Sun WorkShop TeamWare User's Guide

901 San Antonio Road
Palo Alto, , CA 94303-4900
USA 650 960-1300 fax 650 969-9131

Part No: 805-4949
Revision A, February 1999

Adobe PostScript™

Please Recycle

Contents

Preface

The *Sun WorkShop TeamWare User's Guide* describes how to use the Sun™ Workshop™ TeamWare code management tools. The concepts and information discussed apply to both command line and graphical user interfaces.

Who Should Use This Book

This manual is intended for the software developer, but also addresses integrators, administrators, and release engineers in their tasks involving code management.

As a software developer, you typically acquire code from a code integration area or integration workspace. You then:

- Add new features to your program module
- Test and debug the program
- Put the code back in the implementation or integration workspace from which it was acquired

The descriptions of the Configuring tool are primarily addressed to the software developer, but also address the needs of integrators, administrators, and release engineers.

Use the chapters on Versioning and Freezepointing if you write programs coded in ASCII text source. These chapters assume that you are familiar with programming constructs and processes. You need not have previous experience with the Source Code Control System (SCCS).

The Building and DistributedMake chapters are a supplement to the standard `make` documentation. They describe how to use Building and Distributed Make to make the process of building programs more efficient. Use these chapters if you maintain programs using the make utility and wish to speed up the build process. These chapters also assume that you are familiar with the standard make utility.

This manual assumes some understanding of the Solaris™ operating environment and UNIX® commands.

How This Book Is Organized

Chapter 1" provides instructions for quickly getting started using the Sun WorkShop TeamWare code management tools.

Chapter 2" provides a full introduction to the Sun WorkShop TeamWare product, and provides you with ways to get the best information.

Chapter 3" presents an overview of the Configuring utility in TeamWare. Basic concepts are discussed that are vital to understanding how Configuring works.

Chapter 4" presents an overview of the Merging utility in TeamWare, and describes the graphical interface.

Chapter 5" presents information on how to start a new project, and how to move existing work projects into the TeamWare environment.

Chapter 6" describes the Configuring user interfaces.

Chapter 7" describes the Configuring workspace and the associated commands.

Chapter 8" describes the TeamWare Configuring transactions used to transfer files between workspaces.

Chapter 9" explains how you resolve conflicts between files in parent and child workspaces.

Chapter 10" describes how TeamWare Configuring manipulates SCCS history files during file transfer transactions.

Chapter 11" contains an example that demonstrates the TeamWare Configuring bringover, putback, and resolve transaction cycle.

Chapter 12" describes common SCCS functions such as checking out and editing a file, checking in a new file, and displaying delta differences, using the Versioning utility. It covers the basic operational tasks and walks you through step-by-step instructions.

Chapter 13" presents Merging, a tool that allows you merge files to resolve conflicts. It provides an introduction to the graphical interface and a tour of the command line options.

Chapter 14" examines the process of using Merging to resolve differences between files, including automatic merging, and provides a detailed example.

Chapter 15“ presents Freezepointing, a tool that allows you to create snapshots of a project. It provides an overview of the graphical interface and shows you how to use this tool in conjunction with the other TeamWare development tools.

Chapter 16“ provides a problem checklist to consider before calling the Sun Support hotline. It also gives information on how to report a problem, as well as a list of error messages—their meanings and what to do next.

Chapter 17“ presents a discussion of the build process with specific targets, as well as hints on fixing build errors.

Chapter 18“ describes `dmake`, a tool that allows you to distribute builds over several hosts concurrently. The operation of `dmake` is described, and instructions given on how to distribute your build efficiently.

Chapter 19“ lists TeamWare Configuring error messages and warnings. Each message is defined, and a possible remedy is provided.

“Glossary“ provides a clear explanation of the special terms used in this manual.

Appendix A“ explains why SCCS Mergeable IDs (SMIDs) are necessary, how to translate SCCS delta IDS (SIDs) to SMIDs, and how to translate SMIDs to SIDs.

Multiplatform Release

Note - The name of the latest Solaris operating environment release is Solaris 7 but code and path or package path names may use Solaris 2.7 or SunOS 5.7.

The Sun™ WorkShop™ documentation applies to Solaris 2.5.1, Solaris 2.6, and Solaris 7 operating environments on:

- The SPARC™ platform
- The x86 platform, where x86 refers to the Intel implementation of one of the following: Intel 80386, Intel 80486, Pentium, or the equivalent

Note - The term “x86” refers to the Intel 8086 family of microprocessor chips, including the Pentium, Pentium Pro, and Pentium II processors and compatible microprocessor chips made by AMD and Cyrix. In this document, the term “x86” refers to the overall platform architecture. Features described in this book that are particular to a specific platform are differentiated by the terms “SPARC” and “x86” in the text.

Related Books

The following Sun manuals and guides provide additional useful information:

Other Sun WorkShop Books

- *Sun WorkShop Quick Install* provides installation instructions.
- *Sun WorkShop Installation and Licensing Reference* provides supporting installation and licensing information.
- *Using Sun WorkShop* gives information on performing development operations through Sun WorkShop.
- *Debugging a Program With dbx* provides information on using `dbx` commands to debug a program.
- *Analyzing Program Performance with Sun WorkShop* describes the profiling tools; LoopTool, LoopReport, LockLint utilities; and use of the Sampling Analyzer to enhance program performance.
- *Sun™ WorkShop Visual User's Guide* describes how to use Visual to create C++ and Java™ graphical user interfaces.

Solaris Books

- The Solaris *Linker and Libraries Guide* gives information on linking and libraries.
- The Solaris *Programming Utilities Guide* provides information for developers about the special built-in programming tools available in the SunOS™ system.

Ordering Sun Documents

The SunDocsSM program provides more than 250 manuals from Sun Microsystems, Inc. If you live in the United States, Canada, Europe, or Japan, you can purchase documentation sets or individual manuals using this program.

For a list of documents and how to order them, see the catalog section of the SunExpress™ Internet site at `http://www.sun.com/sunexpress`.

Accessing Sun Documents Online

Sun WorkShop documentation is available online from several sources:

- The `docs.sun.com` Web site
- AnswerBook2™ collections
- HTML documents
- Online help and release notes

Using the `docs.sun.com` Web site

The `docs.sun.com` Web site enables you to access Sun technical documentation online. You can browse the `docs.sun.com` archive or search for a specific book title or subject. The URL is `http://docs.sun.com`.

Accessing AnswerBook2 Collections

The Sun WorkShop documentation is also available using AnswerBook2 software. To access the AnswerBook2 collections, your system administrator must have installed the AnswerBook2 documents during the installation process (if the documents are not installed, see your system administrator or Chapter 3 of *Sun WorkShop Quick Install* for installation instructions). For information about accessing AnswerBook2 documents, see Chapter 6 of *Sun WorkShop Quick Install*, Solaris installation documentation, or your system administrator.

Note - To access AnswerBook2 documents, Solaris 2.5.1 users must first download AnswerBook2 documentation server software from a Sun Web page. For more information, see Chapter 6 of *Sun WorkShop Quick Install*.

Accessing HTML Documents

The following Sun Workshop documents are available online only in HTML format:

- Tools.h++ Class Library Reference
- Tools.h++ User's Guide
- *Numerical Computation Guide*

- Standard C++ Library User's Guide
- *Standard C++ Class Library Reference*
- *Sun WorkShop Performance Library Reference Manual*
- *Sun WorkShop Visual User's Guide*
- Sun WorkShop Memory Monitor User's Manual

To access these HTML documents:

1. **Open the following file through your HTML browser:**

   install-directory`/SUNWspro/DOC5.0/lib/locale/C/html/index.html`

   Replace *install-directory* with the name of the directory where your Sun WorkShop software is installed (the default is `/opt`).

   The browser displays an index of the HTML documents for the Sun WorkShop products that are installed.

2. **Open a document in the index by clicking the document's title.**

Accessing Sun WorkShop Online Help and Release Notes

This release of Sun WorkShop includes an online help system as well as online manuals. To find out more see:

- Online Help. A help system containing extensive task-oriented, context-sensitive help. To access the help, choose Help Help Contents. Help menus are available in all Sun WorkShop windows.
- Release Notes. The Release Notes contain general information about Sun WorkShop and specific information about software limitations and bugs. To access the Release Notes, choose Help Release Notes.

What Typographic Changes Mean

The following table describes the typographic changes used in this book.

TABLE P–1 Typographic Conventions

Typeface or Symbol	Meaning	Example
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls -a` to list all files. `machine_name% You have mail.`

`AaBbCc123`	What you type, contrasted with on-screen computer output	`machine_name%` **`su`**
		`Password:`

AaBbCc123	Command-line placeholder: replace with a real name or value	To delete a file, type `rm` *filename*.
AaBbCc123	Book titles, new words or terms, or words to be emphasized	Read Chapter 6 in *User's Guide*. These are called *class* options. You *must* be root to do this.

Shell Prompts in Command Examples

The following table shows the default system prompt and superuser prompt for the C shell, Bourne shell, and Korn shell.

TABLE P–2 System Prompts

Shell	Prompt
C shell prompt	`machine_name%`
C shell superuser prompt	`machine_name#`
Bourne shell and Korn shell prompt	`$`
Bourne shell and Korn shell superuser prompt	`#`

CHAPTER 1

QuickStart

Use this chapter to quickly get started using Sun WorkShop TeamWare. For more details, see the other parts of this guide. Also refer to the online help for immediate assistance in completing specific tasks.

This chapter contains the following sections:

- "Basic Concepts" on page 1
- "Using TeamWare Configuring" on page 3
- "Using TeamWare Versioning" on page 10
- "Using TeamWare Freezepointing" on page 13
- "Distributed Make" on page 16

Basic Concepts

TeamWare is based on a concurrent development model called *Copy-Modify-Merge*. Isolated (per developer) workspaces form the TeamWare model. A workspace is a specially designated UNIX directory and its subdirectories. With TeamWare, you *copy* source from a central workspace into your own workspace, *modify* the source, and *merge* your changes in the central workspace with the changes made by other developers.

In addition to providing isolated workspaces, TeamWare enables you to easily and intelligently copy files between workspaces and then merge changes that exist between corresponding files. The intelligent copy feature enables you to copy project files in groups that you (or the project administrator) determine are logically linked, and automatically determines differences between files in the originating workspace and the destination workspace.

TeamWare further assists the concurrent development process by determining whether differences exist between the files in the central workspace and your workspace. If differences are found to exist, TeamWare commands prevent you or another developer from copying over those changes; TeamWare then provides sophisticated window-based tools that help you to merge these differences.

Parent and Child Workspaces

When you copy files from a central workspace to create a new workspace, a special relationship is created between the central workspace and the new one. The central workspace is considered the *parent* of the newly created *child* workspace. You can acquire files from any Configuring workspace in this manner, and workspaces can have an unlimited number of children. The portion of the file system that you copy from the parent workspace is determined at the time you copy it. You can copy the entire contents of the parent to the child, making it a clone of the parent, or you can copy only portions of the file system hierarchy that are of interest to you. The TeamWare Configuring transaction used to copy files from a parent workspace to a child workspace is called *Bringover*.

When development and testing are complete in the child, you copy changes in files that were modified or added in the child, back into the parent workspace. Once the altered files are present in the parent, they can be copied by other children or passed up another level to the parent's parent workspace. The TeamWare Configuring transaction for copying changes in files from a child workspace to a parent workspace is called *Putback*.

If any of the files you attempt to put back are changed in *both* the parent and child workspace, the files are in *conflict*. If this is the case, Configuring will block the transaction. You must then use the Bringover transaction to bring over the changed information from the parent and use the *Resolve* transaction to resolve the conflict in the child workspace before you can put your work back to the parent.

Source Code Control System

TeamWare Configuring acts only upon files under the Source Code Control System (SCCS). When considering Configuring file transfer transactions, remember that source files are derived from SCCS deltas and are identified by SCCS delta IDs (SIDs). When a file is copied by either a Putback or Bringover transaction, Configuring acts upon (copies or merges) the file's SCCS history file (also known as the "s-dot-file"). How Configuring manipulates and merges the history files is described in Chapter 10."

Changing Names

The current release of TeamWare uses new command names, so Table 1–1 summarizes the correspondences for you. Note that the old commands still work, however this manual uses the new commands and GUI names.

TABLE 1–1 Correspondences Between Old and New TeamWare Commands

Old Command	New Command	Old Tool Name	New GUI Name
codemgrtool	twconfig, teamware	CodeManager	Configuring
vertool	twversion	VersionTool	Versioning
filemerge	twmerge	FileMerge	Merging
maketool	twbuild	MakeTool	Building
freezepttool	twfreeze	FreezePoint	Freezepointing

Using TeamWare Configuring

You can use TeamWare Configuring through either a graphical user interface (GUI) or command-line interface (CLI). The following procedures use the GUI. For information about the CLI, refer to the `bringover(1)` and `putback(1)` man pages.

Getting Started

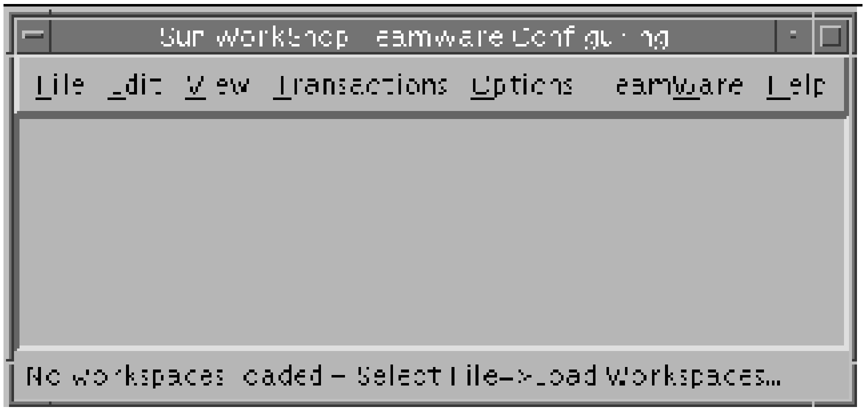
You can start TeamWare Configuring by typing `twconfig` at a shell command prompt, followed by the ampersand symbol (&).

If you are running Sun™ WorkShop™, you can start TeamWare Configuring by:

- Clicking the TeamWare button on the tool bar in the Sun WorkShop main window
- Selecting TeamWare from the Tools menu

When you start TeamWare Configuring, the Configuring window (see Figure 1–1) opens.

Note - Before you begin using TeamWare Configuring on your project, you must know the path name of the workspace from which you are to bring over your work



No workspaces loaded – Select File->Load Workspaces...

Figure 1–1 Configuring Window

Creating a New Workspace

You can create a new workspace by:

- Making an existing hierarchy of files into a workspace
- Creating an empty workspace and then populating it with a hierarchy of new directories and files
- Copying files from a central (parent) workspace to create a child workspace

Creating a Workspace From an Existing Hierarchy of Files

If you have a directory containing the files that you want to make into a workspace, do the following to create a new workspace:

1. **Start TeamWare Configuring.**

2. **Choose File > Create Workspace.**

3. **In the Workspace Directory text box, enter the path name to the directory that contains the files.**

4. **Click OK.**

   TeamWare Configuring creates a workspace in the directory you have specified and automatically creates an icon for it in the Configuring window. You may notice that a new directory called `Codemgr_wsdata` has been created in your workspace directory. This is a special directory that TeamWare uses to maintain information about the workspace such as access control, notification lists, and transaction history.

5. **TeamWare recognizes only files that are under SCCS version control. If you have not already done so, make sure to check in any files in the workspace directory tree that you wish to be included in the workspace by TeamWare.**

   You may do this by typing `sccs create` commands at the command line or by using TeamWare Versioning.

Creating an Empty Workspace

To create a new empty workspace:

1. **Start TeamWare Configuring.**

2. **Choose File > Create Workspace.**

3. **In the Workspace Directory text box, type the name of an existing directory with a new workspace name.**

4. **Click OK.**

The new workspace is populated with `Codemgr_wsdata` files only.

Creating a Child Workspace

To create a child workspace:

1. **Start TeamWare Configuring.**

2. **If the workspace from which you must obtain your files is not automatically loaded, choose File > Load Workspaces.**

3. **Select the workspace in the Load Workspaces dialog box and click Load Workspaces.**

   The parent workspace is loaded and its icon appears in the Configuring window.

4. **Drag and drop the parent workspace icon onto a vacant space in the Configuring window, or choose Transactions > Bringover > Create, to open the Bringover Create version of the Transactions window (see Figure 1–2).**

5. **In the Bringover Create Transactions window, type the child workspace path name in the text box labeled: To Child Workspace Directory.**

6. **In the Directories and Files pane, create the list of directories and files you wish to bring over to your workspace from the parent workspace.**

   Click Add to open the Add Files dialog box and select files to bring over. Shift-click to select more than one file. To select all of the files, type a period (.) in the Name text box. Click Add Files when you have selected all the files you want.

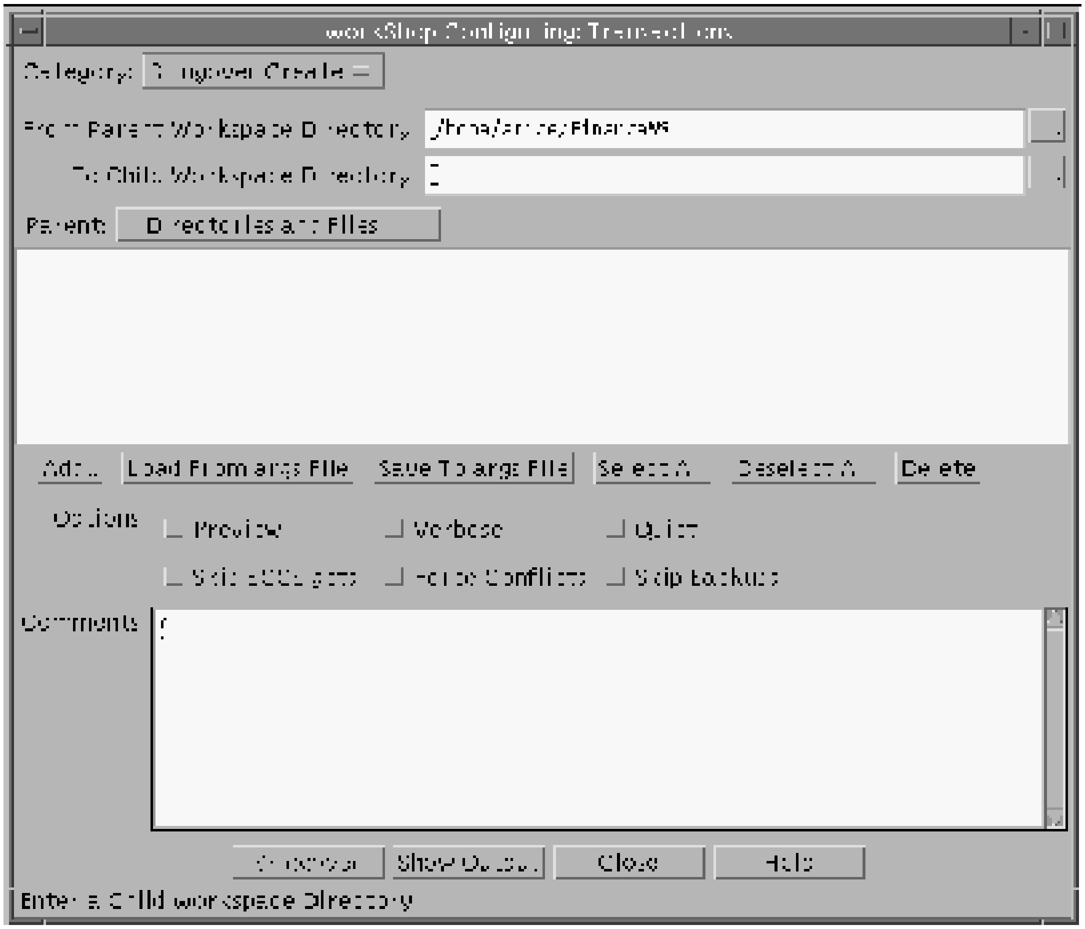
   Note - You can select the Preview option to verify your transaction before you actually transfer any files.

   ---

7. **Click Bringover in the Transactions window to initiate the Bringover Create transaction.**

8. **View transaction output in the Transaction Output window.**

   For more information about the Bringover Create transaction, see “Creating a New Child Workspace (Bringover Create)” on page 93.



From Parent Workspace Directory: /home/[illegible]/FinanceWS

Enter a Child workspace Directory

Figure 1–2 Bringover Create Version of Transactions Window

Changing Files in a Child Workspace

To change files in a child workspace, you need to start Versioning (see “Starting Versioning” on page 10).

Check out the files, edit them, and check them back in under SCCS (see “Checking Files In and Out of SCCS” on page 11).

Putting Back Changes to the Parent WorkSpace

You can update the parent workspace with the changes you make. This Configuring transaction is called Putback.

To put back changes to the parent workspace:

1. **Start a Putback transaction by dragging and dropping your child workspace icon onto the parent workspace icon, or by choosing Transactions > Putback, to open the Putback version of the Transactions window.**

   Configuring automatically fills in the names of the parent and child workspaces in the Putback Transaction window and includes the same directories and files that you included when you created the child workspace.

2. **Type a comment in the Comments text window.**

   Note - You can select the Preview option to verify your transaction before you transfer any files.

   ---

3. **Click Putback to initiate the Putback transaction.**

4. **View transaction output in the Transaction Output window.**

   For more information about the Putback transaction, see "Updating a Parent Workspace Using Putback" on page 102.

Updating the Child Workspace

If any of the files have changed in the parent since you brought them over, the Putback transaction is blocked. In this case, you will have to use the Bringover Update transaction to bring those changes into your child workspace.

To update the child workspace:

1. **Resolve any conflicts (see "Resolving Conflicts" on page 9).**

2. **Put back the files to the parent workspace.**

   A popup window advises you that the transaction is blocked.

   Note - You can select the Preview option to verify your transaction before you transfer any files.

   ---

3. **Initiate the Bringover Update transaction by clicking Bringover now in the popup window to open the Bringover Update version of the Transactions window.**

In the Bringover Update Transactions window, Configuring automatically fills in the names of the parent and child workspaces, and includes the same directories and files that you included when you created the child workspace.

4. **Click Bringover to initiate the Bringover Update transaction.**

5. **View your transaction output in the Transaction Output window.**

   For more information about the Bringover Update transaction, see "Updating an Existing Child Workspace (Bringover Update)" on page 97".

Resolving Conflicts

If any of the files you changed in your child workspace were also changed in the parent workspace, they are in *conflict*. If Configuring discovers any conflicts during the Bringover Update transaction, it automatically activates a popup window advising you of this.

To resolve conflicts:

1. **Initiate the Resolve transaction by clicking Resolve now in the popup window to open the Resolve version of the Transactions window.**

   Configuring automatically alters the workspace icon to alert you that a workspace contains unresolved conflicts. It also:

   - Lists the path names of the files that are in conflict in the Resolve Transaction window
   - Starts the Merging program, loading the first file in the list

     Merging displays two text files (the versions of the file from the parent and child workspaces) for side-by-side comparison, each in a read-only subwindow. Each version is shown in comparison (using glyphs) to the version that existed before the changes were made. Beneath them, Merging displays a subwindow that contains a merged version. The merged version contains selected lines from either or both deltas.

2. **Merging automatically merges the files for you in the bottom window.**

   You can override the choices made by the program by using the Left and Right buttons to accept the changes found in the left or right window.

Note - If at any point during resolution, you want to reload the files, abandoning all the conflicts that have been resolved and starting over, click the Reload button. The files are reloaded from disk, and any non-conflicting differences are resolved if the Auto Merge option is selected. You can then proceed with accepting one or the other version of the remaining conflicting lines.

---

3. **When you are satisfied with the merged file, click Save to save the file.**

   If there are more files in the Transactions window conflict list, Configuring automatically loads the next file in the list into Merging.

   For more information about resolving conflicts and merging files, see Chapter 9."

