Transaction-Oriented Configuration Management:

A Case Study

Peter Feiler
Grace Downey

November 1990

Transaction-Oriented Configuration Management: A Case Study

Peter Feiler
Grace Downey

Software Development Environments Project

This report was prepared for the SEI Joint Program Office HQ ESC/AXS

5 Eglin Street

Hanscom AFB, MA 01731-2116

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

FOR THE COMMANDER

(signature on file)

Thomas R. Miller, Lt Col, USAF, SEI Joint Program Office

This work is sponsored by the U.S. Department of Defense.

Copyright 1990 by Carnegie Mellon University.

Table of Contents

List of Figures

Transaction-Oriented Configuration Management: A Case Study

Abstract: Software configuration management (SCM) is a key element of the software development process. A number of new configuration management techniques in commercial SCM tools and environments with SCM capabilities have been observed. This report illustrates some of the advances in SCM concepts by example of a particular commercial system: the Sun Network Software Environment (NSE). NSE embodies a transaction model of configuration management. In order to demonstrate the capabilities and limitations of the transaction model, NSE is applied to three problem areas for configuration management: adaptation for parallel development and team support, development and maintenance in software families and development in a distributed and heterogeneous network.

1. Introduction

Configuration management is an integral part of the software development process. SCM tools and capabilities in a software development environment or environment framework automate certain elements of the software process. The functions provided by an environment may be low-level. To be useful to developers a set of conventions or procedures must reflect a particular desired SCM method. A number of commercial systems have become available offering higher-level SCM functions such as Rational's Ada Development Environment [2], Apollo's Domain Software Engineering Environment [3], BiiN SMS [6], and Software Maintenance and Development Systems' Aide-De-Camp [1]. On one hand, their functions are closer to the desired support of a particular SCM method, by automating the conventions and procedures to be adhered to. On the other hand, the functions may restrict the range of development processes which can be supported.

The report is organized as follows. Section 2 defines the scope of SCM as a developer support function and states common practice. It concludes with a description of SCM advances, including a description of the transaction model to control software evolution. Section 3 provides the properties and functionality of NSE that are important to SCM support. Sun's NSE is described in this report because it embodies several advances that can be observed in a number of recently released commercial systems. One of the advances is NSE's use of a long-term transaction model as its primary SCM concept. We are not critiquing the implementation of the model as found in Version 1.2 of NSE, but using NSE as a vehicle to probe the transaction model and other advances. We explore how this advance impacts three common development situations and illustrate the benefits and limitations of the transaction model. Section 4 describes how the effort of individual team members is coordinated. Section 5 explores different ways to support the development of software products that share portions of their source code. Section 6 describes how to manage software development taking place in a heterogeneous network of workstations. The report

concludes with a summary of the benefits and limitations seen in the SCM advances found in NSE. There is also a glossary of terms provided as Appendix A. The glossary defines software configuration management terms and describes the function of some NSE specific commands.

2. Software Configuration Management

The term software configuration management is subject to a wide range of interpretation. In order to reduce misunderstandings, we first define SCM and describe the particular focus we have taken in this report. Then, we summarize common practice in SCM developer support. Section 2.3 describes new configuration management techniques found in commercially available environments and frameworks.

2.1. Definition and Focus

Software configuration management can be defined as the discipline of controlling the evolution of software. As illustrated in Figure 2-1, SCM is one of the control disciplines complementing management disciplines and development disciplines in producing and maintaining quality software products. It is one of the key elements of the software process. SCM provides stability and consistency by tracking and recording all changes, by uniquely identifying versions and configurations, and by managing change. SCM affects the other disciplines as it provides the necessary stable context in which to achieve valid measurements and results.

Management Disciplines

Controlling Disciplines

- Software Quality Assurance
- Software Configuration Management
- Independent Verification and Validation

Product Integrity

Development Disciplines

- Specification
- Design
- Code
- Test
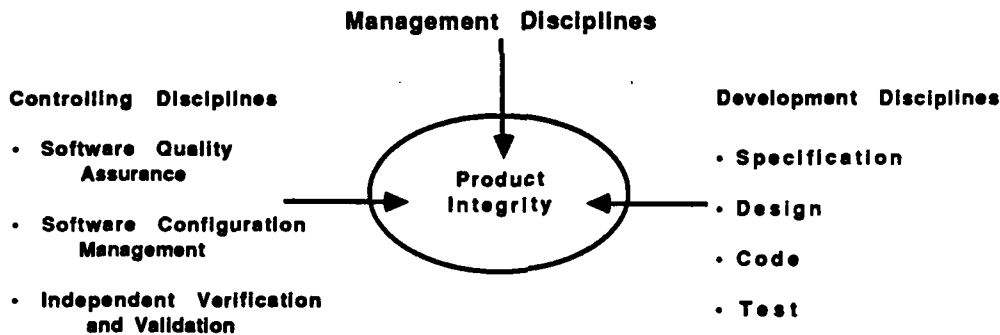
Figure 2-1: The Controlling Disciplines

Software configuration management has a number of facets; some emphasize the management aspect of software development, while others focus on technical aspects. Examples of SCM facets with management emphasis are:

- Change management: initiation, evaluation, and approval of change through mechanisms of change requests and change control boards.
- Release management: identification and packaging of releases, tracking of customer installations, relating problem reports and fixes to releases.
- Corporate product management: identification and tracking of product spectrum; impact analysis of new or upgraded products and components on spectrum.

- Contract development management: identification and tracking of deliverables in contracted software development.
- Acquisition management: acquisition of off-the-shelf products and their upgrades from multiple vendors.

The technical aspects of SCM include:

- Developer support: storage of and access to actual product components, support for product composition and manufacturing, coordination of concurrent change activity.
- Customer adaptation: adaptation to computing environment at customer site, and tailoring organizational and project needs and individual preferences; management of product configuration refinements.
- Installation management: management of installed product configurations including vendor upgrades, co-residence of multiple versions, multiplicity of installations, and roll-back to old configurations.
- Dynamic reconfiguration: management of incremental update to system configurations, especially in non-stop systems.

In this report we focus on the developer-support facet of SCM for two reasons. First, development and maintenance of software artifacts are performed by a large part of project personnel. However, SCM is often perceived by developers as intrusive and practiced as an afterthought. As a management-oriented control discipline it places additional burden on developers with little apparent benefit. This intrusion may be due to a rigid SCM method that does not adapt to desirable variations in control as more of the developers' activities are included. It may also be due to inappropriate tool support by offering either inadequate functionality to support the SCM method or insufficient performance for frequently executed operations that intuitively seem simple. Thus, tool support for SCM in this area can potentially improve productivity of developers, leading us to the second reason for our focus. In the last several years many commercial software development environments and SCM tools have become available with new services in SCM developer support. These new services have made advances in better supporting software development and reducing the intrusion of SCM tools and the reluctance of developers to use them in their day-to-day work.

2.2. Common Practice in SCM Support

In order to discuss the advances found in SCM support, it is first necessary to outline the current practice. Common practice in SCM support provides three types of tools. The first is a record keeping tool to track information about the product under configuration management. It often is a database application, that encodes a well-defined SCM process and method, which previously was enacted on paper. Identification, characteristics, and development history of product artifacts, as well as artifacts of the change process such as problem reports, change requests, and their relationships to the product are recorded on forms and stored in the database. Report generation capabilities are used to provide project status and accounting information.

The second type of tool is a software artifact filing capability, referred to as software repository or software library. This is the place where the actual artifacts (in their online form) are kept. Typically, its functions are modeled after those of a librarian—the gatekeeper of project products. The term *development repository* reflects the more passive role as an information sink. Product artifacts are created and modified outside the realm of this tool. When artifacts are deemed to have reached a certain state of maturity (e.g., handing off to quality assurance or the customer), they are passed into the repository for preservation and record keeping. Emphasis is on enforcement of control and authorization of change. Software repositories are typically used in conjunction with record keeping tools described above. As a result, retrieval of artifacts is performed relatively infrequently—usually for the purpose of preparing a release or for establishing a baseline for further or new development.

Another aspect of the second type of tool is the software library. A library plays a more interactive role in software development. It is intended to be the primary source of artifacts to be evolved (i.e., retrieval of artifacts is a common operation). Tools in the role of software library often are limited to managing "source code" artifacts, restricting the artifacts to be text files. Control tends to be limited to coordination functions through a check-out/check-in protocol of individual artifacts. A historical record of artifacts is maintained in the form of sequential revisions and through branches in the revision graph. These branches can, by convention, reflect variants of the artifact, separate development activity of different artifact releases, or concurrent change. Similar to the repository, the library has a flat name space, that is, product structures have to be recorded through naming conventions or through auxiliary record keeping. Because artifacts must be explicitly retrieved before they can be accessed and retrieval can take a considerable amount of time, developers tend to maintain copies of the artifacts in their work area—introducing potential inconsistencies, as it is the developers' responsibility to keep the work area up-to-date with the library.

The third type of tool is a facility for automating the manufacture of software (i.e., system build). System build automates the derivation of artifacts such as machine code from others by application of tools such as compilers. It does so based on a description of the components of a system, of the system structure based on these components, and of dependencies between components. Language-independent system build tools tend to limit their dependency records to cases where one component includes the contents of another entirely before processing (e.g., "include" files in C). Typically, existence of files and file dates are used to determine whether derived artifacts need to be generated. To generate a derived artifact, predefined or user-specified build rules (i.e., tool invocation templates) are elaborated and executed.

In common practice, the three types of tools: record keeper, repository (or library) and build, are usually implemented as three independent services. For example, system build tools have little awareness of a software repository, that is, they assume that all artifacts are available in the file system. The following subsection focuses on changes in repository or library support that represent advances in software development support.

2.3. Advances

The software configuration management systems found in new software development environments are supporting more advanced concepts and developer interaction. Repositories can capture more than a collection of files that are processed into a software system. Repositories are becoming a place that provides context in which to process source rather than a library of source versions. SCM systems are also supporting methods to evolve a software configuration from one version to the next. Sections 2.3.1 through 2.3.3 explore each of these changes in support in more detail.

2.3.1. Configurations as Managed Objects

In a traditional repository model for source code, such as the Revision Control System (RCS) or Source Code Control System (SCCS), versions of individual source code files are managed by numbering them and placing them in an SCCS or RCS subdirectory. Each directory in the UNIX directory hierarchy containing the source has its own RCS or SCCS subdirectory. Figure 2-2 depicts the typical scheme for structuring a directory hierarchy when using SCCS. Directory names are shown with a leading "/" and underlined. Directories may contain files and other directories. Figure 2-2 shows files contained in a directory by listing them beneath the directory name. Directories are shown to contain other directories by drawing an arc to the contained directory's name. This depicted solution is limiting, in that it is unable to capture a complete version of a configuration from one place in the event of a system release. A configuration is the collection of all the sources that are compiled into one consistent executable. The system structure may be recorded in a descriptive file, such as a *Makefile*. However, the descriptive file will only be treated as any other artifact. The repository, in this case a collection of SCCS subdirectories, has no knowledge of the system structure.

An advance in newer SCM systems is the ability to represent system structure. The SCM system captures a collection of files as a composite. The configuration is explicitly supported as an object that can have a series of versions. A given version of a configuration may contain source objects, derived objects, files containing build rules, documentation sources, and a directory hierarchy. By capturing all the software artifacts as they exist in the directory hierarchy the system structure is represented.

Another advantage to capturing more than source files, is that derived objects now come under the control of the configuration management system. Configurations provide a context in which individual files are modified, and derived objects are produced and stored in this context.

2.3.2. Transparent Access to Repository

When an SCM system is based upon a software repository, the repository serves as a place to store versions of the source code of the system. In this case a developer executes specific commands to preserve artifacts in the repository. When an SCM system is based on a library, the library becomes the source of objects to be changed. The developer executes

```
                        /product
                        Makefile    .
                        Ourprogram _____
                                                      ___  /sccs
                                                           s.Makeflle

    /subsys1                     /subsys2
      Makeflle                     Makeflle
      bar.c      /sccs             back.c     /sccs
      bar.h        s.Makeflle      back.h       s.Makeflle
      bar.o        s.bar.c         back.o       s.back.c
      baz.c        s.bar.h         front.c      s.back.h
      baz.o        s.baz.c         front.o      s.front.c
      foo.c        s.foo.c         In.c         s.In.c
      foo.h        s.foo.h         In.o         s.out.c
      foo.o                        out.c
      subsys1.a                    out.o
                                   subsys2.a
```
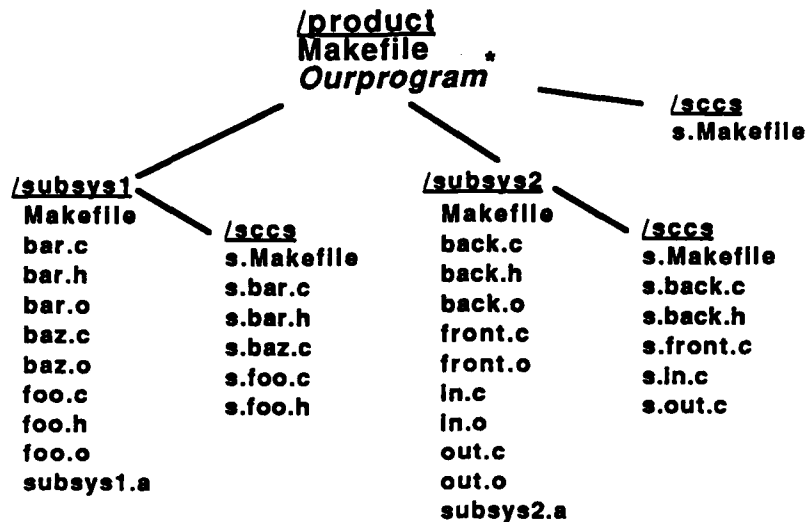
Figure 2-2: Typical Directory Hierarchy Using SCCS

specific commands to populate his workspace with versions of objects which will be changed. Once an object is changed it is returned to the library, where it is again available to others. An SCM advance is the ability to provide direct access to the contents of the repository itself. With this advance, a developer works in the repository changing files as they exist there. By mapping a configuration of artifacts into the file system transparently, the SCM system eliminates the need for moving artifacts in and out of the repository.

