections of changed components. These collections, however, are not directly retrievable.

A component or configuration version actually contains a collection of changes—the cumulative deltas of a version sequence in the version graph, as illustrated in Figure 5-2. Thus, determining whether a change is included in a particular configuration is not a simple query on a version label, but would require traversing the version graph.

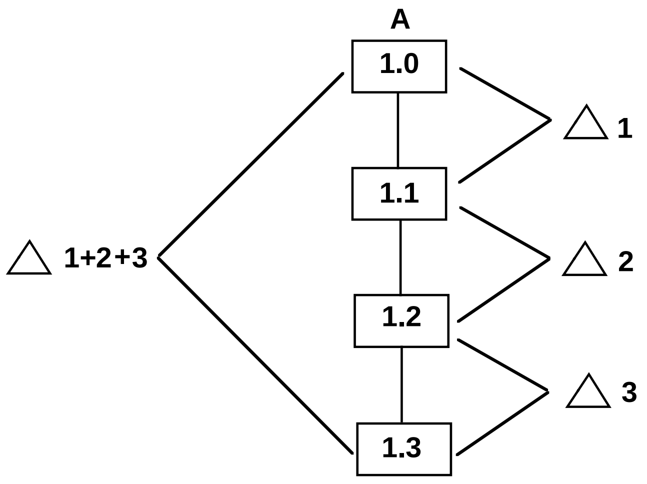

Figure 5-2: Cumulative Changes

Changes are propagated by merging versions. When a particular version is merged, the cumulative set of changes is propagated. Selective inclusion of changes in a configuration is not easily accomplished. This limits the ability to create tailored configurations by combining subsets of logical changes. Restrictions on merging to ancestor branches (see Section 2.1.1) and on committing configurations in a transaction hierachy (see Section 4.1.3) may further limit the ability to support propagation.

In contrast to version-oriented configuration management, change-oriented configuration management views configurations as being defined by a baseline configuration and a combination of change sets, i.e., a set of logical changes. In some respects, the two views are dual representations of configuration versions. This is illustrated in Figure 5-3. The figure shows both the version graph of a configuration (boxes) and the dual version graph of change sets (shaded ellipses). The apparent duality is due to the fact that the configuration versions can be expressed in terms of the baseline configuration and the change sets.

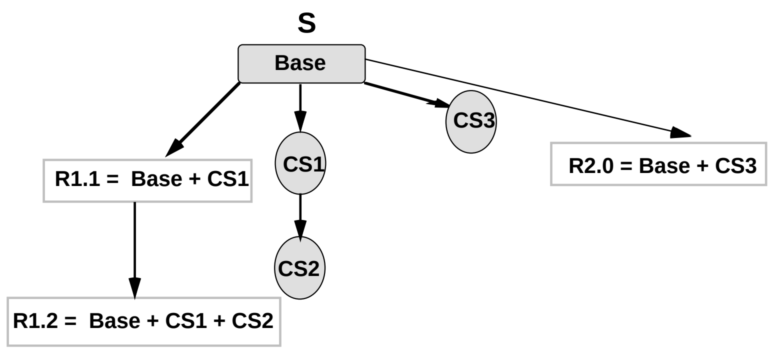

Figure 5-3: Version Graph of Configuration and Corresponding Change Sets

The change-oriented view differs from the version-oriented view in two ways. First, the explicit representation of logical changes permits tracking of change sets with respect to both individual components and configurations. Second, the ability to refer to individual change sets and selectively include them in configurations provides support for managing the evolution of a system based on propagation of logical changes to a set of maintained system configurations. This latter ability breaks the duality of configuration and change set versions. The next two sections address the two differences.

5.1.2. Change Set Tracking

Having change sets as named and manipulable entities allows developers to ask questions about their relationship to system components and configurations. These queries fall into the following categories:

- Determining which components have been modified as part of a logical change. This information can provide insight into whether for certain logical changes the same set of components requires modification, i.e., there exists a logical dependency that may potentially not be recorded explicitly in the system structure.
- Determining the collection of change sets a particular components is part of. This information can provide insight into the stability of a component based on the frequency and type of logical change. The type of a logical change may be recorded as part of the history log or as a user-defined attribute of the change set, e.g., bug fix, enhancement, generalization, portability improvement.

- Determining which change sets are included in a particular configuration. This information can provide insight into whether certain requested features or fixes have been included in the configuration in hand. The capabilities of the system configuration can be determined in terms of the characteristics of the included logical changes.
- Determining which configurations include a particular change. This information permits a developer to determine to what degree a particular logical change has been propagated to a set of maintained system configurations.