Using TeamWare Versioning

Versioning is a GUI to SCCS that enables you to manipulate files and perform SCCS functions without having to know SCCS commands. It provides a way to check files in and out, and to display a file's delta history and show differences between deltas. With Versioning, you can do the following:

- Check out a version of the file for editing
- Check in files
- Retrieve copies of any version (delta) of a file
- Visually peruse the branches of an SCCS history file
- Scan a workspace for checked out files
- Scan a workspace for history files that contain unmerged deltas
- Highlight removed or unmerged deltas
- Back out changes to a checked-out copy
- Display differences between selected deltas using Merging
- Display the version log summarizing executed commands
- Create new SCCS files

Starting Versioning

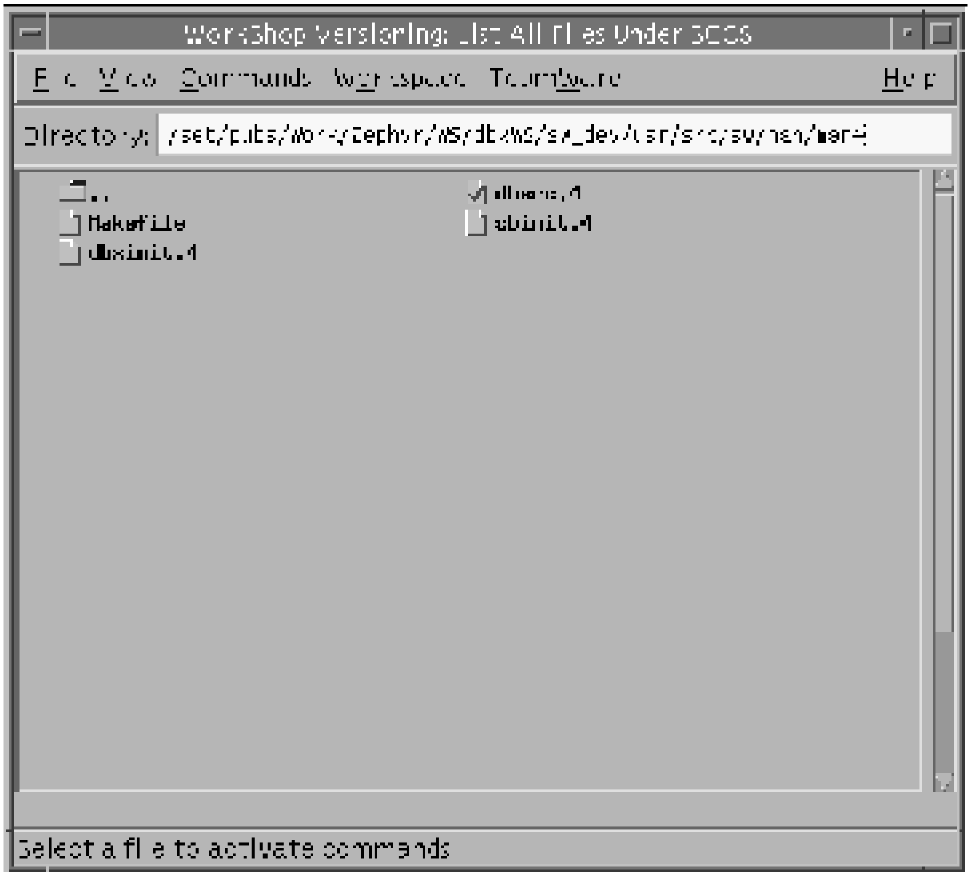
You can start Versioning in three ways:

- Type `twversion` at a shell command prompt, followed by the ampersand symbol (&).
- Choose TeamWare > Versioning in the Configuring, Merging, or Freezepointing window.
- Double-click on a workspace icon in the Configuring window.

To use Versioning, select a file (or group of files) in the File List pane of the Versioning window (see Figure 1–3) and choose a menu item to operate on it. Commands are located in the:

- Commands menu
- View menu
- File List pane floating menu

Following are two examples that describe how to use Versioning to check out and check in files, and to view and compare a file’s delta history.



Directory: /set/pubs/work/zephyr/WS/dbx/WS/sw_dev/usr/src/sv/new/mar4

..
Makefile
dbxinit.4
dbxenv.4
sbinit.4

Select a file to activate commands

Figure 1–3 Versioning Window

Checking Files In and Out of SCCS

You can check files in and out of SCCS using Versioning.

Checking a File Out

To check a file out of SCCS:

1. **If the directory that contains the file is not automatically loaded, type the directory path name (followed by Return) in the Directory text box.**

2. **Click a file icon to select a file; use the middle mouse button to extend the selection.**

3. **Choose either Commands > Checkout > Default or Commands > Checkout > Check Out, and Commands > Edit. As the file is checked out a check mark appears in its icon.**

Checking a File In

To check a file in to SCCS:

1. **When you are ready to check the file back in, select the file and choose Commands > Check In to open the Check In popup window.**

2. **Enter a comment in the text pane that describes your changes and click Check In to complete the check in process.**

   The check mark is removed from the file icon as the file is checked in.

Viewing and Comparing a File's Delta History

To view a graph of a file's delta history:

1. **Select the file's icon in the main window and choose View > File History.**

2. **Select two deltas in the graph.**

3. **Choose Differences > Use Merging.**

   Merging displays the two deltas side by side, marking differences with glyphs. For more information about Merging, see Chapter 10."

Merging Previously Unmerged Branch Deltas

To show the version history for a file:

1. **Select the file's icon in the main window and choose View > File History.**

2. **In the History window, select the two versions that you want to merge.**

3. **Click the right mouse button to open a popup menu that contains the Merge Branches command.**

4. **Choose Merge Branches from the popup menu.**

For more information about TeamWare Versioning, see Chapter 12."

Using TeamWare Freezepointing

During software development it is often useful to create freezepoints of your work at key junctures. These freezepoints serve as "snapshots" of a project that enable you to recreate the state of the project at key development points. With the Freezepointing program, you preserve these freezepoints using small storage resources. You can use Freezepointing through two functionally equivalent user interfaces:

- Graphical user interface (`twfreeze`)
- Command-line interface (`freezept`)

The concepts discussed in this section apply to both the GUI and the CLI. Descriptions and examples are included for the GUI only. For information on the CLI, see the man pages: `twfreeze`(1), `freezept`(1), and `freezepointfile`(4).

Freezepointing enables you to create freezepoint files from Configuring workspaces. Freezepointing files are text files that list the default deltas in SCCS history files contained in the workspace. When you later recreate (extract) the files, Freezepointing uses those entries as pointers back to the original history files and to the delta that was the default at the time the freezepoint file was created.

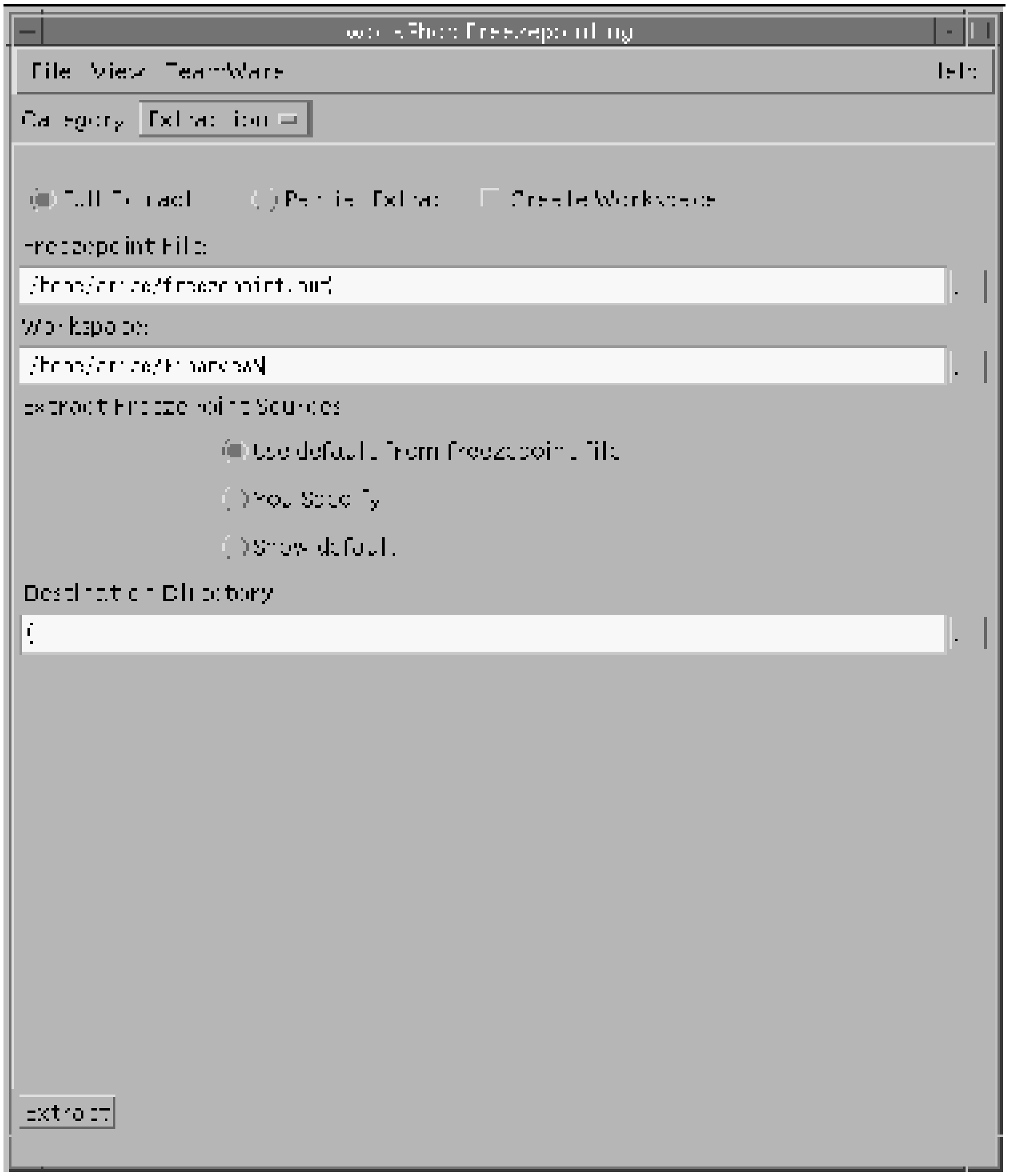
You can also create workspaces from freezepoint files. When you do this, you have the option of extracting the full set of frozen files or a partial set.

Note - Freezepointing gives you the option of recreating a workspace containing the SCCS histories for all extracted files. Alternatively, you may choose to extract the derived files only; that is, retrieve the default delta using the SCCS `get` command with no options. In this case, SCCS histories are not recreated and TeamWare does not create a workspace.

Starting Freezepointing

You can start Freezepointing by:

- Typing `twfreeze` at a shell command prompt, followed by the ampersand symbol (&)
- Choosing TeamWare > Freezepointing in the Configuring, Merging, or Versioning window



Category: Extraction

● Full Extract ○ Partial Extract □ Create Workspace

Freezepoint File: /home/[illegible]/freezepoint.out

Workspace: /home/[illegible]

Extract Freezepoint Sources

● Use default from Freezepoint File

○ You Specify

○ Show default

Destination Directory:

Figure 1–4 Freezepointing Window

Creating and Extracting Freezepoints

The pane below the Control area in the Freezepointing window (see Figure 1–4) is used for both creating and extracting freezepoints. Switch between the Create and Extract panes by choosing the appropriate item from the Category menu. The Create pane is the default and is displayed when you start Freezepointing.

Creating a Freezepoint File

To create a freezepoint file:

1. **Select Creation from the Category list box.**

2. **Type the path name of a freezepoint file in the Freezepoint File text box.**

   Freezepointing automatically inserts the file name `freezepoint.out`. Delete it and replace it with a path name of your choosing.

3. **Type the path name of the source workspace in the Workspace text box.**

   This is the workspace that you are “freezepointing.”

4. **In the Directories and Files text window, compose a list of directories, or files, or both that you wish to freezepoint.**

   Click Add to open the Add Files dialog box and select files to freezepoint. Shift-click to select more than one file. To select all of the files, type a period (.) in the Name text box. Click Add Files when you have selected all the files you want.

5. **Click Create/Update to create the freezepoint.**

Extracting a Source Hierarchy

To extract a source hierarchy from a freezepoint file:

1. **Select Extraction from the Category list box.**

   This changes the pane from Creation to Extraction.

2. **Type the path name of an existing freezepoint file in the Freezepoint File text box.**

3. **Type the path name of the destination directory in the Destination Directory text box**

   This is the directory into which the newly extracted files are placed.

4. **Click Extract to execute to extract the freezepoint.**

   For more information about FreezePointing, see Chapter 15.”

Distributed Make

Distributed Make (dmake) marks the evolution of the make utility into a powerful and flexible tool that permits you to take full advantage of the potential of today's networks and powerful multiprocessor workstations. Using dmake, you can concurrently distribute the process of building large projects, consisting of many programs, over a number of workstations and, in the case of multiprocessor systems, over multiple CPUs.

You execute dmake on a *dmake host* and distribute jobs to *build servers*. You can also distribute jobs to the dmake host, in which case it is also considered to be a build server. The dmake utility distributes jobs based on makefile targets that it determines (based on your makefiles) can be built concurrently. You can use any machine as a build server that meets the following requirements:

- From the dmake host (the machine you are using) you must be able to use rsh, without being prompted for a password, to remotely execute commands on the build server. For example:

```
demo% rsh build_server which dmake
   /opt/SUNWspro/bin/dmake
```

Note - For more information about the rsh command see the rsh(1) man page or the system AnswerBook.

- The bin directory in which the dmake software is installed must be accessible from the build server.

Note - For more information see the share(1M) mount(1M) man pages or the system AnswerBook.

- The bin directory in which the dmake software is installed must be in your execution path when you rsh to the build server. Be sure this directory is added to the PATH variable in your .cshrc file (or equivalent), *not* in your .login file. You can verify this as follows:

```
demo% rsh build_server which dmake
   /opt/SUNWspro/bin/dmake
```

- The source hierarchy you are building must be accessible from the build server.

From the dmake host you can control which build servers are used and how many dmake jobs are allotted to each build server. The number of dmake jobs that can run on a given build server can also be limited on that server.

For more information about dmake see Chapter 18."

CHAPTER **2**

Introduction

Sun WorkShop TeamWare (TeamWare) is a source management product that provides you and your team with:

- Software configuration management
- Version control
- Change control
- Integration support
- Release support

You can access TeamWare through the command-line interface (CLI) or through several graphical-user interfaces (GUIs). The GUIs and a description of what they do is in the following table:

TABLE 2–1 TeamWare GUI Descriptions

GUI	Description
Configuring	Software configuration management, previously called CodeManager.
Versioning	Version control. Versioning is a GUI for SCCS. Previously called VersionTool.
Building	Serial, parallel, or distributed make. Building is a GUI for the make utility. Previously called Maketool
Freezepointing	Obtaining a snapshot or freezepoint. Previously called FreezePoint.
Merging	Selectively merging two files. Previously called FileMerge.

CHAPTER **3**

Introduction to TeamWare Configuring

This chapter contains the following sections:

- "Coordinating the Work of Software Developers" on page 19
- "Copy-Modify-Merge Model" on page 20
- "Default Configuring" on page 22

Coordinating the Work of Software Developers

Managing large programming projects involves coordinating the work of developers who share common and interdependent files.

If developers have private copies of the source code, the changes they make to the source base are difficult to track when all of the code is finally (or even periodically) merged. Often the incompatible changes are subtle, and they can affect the entire project. Preparing the code for a final build and release can be difficult.

One solution is to allow serial access to the common files, one developer at a time. This approach eliminates conflicts due to changes that are made simultaneously. Unfortunately, this approach produces a productivity bottleneck because only one programmer at a time has access to the code.

Developers often change the way source files are grouped and used to build the intermediate and final product. A developer must know what source files, header files, and libraries are required to build a particular program. Often a developer copies a set of files, then later finds that it is incomplete. Only after repeated failed attempts to build the program is the developer able to determine which files are required to successfully build the program. Also, changes not only occur to files, but often to the file system structure as well. New files and directories are constantly created, renamed, and deleted.

Maintaining a consistent, buildable set of sources in preparation for a product release is also difficult on a large software project. When developers integrate their work directly into the mainline source hierarchy, a set of sources that built correctly one day can be made incompatible the next.

Another problem common to large software projects is the inability to recreate the product at a certain stage of development (for example, a past release). Preserving source code deltas becomes difficult when different copies of files are changed concurrently. Developers generally do not take the time to apply more than one delta. To accurately represent concurrent development, SCCS branch deltas must be used. When deltas are collapsed together, or when parallel deltas are represented sequentially, the true history of the file is lost.

Sometimes development of a feature is begun for a given release and later (often quite near the release date) a decision is made to include the feature in a different release. Backing out the changes and then including them in a different release can be difficult.

Copy-Modify-Merge Model

Sun WorkShop TeamWare assists in the development and release of large software projects. TeamWare is based on a concurrent development model called *Copy-Modify-Merge*. Isolated (per developer) workspaces form the basis of the TeamWare Configuring model. With TeamWare Configuring, you (the developer) *copy* source you want to change from a central workspace into your own workspace, *modify* the source to your liking, and then *merge* your changes with changes made by other developers in the central workspace.

Note - A workspace is simply a specially designated SunOS™ directory and its subdirectories.

The inconvenience of merging changes is outweighed by the productivity increase that results from developers working concurrently. TeamWare Configuring is designed to minimize (and in some cases, eliminate) the inconvenience of merging changes.

Besides providing isolated workspaces, TeamWare Configuring enables you to easily and “intelligently” copy files between workspaces and then merge changes that exist between corresponding files. Configuring’s “intelligent” copy feature enables you to copy project files in groups that you (or the project administrator) determine are logically linked; it also automatically determines for you whether differences exist between the files in the originating workspace and the destination workspace.

You copy project files from a central workspace into your own private workspace, make changes to files (or the file system), and then copy your changes back to the central workspace. You can group source files, header files, libraries, and so on, together in logical units that are copied in unison; TeamWare Configuring further assists the concurrent development process by determining whether differences exist between the files in the originating workspace and the destination workspace. If differences *are* found to exist, Configuring prevents you (or another developer) from copying over those changes. Configuring then provides sophisticated window-based tools that help you to merge these differences.

Copy-Modify-Merge Example

The following is a Copy-Modify-Merge concurrent development model employed by TeamWare Configuring. This example describes a common software development scenario where two developers are working simultaneously on the same or related parts of a project.

TABLE 3–1 Copy-Modify-Merge Example

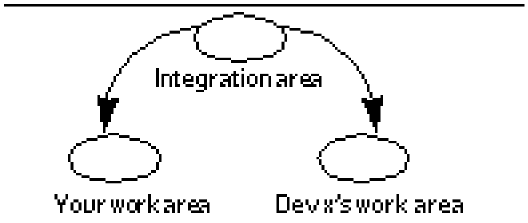
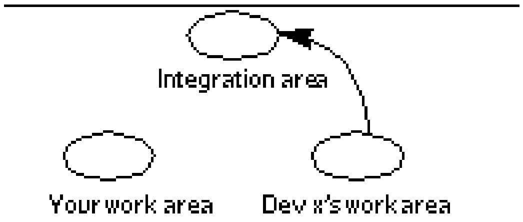
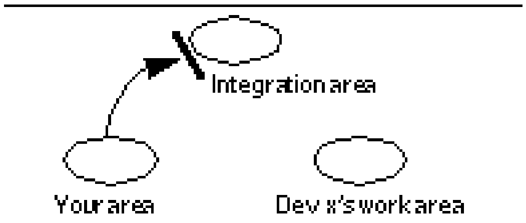
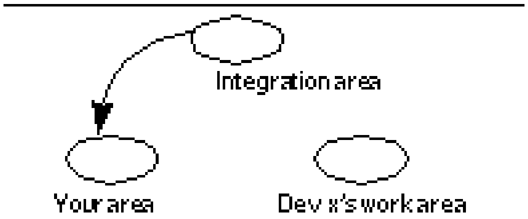
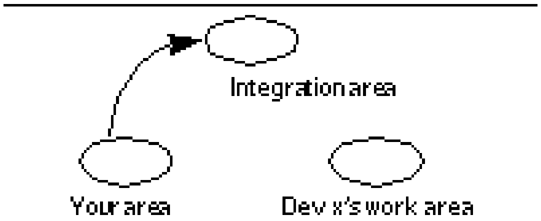
	■ Both you and Developer x copy the same file from the project integration area to your separate work areas.
	■ Developer x changes the file and copies the changed file back into the integration area.
	■ You modify the same file in your work area and attempt to copy the file back into the integration area. TeamWare Configuring blocks your attempt to copy since it would overwrite Developer x's changes.

TABLE 3–1 Copy-Modify-Merge Example *(continued)*

	- Configuring informs you of the conflicting changes and you copy the file containing Developer x's changes from the integration area to your work area.
	- With Configuring's assistance, you resolve the conflicts, merge the changes, test the changes, and successfully copy the file back to the integration area.

Default Configuring

TeamWare Configuring can be customized in ways that modify its default behavior; many of those customizations are discussed in Chapter 7". All source files in a Configuring project are maintained under the UNIX SCCS. *TeamWare Configuring only copies files that are under SCCS.* Within your workspaces, you use SCCS in the normal way. For example, you:

- Create files
- Create deltas
- Edit files
- Add comments
- Check in files using SCCS commands

SCCS history files are in SCCS subdirectories, as they would be if the project were not using Configuring. When you copy files between workspaces and merge files that have changed, TeamWare Configuring manages SCCS history files for you, preserving all comments and deltas.

Workspace

The *workspace* is the basis of the Configuring system. The workspace provides the isolation in which developers work concurrently with other developers programming in other workspaces. Project files are propagated between workspaces by Configuring commands. The workspace is a directory and its subdirectory hierarchy. When the workspace is created, Configuring creates a special subdirectory under the workspace, called `Codemgr_wsdata,` to store workspace information.

A Configuring project is created in a top-level workspace from which all others are derived. When other workspaces are created from the original workspace, the original file system hierarchy is recreated to form the new workspace. In the following example, work is begun by a developer in a workspace whose top-level directory is `boatspex`. The workspace exists under the directory `/usr/src/ws`.

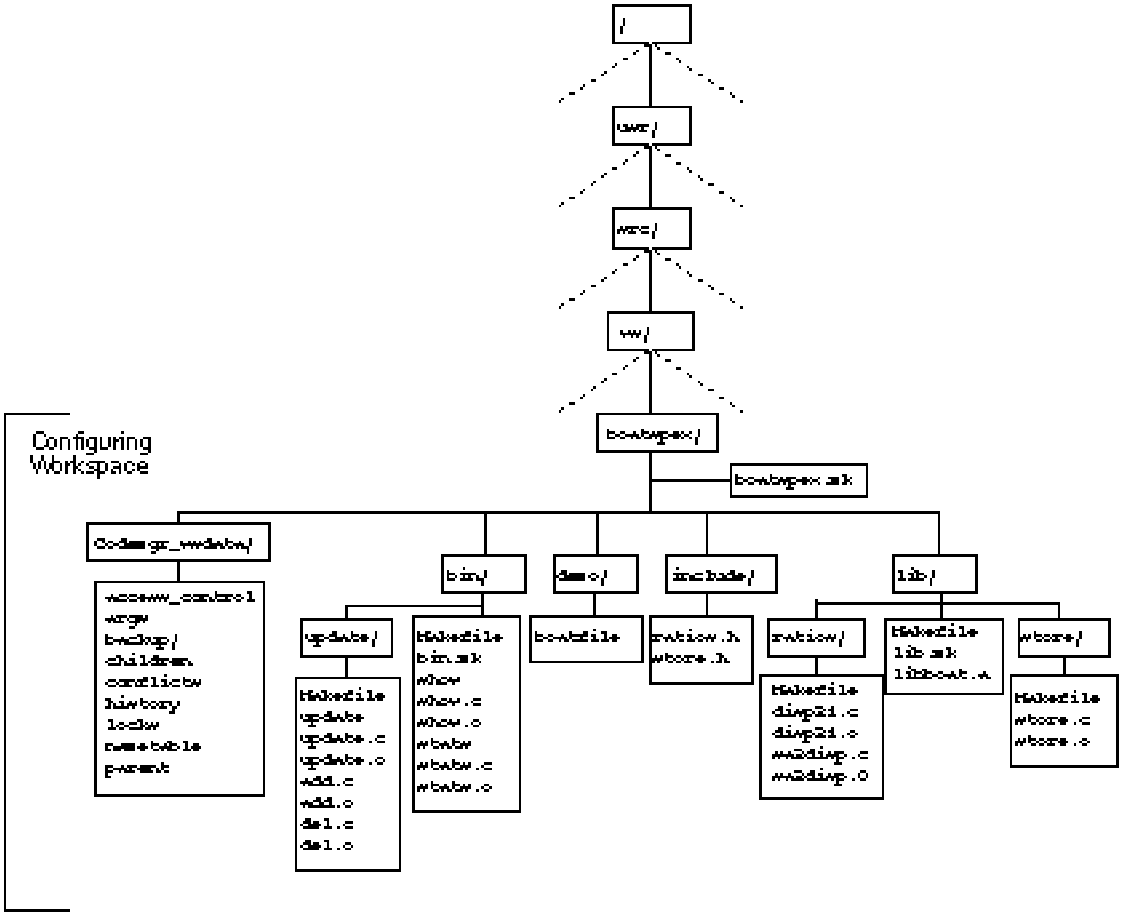


Figure 3–1 Project File System Hierarchy

If you are assigned to work on the Boatspex project you create a copy of the original workspace in a file system of your choice; the workspace portion of the file system in the new workspace is identical to that of the original workspace. If you create the new workspace in your home directory, it appears something like Figure 3–2.