An additional advantage to this approach is that the repository can contain complete source context for the derivation of the software system. If one source file depends on the presence of another, both files will be available in the repository. A new version of a source object is created only when a user indicates intent to modify the object. The entire configuration need not be duplicated to supply the derivation context for a few changing files.

2.3.3. Transaction-Style Software Evolution

The transaction model of software evolution relies on the notion of a transaction as the basis for managing change. The transaction model found in the context of configuration management has similarities to the database transaction model. A configuration of software files resides in a repository which is analogous to a database. A transaction begins with an indication that objects in the repository are to be modified, and are locked to prevent concurrent modification. Changes are made to the files, the repository is updated and the files are unlocked to allow subsequent modification. There are two differences in database transactions and SCM transactions. First, a database transaction performs an update to stored data, while an SCM transaction installs a version of data in the repository. Second, the duration of a transaction as used in SCM is much longer than a transaction used to update a database.

It is possible to have parallel transactions in the SCM transaction model. A repository contains an original baseline of the software artifacts. If there is no locking to prevent two or more transactions from acquiring the same objects from the repository, then several simultaneous transactions may be created. Changes in each of the parallel transactions will be isolated from each other and each transaction can contain enough context to fully process the changes. If parallel transactions are permitted, then there is a mechanism to merge changes that could occur on the same object. The first completed transaction returns its version of the changed objects to the repository. When a parallel transaction attempts to return the same objects to the repository, it fails. It must update to the latest version of objects from the repository, and merge and test its changes to the latest version of objects. In order to prevent the race condition, where a transaction can never complete because the repository changes too quickly, there is a mechanism to lock the repository from any further change until the given transaction can return its changes.

An additional aspect of the transaction model is support for nested transactions. A transaction starts by acquiring objects from the repository. In turn, a sub-transaction may be created by acquiring objects from the initial transaction. The sub-transaction will return its modified objects to its parent (i.e., its enclosing transaction). When a sub-transaction returns its modified objects to its parent, the scope of visibility for those objects widens. They become accessible by any parallel sub-transactions of the parent. An SCM system that supports nested transactions maintains the relationships between the transactions. A given transaction can be queried to determine its parent and any child transactions it has spawned. A transaction may migrate its changes only to its parent or its children.

A final feature of the transaction model is the ability to take snapshots of the contents of a transaction at any time. The snapshot becomes an immutable and retrievable version of the configuration of objects. This creates a linear history of the objects contained in a transaction. It allows a developer to preserve a version of the configuration without having to change its scope of visibility by returning the objects to the parent transaction. When a series of changes is returned to the enclosing transaction, either the series may be individually stored for retrieval or the series may be compressed and stored as one atomic change.

In a pure transaction model for SCM, the repository is implemented as a non-terminating transaction. It captures revisions of the configuration it contains through the local history mechanism. A new revision is created in the non-terminating transaction or repository by each of its completed sub-transactions. In this manner a linear history of the configuration is maintained in the non-terminating transaction.

Transactions can be used to capture and represent software configuration evolution. Simultaneous change of a configuration can be afforded by allowing simultaneous transactions with a merge mechanism. The scope of visibility of a change can be controlled through a series of nested transactions. A linear history of the change in a configuration can be maintained through a snapshot capability local to one transaction. This combination of features represents a configuration management advance that has implications for how software may be controlled during its life cycle.

3. Network Software Environment

The Network Software Environment, commercially available and supported by Sun Microsystems, Inc., provides software evolution support for single developers and groups of developers. It is an example of a commercial system that exhibits and combines the advances explained in Section 2.3. The following sections describe how NSE Version 1.2 implements the advances and introduces NSE commands and objects.

3.1. The Roles of an NSE Environment

A key concept to the software configuration management that NSE provides is the *environment*. The environment serves as a repository, a workspace and it offers scoping and coordination of change through a transaction mechanism. As a repository it offers the ability to capture derived objects and the UNIX directory structure as well as source code. It also captures build and dependency information, and logical structure. Finally, an environment serves to capture versions of the entire configuration of objects.

3.1.1. Repository of Objects
NSE places an entire development directory tree and all the files it contains under configuration management control in an environment. Instead of capturing versions of all the individual source files independently, it captures the aggregate of all the files in the configuration.

In addition to maintaining the system structure present in the UNIX directory structure, NSE tracks the files that are used to build an executable. It also tracks the cases where one source file may depend on the presence of a separate source file. NSE uses *Makefile* information to distinguish between source files and derived files, and manages the relationship between them.

A third mechanism allows logical system structures to be expressed independently of the other system and build structures. Hierarchies are expressed through *components* and can contain components and other logical objects. The basic logical objects are *files* referring to UNIX files and *targets* referring to composition structures in *Makefiles*. For example, a compilation system consists of a compiler, debugger and linker; the source files for these may all reside in the same UNIX directory. However it is possible to group source files as members of components. The compiler, debugger and linker would each be represented as a component. The logical system structure is maintained in an environment.

An environment can capture the directory structure used to store the source and derived objects of a system, the build structure and the logical structure.

3.1.2. Recording the Development Path

Each environment also captures a series of "snapshots" of its contents: the source and derived objects, UNIX directory structure, build structure and logical structure. All the snapshots in the sequence can be named and are immutable, that is, they cannot be changed. Users make a particular version of the environment's contents accessible in the file system by accessing an environment and specifying a particular configuration version or NSE *revision*. By default, the last configuration version is made accessible, and its contents can be changed. Since a particular change often spans an entire series of files, it is viewed as an evolution of the configuration, rather than discrete changes to distinct source files. In Figure 3-1, the series of boxes can all be referenced through one NSE environment. A given box drawn around the UNIX directory structure represents one version of the configuration. A sequence of i`·ese boxes depicts a sequence of versions in a development path.

As a software configuration evolves, often source code files are renamed. NSE provides the ability to rename *files*. The rename command associates the previous history of the *file* with its new name. If the logical structure is revised during development, *environments* and *components* may also be renamed. The renaming capability provides additional continuity when a user wishes to examine the development path.

3.1.3. Transparently Accessible Configurations

The introduction of a new technology to the software development and maintenance process is often frustrated by the human tendency to resist change. An important characteristic of NSE is that it is transparent in the way in which a developer and tools interact with UNIX. To access a configuration, a developer *activates* an environment, and connects to the directory hierarchy using the *cd* command. The contents of the configuration will appear just as if it resided in the native UNIX file system. Thus an environment makes repository access transparent. The repository or NSE environment is not a separate area from the workspace, and objects are not moved in and out of the repository. Rather, a developer *activates* the environment and can carry out changes, re-compilation and testing directly "in" the repository.

Once a developer's workspace is set up, work can proceed as if NSE were not there. NSE intercepts commands to the UNIX file system, and directs them to the appropriate configuration of file versions. A developer does not have to learn an extensive set of commands or procedures to insure the selection of individual file versions which work together.

3.1.4. Workspace

While environments contain many objects and their relationships, environments in themselves are named entities that NSE tracks and manages through a scheme of nested transactions. The *nseenv create* command provides a non-terminating transaction. The *acquire* command is used to start a transaction to modify the contents of the non-terminating transaction. By allowing the creation of more than one sub-transaction, NSE allows a pattern of optimistic or parallel development. A team of developers can spawn any number of copies of a given environment. Each developer can work in a private copy and be isolated from
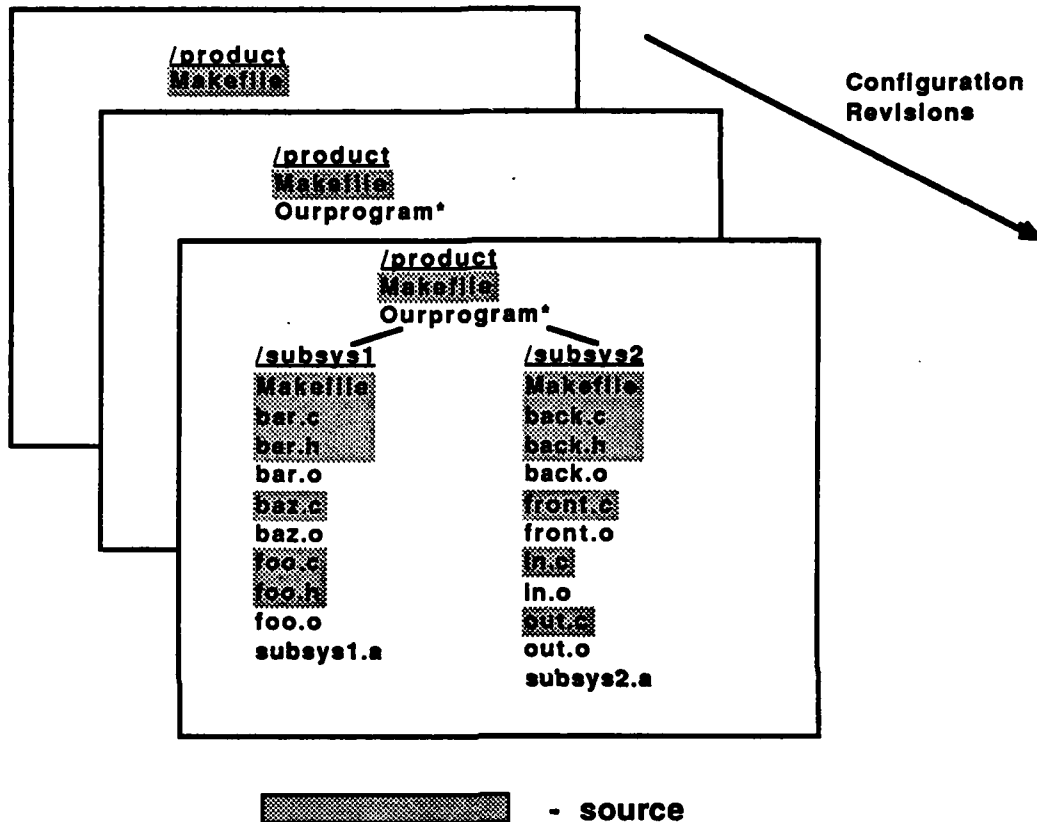
Figure 3-1: NSE Environment and Directory Hierarchy

changes made by others, and can choose when to integrate with others' changes. In this manner an environment provides a scope to the effect of changes. Each *child environment* or workspace retains the full range of commands for establishing a sequence of local revisions, and may even spawn child environments of itself. NSE maintains the relationships between the copies of environments, preventing uncontrolled change. It is always possible to determine the baseline to which a change was made by looking to an environment's *parent*. NSE also provides a paradigm supported by commands and interactive multiwindow merge tools to coordinate the diverse pieces of work from the child workspaces.

4. Support for Evolution by Team

The development and maintenance of software today is usually carried out by teams of programmers. The software configuration management advances as implemented in NSE have implications for the way a team of developers can control change in a system composed of many software artifacts. This section describes how the transaction model supports the creation of a developer's workspace, development paths and the coordination of parallel change.

4.1. Workspace Management

An NSE environment serves as a named, controlled workspace which can contain an entire UNIX directory structure populated with source and derived objects. This provides an advantage over development as often performed on UNIX, where a developer copies a set of the sources into an arbitrary UNIX directory to make upgrades and modifications. The developer may or may not merge his changes properly with others, or may even unwittingly overwrite correct sections of code.

A repository or initial environment is created by the *bootstrap* command. A developer can *activate* this original environment, and perform useful work there. However, it is more advantageous to use the NSE's transaction style mechanism for creating working copies of the initial environment. Through the *acquire* command a user may create a child or *sub-environment* of the original repository. The child environment contains a virtual copy of everything or user-selected portions of what appears in the original repository. The creation of the child environment locks the original environment from any further changes carried on within the original environment. The user can then *activate* the working copy, as a private workspace. Here the developer can access files, make and test changes, without affecting any other environment or workspace. A developer may *activate* or *exit* an environment at any time; when activated, the environment presents the latest version of all of its contents in the directory designated at the time the environment was created. The designated directory is called the *control point* of the environment. Upon exiting, the contents of the environment no longer appear under the control point.