5.1.3. Consistent Combination of Change Sets

The change set model supports creation of new configurations by adding a combination of change sets to a baseline configuration. There are, however, constraints on which change set combinations are consistent. Some change sets are dependent on the presence of other change sets. Some change sets are in conflict with other change sets.

The evolution history of a system, reflected in its configuration version graph, provides some insight into the dependence of one change set on the presence of another. The configuration version graph (and its dual change set version graph) indicates configuration versions of change sets that were present when a particular logical change was made. This time ordering, however, does not necessarily imply a logical dependence of one change set on another. Two logically independent changes performed sequentially would be recorded in the version graph as time ordered. Determining whether two change sets are logically independent is more difficult than checking whether the sets of components involved in both change sets overlap. Such a determination requires consideration of both the write set, i.e., the set of modifications of components in the change set, and the read set, i.e., the set of component properties made use of in a change.

The same information is necessary to determine whether two change sets are in conflict. Potential conflicts can occur when developers propagate a logical change to another existing configuration, e.g., when merging a change from one development path into another, or when developers define new configurations by combining change sets, e.g., when pursuing a strategy of integrating a number of changes to a large system into one release. In either case, the developer effectively merges change sets from two branches of the version graph. Pragmatically, the CM system will determine which components have been modified in either branch. If conservative, the CM system invokes a merge tool on all modified components, highlighting the changes, and requires the developer to interactively confirm the safe inclusion of the modification or interactively correct the conflict. A more optimistic CM system invokes an interactive merge tool only on those components that are part of both change sets.

Change sets are related to long transactions in that they are the persistent record of preserved and committed changes to configurations that are performed in the context of a long transaction. Defining new configurations by adding change sets or combining change sets can be interpreted as establishing new transaction schedules. The consistency of the configuration corresponds to the validity of the schedule. unlike database transactions, developers can be asked to interactive resolve some of the conflicts.

5.2. The Change Set Model in Use

In this section we focus on CM systems supporting the change set model as their primary model. Such CM systems typically complement this model with the checkout/checkin model for components to provide concurrency control. The section focuses on the creation of change sets, the granularity of changes, promotion of changes, support for system families, and distributed concurrent changes.

5.2.1. Change Set Creation

Change sets can be viewed as the result of a long transaction. However, CM systems supporting change sets typically do not fully support long transactions. Instead, a change set is created as part of a working context. The CM system establishes a change set name and logs modifications preserved through component checkin as part of the change set. Some CM systems allow modifications to be added to a change set at any time. In this case, conventions must be followed by developers to keep a logical change stable, once it is completed. The full semantics of long transactions with atomic commitment of changes, scopes of visibility through nested transactions, and configuration level concurrency control have to be emulated through conventions.

5.2.2. Granularity of Changes

Change sets interact with the system structure. Change sets can be recorded for individual components, for configurations of subsystems, and for complete system configurations. After change sets have been recorded at a certain granularity of the system structure, change sets at a larger granule can be defined as sets of existing change sets on the lower granule. If the changes were made in different subsystems and interfaces between them are well established, such cumulation of change sets will encounter few, if any, conflicts. This scenario describes planned concurrent change to systems through partitioning. If strict partitioning is not appropriate, developers can fall back on the conflict detection and resolution capabilities when combining change sets.

Grouping of change sets into named units is useful, if separately carried out changes are later recognized as being interrelated, and this fact should be recorded. After-the-fact splitting of change sets into smaller ones is not supported directly. The effect of such a split can be recreated, though, by composing change sets from component change sets, i.e., from component deltas if they are accessible to the user as named entities.

5.2.3. Promotion of Changes

The change set model supports evolution in that a change set reflects the logical changes as a system configuration evolves into a new version. Change set concept itself does not provide scopes of visibility and coordination of concurrent change. Scopes of visibility of changes, however, can be modeled through appropriate use of branches in the version graph of a configuration in a manner similar to that illustrated in Figure 2-3. Each branch represents a different state of maturity. Access control restrictions on the branches control their visibility. The state of a configuration is promoted by adding the change set to the configuration in the respective branch.

5.2.4. Support for System Families

The change set concept supports product families in the following way. Members of a system family, i.e., variants, are typically represented as branches in the configuration version graph and the dual change set version graph. Thus, variants can be be chosen by selecting from configurations or change sets in a manner similar to that of the composition model. Currently available CM systems offer a simple change set concept without the ability to attach attributes and to perform selection based on selection rules. Such a capability would have to be emulated through choice of appropriate naming conventions.