Note - If you were only working on a portion of the project, you could copy only that portion.

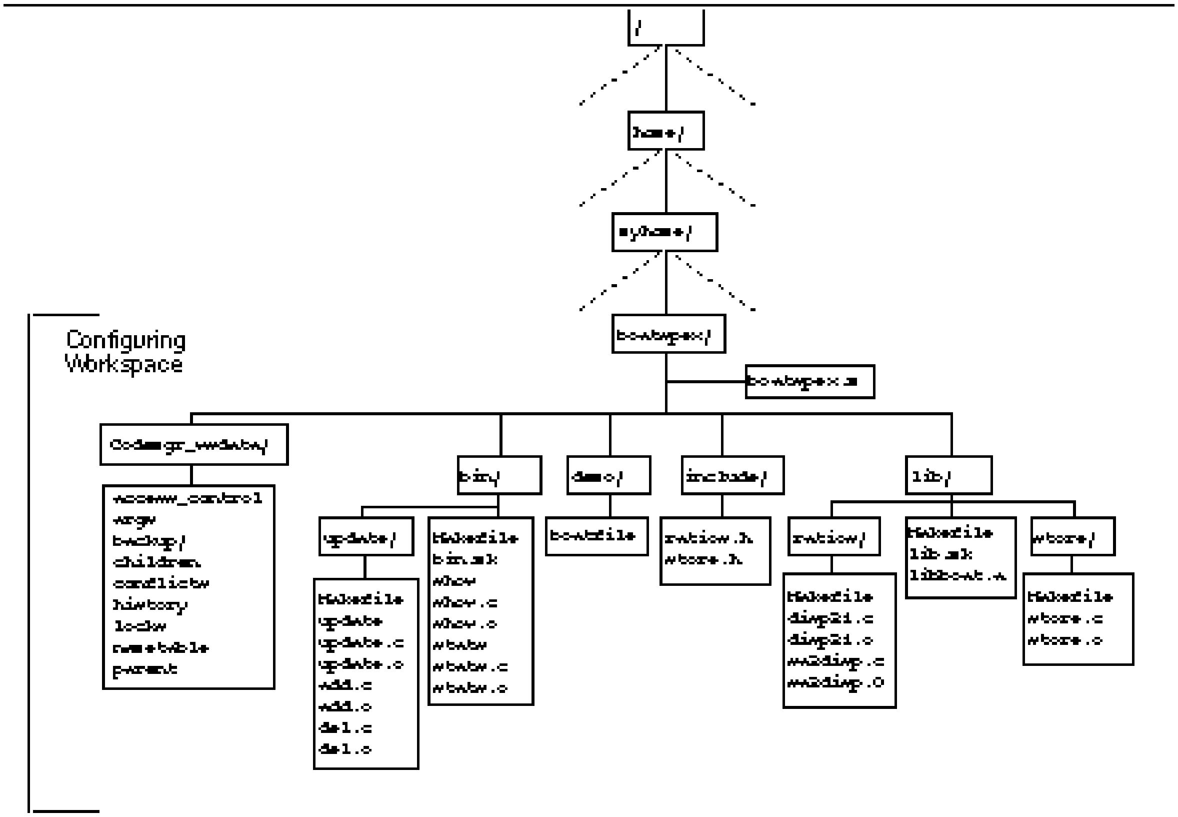


Figure 3–2 Your New Workspace

The directories previous the workspace directory (`boatspex`) are variable. They change depending on where in the file system you locate the workspace. Below the workspace directory, the file system is a duplicate of the original workspace

Parent and Child Relationship

When you copy files from a workspace to create a new workspace, a special relationship is created between the original workspace and the new one. The original workspace is considered the *parent* of the newly created *child* workspace. You can acquire files from any Configuring workspace in this manner, and workspaces can have an unlimited number of children. The portion of the file system that you copy from the parent workspace is determined at the time you copy it. You can copy the entire contents of the parent to the child, making it a clone of the parent, or you can copy only portions of the file system hierarchy that are of interest to you. The Configuring transaction used to copy files from a parent workspace to a child workspace is called Bringover.

Note - If you use the Bringover transaction to copy files to a workspace that does not already exist, the transaction creates a new child workspace and then copies files to it. This special case is called a *Bringover Create* transaction. You use the *Bringover Update* transaction to update an existing child workspace.

The parent and child relationship is special because project data is exchanged only between parent and child workspaces. All files contained in a child workspace were either brought over from a parent workspace or created in the child workspace. When development and testing are complete in the child, you can copy the files that were modified or added in the child back into the parent workspace. Once the altered files are present in the parent, they can be copied by other children or passed up another level to the parent's parent workspace. The Configuring transaction for copying files from a child workspace to a parent workspace is called Putback.

Note - Unless the child is itself a parent, in which case new files can also be copied to it from its children.

Workspace hierarchies are formed by repeating Bringover transactions to create child workspaces. The hierarchy of parent and child workspaces forms a pathway through which data is moved throughout the project.

In the following example, a project is originally created in a workspace and then a three-level workspace hierarchy is created by means of the Bringover transaction. The original workspace is considered to be the parent of the integration workspace and, conversely, the integration workspace is considered to be the child of the original workspace. Developers (Jon, Jack, and Jill) then use the Bringover Create transaction, shown in Figure 3–3 to create child workspaces from the integration workspace, which forms a three-tiered hierarchy of workspaces.

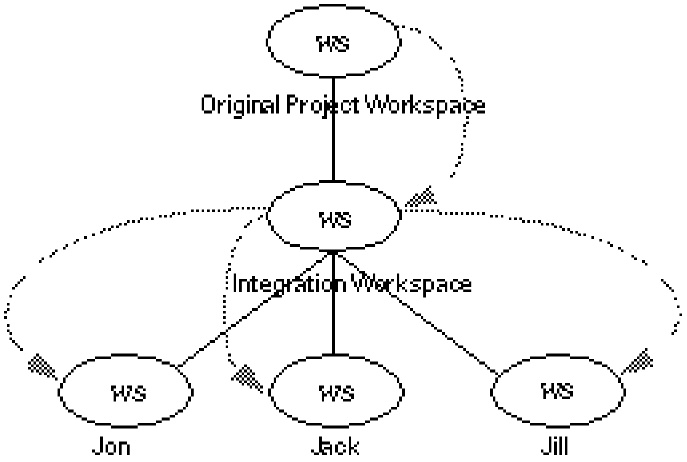


Figure 3–3 Using the Bringover Create Transaction to Create a Workspace Hierarchy

In this hierarchy, files can be disseminated from Jon's workspace to its "sibling" workspaces owned by Jack and Jill. Jon uses the Putback transaction to copy modified files from his workspace into the common parent (step 1) and then Jack and Jill use the Bringover Update transaction to copy the files from the parent into their workspaces (step 2), shown in Figure 3–4.

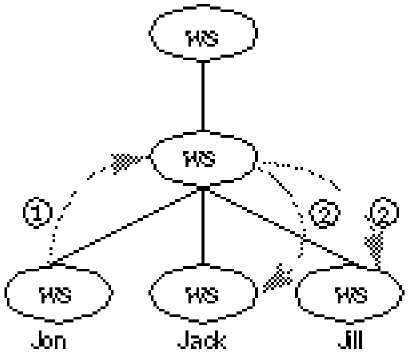


Figure 3–4 Copying Files between Workspaces

Reparenting

Parent and child relationships can be changed. Configuring permits child workspaces to be "reparented" to new parent workspaces. Reasons that you might want to reparent a workspace include the following:

- To reorganize workspace hierarchies
- To populate a new project hierarchy (new top-level workspace)

- To move a feature into a new release
- To apply a bug fix to multiple releases

Refer to “Reparenting a Workspace” on page 70” for more information

Codemgr_wsdata Directory

Every Configuring workspace contains a directory named Codemgr_wsdata that is a subdirectory of the workspace top-level (root) directory. This directory contains text files that Configuring uses to log its actions, and store temporary and permanent data. You can view and alter these files using standard text utilities. Refer to “The Workspace Metadata Directory ” on page 65” for more information.

Modifying Files

Since Configuring workspaces are simply directories within the SunOS file system, all your usual tools and utilities can be used on files and directories in workspaces. Your normal edit/compile/debug process is not altered by Configuring.

Copying Files between Workspaces

Once you make and test modifications in a child workspace, you must disseminate them to the rest of the developers working on the project and ultimately to an integration/release workspace.

Every developer in a project needs up-to-date data with which to work. If a modification is made to a module in one part of the project, it could have profound implications for the testing of a different module in another part of the project. Perhaps even more important is the sharing of information between developers working on the same or closely related modules.

Newly modified files (or groups of files) are transferred between parents and children up and down the workspace hierarchy in order to keep workspaces consistent. The decision as to when the data is ready for dissemination is, of course, left to the developer’s discretion.

The Putback and Bringover transactions are generally applied to groups of files so that files need not be specified individually. Configuring provides the means for you (or your project administrator) to specify groupings of files that should logically be copied together. Three examples of this type of grouping are as follows:

- Directories
- Files required to build a particular program
- All of the child workspace

How files are grouped for Bringover and Putback transactions between workspaces is discussed in detail in Chapter 8."

Bringover and Putback transactions are always initiated from within the child workspace. Both transactions are viewed from the perspective of the child workspace—not the parent's.

Source Code Control System Files

When considering Bringover and Putback transactions, remember that source files are derived from SCCS deltas and are identified by SCCS delta IDs (SIDs). When a file is copied by either a Putback or Bringover transaction, Configuring is manipulating the file's SCCS history file (also known as the s-dot-file).

When a file is copied from one workspace to another, Configuring decides how to manipulate the SCCS history file used to derive the file. If the file does not exist in the target workspace, Configuring copies the history file from the source workspace to the target. In the more complicated case—when the file (and thus the SCCS history file) exists in both the source and the target—the SCCS history files must be merged to maintain the file's delta, administrative, and comment history.

Remember, files consist of both the file derived from the latest delta and its predecessors by the SCCS `get` command and the SCCS history file from which it is derived. When files are copied from workspace to workspace, SCCS history files are adjusted appropriately.

Bringover and Putback Transactions

When you initiate a Bringover Update or Putback transaction, Configuring must make a number of determinations before taking any action. Copying files indiscriminately from one workspace to another could overwrite work that you or another developer want to keep. Configuring must check all files specified for transfer to determine where they stand in relationship to each corresponding file in the other workspace.

For example, suppose a file was modified in the parent (perhaps put back from another child) since it was last brought over into your child. You have modified your copy of the same file in your child workspace. When you attempt to put back that file (or a group of files that contains that file) from your child workspace to the parent, Configuring will not allow your Putback transaction to proceed because it would cause the revised version of the file in the parent to be overwritten by the version of the file from your child. In this case, Configuring blocks your attempt to put back the files into the parent and informs you of the conflicting change.

When a Putback or Bringover Update transaction is blocked, none of the files in the group are copied, even those that don't conflict.

The conflicts between your versions of the files and the versions in the parent must be resolved in your (child) workspace. *Conflicts are always resolved in the child workspace to preserve the integrity of the parent.*

You use the Bringover Update transaction to copy the conflicting files from the parent to your workspace, and using Configuring's merge tool, you merge your changes with those made by the other developer. After testing the changes you then put back the merged files to the parent workspace.

TABLE 3–2 Keeping Work Synchronized

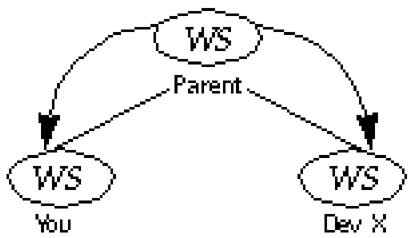
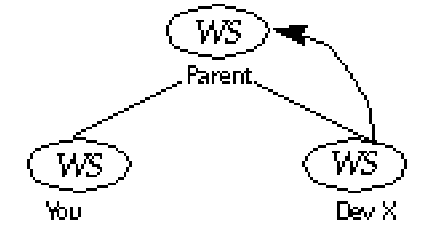
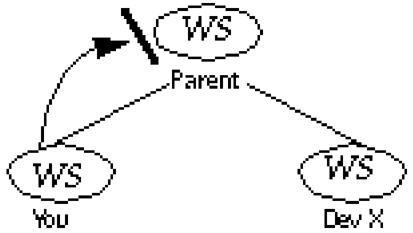
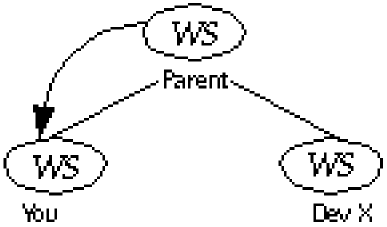
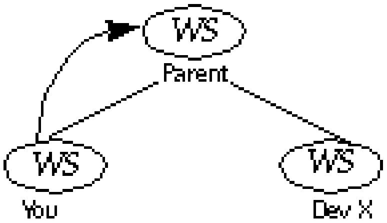
	■ Both you and Developer x bring over the same file to your workspaces.
	■ Developer x changes the file and puts the changed file back into the parent.
	■ You change the same file in your workspace and attempt to put the file back into the parent. Configuring blocks the Putback.

TABLE 3–2 Keeping Work Synchronized *(continued)*

	■ Configuring notifies you of the conflicting changes and you bring the file over to your workspace (actually, the SCCS history files are merged).
	■ You resolve the conflict, test the changes and successfully put back the file back to the parent workspace.

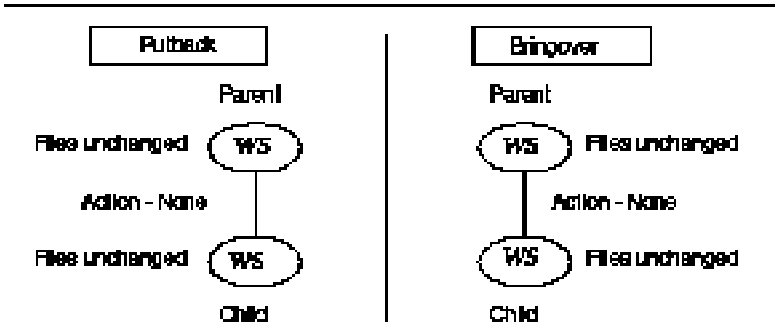
Relationships between Files in Parent and Child Workspaces

The previous example describes only one of four possible states of relationship that can exist between corresponding files in parent and child workspaces. The relationship between files in parent and child workspaces governs the way that Configuring behaves when you attempt to copy files via Putback and Bringover Update transactions. Following are descriptions of the four cases and the action Configuring takes in each case:

Case 1

Neither the files in the parent nor the corresponding files in the child have been modified since they were put back into the parent or brought over into the child

In this case no action is required by Configuring in either case. The files are exactly the same in both the parent and child.



Case 2

The specified files were not modified in the parent since they were brought over from the parent into the child or put back from the child into the parent. The corresponding files *were* modified in the child.

In this case when you use the Putback transaction to copy the file to the parent, the changed files are updated from the child into the parent, replacing the corresponding files in the parent. This new data is available for acquisition by other children of that parent or to be further propagated up to the parent's parent workspace.

When you use the Bringover Update command in this case, no action is taken because copying the file from the parent would overwrite changes made in the child.

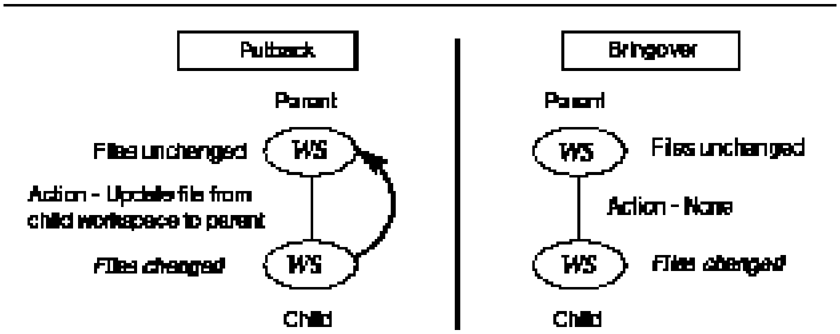
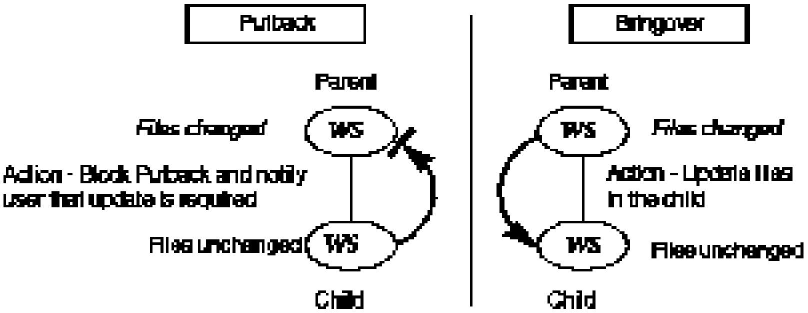


Figure 3–5 Case 3

One or more files in the parent were modified since their corresponding files were brought over into the child or put back into the parent from that child. The corresponding files in the child were not modified.

In this case the parent's copy of the file being put back from the child was modified (probably by one of its other children) since it was last brought over to the child; the corresponding file in the child was not modified since it was last brought over into the child.

When Configuring detects this situation during the Putback transaction, it cannot update the parent workspace until the child workspace is updated by means of the Bringover Update transaction. Even if the changes are in files that you have not altered (remember you're copying *groups* of files), they might impact the changes you have made. In this case, the Putback transaction is blocked and the user is notified. It is the user's responsibility to execute the Bringover Update transaction in order to update the child workspace.



Case 4

Corresponding files were modified in both the parent and child workspaces.

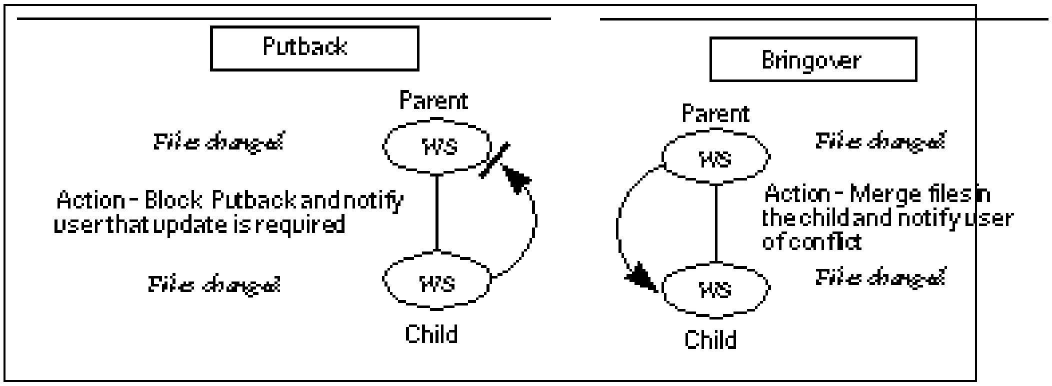
This is the most complicated of the four cases. Configuring cannot allow the file to be put back from the child into the parent because the transaction will obscure modifications there. Likewise, Configuring cannot allow the file to be brought over from the parent into the child because the transaction will overwrite modifications there.

As in case 3 above, Configuring blocks the Putback transaction and notifies the user. When the user attempts to update the child workspace by means of the Bringover transaction, Configuring detects that the file in the child has also been changed; the file cannot be updated without overwriting the newly created work in the child. In this case Configuring merges the parent and child SCCS history files for the conflicting file in the child workspace.

Configuring merges the parent and child SCCS history files together in the child workspace; the SIDs that were created in the child are renamed and placed on an SCCS branch off of the current line of work brought down from the parent. Although it is a branch, the child's SCCS version tree remains the default for any additional deltas so that work on the file may proceed in the child as if nothing had changed.The merge process places all needed deltas in the SCCS history file so that the conflicting files can be merged at the user's discretion. All SCCS comments are preserved in this process since the entire SCCS delta history is preserved.

At this point the conflict between the parent and child versions of the file is still open. Work can continue on the branch that contains the deltas created in the child;

any new deltas will be added to the branch. *However, the user must resolve the conflict before the group of files that contain the conflicting file(s) can successfully be put back to the parent.* Conflict resolution is discussed in "Resolving Conflicts" on page 34.


Putback
Parent
File changed
WS
Action - Block Putback and notify user that update is required
File changed
WS
Child
Bringover
Parent
WS
File changed
Action - Merge files in the child and notify user of conflict
WS
File changed
Child

Summary

The following two tables summarize, first, the action taken by Configuring during a Putback transaction in each of the four cases described above and secondly, for the Bringover transaction.

TABLE 3–3 Putback Transaction

Case	File in Parent	File in Child	Action by Configuring
1	Unchanged	Unchanged	None
2	Unchanged	Changed	Update file in parent
3	Changed	Unchanged	Block Putback, notify user
4	Changed	Changed	Block Putback, notify user

TABLE 3–4 Bringover Transaction

Case	File in Parent	File in Child	Action by Configuring
1	Unchanged	Unchanged	None
2	Unchanged	Changed	None

TABLE 3–4 Bringover Transaction *(continued)*

Case	File in Parent	File in Child	Action by Configuring
3	Changed	Unchanged	Update child (extend SCCS files)
4	Changed	Changed	Merge SCCS history files and notify user of conflicts

Resolving Conflicts

During the Putback transaction, Configuring may determine that a file in the parent has been modified since it was last put back from that child or brought over into the child. In that case it blocks the Putback so that the changes are not overwritten and then notifies the user of the potential conflict.

Generally the owner of the child workspace will then attempt to update the child by bringing over the changed file. If, during the Bringover Update transaction, Configuring determines that the corresponding file in the child has *also* been modified since it was last brought over, a conflict exists.

Conflicts arise when corresponding files in both the parent and child have been modified. If Configuring were to overwrite either of the files, a loss of data would result. Before the specified file can be put back or brought over the user must resolve any conflicts.

When Configuring detects a conflict during the Bringover Update transaction, as described in the previous section, it then does the following:

- Merges the parent and child SCCS history files for the conflicting files in the child workspace
- Notifies the user of the conflict
- Assists the user in resolving the conflict

Note - All conflicts are resolved from within the child workspace and from the perspective of the child workspace.

In the case of most conflicts, the options available to the user for resolving conflicts are:

- Install the latest delta from the parent as the resolved version in the child.
- Accept the latest delta from the child as the resolved version of the file. Since it has been through the resolve process, its Putback transaction will no longer be blocked in the parent.

- Merge the contents of latest delta from the parent with that of the child.

Configuring provides tools that aid in resolving conflicts, however, the conflicts must be resolved by the user. Refer to Chapter 9,” for a detailed discussion about conflict resolution.

CHAPTER **4**

Introduction to Merging

This chapter deals with the basic concept of Merging files, and is organized into the following sections:

- "Differences Defined" on page 38
- "Merging Window" on page 39

For explicit details on using Merging, together with an example, see Chapter 13."

Merging loads and displays two text files for side-by-side comparison, each in a read-only text pane. Merging marks lines that differ between the two files and displays a merged version in a third text pane. When automatically activated, the merged version contains two types of lines:

- Lines that are common to both input files (these lines always appear in the output file)
- Marked lines that are different in each file (these lines appear as the result of the default automerge process)

You can edit the merged version and save it as an output file.

At the time you load the two files to be merged, you can also specify a third file, called the *ancestor* of the two files (which are called its *descendants*). When you have specified an ancestor file, Merging marks lines in the descendants that are different from the ancestor and produces a merged file based on all three files. To automatically merge (automerge) the two input files, you must specify an ancestor file.

Differences Defined

Merging operates on *differences* between files. Although you probably have a good intuitive grasp of what a difference is, the following describes how Merging recognizes and classifies differences.

Difference

When Merging discovers a line that differs between the two files to be merged (or between either of the two files and the ancestor), it marks with glyphs the lines in the two files. Together, these marked lines are called a *difference*. While Merging is focusing on a difference, it highlights the glyphs.

Current, Next, and Previous Difference

The difference on which Merging is focusing at any given time is called the *current* difference. The difference that appears immediately later in the file is called the *next* difference; the difference that appears immediately earlier in the file is called the *previous* difference.

Resolved and Remaining Difference

A difference is *resolved* if either you or Merging accept the changes to a line. Differences are resolved one of two ways:

- While focusing on a difference, you can accept a line from one of the original files, or you can edit the merged version by hand. When you indicate that you are satisfied with your changes (by clicking on a command button), the current difference is then resolved.
- If the Auto Merge feature is on, Merging resolves differences automatically. For more information on how Merging resolves differences, see the discussion in "Merging Glyphs" on page 41.