An activated workspace may be shared by more than one developer simultaneously. Their work on source code is coordinated by the Version Control System (VCS), the NSE service that locks and serially versions each source file. When a file is listed in a workspace, it is only a virtual copy, made to appear as if actually there through NSE's Translucent File Service. The contents of the file may be browsed or used as input to a tool. It is only when a file is checked-out through VCS, that the user is provided with an actual copy for modification. VCS provides *vcs checkout* and *vcs checkin* commands similar in operation to RCS and SCCS. The *vcs checkout* command locks a file for write access by one user within the workspace. This VCS file lock is local to only the current activated workspace, and does not affect the file as found in parent, child or sibling workspaces. Upon *vcs checkin*, a local copy is frozen and stored within the workspace. *vcs checkin* also prompts the developer for

a comment about the change. Within the workspace, from that point onward, VCS is similar to RCS and SCCS in that it uses a delta mechanism to store changes between successive versions within a workspace. The delta mechanism has the limitation of only being useful in dealing with textual files such as source code. An additional similarity to RCS and SCCS is that a developer can list the comment log associated with a source file, and request previous versions of a checked-in file to become the current version.

4.2. Single and Multiple Development Paths

A workspace contains an entire configuration of source, derived objects and any additional files a developer considers relevant. A development project can capture the evolution of the configuration by taking a snapshot of the entire workspace with the *preserve* command. Through successive applications of the *preserve* command the project creates a sequence of revisions of the contents of a workspace. *Preserve* is available in two forms: named and unnamed. In a series of unnamed revisions, only the latest is available for retrieval. A named revision is not destroyed, and can still be accessed after subsequent revisions have been made. When *preserve* is invoked, the user chooses a name and supplies it as a command parameter. The choice of names is left entirely to the user, and may reflect a revision numbering scheme or a configuration characteristic. The sequence of revisions accessible through an environment defines a development path for the configuration. Figure 4-1 depicts a series of named revisions. The names for revisions in this example reflect a simple sequential numbering scheme.
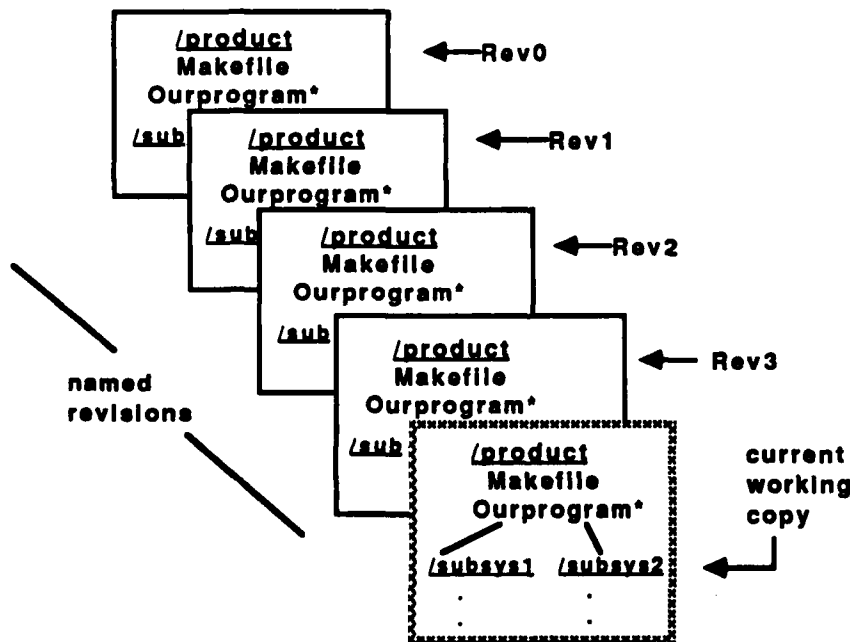
Figure 4-1: Development Path of a Configuration

It is often useful to maintain two versions of a system concurrently. For example, a field release should be available to upgrade for quick response to customer error reports, while in-house development continues on a more steady baseline. RCS supports branches in the line of development in an individual file while NSE supports branches of the configuration. At any time during development managed in an environment, a sub-environment may be created to carry on a different branch of development as shown in Figure 4-2.

The semantics of the NSE parent-child relation require that for change to occur in the parent, it must be *locked* while it has a child environment. The locking of the parent prevents a *reconcile* from the child environment overwriting changes made in the parent. When the error corrections which were made in the child environment are transferred to the major development branch in the parent, the environment must first be *unlocked* and the changes may ge transferred using the *reconcile* command.

Another option for providing parallel development paths is to create two children from the same parent environment, each to represent a separate branch of development. One child can carry on the implementation of fixes for error reports, the other child can carry on the business of planned upgrades.

Major Development Branch

Development1.0

Release1.0

Upgrade1.1

Upgrade1.2

Release2.0

Acquire

Reconcile & Merge

Bugfix1

Bugfix2

Error Correction Branch
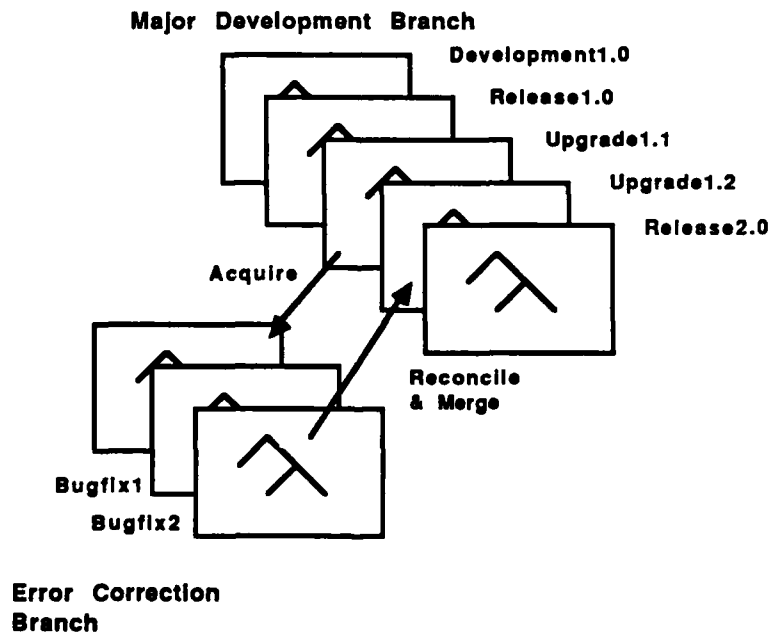
Figure 4-2: Development Subpath

Branches of the configuration may be created for the purpose of temporary concurrency. The information contained in the separate branches may be merged later using some of NSE's functionality. If two or more children were created the merge of changes from along the independent branches can occur at agreed to intervals by executing the *reconcile* command from each branch as depicted in Figure 4-3.
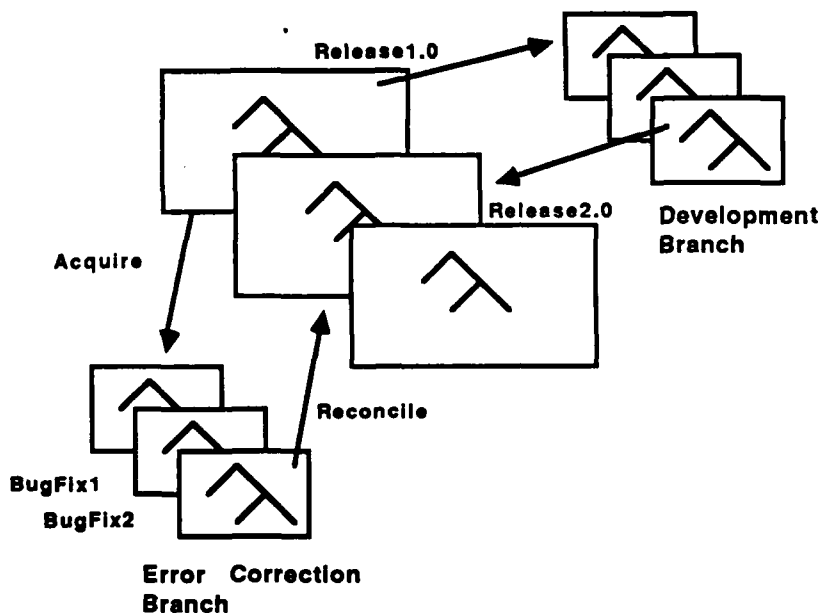
Figure 4-3: Development through Two Paths

It is also possible to achieve continued concurrency by never terminating the child environments as shown in Figures 4-2 and 4-3. However, the merge mechanisms may be used to move shared changes at desired intervals between the various branches of development.

4.3. Team Support

Once an environment has been created manually or through the use of the *bootstrap* command, child environments may be established for individual developers to work in through the *acquire* command (see Figure 4-4). The *acquire* command creates a duplicate, or a duplicate of part of the contents of an environment in a child environment. A parent environment may have more than one child. The child environments can serve as private workspaces for individual developers. A developer is able to place local changes back into the parent environment through the *reconcile* command. This action serves to make the changes immediately visible in the parent, and available to be moved to any other child environment of the parent. A developer checks if any code has changed in the parent environment with the *resync* command. If the same object in the parent environment and child environment has changed, then the *resolve* command can be invoked to aid in performing a merge of the changes.
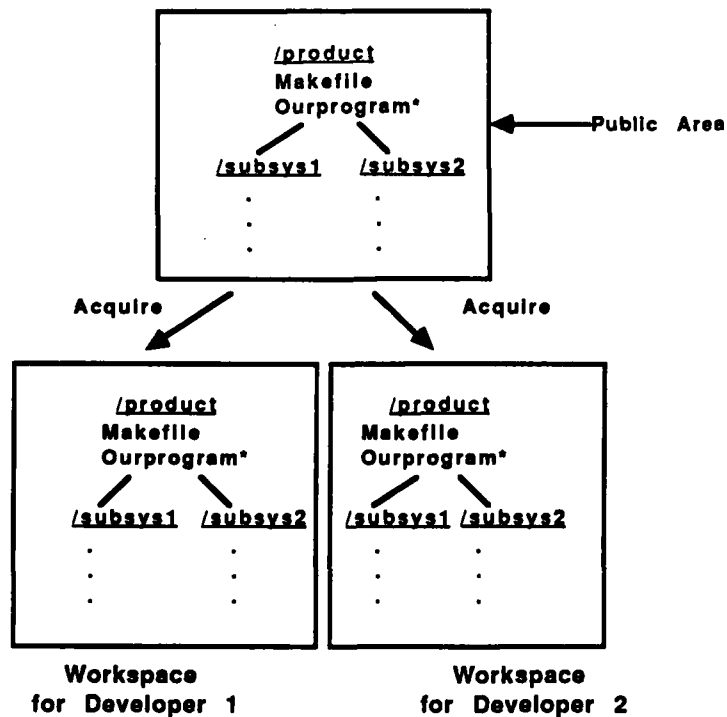
Figure 4-4: Workspaces in NSE