The propagation support of the change set model can be used to manage the evolution of a system family in a flexible way. Changes are initially made to one product variant or in one development path, and are recorded in change sets under one configuration version branch. These changes can then be independently propagated to other variants or development paths by adding the appropriate change sets to the respective version branches of the product configuration. Restrictions on initiation of new changes and on the propagation flow result in a more coordinated evolution.

5.2.5. Distributed Concurrent Change

The change set model itself does not provide locking mechanisms to control concurrency. Therefore, CM systems supporting change sets also support the checkout/checkin model for that purpose. However, change sets can be used to emulate some variants of concurrency control. Restrictions can be placed on the propagation of change sets that are records of concurrent logical changes, represented as separate branches in the change set version graph. For example, an optimistic concurrency scheme is emulated by restricting that a change set can only be migrated to a branch representing a parent transaction if all change sets in the parent have been included in the child branch.

Change sets can also be used to support distributed concurrent change without centralized coordination. Each site generates change sets independently. Once the change sets are exchanged between sites, each site can, at its leisure, combine change sets. The result is that the system evolves at both sites. If assignment of changes to sites is planned carefully, conflicts in change sets can be kept at a minimum.

5.3. Summary of the Change Set Model

The change set model supports change-oriented configuration management in a natural way. It provides a link to change requests, which are a management tool for controlling the software process through change authorization and status tracking. It allows developers to view configurations in terms of collections of logical changes. Logical changes are propagated by adding change sets to appropriate configurations. Different integration strategies can be pursued for collections of concurrent logical changes by combining them in particular order. Appropriate queries can determine the degree to which logical changes have spread throughout a collection of system configurations being maintained.

Although change sets relate to transactions, they do not provide concurrency control. Thus, CM systems complement the change set model with the checkout/checkin model. Merge mechanisms necessary to support optimistic concurrency control are also applicable to determining and resolving change set conflicts.

Current CM systems either provide limited support for logical changes through naming of component and configuration versions, or offer support for the change set concept, but place little restriction on consistent combination of change sets into configurations. However, many practical development scenarios have change propagation patterns that result in few change conflicts.

6. Summary and Conclusions

This report has focused on a discussion of four CM models that can be observed in commercial systems. Each of the models focuses on a particular aspect of version and configuration management. One of the models is the well-known checkout/checkin model. It provides versioning of individual components and concurrency control on components through locking as well as through branching and merging. The other three models reflect advances in CM functionality over this basic model. The composition model focuses on the creation of configurations in a two-step process—the description of the composition structure of a system in terms of its components, and the selection of an appropriate version for each of the components. In this model, the link to system build capabilities is most prevalent. The long transaction model focuses on the evolution of a system as a series of configuration versions and the coordination of development teams changing a system simultaneously. In contrast to the composition model, in this model developers deal primarily with versions of configurations. Once selected, the appropriate component versions are implicitly inferred from the configuration. This model emphasizes support for developers actively making modifications by managing their workspaces. The change set model focuses on the evolution of a system in terms of logical changes. This change-oriented view of configuration management emphasizes support for managing the propagation of logical changes throughout a system family. Change sets represent the actual modifications making up a logical change. They provide the link to managing the change process through change requests.

The four CM models have been derived from examining a number of commercial systems providing CM functionality, be it CM tools, multi-user CASE tools, or environment frameworks with CM capabilities. Figure 6-1 characterizes a number of CM systems based on the four models. The systems shown in the table are UNIX SCCS and RCS, Apollo DSEE, Sun NSE, Software Maintenance & Development Systems Aide-De-Camp, Softool CCC, CaseWare Amplify Control, Rational Environment, Procase SmartSystem, and Interactive Development Environments Software Through Pictures. (Aide-De-Camp is labeled as ADC and Software Through Pictures is shown as StP.) Several systems offer turn-key systems, i.e., versions of the system tailored to support a particular software process. The purpose of the table is to illustrate the spectrum of services offered in different systems. As can be seen from the table, a particular system typically focuses on one CM model as its primary model, possibly complementing it with a second model. The discussions in this report on the use of the CM model in different software process scenarios have shown that the CM support of a particular system matches certain scenarios well, while others can be supported with limitations by following certain usage conventions. A single CM system may have difficulties meeting all needs throughout the software process.

System	CM model	Comments
SCCS	checkout/checkin (co/ci)	
RCS	co/ci	configurations as labels
DSEE	composition + co/ci	logical change record in task log
NSE	transaction	multi-level transactions with optimistic concurrency
ADC	change set + co/ci	turnkey systems
CCC	composition + co/ci	labels for logical changes; turnkey systems
Amplify	composition + co/ci	turnkey systems
Rational	composition + transaction	two-level transactions with component locking; composition of subsystems
SmartSystem	transaction	one-level transactions on single development path; locking & optimistic schemes
StP	composition + co/ci	no data dictionary versioning

Figure 6-1: CM Models in Systems

The four models discussed here are also in various stages of investigation by the research community. [25] [26] [27] Much of the current research in support for configuration management tends to focus on a particular model and the refinement of its concepts. Given this state of support for software configuration management, several concluding observations can be made.