After a difference is resolved, Merging identifies it by changing its associated glyphs from solid to outline font. Merging then automatically advances to the next difference (if the Auto Advance property is on) or moves to the difference of your choice.

A *remaining* difference is one that has not yet been resolved.

Merging Window

The graphical interface for Merging consists of the Merging window, in which you do most of your work.

Merging Window at Startup

Figure 4–1 shows the Merging window with the left and right text panes at the top displaying the files to be compared; the text pane at the bottom displays a merged version of the two files that you can edit.

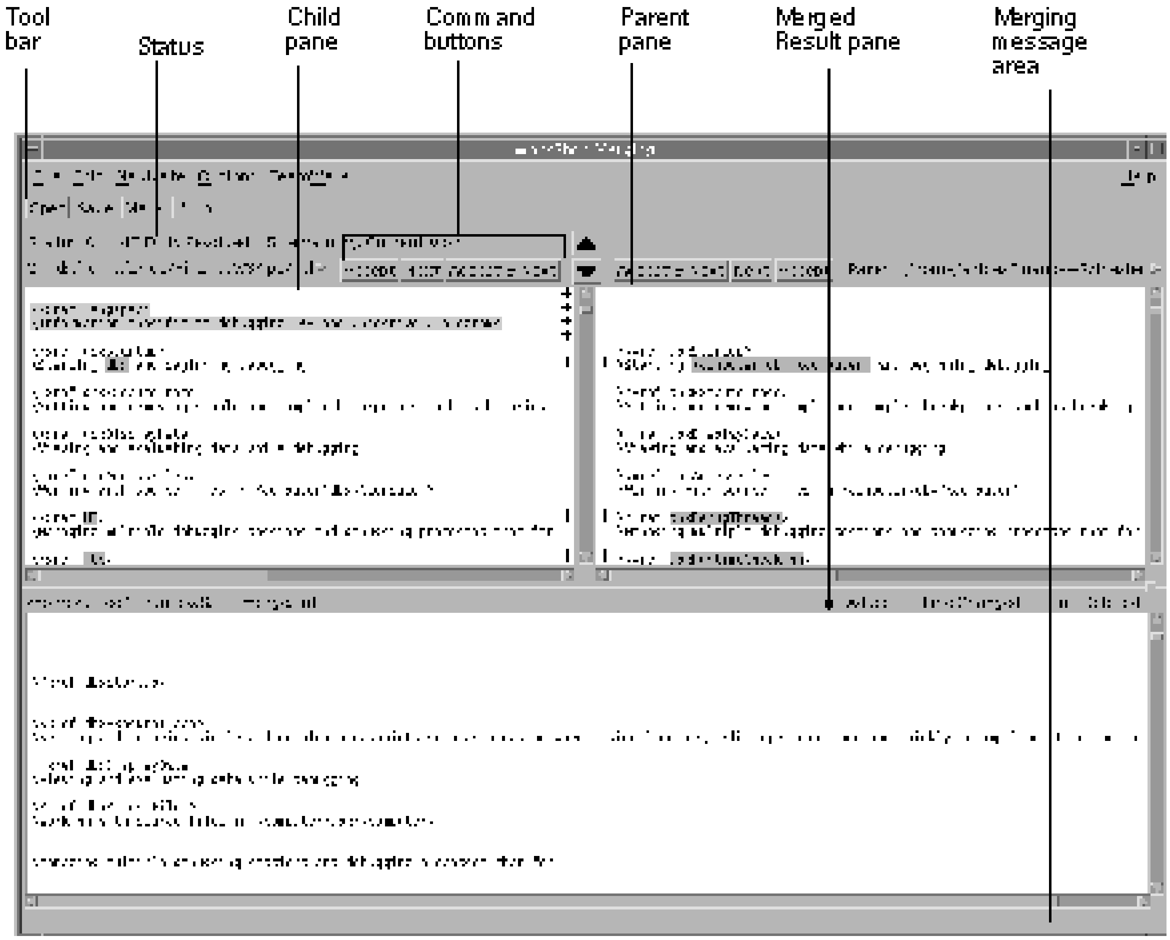


Figure 4–1 Merging Window at Startup

File menu	Provides commands for opening and saving files
Edit menu	Provides commands for managing differences between files
Navigate menu	Provides commands for moving among differences in both files
Option menu	Provides commands for setting merging, scrolling, display, and diffing options
TeamWare menu	Provides command for starting other TeamWare tools
Tool bar	Provides quick access to file management operations

Status field	Displays information about the state of your merge
Child pane	Shows the child file with markings
Command buttons	Provide quick access to navigating and editing functions
Parent pane	Shows the parent file with markings
Merged Result pane	Shows you the combined file with markings
Merging message area	Displays messages about operating in the Merging window

Merging Glyphs

When files are loaded in the text panes, glyphs appear to indicate the disparities. There is a difference between two files being merged without a common ancestor, and two files that have a common ancestor (this case is actually a three-way merge). The meaning of the glyphs in each case is slightly different, as explained below.

Two Input Files

When only two files have been loaded into Merging, lines in each file are marked by glyphs to indicate when they differ from corresponding lines in the other file:

- If two lines are identical, no glyph is displayed.
- If two lines are different, a vertical bar (|) is displayed next to the line in each input text pane, and the different characters are highlighted.
- If a line appears in one file but not in the other, a plus sign (+) is displayed next to the line in the file where it appears, and the different characters are highlighted.

Three Input Files

When an ancestor file has been specified for the two files to be merge, lines in each descendant are marked according to their relationship to the corresponding lines in the common ancestor:

- If a line is identical in all three files, no glyph is displayed.
- If a line is not in the ancestor but was added to one or both of the descendants, a plus sign (+) is displayed next to the line in the file where the line was added, and the different characters are highlighted.
- If a line is present in the ancestor but was removed from one or both of the descendants, a minus sign (-) is displayed next to the line in the file from which

the line was removed, and the different characters are highlighted and in strikethrough.

- If a line is in the ancestor but has been changed in one or both of the descendants, a vertical bar (|) is displayed next to the line in the file where the line was changed, and the different characters are highlighted.

Resolved differences are marked by glyphs in outline font.

Table 4–1 summarizes the automerging algorithm. Ancestor is the version of a text line that is in the ancestor file; Change 1 is a change to that line in one of the descendants; Change 2 is another change, different from Change 1. Only when a line is changed differently in the left and right descendants does automerging fail.

TABLE 4–1 Automerging Rules Summary

Left Descendant	Right Descendant	Automerged Line
Ancestor	Ancestor	Ancestor
Change 1	Ancestor	Change 1
Ancestor	Change 2	Change 2
Change 1	Change 1	Change 1
Change 1	Change 2	No Automerge

When Merging automatically resolves a difference, it changes the glyphs to outline font. Merging lets you examine automatically resolved differences to be sure that it has made the correct choices.

You can disable automatic merging by deselecting Auto Merge on the Options menu. When automatic merging is disabled, Merging constructs a merged file using only lines that are identical in all three files and relies on you to resolve the differences.

If you do not specify an ancestor file, Merging has no reference with which to compare a difference between the two input files. Consequently, Merging cannot determine which line in a difference is likely to represent the desired change. The result of an automerge with no ancestor is the same as disabling automatic merging: Merging constructs a merged file using only lines that are identical in both input files and relies on you to resolve differences.

CHAPTER **5**

Starting a Project

When you begin to use Sun WorkShop TeamWare, you are probably moving an existing software development project to TeamWare or you may be starting a new software development project. This chapter contains information about:

Moving an Existing SCCS-Based Project to Sun Workshop TeamWare

To move an existing SCCS-based software project to TeamWare do the following:

1. **Ensure that all SCCS history files (s-dot files) are in directories named** `SCCS` **and that these files are directly beneath directories containing source files.**

   Configuring works only on files under SCCS version control.

2. **Be sure that your project directory structure is current and organized.**

3. **Choose File > Create Workspace, specifying the top-level directory as your workspace.**

   The Create Workspace command creates the `Codemgr_wsdata` directory under the top-level directory.

4. **Use the Bringover Create transaction to form a workspace hierarchy (see "Creating a Workspace From an Existing Hierarchy of Files " on page 4).**

   See "Configuring a Workspace Hierarchy" on page 47 for guidelines regarding workspace hierarchies.

If your project is structured so that compilation units can be easily grouped on a directory basis during transfer operations, you can use the default Configuring FLP. See "Grouping Files for Transfer Using File List Programs" on page 89 for a description of the default FLPs.

If your project requires files to be grouped for transfer operations in special ways, you will have to write your own FLP(s).

Moving a Non-SCCS Software Development Project to Sun Workshop TeamWare

To move an existing project into Sun Workshop TeamWare:

1. **Choose File > Create Workspace to create your project's top-level directory (with its `Codemgr_wsdata` directory)**

2. **Change to the directory where the existing development files live.**

3. **Choose TeamWare > Versioning.**

4. **In the Versioning window, choose Commands > Check In New**

   In the Check In New Files dialog box, select the files you want to include in the development project, and type an initial comment in the Initial Comment text box.

Converting an RCS Project to Sun Workshop TeamWare

`rcs2ws` is a program that produces a Configuring workspace from an RCS source hierarchy. It converts a project developed in RCS (Revision Control System) and works its way down through the hierarchy to convert the RCS files to SCCS.

`rcs2ws` operates on RCS files under the parent directory and converts them to SCCS files, then puts the resulting SCCS files into a workspace. If a workspace doesn't already exist, it will be created. The parent directory hierarchy is unaffected by `rcs2ws`.

`rcs2ws` searches directories recursively.

To convert files, `rcs2ws` invokes the RCS `co` command and the SCCS `admin`, `get` and `delta` commands. These commands will be found using your `PATH` variable. If `rcs2ws` cannot find the SCCS commands, it looks for them in the `/usr/ccs/bin` directory.

`rcs2ws` does not convert RCS keywords to SCCS keywords. Keywords are treated as text in the SCCS delta.

Using rcs2ws

To use `rcs2ws`:

1. **Type** `rcs2ws -p [`***parent_dir***`] -n [files | directory]` **at the prompt.**

   The `-p` option is required to name the RCS source hierarch.

   The `-n` option allows you to see what would be done without actually doing anything.

Relative file names are interpreted as being relative to *parent_dir*.

The `CODEMGR_WS` environment variable contains the name of a user's default workspace. This workspace is automatically used if the `-w` option is not specified. If you wish to designate the child workspace, use the `-w` option. See man `rcs2ws(1)`.

If the current directory is contained within a workspace, the containing workspace is used as the child workspace. If workspace does not exist, `rcs2ws` creates it.

Note - Use the "." directory to specify that every RCS file under *parent_dir* should be converted.

When all is as you wish, go on to Step 2.

1. **Type the** `rcs2ws` **command using only the** `-p` **option.**

   Normally, SCCS gets (g-files) are extracted after the files are converted from RCS. If you want to halt the process, do the conversion using the `-g` option. (`rcs2ws -g`)

Starting a New Software Development Project with Sun Workshop TeamWare

To start a new project with Sun Workshop TeamWare:

1. **Choose File > Create Workspace to create your project's top-level directory (with its** `Codemgr_wsdata` **directory)**

2. **Proceed as you would to set up an SCCS-based development hierarchy.**

   Ensure that all SCCS history files ("s-dot-file") are in directories named `SCCS` located directly beneath directories that contain source files.

3. **Use the Bringover Create transaction to form a workspace hierarchy (see "Creating a Workspace From an Existing Hierarchy of Files " on page 4).**

   See "Configuring a Workspace Hierarchy" on page 47, for guidelines regarding workspace hierarchies.

   Note - The default Configuring FLP groups files recursively by directory; if you intend to use that FLP, be sure to arrange files in compilation units accordingly. If your project requires that files be grouped differently during transfer, be sure to arrange your project hierarchy in such a way that it works well with the FLP(s) you will create.

Configuring a Workspace Hierarchy

How you configure a project workspace influences the interworkspace file-transfer process and the way you prepare product releases. This section will help you choose the workspace hierarchy best suited for your project. You can change any decisions you make now regarding workspace hierarchies by using the Configuring workspace reparenting feature. See "Reparenting a Workspace" on page 70 for details.

A workspace hierarchy is a chain of parent and child workspaces that is two or more layers deep. The number of layers in a hierarchy bears no relation to the number of workspaces comprising it. A parent workspace and its child comprise two layers. A parent workspace and three children also comprise two layers. A parent workspace and its child and grandchild comprise three layers. Figure 5–1 depicts a "flat" (three-tiered) hierarchy, and Figure 5–2 shows a "multitiered" (four-tiered) hierarchy.

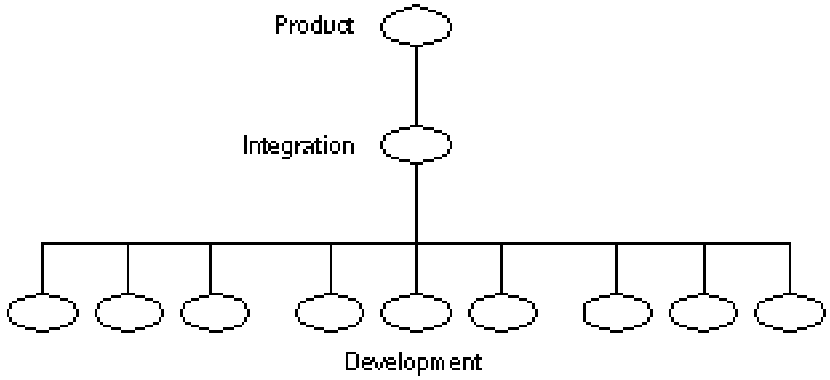


Figure 5–1 A "Flat" (Three-Tiered) Hierarchy

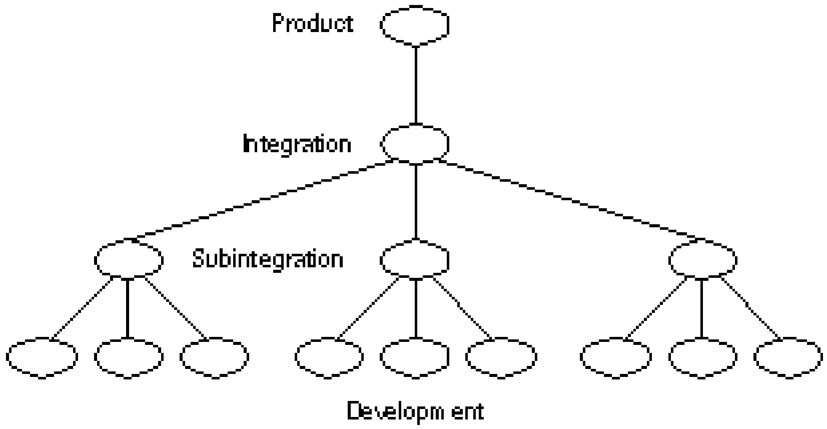


Figure 5–2 A “Multitiered” (Four-Tiered) Hierarchy

File Transfer Considerations

The way in which you set up your workspace hierarchy can have an impact on the transfer of files up and down the hierarchy.

File System Accessibility

In order to transfer (Bringover/Putback) files between workspaces, both the parent and the child must be mounted on the same file system. The automounter can be used to connect file systems.

Flat Hierarchy vs. Multitiered Hierarchy

To properly design your workspace, you need to be aware of the advantages and disadvantages of flat and multitiered hierarchies.

Advantages of a Flat Hierarchy

A flat workspace hierarchy is one in which many developers put back files to a single integration workspace. The advantage of a flat hierarchy is that all developers have immediate access to one another’s work. The moment that Jack (a developer) puts back his work to the integration workspace, Jon (another developer) can use the Bringover Update transaction to have immediate access to the changes made by Jack.

Disadvantages of a Flat Hierarchy

The disadvantage of a flat hierarchy is that time is often wasted because the integration workspace changes frequently, requiring developers to do frequent Bringover transactions, builds, and tests in order to keep their source base up-to-date. There is a cumulative effect of doing Putback transactions; the first developer to do a Putback resolves only one set of changes, the next developer resolves two, and so on till the last developer, who must resolve all of the changes that have been made within her development group.

Advantages of a Multitiered Hierarchy

The amount of time required for a developer to put her work back to the integration workspace can be sharply reduced by interposing a tier of subintegration workspaces between the integration and development level workspaces.

Whenever a developer puts back work to an integration workspace, there is some chance that the next developer to do a Putback transaction will not be able to put back their changes until they bring over the earlier changes, rebuild the modules, and test the new changes with their own—the more Putbacks that occur the higher the potential for conflict.

When many developers work on a project, the Bringover, rebuild, test cycle can become onerous and time consuming. If smaller groups of developers working on related portions of code integrate into a subintegration workspace, that workspace will be more stable and require fewer builds and less testing. Of course when the subintegration workspaces are themselves put back to their common integration area, changes made in the other development workspaces will have to be integrated. Experience has shown, however, that doing larger integrations, less frequently, is more efficient.

Disadvantages of a Multitiered Hierarchy

The disadvantages of multiplying subintegration workspaces are as follows:

- Each new workspace consumes disk space.
- Developers who ought regularly to be looking at one another’s work may find it harder to do so because they do not put back to the same integration workspace
- Integration of the subintegration workspaces to the higher integration workspace can become more complicated than more frequent, smaller integrations.

Product Release Considerations

When you plan your project hierarchy structure, consider how you plan to release your product. There are a number of ways that you can structure workspace hierarchies to facilitate the preparation of major, minor, and patch releases. The following discussion presents some ideas for you to consider; your product may not lend itself to this model, or your product may have considerations that suggest an alternate scheme.

You should consider dedicating a workspace as a product release staging area for each release. “Hang” the release workspace off a top-level “product” workspace. The product workspace should be located hierarchically above the workspaces in which normal development integration is done. Locating the product workspace this way permits you to begin development of your next release without corrupting the current release.

After the files are transferred to the product workspace, you use the Bringover transaction to transfer the files down to the release workspace. The release workspace can be used to make masters and can serve as an area in which to save work for subsequent releases if necessary.

Figure 5–3 shows a hierarchy that contains a product workspace and release workspaces for six different releases.

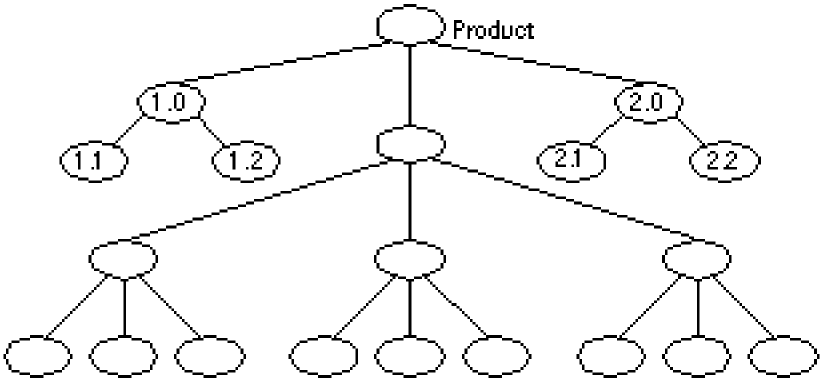


Figure 5–3 Hierarchy with Product and Release Workspaces

Note - You can use the reparenting feature to transfer data between release workspaces directly. See “A Reparenting Example” on page 71 for details.

Coordinating Access to Source Files

Coordinating write access to source files is important when changes will be made by several people. Maintaining a record of file updates allows you to determine when and why changes were made.

The source code control system (SCCS) allows you to control write access to source files and monitor changes made to those files. The SCCS allows only one user at a time to update a file, and it records all changes in a history file.

Versioning is a GUI to SCCS. Versioning allows you to manipulate files and perform most of the basic SCCS functions without having to know SCCS commands. It provides an intuitive method for checking files in and out, as well as displaying and moving through the history branches.

With Versioning, you can:

- Check in files under SCCS
- Check out and lock a version of the file for editing
- Retrieve copies of any version of the file from SCCS history
- Visually peruse the branches of an SCCS history file
- Back out changes to a checked-out copy
- Inquire about the availability of a file for editing
- Inquire about differences between selected versions using Merging
- Display the version log summarizing executed commands

Versioning helps you perform these tasks and expedites the progress of concurrent development projects.

Branches

You can picture the deltas applied to an SCCS file as nodes of a tree with the initial version of the file as the root. The root delta is numbered 1.1 by default. These two parts of the SCCS delta ID (SID) are the release and level numbers. Successive deltas (nodes) are named 1.2, 1.3, and so forth. This structure is called the trunk of the SCCS delta tree. It represents the normal sequential development of an SCCS file.

It may be necessary to create an alternative branch on the tree. Branches can be used to keep track of alternate versions developed in parallel, such as for bug fixes.

The SID for a branch delta consists of four parts: the release and level numbers and the branch and sequence numbers, or release.level.branch.sequence. The branch number is assigned to each branch that is a descendant of a particular trunk delta; the first branch is 1, the next 2, and so on. The sequence number is assigned, in order, to each delta on a particular branch. Thus, 1.3.1.1 identifies the first delta of the first branch derived from delta 1.3. A second branch to this delta would be numbered 1.3.2.1 and so on.

The concepts of branching can be extended to any delta in the tree. The branch component is assigned in the order of creation on the branch, independent of its location relative to the trunk. Thus, a branch delta can always be identified from its name. While the trunk delta can be identified from the branch delta's name, it is not possible to determine the entire path leading from the trunk delta to the branch delta.

For example, if delta 1.3 has one branch, all deltas on that branch will be named 1.3.*n*. If a delta on this branch has another branch emanating from it, all deltas on the new branch will be named 1.3.2.*n*. The only information that can be derived from the name of delta 1.3.2.2 is that it is usually the second chronological delta on the second chronological branch whose trunk ancestor is delta 1.3. In particular, it is not possible to determine from the name of delta 1.3.2.2 all of the deltas between it and its trunk ancestor (1.3).

CHAPTER **6**

TeamWare Configuring User Interfaces

You can work with TeamWare Configuring in two ways:

- Use the command-line interface (CLI).
- Use the Configuring graphical user interface (GUI).

Both interfaces are included with TeamWare to accommodate different computing styles. Complete TeamWare Configuring functionality is implemented in both interfaces. The interfaces can be used interchangeably. You can simultaneously use the GUI for some functions and the CLI for others. Both interfaces employ the same underlying Configuring functionality and command structure; the difference is an easy-to-use, graphical, interface for the GUI.

The concepts in this chapter generally apply to both the GUI and the CLI. Except in cases where there are special considerations regarding the CLI, descriptions and examples are included for the GUI only—information specific to the CLI can be obtained online through the man pages. The remainder of this chapter is an introduction to the Configuring CLI and GUI.

This chapter is organized as follows:

TeamWare Configuring Command-Line Interface

The TeamWare Configuring CLI is accessible from any Solaris shell. The CLI is useful when you are not working on a window-based system.

Like SCCS commands, all CLI commands may be executed through a central "umbrella" command. The individual commands may also be executed directly by specifying the individual command name.

The umbrella command `codemgr` lets you list Configuring commands. You can list Configuring commands with their use summaries by typing `codemgr` without specifying any arguments. You can achieve the same results by typing `codemgr` with the `help` subcommand.

```
example% codemgr
   bringover ...
   codemgrtool
   help
   putback ...
   resolve ...
   ws_undo ....
   workspace ....
```

To use the umbrella command to execute commands, type `codemgr` followed by the name of the subcommand you wish to execute. For example:

```
% codemgr bringover -w my_child -p their_parent /usr/ws/project
```

You can also execute the commands directly (without typing `codemgr`). For example:

```
% bringover -w my_child -p their_parent /usr/ws/project
```

Note - You must use the individual command name when you access TeamWare man pages.

TeamWare provides several ways to reduce typing long command-lines, including environment variables and argument files that store previously specified arguments. See the respective man pages for details.

TeamWare Configuring Graphical User Interface

The TeamWare GUI (Configuring) is a tool that enables you to view workspace hierarchies and to execute menu-based commands on workspaces and their contents. Key features include the following:

- Graphical display of workspaces in a Workspace Graph pane. This feature enables users to:
    - Conveniently view workspace hierarchies
    - Use the mouse to select workspace icons
    - Execute menu-based commands on selected workspaces and their contents
- Menu lists that reduce the need for you to remember and type command names, options, and arguments
- Facilities to customize the GUI to meet your individual style and needs
- Online help to assist you at all levels, including explanation of error messages

---

Note - TeamWare windows, menus, and buttons are documented online using the Help feature. With Help, you can obtain information regarding any object on the screen. Therefore, this section does not discuss these objects in detail; rather, it serves as a guide to other aspects of Configuring. Throughout the rest of this chapter, Configuring tasks such as Bringover and Putback transactions and conflict resolution are discussed in detail.