The private workspaces provided by the NSE environment allow for parallel development by a team of programmers through an optimistic change and merge scheme. A parent environment can be viewed as a central repository where all teamwork will eventually be collected. Each developer can have his own child environment, to make changes to and test, independent of the work of others. As a developer completes a set of changes, the changes can be moved to the parent environment with the *reconcile* command as shown in Figure 4-5. When a second developer finishes and issues the *reconcile* command, it will fail if both developers modified the same files. Figure 4-6 depicts how the second developer's work area is automatically updated with the files containing the first developer's changes, and local derived objects are marked as out of date. The second developer can use the *resolve* command to merge the conflicts between the later work, and the work submitted by the first developer. The merged code can be compiled and tested, and then submitted to the central repository through the *reconcile* command. Figure 4-7 depicts a merge of the first and second developers' changes and successful *reconcile*. This is referred to as optimistic development, as concurrency is not prevented through *a priori* locking in the parent or team environment. Child environments can be deleted as particular tasks are finished or remain as permanent workspaces that are updated with changes from the parent as needed.
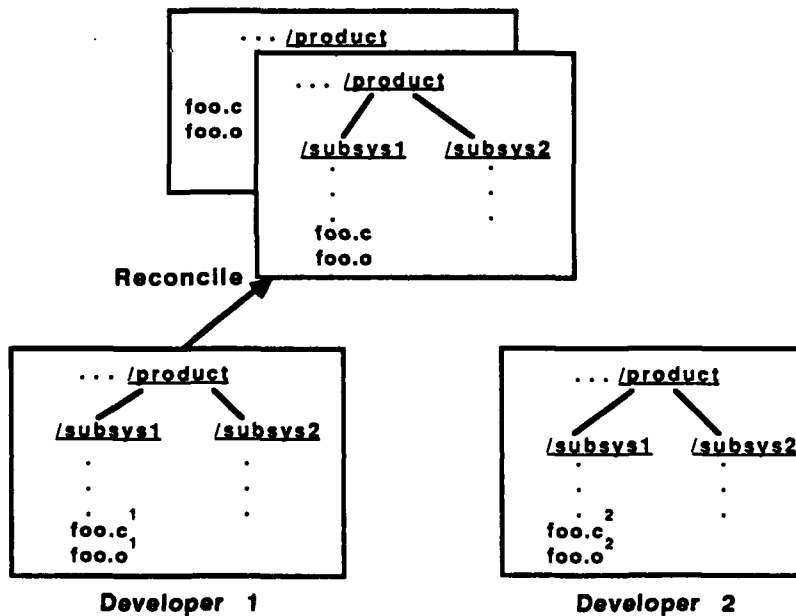
Figure 4-5: Release of a Developer's Changes
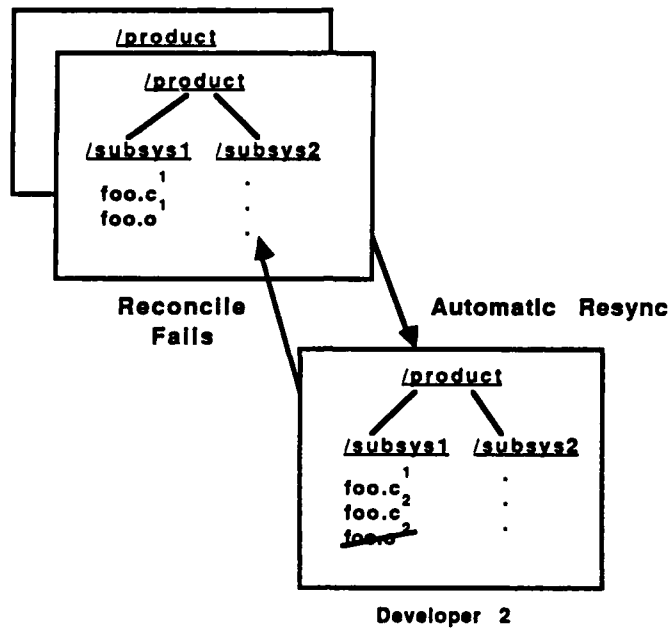
Figure 4-6: Failed Release of a Developer's Changes
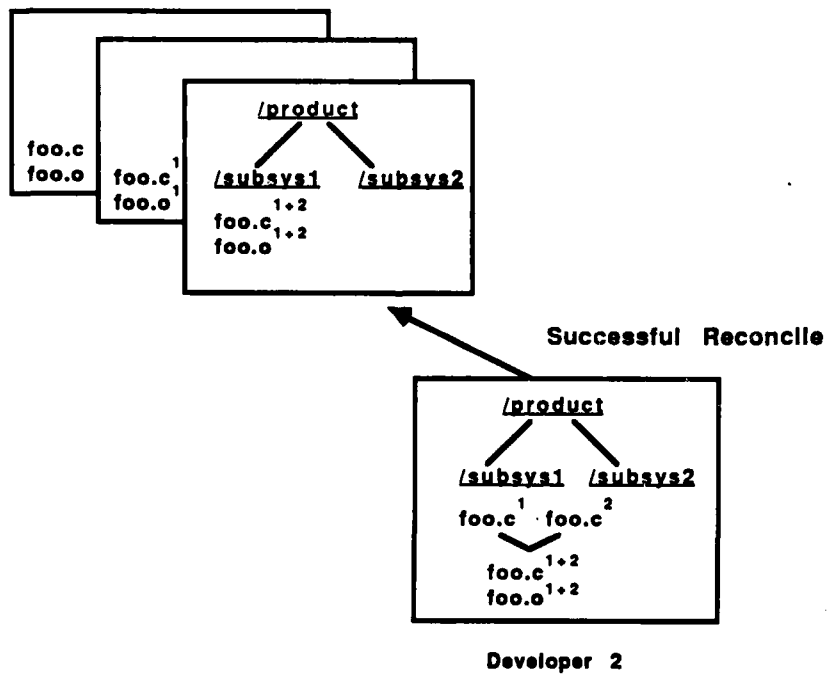
Figure 4-7: Merge and Release of Changes

5. Support for Evolution of Software Families

This section explores how well NSE supports the development and maintenance of a software family. The term *software family* refers to a collection of programs with essentially the same functionality. Members of the family may differ in alternative implementations of certain components or may represent adaptations to differences in the computing platform, such as different window systems, operating systems, or different hardware. The software may be maintained as a single set of source files, from which alternative executables are generated, or as alternative source files, from which a particular family member is produced by selection and composition.

The evolution of a software family is a challenging task. The family evolves as a system, i.e., new functionality may be added and improvements made, while old releases have to be maintained. Members of the family may evolve in a variety of patterns, and changes have to be propagated and merged. In practice, development patterns range from maintaining a family centrally from a single source, to development of a primary family member (e.g., for an initial target) followed by adaptation for other members (e.g., porting to other targets), and to independent development of different family members. In the former cases the change process is fairly restricted, while in the latter case decentralized initiation of change provides more autonomy, but requires stronger support for propagation and integration of change.

The NSE concept of environment can be used in different roles to support evolution in a software family. The commands which manage change and manipulate environments are based on the notion of a transaction. Particularly, the acquisition of a child environment, modification of code, and successful reconciliation of changes to a parent environment can be viewed as a transaction. Also, a transaction limits the scope of visibility of changes—isolating others from local changes and being isolated from others' changes. A transaction has local memory (i.e., revisions of configurations can be locally preserved in a linear history). The changes in a transaction are committed in an atomic operation to an enclosing transaction. Non-terminating transactions play the role of a repository. Once the work is completed it is committed from the workspace into an enclosing environment. Thus, the environment in the role of a workspace acts as a transaction. In this section, we examine how NSE's implementation of the transaction model lends itself to supporting the different development patterns encountered in the evolution of software families. We focus on the capabilities and limitations of the transaction model to support different degrees of independence for initiation of change in particular family members and propagation of change to other family members.

To illustrate the utility of environments and the transaction model, the following subsections each present a possible scenario for managing change in a software family. Section 5.1 discusses development of the family from a single central location. Therefore, its focus is on the ability of NSE to support multiple sets of derived objects (executables, and support both single source and multiple source families). Section 5.2 examines support for the development pattern of primary development on one family member and adaptation of code for

other members based on the primary change. In this pattern, the flow of changes back to the primary member is minimal. Section 5.3 describes NSE's ability to support the concurrent evolution of different family members and yet still share change among family members. Finally, Section 5.4 assesses NSE's limitations in supporting the composition model. It does so in the context of NSE's ability to support the creation of different family members by selection of alternative variants and versions of components comprising a system.

5.1. Variants

There are two different ways that variation in a software family occurs. The first occurs when a single set of sources generates multiple sets of objects and executables. Either several compilers are used to generate the sets of objects and executables, or switches passed to the compiler on different builds result in different program behavior such as debug or optimization switches. The second way that variation may be present occurs when different sources are used to compose different versions of the executable. If a program is available to run on two different window management systems, the user interface portions will have target-specific code that deals with each window manager. While the bulk of program code must be maintained in parallel across both targets, there is a portion which is selected at build time depending on the desired target.

If the members of a software family are to be developed or maintained in parallel, then the build procedure for different members must be synchronized. For example, using Sun's Network File Server, a Sun 3 and Sun 4 computer can share the same storage through the network. By keeping a single set of sources, changes and upgrades can occur to both Sun 3 and Sun 4 versions simultaneously. Although the Sun 3 and Sun 4 versions of a program share sources, the derived objects can not be accumulated in the same directories due to the name clash of the two variants of the derived object from the same source. A common solution is to construct a *Makefile* such that the compiler places each set of derived objects into a designated directory through a parameterized pathname. Then the invocation of *make* requires the proper parameter to build the correct set of derived objects.

NSE manages multiple derived object sets by storing different sets of derived objects in separate internally managed directories. An NSE environment can contain a version of a set of software sources, and several different *variants* of derived objects. Through parameters in the *activate* or *acquire* commands, the user specifies which directory of derived objects are seen by users and tools with the sources as shown in Figure 5-1. It is possible to construct one version of a *Makefile* without parameterized path names that will build the correct set of derived objects due to the environment managing the build context.
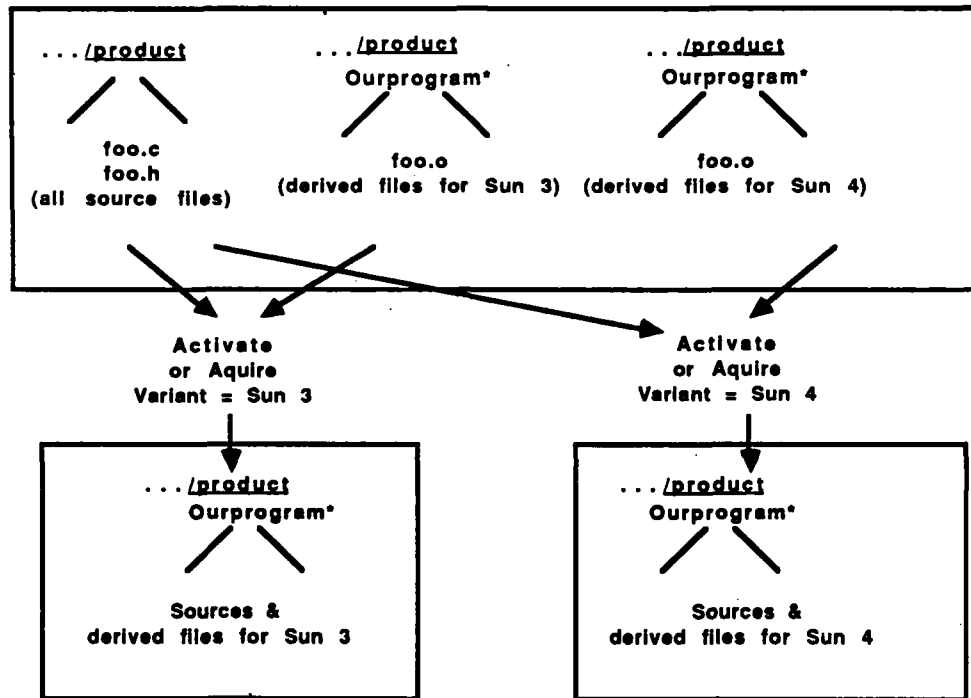
Figure 5-1: Source and Object Management

Parallel development on different variants may occur in two ways. First, a given environment may be activated at the same time by two different developers. Each developer can specify a different variant during the *activate* command. When one developer checks-out a file through VCS, the other developer can not modify it. When the developer checks-in the file the updated file is immediately available to the other developer. The VCS commands in this instance are serving as access control to the source files, and development on the two variants must be more closely coordinated between developers. A second scheme for parallel development occurs by creating child environments with the *acquire* command. When each child environment is created or activated, the developer may request a specific variant. The user and tools will access the sources and derived objects for only the requested variant. Each child environment can then represent the development of a particular variant. There is no file locking across child environments, and as such development can go

on in each environment independently. If two developers simultaneously change shared code in their respective variant environments, then the changes will be coordinated and tested using the same *acquire*, *reconcile* and *resolve* procedures described in Section 4.3.