First, CASE tools, environment frameworks, and CM tools are being integrated into project support environments. Their integration raises several issues specific to the integration with CM. The CM concepts in different systems to be integrated may differ. In such a case, a way must be found for the particular models to interoperate or for some systems to use an externally provided CM facility in favor of their own. Similarly, many tools provide their own data management facilities ranging from separate files for components to data dictionaries and databases containing semantic information for whole systems. These data management facilities must be integrated with the data management facilities of the CM system. Options include exporting data from the tool through an external representation into a CM repository, the tool maintaining its data directly in the CM repository (especially if transparent access is supported), and configurations being maintained by both the tool and the CM system for different aspects of the software process. For example, a CASE tool may provide the support for a team actively making changes in a transaction model. The result of the teamwork is then committed to the repository supported by a CM tool. Its model may emphasize support for composition or management of logical changes.

Second, CM systems cannot be viewed as independent of the software process to be supported. The concepts offered by a CM system impact the software processes to be supported by imposing a particular conceptual model of CM. At the same time, the services offered by a CM system are a basis for tailoring its support to specific software processes. Several CM systems offer capabilities for tailoring, resulting in turn-key systems, and typically offer instances of those to their customers. Software process information can be embedded in a system and supported by it in a number of ways, ranging from structuring the repository to guidance through process steps and event/trigger-based primitives for process automation. Since software processes differ from organization to organization, even from project to project, encoding of process information must be flexible, must support variations in the process, and must allow adaption with limited effort. In its general form, software process modeling is an active research area.

Third, there is a need for evolving to a stable core set of CM concepts that addresses the needs of developers in such a way that it can accomodate changes in implementation technology and offer flexibility to support different CM policies. In the former case, object management system (OMS) technology is emerging and starting to mature. The impact of that technology on the core set of CM concepts should be minimal. In the latter case, the core set of concepts should not include policy decisions that unnecessarily restrict software processes. For example, as the discussion of concurrency control in this report indicates, it is desirable to support several concurrency control schemes, each utilized in a different part of the software process. Similarly, the access control primitives of a CM system should allow different authorization schemes to be supported and enforced.

Finally, there is a need for a unified CM model that provides a framework for configuration management support. This unified model should be a multi-paradigm model that supports a several CM concepts cooperating in harmony. This model can become a framework for adaptation to a range of software processes. Katz [16] has analyzed concepts in support of version and configuration management in computer-aided design (CAD) databases. Several concepts have emerged as desirable core concepts. Our analysis of the four CM models in commercial systems supporting software development complements this work. The set of observed concepts has been expanded. These are first steps toward a unified model, which is a major challenge. The appropriateness of the concepts as a basis for a unified model has to be validated, the interaction between the concepts has to be understood, and the ability to combine and adapt concepts to process needs has to be possible.

Acknowledgements

The findings presented in this report have evolved over time and have benefited from many discussions with colleagues, particularly Susan Dart, Grace Downey, and Kurt Wallnau at the Software Engineering Institute (SEI). I would also like to thank Scott Kramer and Ian Thomas from Hewlett-Packard, Gail Kaiser from Columbia University, Leigh Power from the Microelectronic and Computer Technology Corporation (MCC), and several colleagues at the SEI for their constructive comments on earlier drafts of the report.

References

[1] Barghouti, Naser S. and Kaiser, Gail E.
Concurrency Control in Advances Database Applications.
ACM Computing Surveys , 1991.

[2] Belady, L. A. and Merlin, P. M.
Evolving Parts and Relations - A Model of System Families.
In *Program Evolution*, pages 221-236. Academic Press, 1985.

[3] Softool.
CCC: Change and Configuration Control Environment. A Functional Overview
1987.

[4] Courington, W.
The Network Software Environment.
Technical Report Sun FE197-0, Sun Microsystems Inc., February 1989.

[5] Dart, Susan A.
Concepts in Configuration Management Systems.
In *Third International Software Configuration Management Workshop*. ACM Press, June 1991.
also available as CMU/SEI-90-TR-11.

[6] Estublier, Jacky.
A Configuration Manager: The Adele Data Base of Programs.
In *Proceedings of the Workshop on Software Engineering Environments for Programming-in-the-Large*, pages 140-147. June 1985.