Starting TeamWare Configuring

To start TeamWare Configuring, at a shell command prompt type `twconfig` followed by the ampersand symbol (&).

```
demo% twconfig&
```

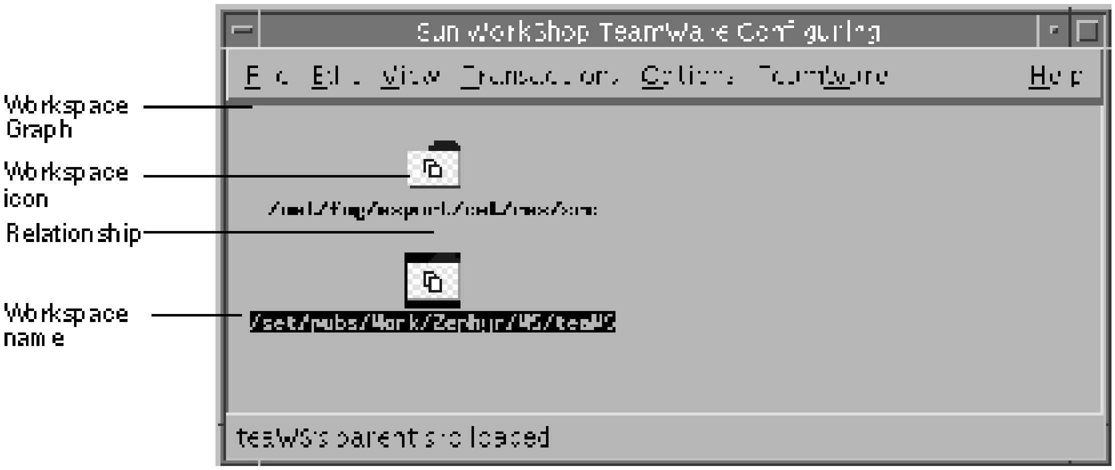
After a moment, the Configuring window appears.

TeamWare Configuring Windows

Configuring consists of a main window (see Figure 6–1) and a number of dialog boxes. Within the main window is a menu bar from which you can choose items to help you accomplish your tasks. The online help contains a full explanation of TeamWare Configuring windows.

Configuring Window

When you start Configuring, the main window opens (see Figure 6–1). In working with Configuring, you select workspace icons in the Workspace Graph pane and then choose commands that act upon the selected workspaces and the files they contain.



tesWS's parent is loaded

Figure 6–1 Configuring Window With Parent and Child Workspaces Loaded

File menu	Provides commands to load, unload, and create workspaces
Edit menu	Provides commands to delete, rename, reparent, and update workspaces
View menu	Provides commands to set view options in the Workspace Graph pane
Transactions menu	Provides commands for synchronizing files with Bringover, Putback and Resolve transactions
Options menu	Provides commands for setting Workspace and Configuring options

TeamWare menu	Provides command for starting other TeamWare tools
Workspace Graph pane	Displays a visual graph of the workspaces you have loaded

Workspace Graph Pane

In the Workspace Graph pane each workspace is represented by a workspace icon. Parent and child relationships are depicted by lines connecting workspaces. The path name of the workspace top-level (root) directory is displayed beneath the icon.

Loading Workspaces into the Workspace Graph Pane

When Configuring is started, it checks the directory (or directories) specified by the environment variable `CODEMGR_WSPATH` to determine if it contains any workspaces. If workspaces are found, they are loaded into the Workspace Graph pane. If `CODEMGR_WSPATH` is not set, Configuring attempts to load workspaces from the directory in which it was started. To load additional workspaces, choose File > Load Workspaces.

Layout

Workspace hierarchy graphs are automatically created in the Workspace Graph pane by the Configuring program as you load workspaces (using the Load menu and the Load Workspaces dialog box (see “Customizing Configuring Using Tool Properties” on page 59). Hierarchies are displayed either vertically or horizontally starting from the upper-left corner and distributed to the right as space permits. You can select the orientation by choosing View > Orientation. Vertical orientation is the default. Layout is done automatically—*you cannot change the layout by moving icons with the mouse.*

Workspace Name Fields

Beneath the workspace icon is a text field that contains the name of the workspace root directory. You can choose to have workspace names displayed one of two ways:

- Using the absolute (full) path name of the root directory
- Using the truncated (short) name of root directory

Choose the display style you prefer by choosing Name >View.

You can change the path name of a workspace by editing the name text box.

Dragging and Dropping Workspace Icons

You can accomplish two types of operations by directly manipulating icons on the Workspace Graph pane. You can drag and drop workspace icons to initiate both Bringover and Putback transactions and to reparent workspaces.

- Interworkspace transactions—If you select and drag a workspace and drop it on top of another icon, Configuring initiates one of the following transactions: Bringover Create, Bringover Update, Putback. You determine which transaction is initiated by which icon you drag, and where you drag it; Table 6–1 summarizes these actions. For more information about interworkspace transactions, see Chapter 8."

TABLE 6–1 Workspace Drag and Drop Action

Drag:	**To:**	**Action**
Any workspace icon	Open area	Activate Bringover Create transaction window
Parent workspace icon	Child workspace icon	Activate Bringover Update transaction window
Child workspace icon	Parent workspace icon	Activate Putback transaction window
Any workspace icon	A nonrelated (not a parent or child) workspace icon	Activate pop-up notice to determine actions

- Reparenting—To use the drag and drop facility to change a workspace's parent, press and hold the SHIFT key while you drag the workspace icon on top of its new parent's icon. If you drag the icon to an open area of the Workspace Graph pane, the workspace will be orphaned (have no parent). The display is automatically adjusted to reflect the new relationship. For more information about reparenting workspaces, see "Reparenting a Workspace" on page 70.

Note - You are prompted to confirm the reparent operation.

Double-Click Action

When you double-click when the pointer is over a workspace icon, the TeamWare Versioning is automatically started (with the selected workspace automatically loaded). See Versioning online help and Chapter 12" for instructions on using Versioning.

If you double-click when the pointer is over the icon of a workspace that contains unresolved conflicts, Configuring automatically activates the Resolve transaction window. Conflicted files from the selected workspace are automatically loaded and ready for processing.

You can customize Configuring double-click behavior using the Tool Properties dialog box (see Table 6–2).

Customizing Configuring Using Tool Properties

Using the Tool Properties dialog box (see Table 6–2), you can customize the behavior of:

- Configuring window functions
- Bringover/Putback transactions
- Resolve transaction

To open the Tool Properties dialog box, choose Options > Configuring. The Category list box in the dialog box lets you to switch between the Configuring, Bringover/Putback, and Resolve panes.

Configuring Defaults Files

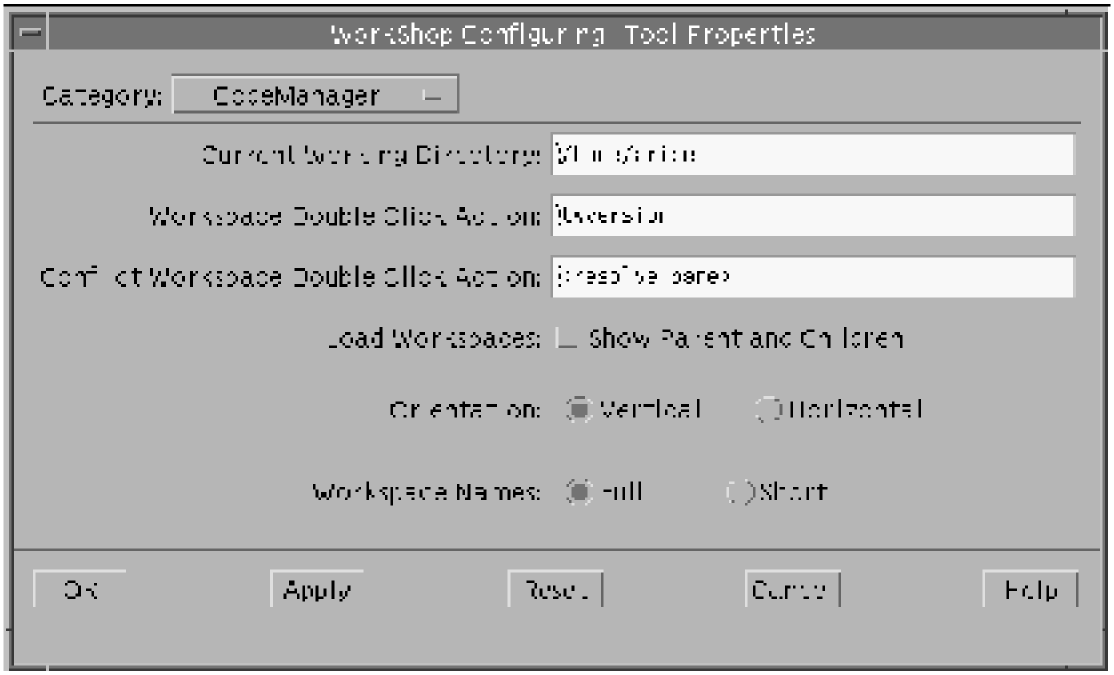
When you change Configuring behavior using the Tool Properties window, you can use the Apply button to preserve the changes in runtime configuration files in your home directory. The runtime configuration files are consulted by Configuring when it starts; your changes are used as the default values.

Changes made in the Configuring and Bringover/Putback panes of the Tool Properties window are written to the file `~/.codemgrtoolrc`. This file is an XWindows resource file.

Changes made in the Resolve pane of the Tool Properties window are written to the runtime configuration file `~/.codemgr_resrc`.

Configuring Pane

The Configuring pane of the Tool Properties dialog box (see Table 6–2) enables you to change the behavior of the Configuring main window. The specific properties are described in Table 6–2.



WorkShop Configuring Tool Properties

Category: CodeManager

Current Working Directory: [illegible]

Workspace Double Click Action: twversion

Conflict Workspace Double Click Action: <resolve_pane>

Load Workspaces: □ Show Parent and Children

Orientation: ● Vertical ○ Horizontal

Workspace Names: ● Full ○ Short

Figure 6–2 Configuring Pane of the Tool Properties Dialog Box

TABLE 6–2 Configuring Tool Properties

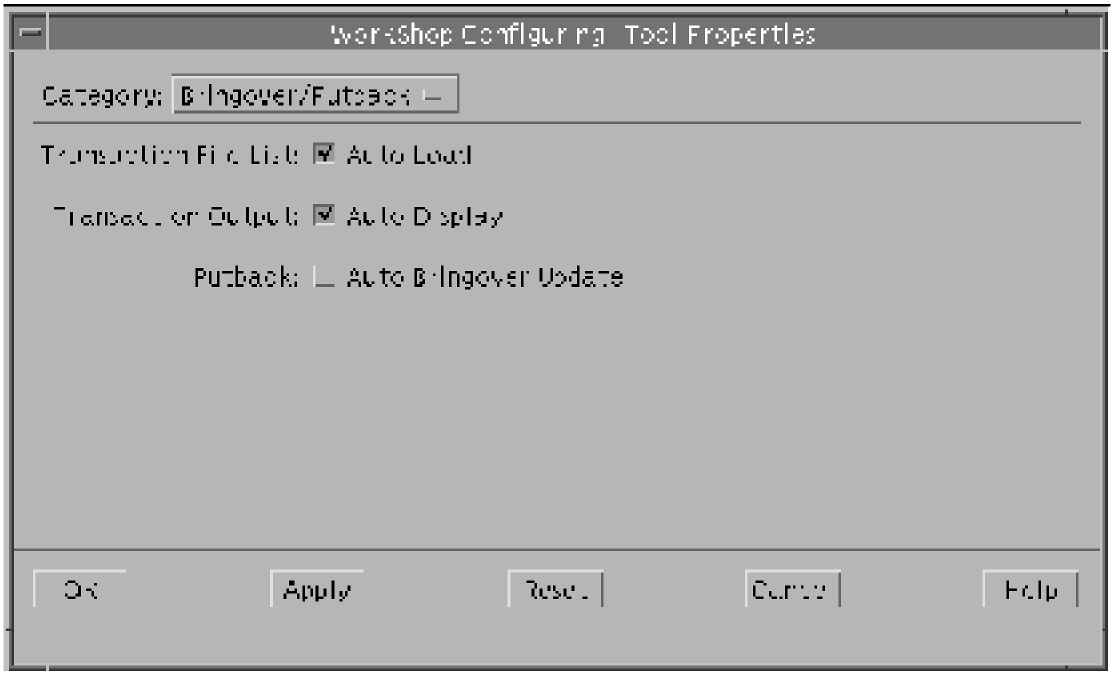
Property	Description
Current Working Directory	Lets you specify the directory to which Configuring actions are relative.
Workspace Double Click Action	Lets you specify the commands you want launched when you double-click on a standard workspace icon. Type the path name required to execute the commands based on the current working directory and your search path. By default, the standard workspace command is Versioning (`twversion`).
Conflict Workspace Double Click Action	Lets you specify the commands you want launched when you double-click icons of workspaces that contain conflicts. Type the path name required to execute the commands based on the current working directory and your search path. By default, the Resolve Transaction window (<`resolve_pane`>) is opened for conflicted workspaces.
Load Workspaces	Select this check box if you want the parent and children of workspaces you load in the Workspace Graph pane automatically loaded with them. By default this box is not checked.

TABLE 6–2 Configuring Tool Properties *(continued)*

Property	Description
Orientation	Select the Horizontal setting if you want the workspace hierarchy displayed horizontally from left to right in the Workspace Graph pane. Select the Vertical setting if you want workspace hierarchy displayed vertically from top to bottom. By default the Vertical setting is in effect. This property corresponds to the choosing View Orientation the Configuring main window.
Workspace Names	Select the Short setting if you want workspaces labelled with the shortest possible name in the Workspace Graph pane. Select Full if you want workspaces labelled with absolute path names. By default the Full setting is in effect. This property corresponds to choosing View Names in the Configuring main window.

Bringover/Putback Pane

The Bringover/Putback pane of the Tool Properties dialog box (see Figure 6–3) enables you to change the behavior of the Bringover and Putback panes of the Transactions window. The specific properties are described in Table 6–3.



WorkShop Configuring: Tool Properties

Category: Bringover/Putback

Transaction File List: ☑ Auto Load

Transaction Output: ☑ Auto Display

Putback: □ Auto Bringover Update

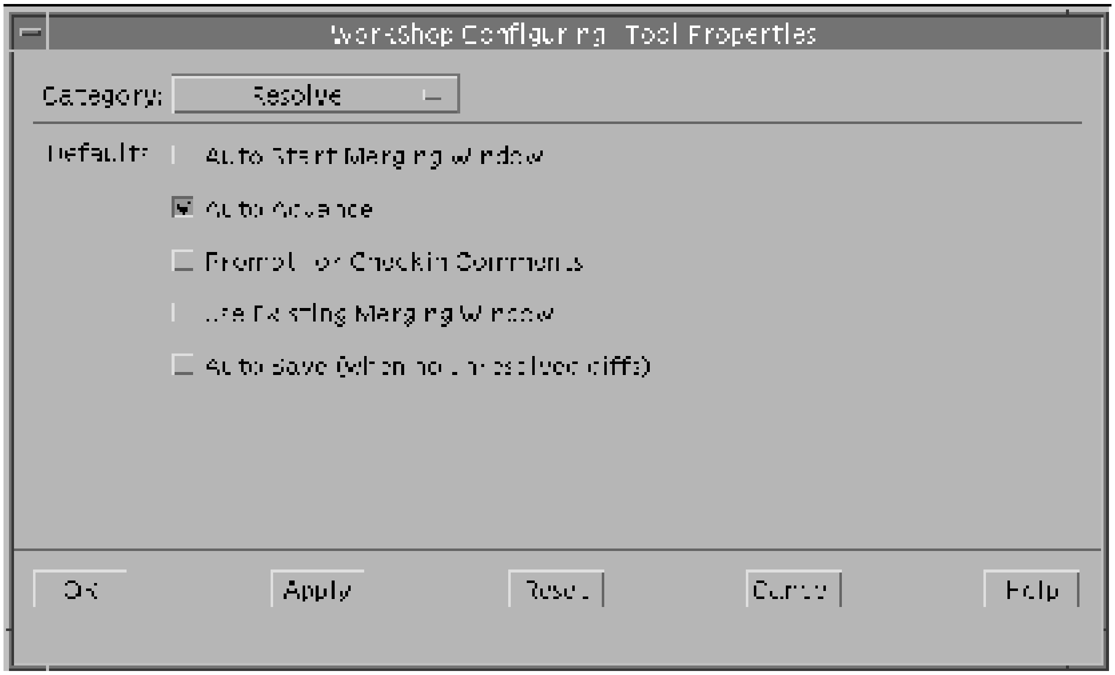
Figure 6–3 Bringover/Putback Pane of the Tool Properties Dialog Box

TABLE 6–3 Bringover/Putback Tool Properties

Property	**Description**
Transaction File List	Selecting Auto Load causes Configuring to reread the `Codemgr_wsdata/args` file and load it into the File List pane whenever a new workspace is selected. You might choose to deselect this property when you want to use the same file list for a number of transactions involving different workspaces.
Transaction Output	If you select Auto Display, Configuring automatically displays the Transaction Output window during transaction execution.
Putback	If you select Auto Bringover Update, and a Putback transaction is blocked, Configuring automatically initiates a Bringover transaction to update the child workspace.

Resolve Pane

The Resolve pane of the Tool Properties window enables you to change the behavior of the Resolve pane of the Transactions window. The specific properties are described in Table 6–4.



Category: Resolve

Default:
☐ Auto Start Merging Window
☒ Auto Advance
☐ Prompt for Checkin Comments
☐ Use Existing Merging Window
☐ Auto Save (when no unresolved diffs)

Figure 6–4 Resolve Pane of the Tool Properties Dialog Box

TABLE 6–4 Resolve Tool Properties

Property	**Description**
Auto Start Merging Window	Causes Merging to start automatically when the Resolve transaction pane is chosen.
Auto Advance	Causes the next file in the list to be automatically loaded into Merging after the current file is resolved.
Prompt for Checkin Comments	A default comment is automatically supplied during checkin after you resolve a file. This property causes you to be prompted for an additional comment that is appended to the standard comment.

TABLE 6–4 Resolve Tool Properties *(continued)*

Property	Description
Use Existing Merging Window	If this property is set, an already running Merging process is reused during subsequent resolve operations.
Auto Save (when no unresolved diffs)	If this property is set, *and* all the changes in the file can be "automerged," the files will also be saved and checked in; you need not select the Merging Save button.

Keyboard Shortcuts

Table 6–5 summarizes the keyboard shortcuts available for Configuring functions.

TABLE 6–5 Configuring Keyboard Shortcuts

Accelerator	Action	Where to Find More Information
Drag and drop workspace icon	Activate Bringover/Putback transaction window	"Dragging and Dropping Workspace Icons" on page 58"
SHIFT+drag and drop workspace icon	Reparent workspace	"Dragging and Dropping Workspace Icons" on page 58"
Click workspace icon name field	Rename workspace	"Workspace Name Fields" on page 57"
Double-click workspace icon	Launch a tool. User configurable, Versioning is the default	"Double-Click Action" on page 58"
Double-click icon of a workspace that contains conflicts	Launch a tool. User configurable, Resolve window is the default	"Double-Click Action" on page 58"

CHAPTER 7

TeamWare Configuring Workspace

As discussed in Chapter 3," the *workspace* forms the basis of the Configuring system. The workspace provides isolation in which you (a developer) work in parallel with other developers programming in other workspaces. For an introduction to the Configuring workspace, refer to "Workspace" on page 23.

This chapter discusses specific aspects of workspaces and the Configuring commands you use to configure, create, manipulate, and administer them, and contains the following sections:

- "The Workspace Metadata Directory " on page 65
- "Creating a Workspace" on page 67
- "Deleting a Workspace" on page 68
- "Moving and Renaming a Workspace" on page 69
- "Reparenting a Workspace" on page 70
- "Controlling Access to Workspaces" on page 73
- "How to Notify Users of Changes to Workspaces" on page 77
- "Viewing Workspace Command History" on page 80
- "Ensuring Consistency Through Workspace Locking" on page 83
- "Configuring Environment Variables" on page 84

The Workspace Metadata Directory

A Configuring workspace is a directory hierarchy that contains a directory named `Codemgr_wsdata` in its root directory. the Configuring program stores data (metadata) about that workspace in `Codemgr_wsdata`. Configuring commands use the presence or absence of this directory to determine whether a directory is a workspace.

TeamWare Configuring provides the tools necessary to maintain the information kept in the Codemgr_wsdata directory. Although not recommended, on rare occasions it may be necessary to modify certain files manually. However, great care must be taken to understand and preserve the format of each file being edited. Table 7–1 briefly describes each of the files and directories contained in the metadirectory. Information regarding the format of these files is available in the man(4) page for each file.

TABLE 7–1 Contents of the Codemgr_wsdata Metadata Directory

File/Dir Name	Description
access_control	The access_control file contains information that controls which users are allowed to execute Configuring transactions and commands for a given a workspace. When workspaces are created, a default access control file is also created. See "Controlling Access to Workspaces" on page 73.
args	The args file is maintained by the Configuring Bringover and Putback transaction commands and contains a list of file, directory, and FLP arguments. Initially, the args file contains the arguments specified when the workspace was created. If you explicitly specify arguments during subsequent Bringover or Putback transactions, the commands determine if the new arguments are more encompassing than the arguments already in the args file; if they are, the new arguments replace the old.
backup/	The backup directory is used to store information that Configuring uses to "undo" a Bringover or Putback transaction. See "Reversing Bringover and Putback Transactions with Undo " on page 107.
children	The children file contains a list of the workspace's child workspaces. The names of child workspaces are entered into the workspace's children file during the Bringover Create transaction. Configuring consults this file to obtain the list of child workspaces. When you delete, move, or reparent a workspace, Configuring updates the children file in its parent.
conflicts	The conflicts file contains a list of files in that workspace that are currently in conflict. See Chapter 9," for more information about conflicts and how to resolve them.
history	The history file is a historical log of transactions and updated files that affect a workspace. See "Viewing Workspace Command History" on page 80 for more information.

TABLE 7–1 Contents of the `Codemgr_wsdata` Metadata Directory *(continued)*

File/Dir Name	Description
`locks`	To assure consistency, Configuring locks workspaces during Bringover, Putback and Undo transactions. Locks are recorded in the `locks` file in each workspace; Configuring consults that file before acting in a workspace. See "Ensuring Consistency Through Workspace Locking" on page 83.
`nametable`	The `nametable` file contains a table of SCCS file names (path names relative to that workspace) and a unique number represented as four 32-bit hexadecimal words. Each entry in the table is terminated by a newline character. The `nametable` file is used by Configuring during Bringover and Putback to accelerate the processing of files that have been renamed. If this file is not available, Configuring rebuilds it automatically during the next Putback or Bringover transaction. See "Renaming, Moving, or Deleting Files" on page 109.
`notification`	The `notification` file is edited by users to register notification requests. This facility permits Configuring to detect events that involve that workspace and to send electronic mail messages in response to the event. See "How to Notify Users of Changes to Workspaces" on page 77.
`parent`	The `parent` file contains the path name of the workspace's parent workspace and is created by the Bringover Create transaction, or by the Reparent command if the workspace was originally created with the Create Workspace command (and thus had no parent). Configuring consults this file to determine a workspace's parent. When you delete, move, or reparent a workspace, Configuring updates the `parent` file in its children.
`putback.cmt`	The `putback.cmt` file is a cache of the text of the comment from the last *blocked* Putback transaction. When a Putback transaction is blocked, the comment is discarded. Configuring caches the comment in `putback.cmt` so that you can retrieve the original text when you reexecute the transaction.

Creating a Workspace

You can create a workspace in one of two ways:

- *Explicitly* by means of File > Create Workspace in the Configuring main window

- *Implicitly* by using the Bringover Create transaction to copy files into a nonexistent child workspace, in which case the child workspace is created and then populated with the files specified as part of the transaction

Using Create Workspace

You can use File > Create Workspace to create new workspaces. Type the name of the new workspace's root (top-level) directory in the Workspace Directory text box and click OK.

If the workspace you are creating already exists as a directory hierarchy, Configuring converts it to a workspace by simply adding the `Codemgr_wsdata` subdirectory in the root directory and displaying its icon in the Workspace Graph pane.

If the directory does not already exist, Configuring creates both the root directory and the `Codemgr_wsdata` subdirectory.

Using the Bringover Create Transaction

Use the Bringover Create transaction (on the Transactions menu) to copy files from a parent workspace to a nonexistent child workspace; the child is automatically created as part of the transaction. See "Creating a New Child Workspace (Bringover Create)" on page 93 for details.

The online help provides further assistance in creating new workspaces. To access the help, choose Help > Help Contents > Starting a Project.

Deleting a Workspace

You delete workspaces by selecting their icons in the Workspace Graph pane and then Edit > Delete.

The Delete submenu provides two items for deleting workspaces:

- Sources and Codemgr_wsdata Directory. Recursively deletes the contents of the workspace. You are prompted to confirm your decision.
- Codemgr_wsdata Directory only. Changes the workspace to a non-workspace directory by deleting only the `Codemgr_wsdata` subdirectory and removing the icon from the Workspace Graph pane.