If a developer is forced by a failed *reconcile* to integrate changes specific to another platform, it can be resolved by using a feature of environment variant activation. At activation time, specific environment variables are defined which indicate which set of derived objects are mapped into the directory structure. The C pre-processor commands **#if** and **#endif** can be used to select appropriate sections of code based on the values of environment variables set at activation. The textual merge editor that is invoked to help resolve such conflicts makes it very clear which areas should be marked for selection by the pre-processor. Figure 5-2 shows how conflicting machine dependent code is pointed out, and Figure 5-3 shows how simple pre-processor commands can be inserted to resolve the conflict.
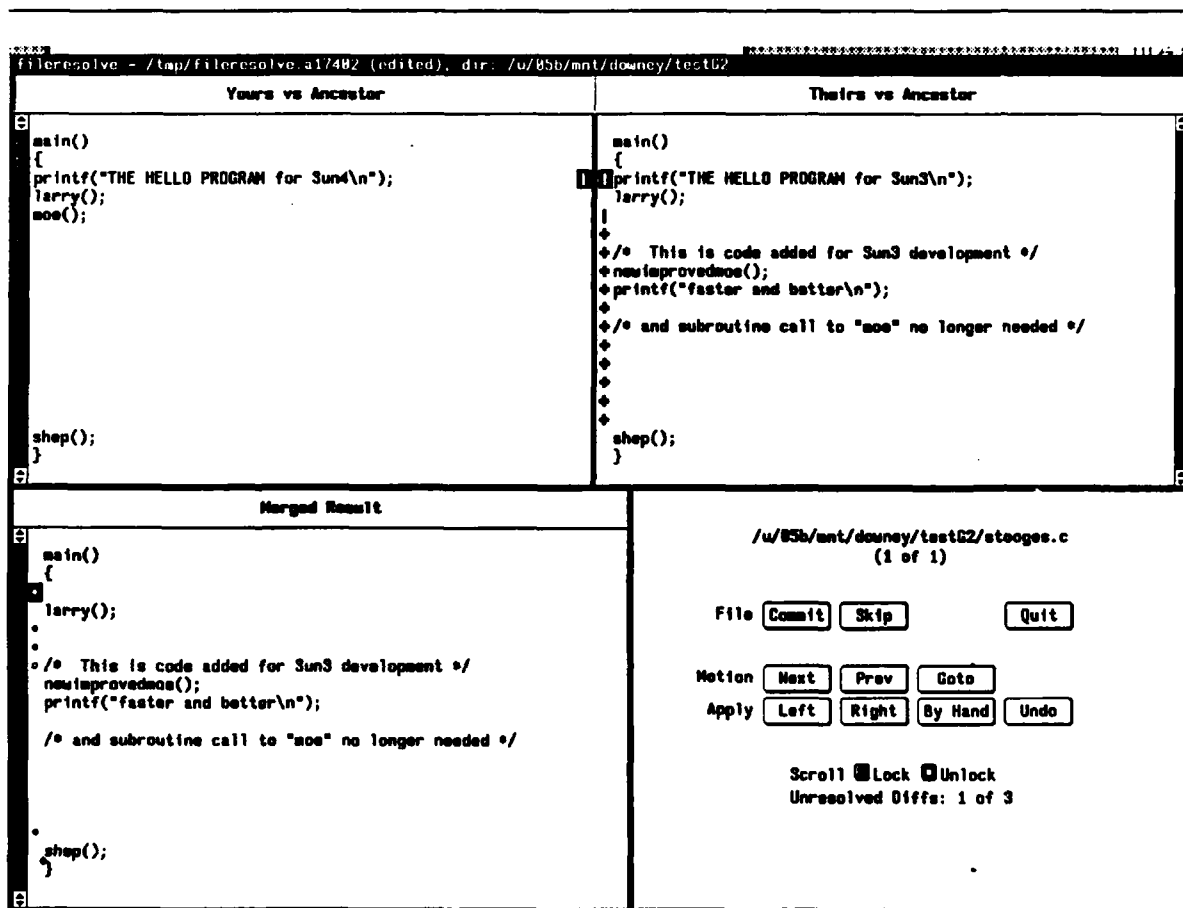
Figure 5-2: Conflict in Machine Dependent Code
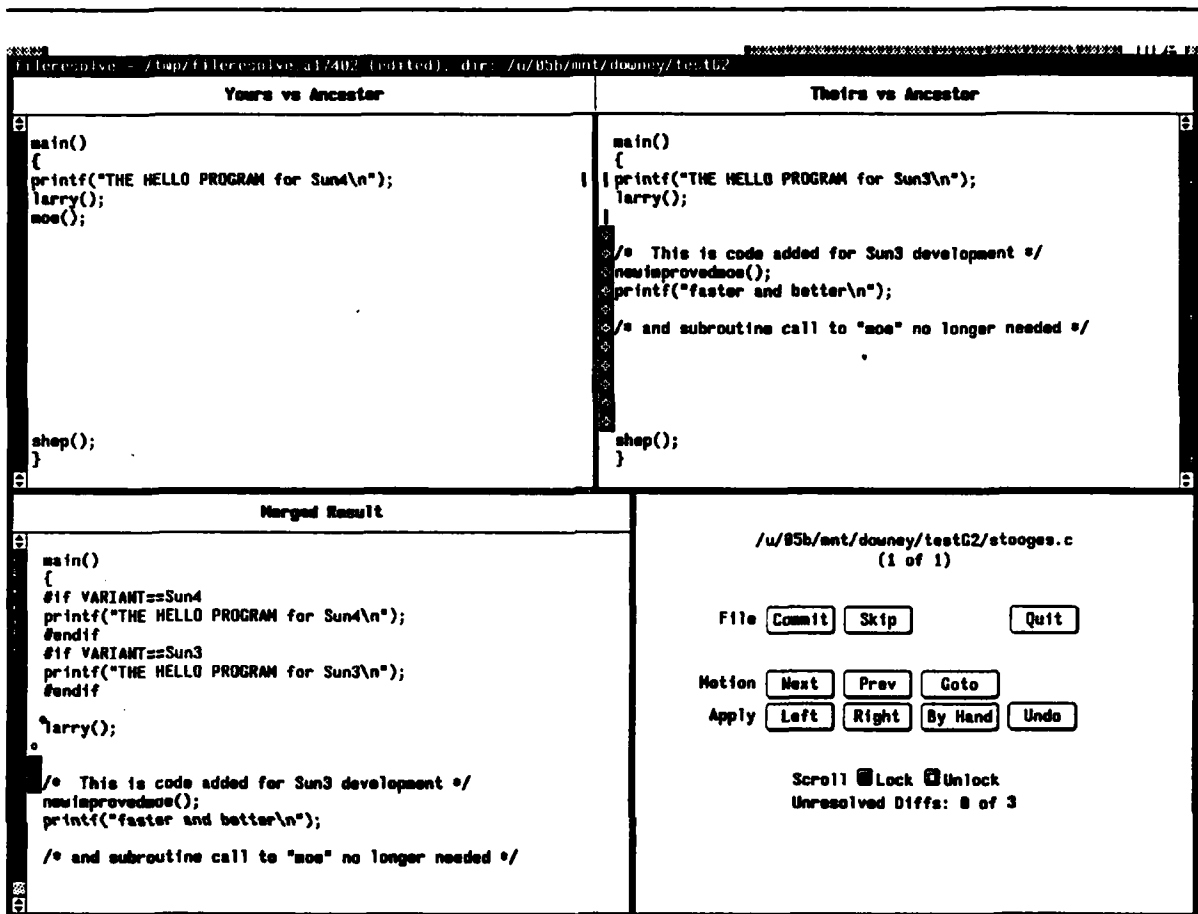
Figure 5-3: Merge of Machine Dependent Code

Variants easily support parallel development when different sources are used to compose different versions of the executable. If a developer of one variant makes a change to a platform-specific source file, the change will be propagated to other variants through the *acquire* and *reconcile* procedures. A developer on a different platform will receive the changed file upon *resync* or failed *reconcile.* He may choose to look at the file, to see if the changes may indicate that analogous changes are needed for his platform. The developer may also choose to ignore the file, since it is not compiled and linked for his target, and proceed by marking it with the command *merged.*

5.2. Primary Development and Adaptation

The use of variants in NSE can accommodate primary development for one platform, providing source for input to a secondary development effort for a different platform. In the relationship between the two efforts, the primary development can be viewed as the "master," while the secondary development is a "slave" to any changes or upgrades made in the primary development path. Advances in functionality may continue in the master environment, at the same time target-specific changes are made in the "slave" environment. This arrangement is best implemented with the "slave" environment being created as a child of the primary development environment. The flow of fixes and changes travels from the "master" environment to the "slave" environment via updates made with the *resync* command. Figure 5-4 depicts the relationship between the two work environments. Any machine dependent changes made for the secondary target, do not need to be ported back, and integrated with the primary development. Since a given environment may have more than one child environment, a master environment may drive more than one slave environment.
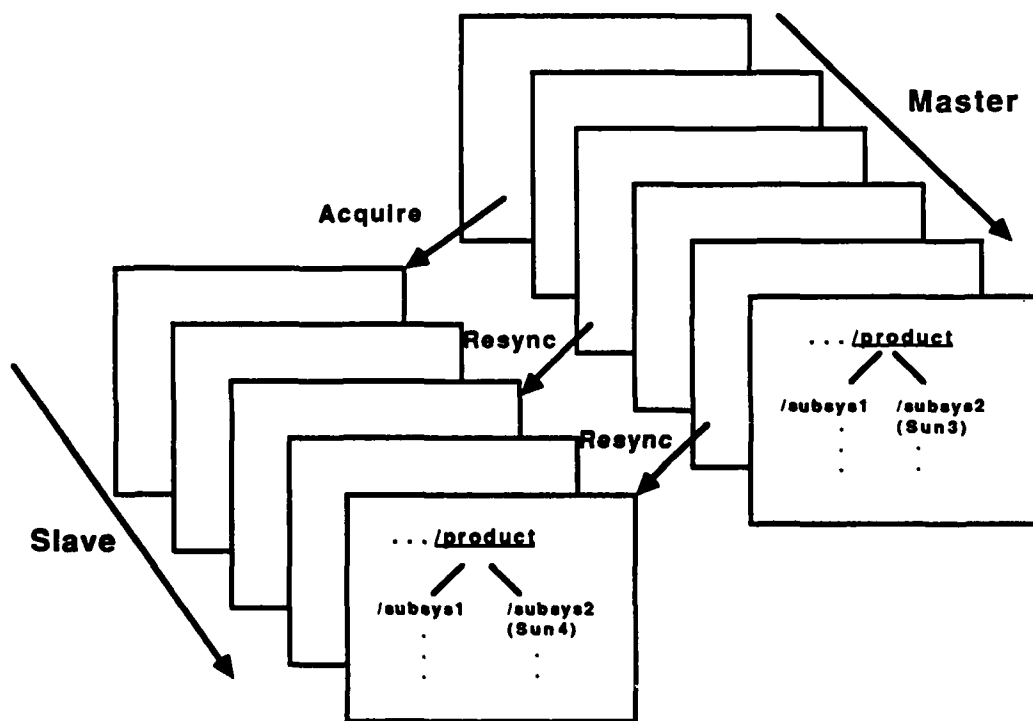
Figure 5-4: Major and Dependent Development

If a slave environment does contain changes which will be useful to the master environment, the semantics of the *reconcile* command impose some restrictions. First, the slave environment must be completely updated with any changes in the parent environment through a *resync* command. Second, the entire sequence of changes in the slave environment will be

propagated back to the master environment upon a successful *reconcile*. While it is useful to have a mechanism to propagate change between environments, it is not possible to be selective about which changes should be moved and applied to the other environment.

A product release from the master environment for the primary platfoim will reflect the latest advances. A product release for another platform made from the slave environment may lag behind the primary development path in terms of functionality.

5.3. Parallel Development for Different Family Members

When development must proceed in a more parallel fashion for multiple target platforms, NSE can take advantage of a traditional directory structuring scheme. Figure 5-5 illustrates how the code that is common to all target platforms resides in one development subdirectory. All code that is parameterized through preprocessor commands can reside in a subdirectory labeled "combined." These are both augmented by a subdirectory for each target platform, which contains code specific to the platform, which may or may not have an equivalent in code for other platforms. The root *Makefile* which invokes *Makefiles* in each subdirectory should be parameterized to build only a selected platform. The root *Makefile* also contains a target that will build all platforms in the case of a major release parallel to all platforms.
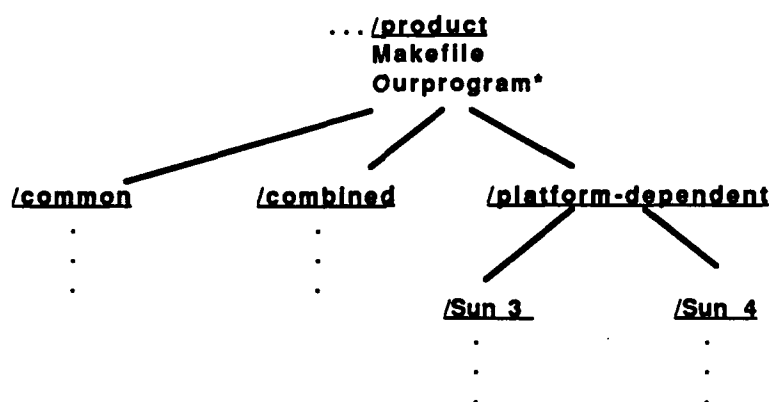
Figure 5-5: Directory Structure for Multi-Platform Source