[7] Feiler, Peter H., Dart, Susan A., and Downey, Grace.
Evaluation of the Rational Environment.
Technical Report CMU/SEI-88-TR-15, ADA198934, Software Engineering Institute, Carnegie Mellon University, July 1988.

[8] Feiler, Peter H. and Downey, Grace F.
Transaction-Oriented Configuration Management: A Case Study.
Technical Report CMU/SEI-90-TR-23, Software Engineering Institute, November 1990.

[9] Feldman, S. I .
Make—A Program for Maintaining Computer Programs.
Software—Practice & Experience 9(4):255-265, April 1979.

[10] Forte, Gene.
Configuration Management Survey.
CASE outlook 90(2), 1990.

[11] Harter, Richard.
Version Management and Change Control; Systematic Approaches to Keeping Track of Source Code and Support Files.
Unix World 6(6), June 1989.

[12] Humphrey, Watts S.
Characterizing the Software Process.
IEEE Software 5(2):73-79, March 1988.

[13] IEEE/ANSI.
IEEE Guide to Software Configuration Management
IEEE Press, 1987.
IEEE/ANSI Standard 1042-1987.

[14] IEEE/ANSI.
IEEE Standard for Software Configuration Management Plans
IEEE Press, 1990.
IEEE/ANSI Standard 828-1990.

[15] Kaiser, Gail E.
A Flexible Transaction Model for Software Engineering.
In *Proceedings of the Sixth International Conference on Data Engineering*, pages 560-567. February 1990.

[16] Katz, Randy H.
Toward a Unified Framework for Version Modeling in Engineering Databases.
ACM Computing Surveys 22(4), December 1990.

[17] Leblang, David B. and Chase, Robert P., Jr.
Computer-Aided Software Engineering in a Distributed Workstation Environment.
In *Proceedings of SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 104-112. Pittsburgh, PA, April 1984.
Published as *SIGPLAN Notices*, Vol. 19 No. 5, May, 1984.

[18] Leblang, David B. and McLean, Gordon D., Jr.
Configuration Management for Large-Scale Software Development Efforts.
In *GTE Workshop on Software Engineering Environments for Programming in the Large*, pages 122-127. June 1985.

[19] Lie, Anund, Conradi, Reidar, Didriksen, Tor M., Karlsson, Even-Andre.
Change Oriented Versioning in a Software Engineering Database.
In *Proceedings of the 2nd International Workshop on Software Configuration Management*, pages 56-65. ACM Press, November 1989.
Software Engineering Notes, Vol. 17, No. 7.

[20] Morgan, Thomas M.
Configuration Management and Version Control in the Rational Programming Environment.
In *Proceedings of the Ada-Europe International Conference*, pages 18-28. Cambridge University Press, June 1988.

[21] Ploedereder, E. and Fergany, A.
A Configuration Management Assistant.
In *Proceedings of the Second International Workshop on Software Version and Configuration Control*, pages 5-14. ACM, USA, October 1989.

[22] Prieto-Diaz, Ruben and Neighbors, James M.
Module Interconnection Languages.
The Journal of Systems and Software 6:307-334, 1986.

[23] Reps, Thomas and Bricker, Thomas.
Illustrating Interference in Interfering Versions of Programs.
In *Proceedings of the 2nd International Workshop on Software Configuration Management*, pages 46-55. ACM Press, November 1989.
Software Engineering Notes, Vol. 17, No. 7.

[24] Rochkind, M. J.
The Source Code Control System.
IEEE Transactions on Software Engineering SE-1:364-370, 1975.

[25] German Chapter ACM, GI, Siemens AG.
Proceedings of the International Workshop on Software Version and Configuration Control, Teubner Verlag, Grassau, W-Germany, January 1988.

[26] ACM SIGSOFT, IEEE CS, GI.
Proceedings of the 2nd International Workshop on Software Configuration Management, ACM Software Engineering Notes, Princeton, NJ, November 1989.

[27] ACM SIGSOFT, IEEE CS, GI.
Proceedings of the 3rd International Workshop on Software Configuration Management, ACM Press, Trondheim, Norway, June 1991.

[28] Tichy, Walter F.
A Data Model for Programming Support Environments and Its Application.
Automated Tools for Information Systems Design.
North-Holland Publishing Company, 1982, pages 31-48.

[29] Tichy, Walter F.
RCS—A System for Version Control.
Software—Practice and Experience 15(7):637-654, July 1985.

[30] Westfechtel, Bernhard.
Structure-Oriented Merging of Revisions of Software Documents.
In *Third International Software Configuration Management Workshop*. ACM Press, June 1991.

Table of Contents

List of Figures

View publication stats