In either case Configuring automatically updates records in parent and child workspaces to reflect the deletion of the workspace.

Moving and Renaming a Workspace

Since a workspace is a directory, you move it by changing its path name. There are two ways that you can move and rename a workspace:

- By editing its name in the Workspace Graph pane

  Select the name field by moving the pointer over a portion of the text and clicking. This selects the text for editing. Use standard text editing to change the name; type Return to enter your changes. Click in an empty portion of the pane to deselect the text.
- By using Edit > Rename

  The path name of the selected workspace is changed to the name that you type in the New Workspace Name text box.

In addition to changing the workspace path name, both methods also update the appropriate data files in the parent and child workspaces to contain the new name. These data files are discussed in "The Workspace Metadata Directory " on page 65.

Caution - Do not use the mv command to rename or move workspaces.

The Configuring Rename command updates files in the workspace's parent and children, as well as logging the event in the `Codemgr_wsdata/history` file.

If you inadvertently use the `mv` command to move and rename a workspace and discover that it has become "disconnected" from its parent and children, you can use the Rename command to reconnect it.

For example, if you used the `mv` command to rename a workspace from A to B:

1. **Use the Rename command to rename B to C.**

   This causes Configuring to update the workspace's new name (C) in the parent and child workspaces. To save time, be sure to use a path name on the same device.

2. **Use the Rename command to change C back to B.**

   Everything should be reconnected.

Reparenting a Workspace

As discussed in Chapter 3, the parent/child relationship is the thread that connects the workspace hierarchy. Configuring provides the means for you to change this relationship at your discretion.

This section discusses how you can explicitly change a workspace's parent. It is also possible for you to implicitly change a workspace parent "on the fly" (for the duration of a single command) by specifying the new parent's path name as part of a Bringover Update or Putback transaction. See the descriptions of the Bringover Update and Putback transactions in Chapter 8" for more information.

Two Ways to Reparent Workspaces

This section describes two equivalent ways to reparent workspaces.

Drag and Drop Workspace Icons

You can change a workspace's parent by selecting its icon in the Workspace Graph Pane, pressing and holding the SHIFT key, and dragging it on top of its new parent's icon. The display is automatically adjusted to reflect the new relationship.

Note - You are prompted to confirm the change.

You may also "orphan" a workspace by selecting its icon, pressing SHIFT, and dragging it to an open area on the Workspace Graph pane. The workspace no longer has a parent: the display is automatically adjusted to reflect its new status.

The Parent Command

You can change a workspace's parent by selecting its icon in the Workspace Graph pane and then choosing Edit > Parent.This opens the Parent dialog box.

When the window is initially opened, the New Parent Workspace Directory text box contains the name of the current parent. Edit that line so that it contains the name of the new parent file and click the Parent button. The Workspace Graph pane is automatically adjusted to reflect the new relationship.

If you do not specify a parent workspace in the New Parent Workspace Directory text box, the workspace is orphaned—it has no parent. The Workspace Graph pane is automatically adjusted to reflect its new status.

Reasons to Change a Workspace's Parent

You might want to permanently or temporarily change a workspace parent for any of these reasons:

- To populate a new project hierarchy (new top-level workspace). You may be completing Release 1 of your product and see the need to begin work on Release 2. In this case you might:
    - Create a new (empty) Release 2 workspace by means of File > Create Workspace
    - Use either of the two methods described in "Two Ways to Reparent Workspaces" on page 70 to make the Release 2 workspace the new parent of the Release 1 workspace
    - Use the Putback transaction to copy files to the Release 2 workspace
    - Reparent the Release 1 workspace to its original parent
- To move a feature into a new release. If a feature intended for a particular release is not completed in time, the workspace in which the feature was being developed can be reparented to the following release's integration workspace. A similar use of reparenting is described in the example in the"A Reparenting Example" on page 71.
- To apply a bug fix to multiple releases.The workspace in which work was done to correct a bug is reparented from hierarchy to hierarchy; the Configuring Putback transaction is used to incorporate the changes into the new parent. An example of this use of reparenting is included in "A Reparenting Example" on page 71.
- To reorganize workspace hierarchies
    - You can add additional levels to the hierarchy.
    - You can remove levels from the hierarchy (do not specify a new parent during reparenting).
    - You can reorganize workspace branches within the project hierarchy.
- To adopt an orphan workspace if its `Codemgr_wsdata/parent` file is deleted. If, for some reason a file is orphaned (for example, its parent is corrupted or its own `Codemgr_wsdata/parent` file deleted) you can use the reparenting feature to restore its parentage.

A Reparenting Example

Often a bug is fixed in a version of a product and a patch release is made to distribute the fixed code. The code that was fixed must usually be incorporated into the next release of the product as well. If the product is developed using Configuring, the patch can be incorporated relatively simply by means of reparenting.

In the following example, a patch is developed to fix a bug in Release 1.0 of a product. The patch must be incorporated into Release 2.0, which has begun development.

1. The workspace in which the patch was developed (or the workspace from which it is released) is cloned by means of the Bringover Create transaction. The reason the workspace is cloned is that it will be altered by its interaction with its new parent (Bringover transaction to synchronize it with its new parent).
2. Either of the two reparenting methods are used to change the cloned workspace's parent from `1.0patch` to `2.0`. (see Figure 7–1).

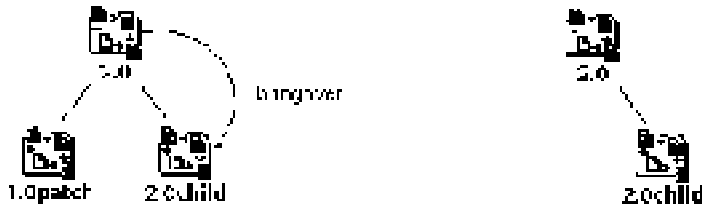


Figure 7–1 Patch Workspace Reparented to New Release

3. The workspace is then updated from its new parent, and any new work is brought over from `2.0`. (see Figure 7–2A).
4. The fixes made for the patch are merged in `patch` with the files from `2.0` and are put back into the `2.0` workspace where they are now available to workspace `2.0child`. (see Figure 7–2B).

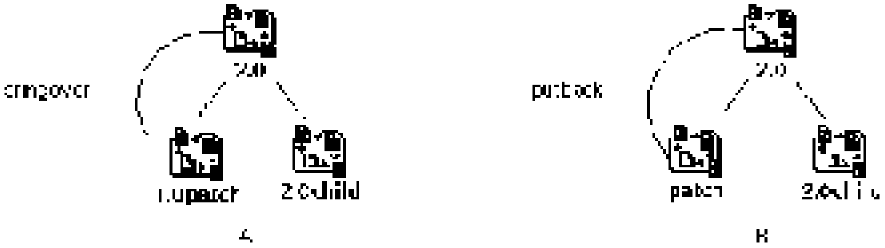


Figure 7–2 Files Brought Over, Merged, and Incorporated into the New Release

5. Files are brought over to `2.0child`, and `patch` is deleted by means of Edit > Delete > Sources and Codemgr_wsdata Directory. (Figure 7–3).

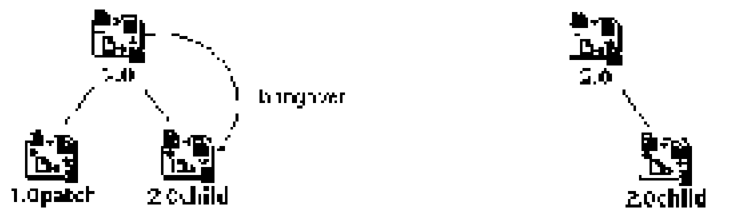


Figure 7–3 Patched Files Brought Over into `2.0child`; `patch` Deleted

Controlling Access to Workspaces

Configuring permits you to control the access that users have to your workspaces. Table 7–2 lists and describes the eight types of access over which you can exercise control.

TABLE 7–2 Operations Over Which You Have Access Control

Type of Access	Description
bringover-from	Controls which users may bring over files *from* this workspace
bringover-to	Controls which users may bring over files *to* this workspace
putback-from	Controls which users may put back files *from* this workspace
putback-to	Controls which users may put back files *to* this workspace
undo	Controls which users may "undo" commands executed in this workspace
workspace-delete	Controls which users may delete this workspace
workspace-move	Controls which users may move this workspace
workspace-reparent	Controls which users may reparent this workspace
workspace-reparent-to	Controls which users may reparent other workspaces to this workspace

TABLE 7–2 Operations Over Which You Have Access Control *(continued)*

Prior to taking any of the actions listed above, Configuring consults a file in the `Codemgr_wsdata` subdirectory named `access_control` to determine whether the user taking the action has access permission to the workspace for that purpose. The `access_control` file is a text file that contains a list of the eight operations and corresponding values that stipulate who is permitted to perform those operations. The `access_control` file is automatically created at the time the workspace is created and is owned by the creator of the workspace.

To view and change access permissions, choose Options > Workspace, then use the Category list box to choose the Access Control pane of the Workspace Properties dialog box (see "Viewing and Changing Access Control Values" on page 75).

Table 7–3 shows the default contents of `access_control` after you create a workspace:

TABLE 7–3 Default Access Control Permissions

Operation	Permissions
bringover-from	
bringover-to	creator
putback-from	
putback-to	
undo	
workspace-delete	creator
workspace-move	creator
workspace-reparent	creator
workspace-reparent-to	

Note - Creator permission indicates that Creator's login name appears.

You can express which users have or do not have access to a workspace in a number of ways. Table 7–4 shows all of the value types you can specify to control access to your workspaces and what the entries mean.

TABLE 7–4 Workspace Access Control Values

Value	Meaning
@engineering	All users in the net group named `engineering` can execute this operation
-@engineering	No users from the net group named `engineering` can execute this operation. Note that “-” denotes negation.
@special -user2 @engineering	All users in the net groups `special` and `engineering` can execute the operation; user2 cannot (unless `user2` is in the `special` netgroup). “-” denotes negation.
user1 user2	The users `user1` and `user2` can execute the operation.
“-”	No user can execute the operation.
creator	Only the user who created the workspace can execute the operation. Note that the creator’s login name actually appears.
(*no entry*)	Any user may execute the operation.

Note - If a user is listed as having both access permission *and* restriction, the first reference is used.

Note - Performance may degrade when net groups are included in the access control file. The time required to look up group membership can add several seconds to the execution of a given operation.

Viewing and Changing Access Control Values

You can view and change the access control status of a workspace using the Workspace Properties dialog box.

Viewing Access Control Status

To view the access control status of a workspace:

1. **Select a workspace icon in the Workspace Graph pane.**

2. **Choose Options > Workspace.**

3. **Select Access Control from the Categories list box.**

Changing Access Control Values

To change the access control status of a workspace:

1. **Select a workspace icon in the base window Workspace Graph pane.**

2. **Choose Options > Workspace.**

3. **Select Access Control from the Categories list box.**

4. **Select an access line in the global Access Control list, then click Edit to open the Access Control Properties dialog box.**

   The operation you selected before clicking Edit is automatically selected for you in the Operation list box.

   Or, use the Operation list box in the Access Control Properties dialog box to select an operation type.

5. **Select the radio button for the type of permission you wish to allow:**
   - None—No users have permission
   - All—All users have permission
   - Specify—Use the Permissions list to construct a list of users and netgroups that are to be granted or denied permission

6. **If you choose to specify individual and/or group permissions, construct your entry using:**
   - The Name text box to type the name of the user or netgroup
   - The Type radio buttons to specify whether the entry is a user or a netgroup
   - The Access buttons to specify whether the specified user/netgroup is granted or denied permission

7. **Click Insert Before or Insert After to add your entry into the Permissions list.**

8. **Click Apply to enter your selection into the global Access Control list.**

9. **In the Workspace Properties dialog box, click Apply to write the changes to the** `access_control` **file.**

How to Notify Users of Changes to Workspaces

You can request Configuring to notify you (through an electronic mail message) when a variety of Configuring events occur in a workspace. Notification requests are entered in the file named `notification` in the `Codemgr_wsdata` directory.

A notification request consists of the following items:

- An address to which mail is sent.
- The event for which you want notification triggered.
- An optional list of directories and files whose changes of status trigger notification. The list is bracketed by BEGIN / END statements.

The following is an example of a `notification` file that contains three requests:

```
chip@mach1 bringover-to
BEGIN
dir1/foo.cc
dir2
END
biff@mach2 bringover-to putback-to
BEGIN
.
END
biff@mach2 workspace-move
```

In the first entry, the user `chip@mach1` requests to be notified when the file `dir1/foo.cc` and *any* file in the directory `dir2` (path names are relative to the workspace root directory) are brought over to the workspace.

Note - File and directory entries for each event are bracketed by BEGIN / END statements. An empty list, a missing list, or a list that consists of only the "." character indicate that all files and directories in the workspace are registered for notification.

In the second entry, user `biff@mach2` requests to be notified when any file in the workspace is brought over to, or put back to, the workspace. The "." character represents all files in the workspace.

In the third entry, `biff@mach2` requests to be notified if the workspace is moved. Events that involve entire workspaces (delete, move, reparent) do not accept directory / file lists.

Table 7–5 lists the events for which you can register notification requests:

TABLE 7–5 Notification Events

Event Name	Description
bringover-from	Send mail whenever files are brought over *from* the workspace in which the notification file is located.
bringover-to	Send mail whenever files are brought over *to* the workspace in which the notification file is located.
putback-from	Send mail whenever files are put back *from* the workspace in which the notification file is located.
putback-to	Send mail whenever files are put back *to* the workspace in which the notification file is located.
undo	Send mail whenever a transaction is "undone" in the workspace in which the notification file is located.
workspace-delete	Send mail if the workspace in which the notification file is located is deleted.
workspace-move	Send mail if the workspace in which the notification file is located is moved.
workspace-reparent	Send mail if the workspace in which the notification file is reparented.
workspace-reparent-to	Send mail if the workspace becomes the new parent of an existing workspace.

Viewing and Changing Notification Entries

To view and change notification entries, select a workspace icon in the Workspace Graph pane and choose Options > Workspace. Select Notification from the Category list box. The requests contained in the `notification` file are similar to those displayed in the Workspace Properties dialog box.

Use the buttons to modify, create, and delete notification entries. Clicking Create Entry or Edit Entry opens the Notification Entry dialog box.

Creating a New Notification Entry

To create a new notification entry:

1. **Click Create Entry with no items selected.**

2. **The Notifications Entry dialog box opens; use this dialog box to specify the following information:**

   a. **Type the mail address to which notification mail is to be sent in the Mail To text box.**

      The mail address can be any valid mail address, including an alias.

   b. **Select the appropriate radio button for the event about which notification mail is to be sent.**

   c. **Select the appropriate radio button for the files to which the notification event applies:**

      - All for any file in the workspace

      - Specify for specific directories or files

3. **If you choose to specify files or directories, create the list of directories and files in the Files text pane.**

   - To add files to the list, click Add to List and select files in the Add Notification Files dialog box
   - To delete files from the list, click Delete or Delete All.

4. **Click Apply to add the entry to the global Notification list.**

5. **Click Apply in the Workspace Properties dialog box to apply changes to the** `notifications` **file.**

Modifying a Existing Entry

To modify an existing entry:

1. **Select the entry in the Notification pane of the Workspace Properties dialog box and click Edit Entry.**

2. **Use the Notification Entry dialog box to modify the entry.**

3. **Click Apply to modify the entry in the global Notification list.**

4. **Click Apply in the Workspace Properties dialog box to apply changes to the** `notifications` **file.**

Note - The following events involve entire workspaces and thus do not require a directory/file list:workspace-delete, workspace-move, workspace-reparent, workspace-reparent-to.

Note - When a directory is specified in the list, all files hierarchically beneath it are automatically registered.

Viewing Workspace Command History

Configuring commands are logged in the text file `Codemgr_wsdata/history`. Commands that affect a single workspace are logged only in that workspace; interworkspace transactions are logged in both the source and destination workspaces.

Note - Although command entries are logged in both the source and destination workspaces, the list of changed files is entered only in the destination directory.

You can view the contents of this file to track or reconstruct changes that have been made to a workspace over time. Log entries consist of the underlying command-line entries and do not correspond to GUI menu item names. If you have any questions about the meaning or syntax of a command, refer to its `man` page for details. Table 7–6 lists the GUI operations and the corresponding CLI command that is entered in the history log.

TABLE 7–6 Corresponding GUI and CLI Commands

GUI Menu Item	Corresponding CLI Command
Create Workspace	`workspace create`
Rename	`workspace move`
Parent	`workspace parent`
Bringover Create	`bringover`

TABLE 7–6 Corresponding GUI and CLI Commands *(continued)*

GUI Menu Item	Corresponding CLI Command
Bringover Update	`bringover`
Putback	`putback`
Undo	`ws_undo`
Resolve	`resolve`

Note - In active workspaces, the `Codemgr_wsdata/history` file can grow very quickly. You may want to periodically prune the file to reduce its size.

The following portion of a history file was generated during a Bringover Update transaction; entries are described in Table 7–7. This entry is taken from the history file in the child; the corresponding entry in the parent is identical except that file status messages are not included.

```
COMMAND bringover -w /home/sponge3/larryh/ws/man_pages -p
/home/sponge3/larryh/ws/manpages man trans/man
update: man/Makefile
update: man/man5/access_control.5
create: man/man5/notification.5
create: man/man1/codemgr.1
rename from: man/man1/def.dir.flg.1
to: man/man1/def.dir.flp.1
update: man/man1/def.dir.flp.1
create: man/man1/codemgrtool.1
rename from: man/man1/fileresolve.1
to: deleted_files/man/man1/fileresolve.1
update children's name history:
 deleted_files/man/man1/fileresolve.1
rename from: man/man1/resolve_tty.1
to: deleted_files/man/man1/resolve_tty.1
update: deleted_files/man/man1/resolve_tty.1
create: trans/man/man1/codemgr_acquire.1
create: trans/man/man1/codemgr_prepare.1
CWD /tmp_mnt/home/sponge3/larryh/temp
RELEASE Beta 1.0
HOST croak
USER larryh
PARENT_WORKSPACE (/home/sponge3/larryh/ws/manpages) (sponge:/export/home/
sponge3/larryh/ws/manpages)
CHILD_WORKSPACE (/home/sponge3/larryh/ws/man_pages) (sponge:/export/home/
sponge3/larryh/ws/man_pages)
START (Mon Jul 13 13:31:16 1992 PDT) (Mon Jul 13 20:31:16 1992 GMT)
END (Mon Jul 13 13:32:08 1992 PDT) (Mon Jul 13 20:32:08 1992 GMT)
STATUS 0
```

TABLE 7–7 History File Entry Descriptions

Entry	Description
`COMMAND`	Underlying command line issued for the operation. File status messages as displayed in the Transaction Output window are included only in the destination workspace history file.
`CWD`	Name of the current working directory when the command was executed.
`RELEASE`	Release number of the TeamWare software
`HOST`	Name of the system from which the command was executed.
`USER`	Login name of the user who executed the command.
`PARENT_WORKSPACE`	The path name of the parent workspace specified in two formats: host-specific and *machine*:*pathname*.
`CHILD_WORKSPACE`	The path name of the child workspace specified in two formats: host-specific and *machine*:*pathname*.
`START`	Time the command started execution, both locally and as measured by Greenwich Mean Time (GMT).
`END`	Time the command completed execution, both locally and as measured by Greenwich Mean Time (GMT).
`STATUS`	Exit status of the command: 0 = Normal completion, any other value indicates an error condition, warning, or other status.

Ensuring Consistency Through Workspace Locking

To assure consistency, the Configuring transactions—Bringover, Undo, and Putback—lock workspaces while they are working in them. These locks only affect Configuring transactions; other commands such as SCCS programs, are not affected. Locks are recorded in the `Codemgr_wsdata/locks` file in each workspace; the

Configuring transaction commands consult that file before acting in a workspace. Two types of locks are used:

- A *read-lock* is used when a command must assure that a workspace does not change while it is examining its contents. Read-locks may be obtained concurrently by a number of commands; no Configuring command can write to the workspace while a read-lock is in force. A read-lock is obtained during a Bringover transaction in the parent when its files are examined in preparation for copying to the child, and during a Putback transaction in the child when its files are examined in preparation for copying to the parent.
- A *write-lock* is used when a command must assure that a workspace does not change while it is writing to it. Only one write-lock may be obtained for a workspace at any time. When a write-lock is in force, only the Configuring command that owns the lock can write to the workspace; other commands cannot obtain read-locks from the workspace. A write-lock is obtained during a Bringover transaction for the child when files are copied into it, and during a Putback transaction for the parent when files are copied into it.

If a Configuring command is unable to remove its lock after completion (for example, the system crashes), you must remove the lock yourself before Configuring commands will again be able to read and/or write in the workspace. You can use the Configuring GUI to view and delete active locks for a workspace, or you can edit the file directly.

To view and delete locks using the Configuring GUI, select a workspace icon from the Workspace Graph pane and choose the Workspace item from the main Props menu. Use the Category menu to choose the Locks pane.

To delete locks, select the line that contains the lock and click on the Delete button. To apply the deletion to the `locks` file, click on the Set Default button.

Configuring Environment Variables

Configuring consults environment variables to direct some of its actions. Configuring uses the `CODEMGR_WS` and `CODEMGR_WSPATH` environment variables.

The `CODEMGR_WS` Environment Variable

If you do not explicitly specify a workspace as the focus of a Configuring command, many of the commands will consult the shell environment variable `CODEMGR_WS` to determine a default workspace as the focus of their action. If you have a workspace that is the primary focus of your work, using the `CODEMGR_WS` environment variable allows you to execute the commands without specifying the workspace argument.

The `CODEMGR_WSPATH` Environment Variable

When it is started, Configuring automatically loads workspaces from directory path names specified in the `CODEMGR_WSPATH` environment variable. You can set the `CODEMGR_WSPATH` environment variable to one or more directories that contain workspace directories. For example, to set `CODEMGR_WSPATH` to the directories `/export/home/ws` and `~/projects/ws`, you would use the following command:

```
% setenv CODEMGR_WSPATH "/export/home/ws ~/projects/ws"
```

Note - The `CODEMGR_WS` environment variable overrides the `CODEMGR_WSPATH`. environment variable. That is, the commands that you execute in Configuring use the default workspaces defined in the `CODEMGR_WS` environment variable.

CHAPTER **8**

Copying Files between Workspaces

Chapter 3," describes copying files up and down the parent/child hierarchy. This chapter describes how you use Configuring to copy files.

The chapter covers the following topics:

- "Configuring Transaction Model" on page 87
- "General File Copying Information" on page 88
- "Copying Files from a Parent to a Child Workspace (Bringover)" on page 92
- "Copying Files from a Child to a Parent Workspace (Putback)" on page 102
- "Reversing Bringover and Putback Transactions with Undo " on page 107
- "Renaming, Moving, or Deleting Files" on page 109

An example demonstrating these transactions can be found in Chapter 11."

Configuring Transaction Model

Configuring is designed so that all interworkspace transactions (Bringover Create, Bringover Update, Putback, Undo, and Resolve) are based upon the same user model; that model is described in Figure 8–1. The ways in which the transactions differ are described later in this chapter (the Resolve transaction is described in Chapter 9").