Since NSE environments not only configure source code, but also derived objects and UNIX directory structure, change in this type of software family can be easily managed. Figure 5-6 shows a hierarchy of environments. The top-most environment is set up to act as repository for the work on the entire software family. In the second level of environments, each environment represents one of the targeted platforms. The third level of environments are workspaces, where changes to common and combined code can occur, in the context of a specific platform inherited from the workspace's parent. As shown in Figure 5-6, Programmer A can modify code common to all platforms, and code specific to Sun3 development. Through the use of variants the second level of environments and their children are limited to one variant of object code. Programmer A can view source code located in the Sun4 subdirectory, but by convention should not modify it, and is unable to derive Sun4 objects from it.
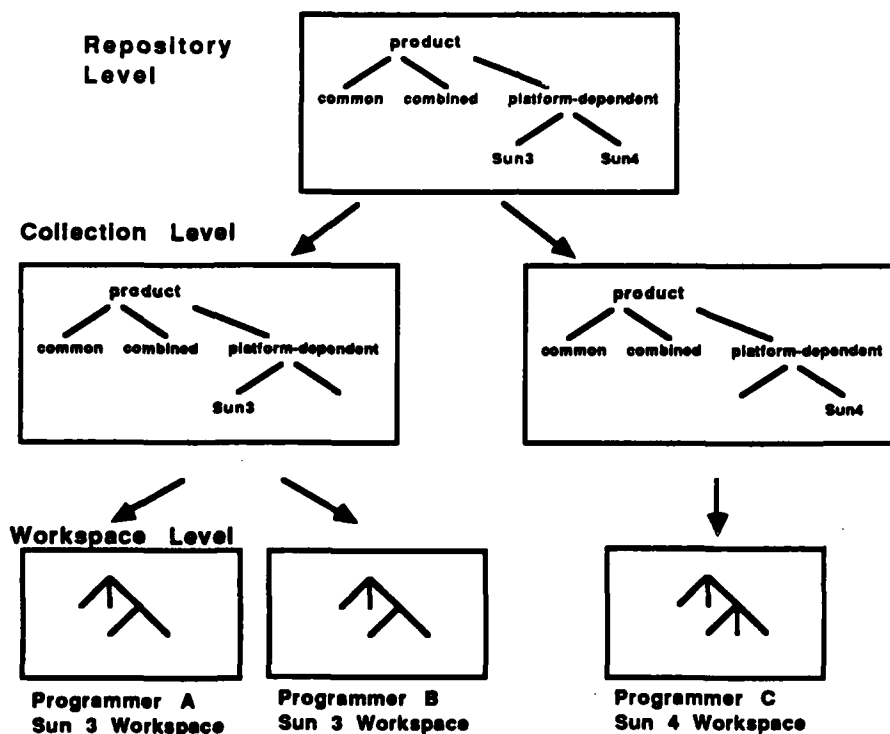
Figure 5-6: Parallel Multi-Platform Development

The second level of environments would coordinate work from multiple child workspaces for a particular platform. Changes to common or combined code, can be propagated to other platforms by reconciling with the top-level repository environment periodically, and accepting changes through *resync* from the top-level repository. Since a programmer is not to make changes to subdirectories for other platforms, during a *resync*, changes in the other subdirectories will be automatically accepted. As long as the programmer does not make changes in another target's subdirectories, he will not have to merge any changes for another target.

The ability to view changes to target-specific code for other platforms may be useful, as it can signal changes needed locally. However, it is possible to limit a working environment's visibility to source and derived objects for only one platform through the logical structure of the software system and the NSE object called a component. A component can represent each logical partition of the system. If a software family is structured as described above, there can be a common component, combined component, and a component for each target platform. Each of these components serves to bundle the source, derived code and sub-directory into one object. The commands which move objects between environments (*acquire*, *reconcile*, and *resync*) all recognize and act upon components. As before, the parent environment serves as the main repository. Child environments would *acquire* a common component, a combined component, but only one platform's component. Within the child environment the sources for common, combined, but only one platform are available for reading and modification. The environment is also limited to the derived objects for that platform through the specification of variant. In this way environment and component hierarchy enforces the rule that no changes can be made to files specific to other platforms. The *reconcile* and *resolve* of changes on common and combined code may proceed as described in the previous paragraph, and platform-specific code can evolve totally independently.

If a product release for each family member is made from the top level environment in the hierarchy, after each family member has reconciled, then the common code across all family members will contain the same functionality. Releases for an individual family member, made from the second level of environments in the hierarchy will reflect changes made for that specific member.

In summary, the use of the UNIX directory structure as shown in Figure 5-5 allows parallel development to occur in a software family. NSE allows for the coordination of evolution of the code that is common to all members of the software family. Using or not using components to logically structure the system provides a choice in whether a developer can monitor member-specific changes made for other family members.

5.4. Composition

A large system is often broken down into several subsystems which can evolve independently of each other during development and maintenance. A successful configuration management system will allow for this partitioning and also serve to support the composition of the system at times of integration test and product shipment. The Rational Environment is one such commercially available environment which provides the ability to construct a system from different versions of subsystems [2]. A system is partitioned into subsystems in such a manner as to minimize compilation dependencies between the subsystems. Each subsystem may evolve its own set of versions independently. Different configurations of the system are composed by selecting a version of each of the subsystems. At first inspection, the transaction model implemented by NSE would seem to preclude composition. However, through the use of environments, named revisions and control points, it is possible to compose selected versions of subsystems into systems.

If a software system can be partitioned such that the subsystems are minimally dependent on each other, each subsystem can be maintained within its own environment. Each environment would be created separately, with no parent-child hierarchy between the environments. Because the environments are not connected to each other, there is no migration of changes between subsystems and the subsystems evolve independently of each other. In order to be able to combine subsystems to perform system integration testing or system product release there must be a coordination of each environment or subsystem's control points. A convenient way to structure the control points is to have one UNIX directory represent the system. The system directory should then contain a separate subdirectory for each subsystem, which serves as the subsystem's environment control point. Each subsystem would populate its control point with the versions of its source, object and executable files organized by subsystem revision. Figure 5-7 depicts system breakdown by subdirectory and environment. A system can then be composed of one of each of the subsystems, by activating in succession the desired revision of each subsystem's environment. The unique control point for each environment allows the subsystem contents of each environment to be viewed together. The composition of a system can be tracked by an *activation file* that lists the environments to be activated and the revision name of each environment. The activation file actually represents a version of the system, and as such can be named so as to indicate a system version, and managed as an object external to the NSE.

control point:
.../product/subsys1

Subsystem 1 Environment

control point:
.../product/subsys2

Subsystem 2 Environment
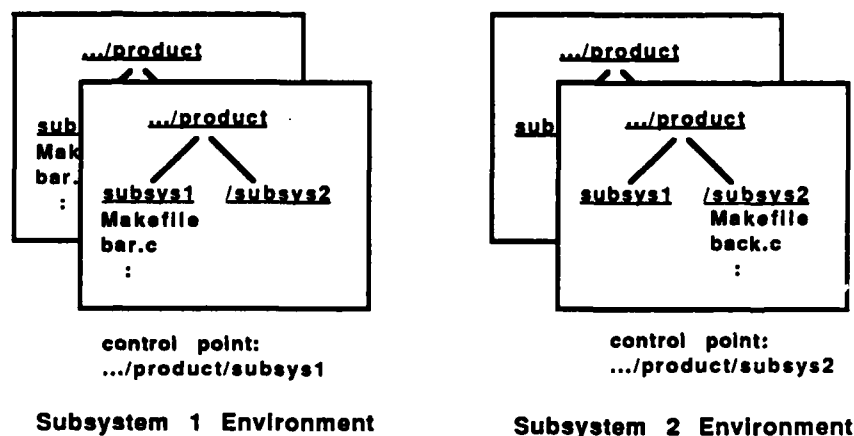
Figure 5-7: System Broken into Subsystems

Some degree of automation is possible in activating a series of environments to compose a system. Instead of the activation file merely containing a table, it can consist of a series of *activate* commands. The activation file would then be an executable shell script. Below is an example of such a script:

```
activate subsystem1 -R Release3.4    \
activate subsystem2 -R Release1.0    \
activate subsystem3 -R Release4.0    \
activate subsystem4 -R Release1.2    \
activate subsystem5 -R Release2.6
```

By executing the activation file, the composition of all the parts can be seen together in one activated environment. When a named revision is specified with the *-R* flag, its subsystem contents will be available in read-only format. This works well for composition from stable environments, and is useful to preserve frozen compositions for release and later re-creation.

During software development in the subsystem composition model it is usually necessary to perform integration testing. Integration testing often leads to a need for modification of the contents of subsystems to arrive at correct system behavior. It is possible to compose an environment of both unchangeable and working versions of subsystems. The following activation script represents a system consisting of two working versions of subsystems in the context of three other unchanging subsystems.

```
activate subsystem1 -R Release3.4   \
activate subsystem2                 \
activate subsystem3 -R Release4.0   \
activate subsystem4 -R Release1.2   \
activate subsystem5
```

If a revision is not specified, the *activate* command defaults to the latest working version of the environment. Due to the nature of the NSE Translucent File Service, only the last activated environment in the list, if it is a working environment, is available for write access. In the above example only the subsystem found in environment subsystem5 would be available for read/write access. It is possible to modify working environments that appear earlier in the activation script such as subsystem2 in the above example. The user must create a separate window and then *activate* the desired working environment from there. This can be done at the same time the composition of environments is active. Changes made in the separate window to the activated subsystem are immediately available in the composition environment. In the example above, the developer will have the ability to *vcs checkout* and modify files for subsystem5 in one window, and concurrently modify files in subsystem2 if it is activated in a second window. If this is done in order to integrate a system, the system may then be released by creating named revisions of the modified subsystems, and then creating an activation file with the named revisions of all subsystems. An activation file which refers to named releases of all its subsystems is by default a frozen configuration of the system.

In addition to the activation file which serves to represent a system, there will be a system *Makefile* or build script, and a resulting system executable. The system *Makefile* or build script will serve to link together derived objects from each of the subsystems. A convention, external to the Network Software Environment will be necessary to link and identify these three files as important to one distinct instantiation of the software system. One such convention is depicted in Figure 5-8. A subdirectory is created to the original UNIX directory structure for each integrated system. Each subdirectory then contains the required activation file, build script and executable.

```
          .../product
         /   |    \      \
        /    |     \      \
  /subsys1  /subsys2  /Release1.0  /Working1.3

  (Control points to      Makefile     Makefile     (build   script)
  composable  subsystems)  Release1.0   Release1.3   (activation  file)
                          product*     product*     (executable)
```
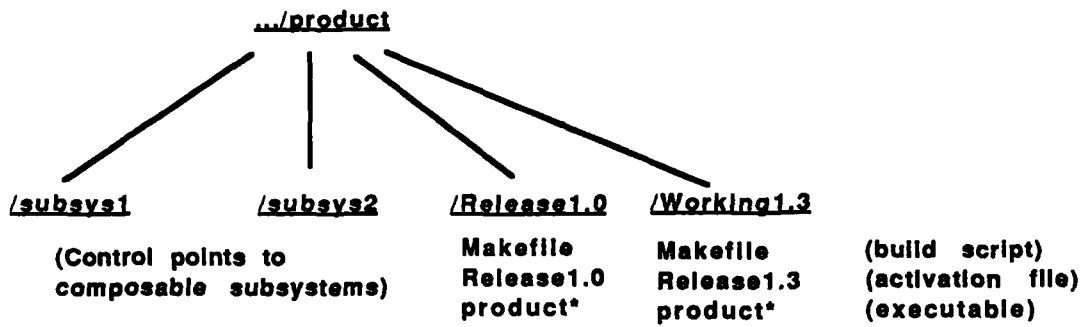
Figure 5-8: Management of System Composition

6. Support for Distributed and Heterogeneous Development

As the name indicates, NSE supports development in a network environment. In this section we examine in detail the support available through NSE for development in a distributed and heterogeneous network environment. First, we examine NSE's support for development in a network of Sun workstations. Then, we discuss the support possible through NSE for development on workstations other than Suns.

6.1. Distributed Development on Suns

Sun workstations come in three flavors: the Sun3 family based on the Motorola 68k processor, the Sun4 family based on the SPARC processor, and the Sun386i family based on the Intel 386 processor. Workstations from these three families can be interconnected. They all run the SunOS version of UNIX including NFS. NSE is available as a product on all three families.

6.1.1. Homogeneous Sun Network

NSE maintains source files and derived files in environments. A single environment contains a sequence of releases, that is, a version sequence of a configuration of files (collection of files in a directory structure). A particular configuration of files is accessible by activating an environment. As a result of the activation, the files and directories of the chosen configuration are made available at a point in the file system, known as the control point. Different environments and their configuration versions can have the same control point. Two environments can be activated on the same control point. The result is that two different processes (and all their subprocesses) see different versions of files if the configurations differ. Thus, the activation of an environment has the effect of a file system mount on a per process basis.

The files contained in an environment actually reside in a *root directory* which is also specified at environment creation time. This root directory can be anywhere in the file system, whether that is on a local disk or a file server accessible through NFS. Users are not aware of the physical location of the files. Access to these files is transparently provided through a NSE server. This server is similar to a NFS server, and understands the way the physical files are organized by not duplicating files that have not changed between configurations.

The names of environments are registered with the Sun network name server *yp*. Thus, they are known throughout the network and environments can be activated on any machine the name server responds to and that has been configured to run NSE.

The files contained in environments can be physically located anywhere in the network and can be made accessible anywhere in the network. The network may be configured such that all workstations access files from one or several file servers, or files may be located on local

---

disks. This allows for load balancing of file storage and file access. NSE specifically supports load balancing in two ways. First, NSE provides administrative functions to relocate the physical location of the files of an environment or a variant within an environment. Relocation may be necessary because the file system storing an environment becomes full, or because it is desirable to spread file access across several disks or file servers. Second, NSE allows environments to be self-contained by having copies of all files including files that have not changed between configurations. Normally, the NSE server may have to check the root directory of several parent environments to locate a file due to the fact that files by default do not get copied if they are not changed. For self-contained environments, file access is guaranteed to be only to the physical location of its own root directory. By locating an environment that represents the work area of a user as a self-contained environment on a local disk network, file access is minimized.

In a network consisting of two local area networks (LANs) and a lower bandwidth bridge connection, but configured as a single network file system, network traffic across the bridge can be reduced by making sure that the physical location of the files of an environment are on the same LAN partition as the majority of environment activations. This is done by creating a child environment that is self-contained. Its root directory can be located on the same partition as the majority of activations by executing *acquire* with the *-R* option. It is also possible to default to all local copies of files with the *-C* option of the *acquire* command. This overrides the default which is making physical copy only after the user indicates intent to modify the file with the *vcs checkout* command.

6.1.2. Heterogeneous Sun Network

In a heterogeneous Sun network, workstations from different families of processors with different instruction sets coexist. Because of the different instruction sets, different executables are necessary. NSE takes these differences into account both for code compiled from source code, and for the executables of tools.

NSE supports the derivation of different sets of objects from the same source. Typically, the derived object variants are the three Sun machine architectures (as indicated by the operation */bin/arch*), but other variants can be defined as well. When an environment is activated the derived files for one variant are made accessible under the control point together with the source files. Newly created derived files are added to the set of derived files of the activated environment variant. Only one set of derived files is accessible within one activation. By default, the variant is that of the architecture of the workstation on which the user is activating the environment. However, the user can specify an alternative variant at activation time. This allows a user to reside on a machine of one Sun family, (the host), and build code for a second Sun family, i.e.,(the target), through cross compilation. The executable code, however, can only be invoked and tested on the Sun family for which it was compiled. This can be done by activating the environment on a machine of the target architecture. The *activate* command will select the variant of code based on current machine architecture, and the corresponding derived objects will appear at the control point.

NSE supports transparent selection of tool versions through the concept of sets of executable tools called *execsets*. An execset contains particular versions of tools (or pointers to tool versions). Execsets are characterized by the host and target architectures they are intended for. For example, in a mixed Sun3 and Sun4 network there may be a separate execset for tools executing on Sun3s generating code for Sun3s, tools executing on Sun3s cross-compiling to Sun4s, and the equivalent tools executing on Sun4s (as illustrated in Figure 6-1). Such groups of execsets are referred to as *execset families*.

	Sun 3 Host	Sun 4 Host
Sun 3 target	68k Compiler	68k Cross Compiler
Sun 4 target	SPARC Cross Compiler	SPARC Compiler

Figure 6-1: Family of Executable Tool Sets

Execset families do not only distinguish between different host and target architectures, but also between different operating system versions for both the host and the target. Different versions of a tool may be dependent on the particular version of an operating system (e.g., compiler versions for SunOS 3.5 and for SunOS 4.0.3 whose executables are not compatible). In this case the execset facility, if set up correctly, will provide for automatic selection of the appropriate tool version based on the operating system version on the host.

Execset families are associated with environments, and can be associated with more than one environment. At environment activation, NSE selects an execset from the associated family based on host and target architecture and operating system version. The tools contained in the execset are transparently mapped into the system directories where the tools normally reside. For this transparent mapping, NSE uses the same per process mount capability as is used for making source and derived files of the environment transparently acces-

sible. The host architecture is determined by the machine the environment activation is performed on, and the target architecture is the architecture of the environment variant being activated. Thus, as a result of the environment activation the appropriate compiler or cross compiler appears in the respective system *bin* directory. Users and build facilities do not have to be concerned with selection of the correct tool version.

Execsets are not restricted to containing only executables of tools. They can also be used to provide transparent access to auxiliary files necessary for system build, but not managed as source files in environments. Examples are UNIX system include files and system library files. Such files may not only differ from architecture to architecture, but also from operating system version to operating system version. The distinction of operating system versions for the target machine allows include files or libraries for SunOS 4.0.3 to be made available transparently to tools executing on SunOS 3.5.

6.2. Development on a Heterogeneous Network

Often, software development facilities face the problem of managing code production on different vendor equipment (Sun, Dec VAX under ULTRIX and VMS, Apollo, etc.). Although NSE is currently only available on Sun workstations, development for and on machines from other manufacturers can be supported to various degrees. The problem with lack of availability of NSE on non-Sun machines is that files in an activated environment are not directly accessible, although those machines may be connected to Suns through a network and have access to files via the Network File System (NFS). One reason is that the NFS server handling client requests from non-Sun machines is not aware of an activated environment. In this section we discuss three development scenarios that can be directly supported with NSE as it is and indicate how NSE can be supplemented to further support non-Sun equipment.

6.2.1. Cross-Development on Suns

The first scenario represents software development on Sun workstations for non-Sun equipment. This scenario assumes that developers work on Suns, that is, they *activate* environments, edit files, and compile and link them on Suns using a cross-compiler and cross-linker. The derived files are kept in a non-Sun target variant, and the cross-development tools are transparently made available through the appropriate execset (see previous section).

Since the files in an activated environment are only accessible by machines that have NSE support, special steps must be taken to make the executables available on the target machine. One alternative is to copy the executables to the file system of the target machine by running a file transfer program from within the activated environment. A second alternative is to copy the executables into a directory that is accessible via NFS by both the developer's workstation and the target machine.

Testing the software on the target machine may require the use of a source level debugger. If the debugger is a cross-development tool, (can execute on a Sun and debug a program on a different target), it can run within the activated environment to access the source files. If the debugger can run only on the target, the source files will also have to be made available by transfer to the target file system or into an area accessible by NFS.

6.2.2. Remote Processing on Target Machines

The second scenario reflects developers editing on Sun workstations, but compilation and linking are being done on the target machine. In this case, the developer edits sources in the activated environment on a Sun workstation. Files necessary for compilation and linking are moved to the target machine or kept in a file system area accessible by the target machine through NFS. This can be accomplished through a tool invocation script (sometimes referred to as *envelope*), which will make the files accessible and invoke the tool on the remote machine. This envelope can also retrieve a copy of the tool output into the activated environment.

In case of C programs, the preprocessor of the C compiler can be run on the Sun, expanding out all macros and include statements. The execset mechanism would be used to pick up the include files appropriate for the target machine. As a result only a single file needs to be made accessible to the compiler running on the target machine.

6.2.3. Remote Development

The third scenario uses NSE and Suns as primary development facilities and performs development and adaptation for non-Sun targets on those machines. In this scenario the software is initially developed on Suns under NSE. As the software has to get ported, adapted, and tested on a non-Sun machine an environment is created on the Sun as a child of the environment containing the software to be worked on. This environment acts as a placeholder for the work on the non-Sun machine. The software is copied from this environment to a file system accessible by the non-Sun machine. Developers work on it using the tools on that machine. To save the work being done on the target machine the developers copy it back into the environment on the Sun representing this target work, and use the NSE *preserve* command. Once the work on the target environment is completed, the changes can be merged with other development on the software utilizing NSE support (through the *reconcile* command).

This last scenario requires the least effort for using NSE in the development of software for non-Sun machines. NSE is used as a repository and it provides support for controlled merging of concurrent changes. In the other two scenarios NSE is also used to support the workspace of individual developers, but tools are either required to support cross-development or part of the workspace is replicated on the target machine for tools residing only on that machine.

7. Conclusion

In this report we have examined the SCM support available in the Sun Network Software Environment (NSE) and explored the impact of its advances on three problem areas in software development. Sun NSE was chosen as a representative for several commercial environments that have recently become available and offer a number of advances over common practice in the area of support for developers. First, it offers SCM in a transparent manner, which makes it easy to use and allows for smooth transition from native UNIX as a development environment. Second, through the concept of environment, NSE provides a transaction-style usage model, which naturally supports developer workspaces, but has some limitations if used to support propagation of change between software family members. Third, NSE supports development in a heterogeneous network, offering a solution to managing multiple variants and versions of tools present in one software development environment. Finally, NSE's solutions demonstrate progress towards integrating support for SCM as both a control discipline and a developer support function.

One of the key elements of NSE's SCM services is its ability to transparently map the repository into the UNIX file system. This transparency eliminates the need for users to deal with two data management facilities, the repository and the file system, and with the transfer of information between them. Users and tools interact with familiar objects, directories and files, while the system maps them into a repository management scheme. Directories are used for structuring of system components into aggregates both in the repository and in the work area. All file system activity in the work area, (the mapped repository) is tracked through NSE. This allows NSE to manage derived files, support multiple variants of derived files for the same set of sources, and a version history local to the work area. NSE preserves the history of aggregates; it manages and evolves versions of configurations. Version identification of individual files is performed automatically by NSE relative to a selected configuration version. The complexities of versioning a large number of files and selecting versions from a variety of version graphs is hidden from the user. In summary, transparency through the repository mapping mechanism is a key facilitator for supporting developers in their work area and for bringing UNIX-based software development under SCM control with little additional effort.

The support for management of developer work areas leads to a transaction style usage model. This model, in a natural way, supports developers evolving a system from a baseline to a new configuration. The transaction model also introduces the notion of scope to change, in that a work area is isolated from other change, and local changes are not visible until promoted. The model also offers control of work areas, is a basis for coordination of concurrent change, and allows promotion of changes as aggregates. It moves away from version graphs for individual files with revisions and branches, to evolution of system configurations, possibly in variants, along several development paths. A team of developers perform each step in the evolution of the configuration as a transaction. A transaction can reflect a task that requires modification to a number of components. Changes by team members are coordinated through optimistic transactions, (changes are not synchronized through *a priori* locking), but update conflicts are detected at commit time, at which time the

conflicting transaction is required to merge the changes of the first transaction with local changes before proceeding. NSE's transactions can be extended with locking schemes. The semantics of a successfully committed transaction are a serialization of changes. In the case of optimistic transactions consistency is maintained through the forced merge of concurrent changes.

NSE supports a pure transaction model. This means that the function of a repository is accomplished through a set of non-terminating transactions. While the forced merge of changes in concurrent transactions is appropriate for supporting team development, it has shortcomings as a repository function. As was illustrated in NSE's support for evolution of a software family, the transaction model forces propagation of change to follow the transaction hierarchy, and change can flow up the hierarchy only when changes in the parent have migrated down and are merged in. This limits the degree of independent development possible even if a system is well partitioned. It also prevents the ability to migrate specific changes between development paths. The transaction model is one usage model that we have observed in recent commercially available software development systems. A future report will discuss the benefits and limitations of all three basic SCM models and the potential of combining them to overcome limitations.

NSE makes use of the transparent mapping mechanism not only to provide access to the repository and manage the work area of developers, but also to manage access to different tool versions in a heterogeneous computing environment. In a manner that is similar to versions of files being transparently accessible in the file system, versions of tool sets are made accessible in the file system. The tool set version is automatically selected based on host and target architecture and operating system, and on the version of the files the tools get applied to. This automation improves the consistency of the files maintained by NSE.

NSE does not offer a complete SCM solution. The set of SCM services offered by NSE is limited to SCM support for developers. However, improvements in SCM support for developers are important in order to bridge the gap between SCM as a control discipline and SCM as a support discipline. SCM as a control discipline, emphasizing the management aspect, controls the product through a repository and the change process through formal change requests and authorizations. Tool support for this aspect of SCM is often perceived as intrusive as it does not aid developers in their day-to-day work, but presents additional burden. SCM as a support discipline extends SCM concepts into developers' work areas, providing stability of workspaces by controlled isolation from change, coordination and controlled propagation of change. By providing these benefits to developers and by realizing that different degrees of SCM control are necessary for different part of the development process, SCM will become a more accepted part of the software process.