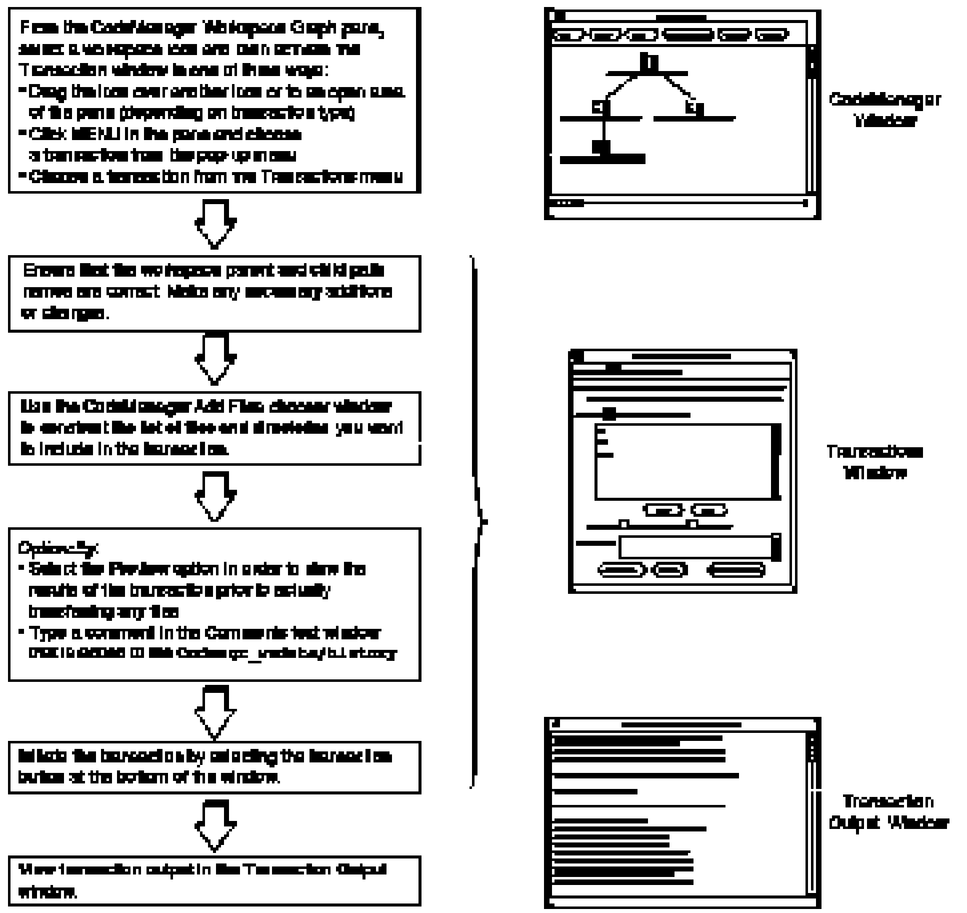

From the CodeManager Workspace Graph pane, select a workspace icon and then activate the Transaction window in one of these ways:
• Drag the icon over another icon or to an open area of the pane (depending on transaction type)
• Click MENU in the pane and choose a transaction from the pop-up menu
• Choose a transaction from the Transactions menu
Ensure that the workspace parent and child path names are correct. Make any necessary additions or changes.
Use the CodeManager Add Files chooser window to construct the list of files and directories you want to include in the transaction.
Optionally:
• Select the Preview option in order to view the results of the transaction prior to actually transferring any files
• Type a comment in the Comments text window that is passed to the Codemgr_wsdata/history
Initiate the transaction by selecting the transaction button at the bottom of the window.
View transaction output in the Transaction Output window.
CodeManager Window
Transactions Window
Transaction Output Window

Figure 8–1 Configuring Transaction Model

General File Copying Information

This section contains background information about copying files between workspaces.

SCCS History Files

When considering Configuring file transfer transactions, it is important to remember that source files are actually derived from SCCS deltas and are identified by SCCS delta IDs (SIDs). When a file is said to be copied by either a Putback or Bringover transaction, Configuring actually acts upon (copies or merges) the file's SCCS history file (also known as the "s-dot-file").

The means by which Configuring manipulates and merges the history files is described in detail in Chapter 10." For specific technical information, refer to the `sccsfile`(4) man page.

Viewing Transaction Output

You view output from Configuring transaction commands in the Transaction Output window. This window is opened automatically when you invoke one of the transactions. You can also activate it yourself by clicking Show Output in any of the Transactions window layouts. Check the online help for details on TeamWare windows.

Note - Configuring transactions are implemented through command-line based programs; some portion of the output contains messages related to the command-line implementation. This manual describes only messages that apply to the actual transactions. If you are interested in more information about the underlying command-line based programs, please refer to the appropriate man pages.

Specifying Directories and Files for Transactions

When you copy files between parent and child workspaces using the Bringover and Putback transactions, you must specify the directories and files you wish included in the transaction. The Bringover Create, Bringover Update, and Putback layouts of the Transactions window contain a File List pane. The File List pane is a scrolling text box in which you construct the list of file and directory names to be included in the transaction. You can accept the default "." convention to bringover or putback all files in a workspace.

Grouping Files for Transfer Using File List Programs

In addition to explicitly specifying files for transfer, you can execute programs that generate that list for you — such a program is called a *File List Program* (or *FLP*). An FLP generates a list of files to `stdout`; the Bringover and Putback transactions read the list of files from `stdout` and include them in the transaction.

Configuring is shipped with a default FLP named `def.dir.flp.` The FLP `def.dir.flp` recursively lists the names of files that are under SCCS control in *directories* that you specify in the File List pane (see "Constructing Directory and File

Lists in the File List Pane" on page 90). The files generated by this (or any) FLP are included for transfer with *files* that you also specify in the File List pane.

If you want to use your own FLPs during a transaction, you can specify their path names in the File List pane.

To use your own FLP:

1. **Select File List Programs (FLPs) in the Parent list box.**

2. **Add FLPs to the list using the Add FLPs dialog box that opens when you click Add.**

   See "Add Files Dialog Box" on page 91 for more information.

Note - You can create your own FLPs that generate lists of files that are useful for your project.

Constructing Directory and File Lists in the File List Pane

Configuring attempts to provide you with a useful initial list of directories and files in the File List pane. You are free to modify the list in any way you wish. The initial list is constructed differently for each type of transaction:

Bringover Create	The initial list is empty.
Bringover Update	The initial list is retrieved from the `Codemgr_wsdata/args` file in the child workspace. This file contains a list of arguments specified during previous Bringover and Putback transactions.
Putback	The initial list is retrieved from the `Codemgr_wsdata/args` file in the child workspace. This file contains a list of arguments specified during previous Bringover and Putback transactions.

Every workspace contains a `Codemgr_wsdata/args` file that is maintained by the Configuring Bringover and Putback transaction commands. The `args` file contains a list of file, directory, and FLP arguments. Initially, the `args` file contains the arguments specified when the workspace was created. If you explicitly specify arguments during subsequent Bringover or Putback transactions, Configuring determines if the new arguments are more encompassing than the arguments already in the `args` file; if the new arguments are of a wider scope, the new arguments replace the old.

Note - You can edit the `args` file at any time to change its contents.

Selecting Files in the File List Pane

Once a list of files and directories exists in the File List pane, you can include or exclude any of them for a given transaction. To be included in a transaction, the file or directory name must be *selected*. You can select or deselect any number of names by moving the pointer over them and clicking. You can select or deselect the entire list by clicking Select All or Deselect All.

Loading and Saving Default Lists

You can reload the default list from the workspace `args` file at any time by clicking Load from args File. This feature is useful if you find that you've made changes to the list that you do not want to keep; you can use Load List from args File to revert the list to its default state.

If you change the default list and wish to make the new list the default in the workspace `args` file, click Save to args File. This is especially useful if you have eliminated files or directories from the list. If you add files, Configuring automatically adds them to the `args` file for you as part of a Bringover or Putback transaction.

Changing the Contents of the File List Pane

You *add* files and directories to the File List pane by using the Add Files dialog box. See "Add Files Dialog Box" on page 91 for details.

You *delete* files and directories from the File List pane by clicking Delete and Deselect All.

Note - You can specify the "." directory as the sole item in the file list to designate that the entire workspace be copied to the child. Enter the "." character using the Name text box in the Add Files dialog box.

Add Files Dialog Box

You can use the Add Files dialog box to conveniently add directories and files to the Transaction window File List pane. Open the Add Files dialog box by clicking Add.

Use the dialog box to navigate down through the file system hierarchy by double-clicking on any directory icon. Double-click on the directory icon to move hierarchically upward in the file system. To move directly to a directory, enter its path name in the Name text box and select the Load Directory button.

Note - The Add Files dialog box does not permit you to navigate outside of the workspace file system.

To add a file or directory to the File List pane:

1. **Select files and directories by moving the pointer over any file or directory icon and clicking.**

   You can extend the selection to include any number of additional files and directories by moving the pointer over them and clicking the middle mouse button.

   You can select entire groups of files by clicking and holding the left mouse button in an empty portion of the dialog box and dragging the bounding box to surround any number of icons. When you release the button, all the files within the bounding box are selected.

   You can also add a file to the File List pane by specifying its path name in the Name text box. If you type Return, the entry will be entered immediately; you may also enter it by clicking Add Files.

2. **Click Add Files to add the file to the File List pane.**

   Note - A check mark in a file icon indicates that the file is checked out from SCCS.

Copying Files from a Parent to a Child Workspace (Bringover)

All Configuring file transfer transactions are performed from the perspective of the child workspace; hence Bringover transactions “bring over” groups of files from the parent to the child workspace. There are two types of Bringover transactions:

Bringover Create	Copy groups of files from a parent workspace to a nonexistent child workspace; the child is created as a result of the Bringover Create transaction.
Bringover Update	Copy files to an existing workspace; the contents of the child are updated as result of the Bringover Update transaction.

Note - You can use the Bringover Update and Create transactions to import directories and files from directories that are not Configuring workspaces. You cannot put back files to directories that are not workspaces.

Creating a New Child Workspace (Bringover Create)

You use the Bringover Create transaction to copy groups of files from a parent workspace to a child workspace that is created as a result of the Bringover transaction. You can display the Bringover Create layout of the Transactions window by any of the following methods:

- Drag and drop a workspace icon onto an empty space in the Workspace Graph pane.
- Select a workspace icon and choose Transactions > Bringover Create.
- Select a workspace icon and choose Bringover > Create from the Workspace Graph pane pop-up menu.
- Select Bringover Create from the Category list box if the Transactions window is already open.

The Bringover Create transaction operates on files that are under SCCS control. When files are said to be copied to the child, the SCCS history file is copied and its g-file (the most recent delta) is created through the SCCS `get` command.

To initiate a Bringover Create transaction:

1. **Specify the parent workspace.**

   If you select a workspace icon on the Workspace Graph pane prior to displaying the Bringover Create window, its path name is automatically inserted in the From Parent Workspace Directory text box. You can edit and change the contents of the text box at any point. You can specify the absolute path name of any accessible workspace; it need not be displayed in the Workspace Graph pane.

Note - You can also specify the path name of directories that are not workspaces to import directories and files into the new workspace.

---

2. **Specify the child workspace.**

   Type the absolute path name of the child that will be created and populated with files from the parent workspace in the To Child Workspace Directory text box.

Note - The parent and child workspaces must be accessible through the file system. Either automounter or NFS® mounts can be used. Accessibility (by users) to workspaces is controlled by the `Codemgr_wsdata/access_control` file in each workspace. Make sure that "bringover-to" and "bringover-from" access for your workspaces are set appropriately. Refer to "Controlling Access to Workspaces" on page 73 for more information.

---

3. **Create a list of directory and file names in the File List pane.**

   You can copy all or part of the contents of the parent workspace to the child. You specify the directories and files you wish to copy in the File List pane. See "Specifying Directories and Files for Transactions" on page 89 for information about specifying directory and file arguments.

Note - If you are using your own FLPs to generate file lists, you also specify them in the File List pane.

Note - If you specify *relative* path names for directory and file names, be aware that they are interpreted as being relative from the top-level (root) directory of the workspace hierarchy (which is assumed to be the same in both parent and child). If you specify these file names using *absolute* path names, the file must be found in one of the two workspaces, or it will be ignored.

---

4. **Select options.**

   - Select the Preview checkbox to preview the results of the transaction. If you invoke the Bringover Create transaction with this option selected, the transaction proceeds without actually transferring any files. You can monitor the output messages in the Transaction Output window and verify the expected outcome of the transaction.
   - Select the Verbose checkbox to increase the information displayed in the Transaction Output window. By default, a message is displayed for each created, updated, or conflicting file. The Verbose option causes Configuring to

print messages for each file, including those that are not brought over. If both the Verbose option and the Quiet option are specified, the Quiet option takes precedence.

- Select the Quiet checkbox to suppress the output of status messages to the Transaction Output window.
- Select the Skip SCCS gets checkbox to inhibit the automatic invocation of the SCCS `get` command as part of the Bringover transaction. Normally g-files are extracted from the SCCS history after they are brought over. This option improves transaction performance although it shifts the responsibility to you to do the appropriate `gets` at a later time.
- Select the Skip Backups checkbox to skip the step of copying the existing files to the `Codemgr_wsdata/backup/files` directory in the destination workspace. This option reduces the disk space occupied by the child workspace and improves transaction performance, but it removes the option of using Undo.

Note - The Force Conflicts option is not applicable to a Bringover Create transaction.

5. **Click Bringover to initiate the transaction.**

Action taken during the Bringover Create transaction can be reversed using the Undo transaction. Refer to “Reversing Bringover and Putback Transactions with Undo ” on page 107 for details.

Checked-out files

When, during a Bringover Create transaction, Configuring encounters files that are checked out from SCCS in the parent, it takes action based on preserving the consistency of the files and any changes to the file that might be in-process.

Table 8–1 shows the different actions that Configuring takes when it encounters checked-out files.

TABLE 8–1 Effects of Checked-out Files on Bringover Create Transactions

File Checked-out in Parent	Configuring Action
g-file and latest delta differ	• Issue a warning • Process file
g-file and latest delta are identical	• Process file

Transaction Output Window

As the transaction proceeds, status information is displayed in the Transaction Output window. Messages are displayed as files are processed during the transaction and a transaction summary is displayed when execution is completed.

Workspace Locks

While Configuring is reading and examining files in the parent workspace during a Bringover transaction, it obtains a *read-lock* for that workspace. When it is manipulating files in the child workspace, it obtains a *write-lock.*

Read-locks may be obtained concurrently by multiple Configuring commands that read files in the workspace; no commands may write to a workspace while any read-locks are in force. Only a single write-lock can be in force at any time; no Configuring command may write to a workspace while a write-lock is in force. Lock status is controlled by the `Codemgr_wsdata/locks` file in each workspace.

If you attempt to bring over files into a workspace that is locked, you will be so notified with a message that states the name of the user that has the lock, the command they are executing, and the time they obtained the lock.

```
bringover: Cannot obtain a write lock in workspace ‘‘/tmp_mnt/home/my_home/projects/mpages’’
because it has the following locks:
 Command: bringover (pid 20291), user: jack, machine: holiday, time: 12/02/91 16:25:23
  (Error 2021)
```

History File

Configuring records information regarding the Bringover transaction in the `Codemgr_wsdata/history` file. This information can be useful to you as a means of tracking changes that have been made to files in your workspaces. Refer to

"Viewing Workspace Command History" on page 80 for further information regarding these files.

Updating an Existing Child Workspace (Bringover Update)

You use the Bringover Update transaction to update an existing child workspace. You can display the Bringover Update layout of the Transactions window by any of the following methods:

- Drag and drop a workspace icon on top of the icon of a child workspace.
- Select a child workspace icon and choose Transactions > Bringover Update.
- Select a child workspace icon and choose Bringover > Update from the Workspace Graph pane pop-up menu.
- Select Bringover Update from the Category list box if the Transactions window is already open.

The Bringover Update transaction transfers files that are under SCCS control. When a file exists in the parent workspace but not in the child, its SCCS history file is copied to the child and its g-file (the most recent delta) is created through the SCCS `get` command. When a file exists in both workspaces and has changed only in the parent, Configuring copies the new deltas from the parent to the child. When a file has changed in both workspaces, Configuring moves the *child's* new deltas into an SCCS branch.

To initiate a Bringover Update transaction:

1. **Specify the child workspace.**

   If you select a workspace icon on the Workspace Graph pane prior to displaying the Bringover Update window, its name is automatically inserted in the To Child Workspace Directory text box. You can insert a new path names, and edit and change the text box at any point.

2. **Specify the parent workspace.**

   The name of the selected child's parent workspace is automatically inserted in the From Parent Workspace text box. The parent workspace name is retrieved from the Configuring metadata file `Codemgr_wsdata/parent`.

   Note - You can also specify the path name of directories that are not workspaces to import files and directories into the workspace.

   You can change a child workspace's parent for the duration of a single Bringover Update transaction by specifying the new parent's path name in the From Parent

Workspace text field. You change the parent for that transaction only; if you wish to permanently change a workspace's parent, choose Edit > Parent in the Configuring main window or drag the child workspace icon over the new parent's icon. See "Reparenting a Workspace" on page 70 for details regarding reparenting workspaces.

Note - If you type the child workspace name and no icons are selected in the Workspace Graph pane, Configuring automatically updates the parent field if you reselect Bringover Update in the Category list box.

Note - The parent and child workspaces must be accessible through the file system. Either automounter or NFS® mounts can be used. Accessibility (by users) to workspaces is controlled by the `Codemgr_wsdata/access_control` file in each workspace. Ensure that "bringover-to" and "bringover-from" access for your workspaces are set appropriately. Refer to "Controlling Access to Workspaces" on page 73 for more information.

---

3. **Create a list of directory and file names in the File List pane.**

   You can copy all or part of the contents of the parent workspace to the child. You specify the directories and files you wish to copy in the File List pane. See "Specifying Directories and Files for Transactions" on page 89 for information about specifying directory and file arguments.

Note - If you are using your own FLPs to generate file lists, you also specify them in the File List pane.

Note - If you specify *relative* path names for directory and file names be aware that they are interpreted as being relative from the top-level (root) directory of the workspace hierarchy (which is assumed to be the same in both parent and child). If you specify these file names using *absolute* path names, the file must be found in one of the two workspaces or it will be ignored.

---

4. **Select options.**

   - Select the Preview checkbox to preview the results of the transaction. If you invoke the Bringover Update transaction with this option selected, the transaction proceeds without actually transferring any files. You can monitor the output messages in the Transaction Output window and verify the expected outcome of the transaction.

- Select the Verbose checkbox to increase the information displayed in the Transaction Output window. By default, a message is displayed for each created, updated, or conflicting file. The Verbose option causes Configuring to print a message for each file, including those that are not brought over. If both the Verbose option and the Quiet option are specified, the Quiet option takes precedence.
- Select the Quiet checkbox to suppress the output of status messages to the Transaction Output window).
- Select the Skip SCCS gets checkbox to inhibit the automatic invocation of the SCCS `get` command as part of the Bringover transaction. Normally g-files are extracted from the SCCS history after they are brought over. This option improves transaction performance although it shifts the responsibility to the user to do the appropriate `get`s at a later time.
- Select the Force Conflicts checkbox to cause all updates to be treated as conflicts.
- Select the Skip Backups checkbox to skip the step of copying the existing files to the `Codemgr_wsdata/backup/files` directory in the destination workspace. This option reduces the disk space occupied by the child workspace and improves transaction performance, but it removes the option of using Undo.

5. Click Bringover to initiate the transaction.

Action taken during the Bringover Update transaction can be reversed using the Undo transaction. Refer to “Reversing Bringover and Putback Transactions with Undo ” on page 107 for details.

Checked-out files

When, during a Bringover Update transaction, Configuring encounters files that are checked-out from SCCS, it takes action based on preserving the consistency of the files and any changes to the file that might be in process.

Table 8–2 shows the different actions that Configuring takes when it encounters checked-out files.

TABLE 8–2 Effects of Checked-out Files on Bringover Update Transactions

File Checked-out in Parent	File Checked-out in Child	Configuring Action
g-file and latest delta differ		• Issue a warning • Process file
g-file and latest delta are identical		• Process file
	g-file and latest delta are identical	• Uncheckout the file • Process the file • Checkout the file
	g-file and latest delta differ	• Create a conflict
	g-file is readonly	• Issue a warning • Do not process the file

Transaction Output Window

As the transaction proceeds, status information is displayed in the Transaction Output window. Messages are displayed as files are processed during the transaction and a transaction summary is displayed when execution is completed.

Conflicts

Bringover Update transactions often produce conflicts (when files are changed in both the parent and child). When this occurs, you are so notified by messages in the Transaction Output window. See Chapter 9,” for details about resolving conflicts.

Workspace Locks

While files are read and examined in the parent workspace during the transaction, Configuring obtains a *read-lock* for that workspace. When Configuring manipulates files in the child workspace, it obtains a *write-lock*.

Read-locks may be obtained concurrently by multiple Configuring commands that read files in the workspace; no commands may write to a workspace while any read-locks are in force. Only a single write-lock may be in force at any time; no Configuring command may write to a workspace while a write-lock is in force. Lock status is controlled by the `Codemgr_wsdata/locks` file in each workspace.

If you attempt to bring over files into a workspace that is locked, you will be so notified with a message that states the name of the user that has the lock, the command they are executing, and the time they obtained the lock.

```
bringover: Cannot obtain a write lock in workspace ‘‘/tmp_mnt/home/my_home/projects/mpages’’
because it has the following locks:
 Command: bringover (pid 20291), user: jack, machine: holiday, time: 12/02/91 16:25:23
  (Error 2021)
```

History File

Bringover Update transaction information is recorded in the `Codemgr_wsdata/history` file. This information can be useful as a means of tracking changes that have been made to files in your workspaces. Refer to "Viewing Workspace Command History" on page 80 for further information regarding these files.

Bringover Action Summary

Table 8–3 summarizes the actions that Configuring takes during Bringover transactions.

TABLE 8–3 Summary of Configuring Actions During a Bringover Transaction

File in Parent	File in Child	Action by Configuring
Exists	Does not exist	Create the file in the child
Does not exist	Exists	None
Unchanged	Unchanged	None
Unchanged	Changed	None
Changed	Unchanged	Update file in the child. (Merge SCCS files and extract [via `get`] a g-file that consists of the most recent delta.)
Changed	Changed	Merge SCCS history files in the child, create conflict, and notify user of the conflict. Current line of work in the child is moved to an SCCS branch.

Copying Files from a Child to a Parent Workspace (Putback)

All Configuring file transfer transactions are performed from the perspective of the child workspace; hence the Putback transaction "puts back" groups of files from the child to the parent workspace.

You use the Putback transaction to make the parent and child workspace identical with respect to the set of files that you specify for the Putback transaction. Use the Putback transaction after you make changes and test them in the child workspace. Putting the files back into the parent usually makes them accessible to other developers.

During a Putback transaction, Configuring may find that it cannot transfer files from the child to the parent workspace without endangering the consistency of the data in the parent. If this occurs, no files are transferred and the Putback transaction is said to be *blocked*. A Putback transaction is blocked because:

- A file in either workspace is currently checked out from SCCS
- A file in the parent workspace contains changes not yet brought over into the child workspace
- A file conflict in either workspace is currently unresolved

The Putback transaction transfers files that are under SCCS control. When a file exists in the child workspace but not in the parent, its SCCS history file is copied to the parent and its g-file (the most recent delta) is materialized through the SCCS `get` command. When a file exists in both workspaces and has changed only in the child, Configuring copies the new deltas from the child to the parent. When a file has changed in the parent, or both the parent and child, the Putback transaction is blocked.

Updating a Parent Workspace Using Putback

You can display the Putback layout of the Transactions window by any of the following methods:

- Drag and drop a child workspace icon onto a parent workspace icon.
- Select a workspace icon and choose Transactions > Putback.
- Select a workspace icon and choose Putback from the Workspace Graph pane pop-up menu.
- Select Putback from the Category list box if the Transactions window is already open.

To initiate a Putback transaction:

1. **Specify the child workspace.**

   If you select a workspace icon on the Workspace Graph pane prior to displaying the Putback window, its name is automatically inserted in the From Child Workspace Directory text box. You can insert new path names, and edit and change the text box at any point.

2. **Specify the parent workspace.**

   The name of the selected child's parent workspace is inserted in the From Parent Workspace text box. The parent workspace name is retrieved from the Configuring metadata file named `Codemgr_wsdata/parent`.

   You can change a child workspace's parent for the duration of a single Putback transaction by specifying the new parent's path name in the To Parent Workspace text box. You change the parent for that transaction only; if you wish to permanently change a workspace's parent, choose Edit > Parent in the Configuring main window or drag the child workspace icon over the new parent's icon. See "Reparenting a Workspace" on page 70 for details regarding reparenting workspaces.

   Note - If you enter the child workspace name by hand and no icons are selected in the Workspace Graph pane, Configuring automatically updates the parent field if you reselect Putback in the Category list box.

   Note - The parent and child workspaces must be accessible through the file system. You can use either automounter or NFS mounts. Accessibility (by users) to workspaces is controlled by the `Codemgr_wsdata/access_control` file in each workspace. Ensure that "putback-to" and "putback-from" access for your workspaces are set appropriately. Refer to "Controlling Access to Workspaces" on page 73 for more information.

   ---

3. **Create a list of directory and file names in the File List pane.**

   You can copy all or part of the contents of the parent workspace to the child. You specify the directories and files you wish to copy in the File List pane. See "Specifying Directories and Files for Transactions" on page 89 for information about specifying directory and file arguments.

   Note - If you are using your own FLPs to generate file lists, you also specify them in the File List pane.

Note - If you specify relative path names for directory and file names, be aware that they are interpreted as being relative from the top-level (root) directory of the workspace hierarchy (which is assumed to be the same in both parent and child). If you specify these file names using absolute path names, the file must be found in one of the two workspaces or it will be ignored.