Appendix A: Glossary of Terms

This glossary is provided to define some software configuration management terms and Sun Network Software Environment terms that are used freely throughout the text of this report. Commands are provided in italics.

acquire (NSE) An NSE command that obtains the latest revision of objects from a parent environment for use within a child environment.

activate (NSE) An NSE command that readies an NSE environment for use by accessing the virtual file system associated with that environment. [4] This causes a version of source, derived objects, and execset containing tools to become available to a developer at a designated point in the UNIX file system.

artifact Any of many different kinds of objects that are used to build a software system, such as: source code files, object code files, executable files, and documentation files.

branch A development path consists of a series of configurations, one version following from changes to a previous version. A branch is the point where one version of the configuration gives rise to two or more simultaneous versions. For example, if a system is released, it must be re-creatable in order to track and fix reported errors. However, while waiting for customer error reports, developers can begin adding changes required for upgrades along one development path. When an error report comes in, the correction should be applied in a separate development path, also directly derived from the release version. At this point there are two branches in the development path.

bootstrap (NSE) The NSE process of putting a UNIX file and directory hierarchy under the control of NSE. [4]

change control board
A panel that may be comprised of product management, software development representatives, quality assurance, and/or customers who are interested in the function of a software system. The panel decides how the software system will evolve. Usually tasked with responding to change requests and determining whether those request are in line with the implied or stated function of the software system.

change request A method of communicating dissatisfaction with the operation of a software system to those in control of modification of the software system. In most common practice it is a form asking the software system user for details of the particular problem.

check-out/check-in A method of coordinating change to the artifacts of a software library. When an artifact is checked-out it typically becomes unavailable for check-out by another developer. When the artifact is checked-in it becomes available for check-out by others.

child environment An NSE environment that was created from an existing environment. It contains a virtual copy of all of the existing environment, or it may con-

tain a virtual copy of a designated component from the existing environment. If a file is edited in a child environment an actual copy of the file is created.

component   A portion of a software system that contains one logical subdivision of the system. A given software system may be seen as composed of its various components.

component (NSE)   An NSE object that is a group of related development objects that constitute a functional unit. A typical low-level component consists of a set of objects: an executable program plus the source, include, object and library files necessary to compile and link the executable program. A component can also include the program's design documentation, test data, and a test driver. Thus, a component gathers into a single unit everything needed to modify a functional piece of the larger system. Higher-level components group lower-level components together; usually, they correspond to subsystems or large programs. [4]

computing environment
Facilities for the development and operation of software comprised of an environment framework (or kernel) and a collection of tools.

configuration   One instance of a software system made distinguishable from another instance by the different versions of one or many of its components.

control point (NSE)   A UNIX file system directory specified at the time of creation of an environment where objects managed by NSE will appear when the environment is activated.

cross compilation   The creation of machine code for one computer architecture by compiling the source code on a architecture. The machine where the compilation occurs is the host. The machine where the compiled code is to execute is the target.

derived object   Any file which is produced when a tool processes input. The derived object is said to depend on the input files to the tool.

development path   A series of revisions to a software system are related in that they move towards a stated goal. For example, a given development path may contain all the error corrections the customers requested. Another development path may contain a series of planned upgrades that are to be present in the next product release.

environment (NSE)   An NSE object that is an individual workspace for developers that provides isolation, configuration management, and team coordination support.

execset or executable set (NSE)
An NSE structure that is associated with an environment and a variant of the environment that provides a stable set of executables necessary to derive the objects found in the variant.

execset family (NSE)
A group of NSE execsets that contain the same tools and are associated with an environment. Each member of the execset family is associated with a particular variant of the environment and contains the tools to produce the object code contained in the variant.

exit (NSE)	An NSE command that closes an active environment by stopping the Translucent File Service. When an environment is active its files are accessible at the control point. Upon exiting, the contents of the environment no longer appear under the control point.
file (NSE)	An NSE object that corresponds to a UNIX file. It may contain source code, and is recognized by NSE as a managed object; it may be derived from source code, and is recognized and managed as a derived object. It may contain any form of input data that is processed by a tool, including text or build scripts.
host	The computer system and operating system on which a program is compiled. See cross compilation and target.
library	The primary source of software artifacts to be evolved (i.e., retrieval of artifacts is a common operation). Often artifacts contained in the library are limited to source code. A historical record of artifacts is maintained in the form of sequential revisions and through branches in the revision graph.
lock (NSE)	An NSE command that is applied to an environment so that only the current user has write access to the environment's contents. A *lock* allows a user to make changes to an environment that has children.
make	A UNIX utility that performs a minimum recompilation based on the time stamps of source code files and derived object files. *Make* takes as input a file that describes derivation rules. The default name for this file is "Makefile."
Makefile	A file used by the UNIX utility *make*, that contains rules for the production of a derived object or set of derived objects.
merge	A procedure whereby one version of a software artifact is created from two separate versions. Usually the two versions evolved from a common ancestor.
merged (NSE)	An NSE command used to declare conflicts resolved in objects that the *resync* or *acquire* commands reported as conflicting. Conflicts are reported when an object has changed in both a parent and a child environment.
nseenv create (NSE)	An NSE command that will establish a new, empty NSE environment. The *bootstrap* command calls this command to create an environment that the *bootstrap* command populates with existing sources and components.
optimistic development	A development style for team programming that is characterized by allowing many changes to the source code to occur in parallel, and relying on a later collection and merge of the work. There is no locking, and conflict of changes is detected at merge time.
parent environment (NSE)	An NSE environment which is virtually duplicated by the creation of a child environment. Once a child is created the contents of the parent environment can not be accessed for change from within the parent environment without first locking the parent environment. All change should be propagated to the parent from child environments.

platform	A term referring to a computer system. The computer system is characterized by the hardware type, the operating system and version it is running, and possibly the window system and version.
RCS	Revision Control System, typically available on the UNIX operating system. It is a configuration management tool based on a software library which allows developers to coordinate change through check-in and check-out procedures. The tool maintains the changes that were made to create the current file and can re-create any earlier version of the file by reversing the changes. It also provides the ability to tag versions of files with some attributes.
reconcile (NSE)	An NSE command that delineates the differences between the hierarchy of objects in a child environment and its parent environment. If there has been no change in the parent environment since the virtual duplication in the child, the changes present in the child are moved to the parent environment.
release	A version of a software system made available to a larger group of users.
repository	The place where the software artifacts are kept in their online form. Its functions are modeled after those of a librarian; it is the gatekeeper of project products. Artifacts are typically created and modified outside the realm of the repository.
resolve (NSE)	An NSE command that merges conflicts that arise when the same object has been simultaneously modified in both a child and parent environment. The conflict is detected when a child environment returns its changes to the parent, or when a child environment tries to update to the latest changes in the parent.
resync (NSE)	An NSE command that updates the objects in a child environment with the corresponding objects in the parent environment.
revision	A snapshot of the current state of a software system; it usually differs from other revisions due to changes made in the source code, or by the tools used to produce derived objects.
revision (NSE)	A snapshot of the current state of an environment that is retrievable at a later time if necessary for bug fixes or support consultations. A revision can include a copy of the tools used to build the revision's object files, ensuring that the user can reliably recreate object files despite the installation of new tools. [4]
root directory (NSE)	A UNIX file system directory specified at the time of creation of an environment where NSE controlled objects actually reside. The Translucent File Service makes the objects residing here, appear under the control point when the environment is activated. Objects created in the environment are transparently stored by the Translucent File Service.
SCCS	Source Code Control System, typically available on the UNIX operating system. It is a configuration management tool based on a software library which allows developers to coordinate change through check-in and check-out procedures. It maintains the original version of a file. The tool also stores the changes that were made to create successive versions, and retrieves the latest version by re-applying the changes.

snapshot	A retrievable picture of a software system as it exists at some point in time. The contents of files, derived objects, system structure, and logical structure are all preserved.
software evolution	The series of changes that a software program goes through from initial design through implementation and even after release as a completed product.
software family	A collection of programs with essentially the same functionality. Members of the family may differ in alternative implementations of certain components or may represent adaptations to differences in the computing platform, such as different window systems or operating systems. The software may be maintained as a single set of source files, from which alternative executables are generated, or as alternative source files, from which a particular family member is produced by selection and composition.
sub-environment (NSE)	
	An NSE environment that was created as a child of an existing environment. At the time of creation it will receive a virtual copy of the contents of the existing environment. Any changes made in the sub-environment, may be returned to the environment it was copied from with the NSE *reconcile* command.
target (NSE)	An NSE object that represents the composition structures in a Makefile. It groups all the objects needed to compose an artifact as indicated in a Makefile, so that they may be operated on as a group by NSE commands.
target	The computer system and operating system for which a program is compiled. See cross compilation and host.
transaction model	A method for the configuration management of evolving source code that is characterized by a central repository containing a consistent version of the software system. The configuration is upgraded by: (1) creating a transaction containing a software object from the central repository, (2) locking the object in the central repository, (3) modifying the copy, (4) replacing the object in the repository with the modified copy, and (5) unlocking the object in the repository. Depending on the software configuration management implementation, a software object may be a single source code object such as one file, or an entire configuration of source code modules.
Translucent File Service (TFS) (NSE)	
	An NSE mechanism that allows a per process mount of the UNIX file system. The TFS provides the ability to see the contents of an environment at the control point when the environment is active.
unlock (NSE)	An NSE command that applies to an environment that was previously accessible for write by one user. *Unlock* releases the environment to accept changes from child environments or if there are no children it releases the environment for change by a different user.
variant (NSE)	is an NSE map that defines a correspondence between a set of object names and a set of object values (contents). The term is commonly used to encompass both the name-to-value map and the object editions

specified by the map entries. When an environment is activated, by default the variant that matches the machine's architecture and operating system release is also activated. [5] The object editions specified by the map entries are typically derived objects.

VCS (NSE)   See Version Control System.

vcs checkin and *vcs checkout* (NSE)

NSE commands that are part of the Version Control System. The *checkout* command locks a file for write access by one user within an environment. The lock is local to only the current activated environment. The *checkin* command creates a local copy that is frozen and stored within the environment. *Checkin* command also prompts the user for a comment about the change and returns the file to read-only status for all users. Within a workspace, *vcs checkout* and *vcs checkin* are similar to RCS and SCCS in that it uses a delta mechanism to store the changes between successive versions of the object in the environment.

Version Control System or VCS (NSE)

An NSE mechanism that records the history of changes to all objects under its control in an NSE environment. VCS stores the latest version of a file with markers to indicate how it was created from previous versions. Previous versions can be created by removing the applied changes. It is also a mechanism that provides file-locking with an environment, that prevents simultaneous editing of the same file if the same environment was activated by two developers.

working environment (NSE)

An NSE environment that is typically a child environment that serves as a private workspace for a developer to make and test changes to a configuration. A working environment is typically at the bottom of an environment hierarchy with no children itself.

workspace   The place where software artifacts are modified and tested online. It is generally easily accessed by a developer, and historically is separate from a library or repository area. Changes made in a workspace are particular to that workspace and must be specifically propagated to a library, repository, or separate workspace.

References

[1]   *Aide-De-Camp Software Management System User Guide*
        Version 7.0 edition, Software Maintenance and Development Systems, Inc., P.O.
            Box 555, Concord, MA 01742, 1989.

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

[3]   Leblang, D.B., and Chase, R.P., Jr.
        Parallel Software Configuration Management in a Network Environment.
        IEEE Software 4(6):16-27, November 1987.

[4]   *Network Software Environment: Reference Manual*
        Sun Microsystems, Inc., 2550 Garcia Avenue, Mountain View, CA 94043, 11 March
            1988.

[5]   Courington, William.
        The Network Software Environment.
        Technical Report, Sun Microsystems, Inc., 1989.

[6]   Schwanke, R.W., et al.
        Configuration Management in BiiN SMS.
        In *Proceedings of the 11th International Conference on Software Engineering,* pages
            383-393. May 1989.

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION		1b. RESTRICTIVE MARKINGS			
UNCLASSIFIED		NONE			
2a. SECURITY CLASSIFICATION AUTHORITY		3. DISTRIBUTION/AVAILABILITY OF REPORT			
N/A		APPROVED FOR PUBLIC RELEASE			
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE		DISTRIBUTION UNLIMITED			
N/A					
4. PERFORMING ORGANIZATION REPORT NUMBER(S)		5. MONITORING ORGANIZATION REPORT NUMBER(S)			
CMU/SEI-90-TR-23		ESD-90-TR-224			
6a. NAME OF PERFORMING ORGANIZATION	6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION			
SOFTWARE ENGINEERING INST.	SEI	SEI JOINT PROGRAM OFFICE			
6c. ADDRESS (City, State and ZIP Code)		7b. ADDRESS (City, State and ZIP Code)			
CARNEGIE MELLON UNIVERSITY PITTSBURGH, PA 15213		ESD/AVS HANSCOM AIR FORCE BASE HANSCOM, MA 01731			
8a. NAME OF FUNDING/SPONSORING ORGANIZATION	8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER			
SEI JOINT PROGRAM OFFICE	ESD/ AVS	F1962890C0003			
8c. ADDRESS (City, State and ZIP Code)		10. SOURCE OF FUNDING NOS.			
CARNEGIE MELLON UNIVERSITY PITTSBURGH, PA 15213		PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT NO.
		63752F	N/A	N/A	N/A

11. TITLE (Include Security Classification)
TRANSACTION-ORIENTED CONFIGURATION MANAGEMENT: A CASE STUDY

12. PERSONAL AUTHOR(S)
Peter Feiler and Grace Downey

13a. TYPE OF REPORT	13b. TIME COVERED		14. DATE OF REPORT (Yr., Mo., Day)	15. PAGE COUNT
FINAL	FROM _____ TO _____		November 1990	53 pp.

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB. GR.	configuration management transaction
			Sun Network Software Environment software evolution
			software development environment

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Software configuration management (SCM) is a key element of the software development process. A number of new configuration management techniques in commercial SCM tools and environments with SCM capabilities have been observed. This report illustrates some of the advances in SCM concepts by example of a particular commercial system: the Sun Network Sosttwar Environment (NSE). NSE embodies a transaction model of configuration management. In order to demonstrate the capabilities and limitations of the transaction model, NSE is applied to three problem areas for configuration management: adaptation for parallel development and team support, development and maintenance in software families and development in a distributed and heterogeneous network.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION	
UNCLASSIFIED/UNLIMITED ☑ SAME AS RPT. ☐ DTIC USERS ☒	UNCLASSIFIED, UNLIMITED DISTRIBUTION	
22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE NUMBER (Include Area Code)	22c. OFFICE SYMBOL
JOHN S. HERMAN, Capt, USAF	412 268-7630	SEI JPO