4. **Select options.**
   - Select the Preview checkbox to preview the results of the transaction. If you invoke the Putback transaction with this option selected, the transaction proceeds without actually transferring any files. You can monitor the output messages in the Transaction Output window and verify the expected outcome of the transaction.
   - Select the Verbose checkbox to increase the information displayed in the Transaction Output window. By default, a message is displayed for each created, updated, or conflicting file. The Verbose option causes Configuring to print a message for each file, including those that are not put back. If both the Verbose option and the Quiet option are specified, the Quiet option takes precedence.
   - Select the Quiet checkbox to suppress the output of status messages to the Transaction Output window.
   - Select the Skip SCCS gets checkbox to inhibit the automatic invocation of the SCCS `get` command as part of the Putback transaction. Normally g-files are extracted from the SCCS history after they are brought over. This option improves transaction performance although it shifts the responsibility to the user to do the appropriate `get`s at a later time.
   - Select the Auto Bringover checkbox to cause Configuring to automatically start a Bringover Update transaction to update files in the child if the Putback transaction is blocked.
   - Select the Skip Backups checkbox to skip the step of copying the existing files to the `Codemgr_wsdata/backup/files` directory in the destination workspace. This option reduces the disk space occupied by the parent workspace and improves transaction performance, but it removes the option of using Undo.

5. **Type a comment.**

   Type a comment that describes the Putback transaction. This comment is included with the transaction log written into the file called `Codemgr_wsdata/history` in the parent workspace. The comment can be up to 8 Kbytes long.

6. **Click Putback to initiate the transaction.**

Action taken during the Putback transaction can be reversed using the Undo transaction. Refer to "Reversing Bringover and Putback Transactions with Undo " on page 107 for details.

Checked-out files

When, during a Putback transaction, Configuring encounters files that are checked-out from SCCS, it takes action based on preserving the consistency of the files and any changes to the file that might be in-process.

Table 8–4 shows the different actions that Configuring takes when it encounters checked-out files.

TABLE 8–4 Effects of Checked-out Files on Putback Transactions

File Checked-out in Parent	File Checked-out in Child	Configuring Action
g-file and latest delta differ		•Block Putback transaction
g-file and latest delta are identical or g-file does not exist)		•Uncheckout the file •Process the file •Check-out the file
	g-file and latest delta differ	•Block Putback transaction
	g-file and latest delta are identical	•Process the file
	g-file does not exist	•Issue a warning •Process the file •No changes made

Transaction Output Window

As the transaction proceeds, status information is displayed in the Transaction Output window. Messages are displayed as files are processed during the transaction, and a transaction summary is displayed when execution is completed.

Workspace Locks

While files are read and examined in the child workspace during the transaction, Configuring obtains a *read-lock* for that workspace. When Configuring manipulates files in the parent workspace it obtains a *write-lock*.

Read-locks may be obtained concurrently by multiple Configuring commands that read files in the workspace; no commands may write to a workspace while any read-locks are in force. Only a single write-lock may be in force at any time; no Configuring command may write to a workspace while a write-lock is in force. Lock status is controlled by the `Codemgr_wsdata/locks` file in each workspace.

If you attempt to put back files into a workspace that is locked, you are notified with a message such as the following that states the name of the user that has the lock, the command they are executing, and the time they obtained the lock.

```
putback: Cannot obtain a write lock in workspace ‘‘/tmp_mnt/home/my_home/projects/mpages’’
because it has the following locks:
 Command: bringover (pid 20291), user: jack, machine: holiday, time: 12/02/91 16:25:23
  (Error 2021)
```

History File

Putback transaction information is recorded in the file called `Codemgr_wsdata/history`. This information can be useful as a means of tracking changes that have been made to files in your workspaces. Refer to “Viewing Workspace Command History” on page 80 for further information regarding these files.

Putback Action Summary

Table 8–5 summarizes the actions that Configuring takes during Putback transactions.

TABLE 8–5 Summary of Configuring Actions During a Putback Transaction

File in Parent	File in Child	Action by Configuring
Exists	Does not exist	Block Putback and notify user
Does not exist	Exists	Create the file in the parent
Unchanged	Unchanged	None

TABLE 8–5 Summary of Configuring Actions During a Putback Transaction *(continued)*

File in Parent	File in Child	Action by Configuring
Unchanged	Changed	Update file in the parent. (Merge SCCS files and extract [via `get`] a g-file that consists of the most recent delta.)
Changed	Unchanged	Block Putback, notify user.
Changed	Changed	Block Putback, notify user.
Checked out	Checked out*	Block Putback, notify user.
Unresolved conflict	Unresolved conflict**	Block Putback, notify user.

*If a file is checked out in either the parent or the child, the transaction is blocked. See Table 8–4 for more information about putting back files that are checked out.

**If a conflict is unresolved in either the parent or the child, the transaction is blocked.

Reversing Bringover and Putback Transactions with Undo

You can reverse (undo) the action of the *most recent* Bringover or Putback transaction in a workspace by using the Undo Transactions window layout. You undo the Putback or Bringover transaction in the destination workspace (the one in which the files are changed). You can undo a Bringover or Putback transaction as many times as you like until another Bringover or Putback transaction makes changes in that workspace; only the *most recent* Bringover or Putback transaction can be undone.

If a file is updated or found to be in conflict by the Putback or Bringover transaction, the Undo transaction restores the file to its original state. If a file is "new" (created by the Bringover/Putback transaction), then it is deleted.

To initiate an Undo transaction:

1. **Specify the workspace in which to reverse the transaction.**

   If you select a workspace icon on the Workspace Graph pane prior to displaying the Undo layout, its name is automatically inserted in the Workspace Directory text box. You can insert a new path name followed by a Return, and edit and change the text box at any point.

2. **Click Undo to initiate the transaction.**

Note - If you have selected Skip Backups in the Transactions window, you cannot undo the transaction.

Workspace Locks

When it is manipulating files in the specified workspace, Configuring obtains a *write-lock* for the workspace. Only a single write-lock may be in force at any time; no Configuring command may write to a workspace while a write-lock is in force. Lock status is controlled by the `Codemgr_wsdata/locks` file in each workspace. If Configuring cannot obtain the lock, it will display an error message and abort.

History File

Configuring records information regarding the Undo transaction in the `Codemgr_wsdata/history` file. This information can be useful as a means of tracking changes that have been made to files in your workspaces. Refer to “Viewing Workspace Command History” on page 80 for further information regarding these files.

How the Undo Transaction Works

When the Bringover and Putback transactions update or create files in the destination workspace (the child in the case of Bringover, the parent in the case of Putback), they make backup copies of the originals before they actually make changes to the files. All existing files are copied to the `Codemgr_wsdata/backup/files` directory in the destination workspace, and the names of all newly created files are entered into a file called `Codemgr_wsdata/backup/new`.

When you decide that you would like to cause a workspace to revert to its state before a Bringover/Putback transaction, the Undo transaction does the following (see Figure 8–2):

- Copies the backed-up files from the `Codemgr_wsdata/backup/files` directory over the transferred files
- Deletes files whose names are contained in the `Codemgr_wsdata/backup/new` file

  The next Bringover/Putback transaction overwrites all data in the `Codemgr_wsdata/backup` directory.

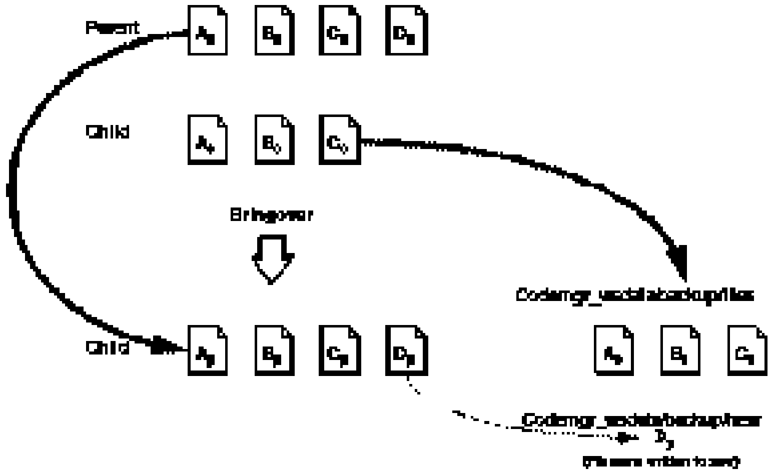


Figure 8–2 How the Undo Transaction Works

Note - All files transferred by Configuring are under SCCS control. Usually, only SCCS history files are backed up during Bringover and Putback transactions; if the files are subsequently restored, the Undo transaction extracts the appropriate g-file (most recent delta) from the history file. If, however, a file in the child is checked out (using `sccs edit`) during the Bringover transaction (Configuring permits files to be checked out during a Bringover transaction, but not during a Putback transaction. If a file that is being put back is checked out, an error condition exists). Configuring backs up *both* the g-file and the SCCS history file in order to preserve the work in progress; the g-file and the SCCS history file are copied to the `Codemgr_wsdata/backup/files` directory and restored by the Undo transaction.

Renaming, Moving, or Deleting Files

When you rename, move, or "delete" files, Configuring tracks those changes so that it knows how to manage the altered files during Bringover and Putback transactions. Although Configuring processes these files automatically, it is helpful for you to understand some of the ramifications of renaming, moving, or deleting files.

Note - For the purposes of this discussion, the terms "rename" and "move" are considered to be the same action and are referred to only as "rename."

The best way to delete and rename files is to use the move and delete commands available from the TeamWare Versioning menu. This section describes the underlying process.

Renaming Files

When you bring over or put back files that you (or another user) have renamed, Configuring must decide whether the files have been newly created or whether they existed previously and have been renamed. When you rename a file, you must rename *both* the g-file *and* the SCCS history file. Configuring propagates the name change throughout the workspace hierarchy using the same rules used with file content updates and conflicts.

During transactions, Configuring processes files individually. When you rename a directory, each file in the directory is evaluated separately as if each had been renamed individually.

Renaming Example

In Figure 8–3, the name of file C in the parent is changed to D. When Configuring brings the file over to the child it must decide which of the following is true:

- D has been newly created in the parent.
- It is the same file as C in the child, only with a new name.

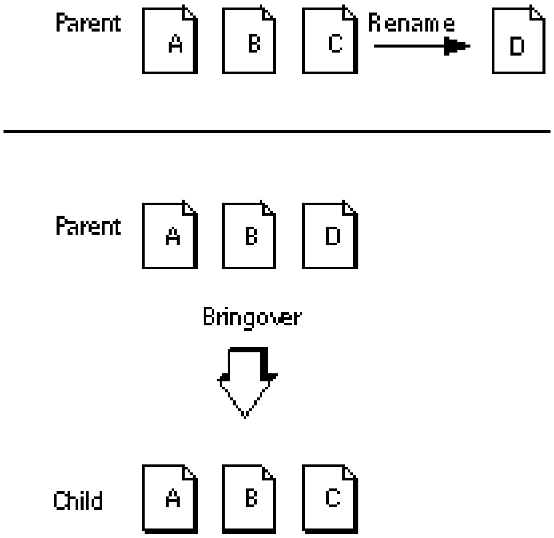


Figure 8–3 File “C” Renamed to “D”

If the same case was the subject of a Putback operation, the same problem would apply: Is “C” new in the child, or has it been renamed from some other file?

The action that Configuring takes is very different in each case. If it is a new file in the parent, Configuring creates it in the child; if it has been renamed in the parent, Configuring renames file “C” to “D” in the child.

Configuring stores information in the SCCS history files that enables it to identify files even if their names are changed. You may have noticed the following message when viewing Bringover and Putback output:

```
Examined files:
```

Configuring examines all files involved in a Bringover Update or Putback transaction for potential rename conditions before it begins to propagate files.

When Configuring encounters renamed files, it propagates the name change to the child in the case of Bringover, and to the parent in the case of Putback. You are informed of the change in the Transaction Output window with the following messages:

```
rename from: old_filename
         to: new_filename
```

Name History

Configuring stores information about a file’s name history in its SCCS history file. The name history is simply a list of the workspace-relative names that have been given to the file during its lifetime. This information is used by Configuring to differentiate between files that have been renamed and those that are new. When you rename a file, Configuring updates the file’s name history during the next Bringover or Putback transaction that includes it. When a name history is updated, you are notified in the Transaction Output window.

```
Names Summary:
   1 updated parent’s name history
   1  updated children’s name history
```

Rename Conflicts

In rare cases, a file’s name is changed concurrently in parent and child workspaces. This is referred to as a *rename conflict*. For example, the name of file “C” is changed to “D” in the parent, and concurrently to “E” in the child.

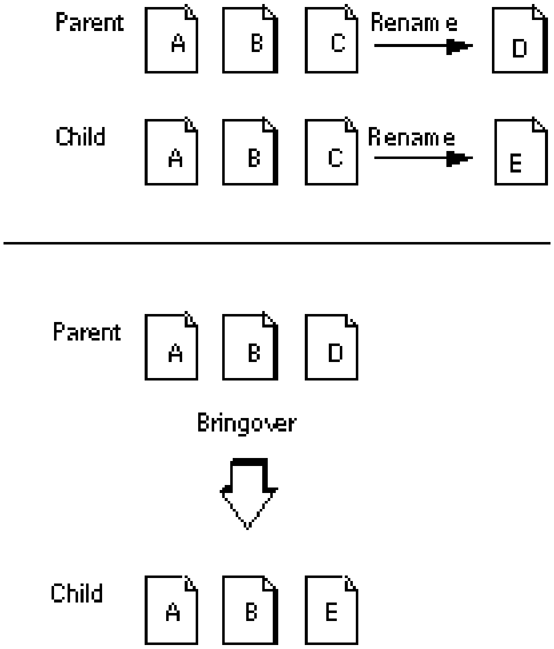


Figure 8–4 File "C" is Concurrently Renamed in both Parent and Child Workspaces

When this occurs, Configuring determines that both "D" in the parent and "E" in the child are actually the same file, but with different names. In the case of rename conflicts:

- Configuring reports the conflict using the name of the file in the child.
- Configuring always resolves the conflict by automatically changing the name of the file in the child workspace to the current (renamed) name in the parent; the name of the file from the parent is *always* chosen, even in the case of a Putback transaction.

When Configuring encounters a rename conflict, you are notified in the Transaction Output window with the following messages:

```
rename conflict: name_in_child
rename from: name_in_child
         to: name_in_parent
```

Deleting Files

Deleting files from a Configuring workspace is a little trickier than it first appears. Deleting a file from a workspace with the `rm` command causes Configuring to think that the file has been newly created in the workspace's parent or child.

Take for instance, the following example. The file “C” is removed from the child workspace using the `rm` command; later the Bringover Update transaction is used to update the child.

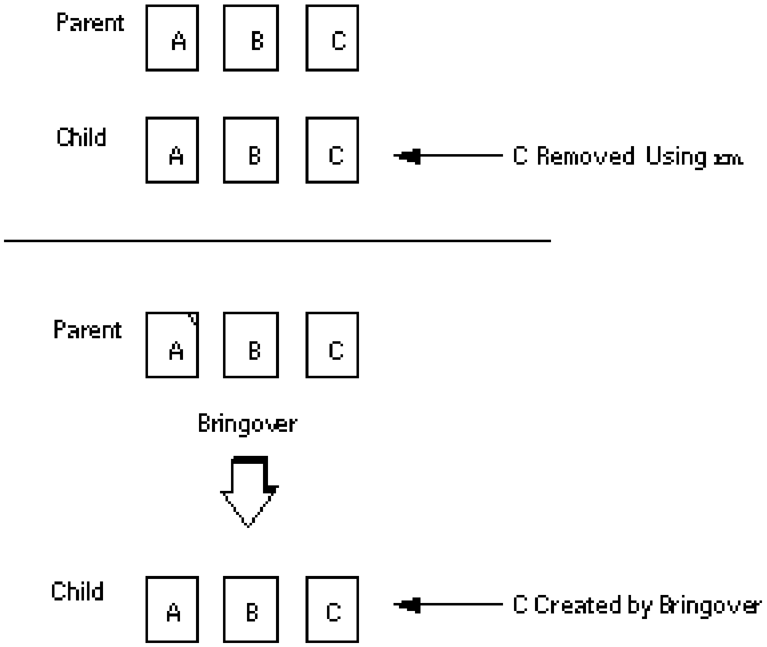


Figure 8–5 File “C” Is Removed From The Child Using the `rm` Command, Then Created Again by Bringover

Configuring examines the two workspaces and determines that the file “C” exists in the parent and not in the child — following the usual Configuring rules, it creates “C” in the child.

The recommended method for “deleting” files in workspaces is to rename them out of the way using a convention agreed upon by everyone working on the project. One recommended method is to rename files you wish to “delete” so that they begin with the `.del-` prefix.

```
example% mv module.c .del-module.c example% mv SCCS/s.module.c SCCS/
s..del-module.c
```

For example:

This method has a number of advantages:

- The file is no longer seen using default commands such as `ls`.
- Configuring does not recreate the file.

- Configuring propagates the change throughout the workspace hierarchy as a rename, “deleting” the file in all workspaces.
- The file remains available to later reconstruct releases for which it was a part (for example, if it was part of a freezepoint (see Chapter 5” and Chapter 15Chapter 15 for more information about freezepoints).

CHAPTER **9**

Resolving Conflicts

This chapter discusses the process by which the Configuring program detects conflicts and then assists you in resolving these conflicts.

The chapter covers the following topics:

Conflict Resolution Process

When files change concurrently in both a parent and child workspace, they are in conflict. Neither the version of the file in the child nor the version in the parent can be copied to the other without overwriting changes. Conflicts are detected during Bringover Update transactions. You must resolve conflicts in the child before the conflicting file(s) can be put back to the parent. The Configuring program assists you in resolving conflicts. Use the following procedures to resolve conflicts if you are using the GUI. If you are using the CLI, see the `resolve`(1) man page for more information.

Detecting Conflicts

Before you can resolve conflicts, Configuring must detect the conflict and prepare the history files of the conflicting files for resolving.

Detecting Conflicts During Bringover Update Transactions

Usually, the conflict resolution process begins when you attempt to put back files that have changed in both the parent and the child workspaces. The Putback transaction blocks the transfer of files from the child to the parent because the version of the file from the child will overwrite changes made in the parent.

After the Putback transaction is blocked, you must use the Bringover Update transaction to update the child. (If Putback is executed with the Auto Bringover option specified, then the Bringover transaction is initiated automatically by Configuring.) If, during the Bringover transaction, Configuring determines that the file in the child has *also* changed, a conflict exists. All files included in the Bringover Update transaction that are *not* in conflict are copied or updated normally.

Preparing Files for Conflict Resolution

When a conflict is encountered during a Bringover Update transaction, Configuring takes special steps to prepare that file so that you can resolve the conflict.

Configuring incorporates the deltas created in the parent into the SCCS history file in the child. The parent and child deltas are placed on separate branches in the child SCCS history file. After the deltas are merged, the history file in the child contains:

- Delta(s) created in the parent
- Delta(s) created in the child
- The delta from which the two versions of the file are both descended (their *common ancestor*)

Note - Versioning enables you to view graphical depictions of SCCS delta histories (including branches).

Access to the three deltas (common ancestor, parent, and child) in the child enables you to use the Resolve transaction and Merging to compare the parent and child deltas — both to their common ancestor, and each to the other.

In addition to merging deltas, Configuring adds the name of the conflicted file to the child’s `Codemgr_wsdata/conflicts` file. The `conflicts` file is a text file that contains the names of all files in that workspace with unresolved conflicts.

The stage is set for you to resolve the conflicts using the Resolve transaction.

Resolving Conflicts

You use two tools to resolve conflicts:

- The Resolve layout of the Transactions window
- TeamWare Merging

Resolve Transaction

The Resolve layout of the Transactions window facilitates resolving conflicts detected during Bringover Update transactions. The Resolve transaction coordinates the merging process, acting as intermediary between you and TeamWare Merging.

When Configuring detects a conflict during a Bringover Update transaction, it does the following:

- Merges new deltas from the parent into the SCCS history in the child
- Enters the file's path name in the child's `Codemgr_wsdata/conflicts` file

By default, Configuring automatically processes sequentially the list of files from the File List pane. After you resolve a conflict, Configuring automatically begins to process the next file in the list. If you want to change the behavior so that it individually processes only files that you select explicitly, deselect the Auto Advance checkbox in the Resolve pane of the Tool Properties dialog box.

Conflicts need not be resolved immediately. You can continue to make changes and create new deltas in conflicted files in the child workspace. New deltas are created on a branch; when you finally resolve the conflict, the latest delta is the one merged with the version brought over from the parent. *Conflicts must be resolved before you can put back the files to the parent*.

When Configuring creates the new delta in the child SCCS history file, it includes the following standard comment:

```
Merged changes between workspaces x and y
```

By default, Configuring does not prompt you for a comment to append to its comment. If you want to be prompted for comments that are appended to the standard comment, select the Prompt for Checkin Comments checkbox in the Resolve pane of the Tool Properties dialog box.

To resolve conflicts in a workspace:

1. **Double-click on the icon of a workspace that contains conflicted files.**

   The Resolve layout of the Transaction window opens with the names of its conflicted files displayed in the File List pane.

2. **Select a file in the File List pane and click Merging.**

   Configuring starts Merging and begins to process the list of files in the File List pane. For each file in the list, Configuring extracts the parent delta, the child delta, and the common ancestor from the SCCS history file and passes their path names to Merging. The Merging window opens with the files loaded and ready for merging.

3. **Use Merging to resolve the differences between the parent and child versions of the file.**

   See Chapter 13" for more information.

4. **Save the file in Merging.**

   After you use Merging to resolve differences between the parent and child versions of the file, Configuring creates a new delta in the child SCCS history file and removes the file name from the `conflicts` file. The new delta contains the merged result you created using Merging.

Merging

This section is a brief introduction to Merging as used with Configuring. For a more detailed description, see Chapter 13."

Merging Window

Merging displays two text files (the parent and child deltas) for side-by-side comparison, each in a read-only pane (see Figure 4–1). Beneath them is a pane that contains a merged version of the two files. The merged version contains selected lines from either or both deltas and can be edited to produce a final merged version.

Each delta in each of the top panes is shown in comparison to the common ancestor delta:

- The child delta is displayed in the left pane labeled Child
- The parent delta is displayed in the right pane labeled Parent

The common ancestor is the delta from which both the parent and child deltas are descended. This arrangement permits you to make a three-way comparison—each delta to the common ancestor, and each delta to the other.

Marking of Differences

Lines in each descendant are marked according to their relationship to the corresponding lines in the common ancestor:

- If a line is identical in all three deltas, then no glyph appears.
- If a line is not in the ancestor but was added to one or both of the descendants, then a plus sign glyph (+) appears next to the line in the delta where the line was added.
- If a line is present in the ancestor but was removed from one or both of the descendants, then a minus sign (-) appears as a placeholder in the delta from which the line was removed.
- If a line is in the ancestor but has been changed in one or both of the descendants, then a vertical bar glyph (|) appears next to the line in the delta where the line was changed.

When Merging discovers a line that differs between either of the two deltas and the ancestor, it marks with glyphs the lines in the two deltas and also in the automatically merged file. Together, these marked lines are called a *difference*. While Merging is focusing on a difference, it highlights the glyphs.

The difference on which Merging is focusing at any given time is called the *current difference*. The difference that appears immediately later in the file is called the *next difference*; the difference that appears immediately earlier in the file is called the *previous difference*.

Resolving Differences

While focusing on a difference, you can accept a line from either of the original deltas, or you can edit the merged version by hand. When you indicate that you are satisfied with your changes (by clicking on a control panel button), the current difference is said to be *resolved*. After a difference is resolved, Merging changes the glyphs that mark the difference to outline (hollow) font. Merging then automatically advances to the next difference (if the Auto Advance property is on), or moves to another difference of your choice.

CHAPTER **10**

How Configuring Merges SCCS Files

This chapter describes the ways the Configuring program manipulates SCCS history files when you copy files between workspaces and resolve conflicts.

- "Merging Files That Do Not Conflict" on page 122
- "Merging Files That Conflict" on page 123
- "An Example of Merging" on page 123

Note - This discussion assumes that you are familiar with SCCS, including the concept of branching. SCCS is described in detail in the Solaris *Programming Utilities* manual.

When considering Bringover and Putback transactions, remember that source files are derived from SCCS deltas and are identified by SCCS delta IDs (SIDs). When a *file* is copied by either a Putback or Bringover transaction, the Configuring program must manipulate the file's SCCS history file (also known as the "s-dot-file").

When a file is copied (by means of Bringover or Putback transaction) from a source workspace to a destination workspace, it appears that a single file has been transferred. In fact, all of the SCCS information for that file (deltas, comments, and so on) must be merged into the destination SCCS history file. By merging the information from the source into the destination history file, the current version (delta) can be derived, and the file's entire delta and comment history are available. (The exception is when the file does not exist in the destination workspace. In this case, the entire history file is copied from the source workspace to the destination workspace.)

Merging Files That Do Not Conflict

If the file in the destination workspace is being updated (the file has changed in the source of a Bringover or Putback transaction and has not changed in the destination), the new deltas from the destination are added to the history file in the destination. The reason that SCCS history files are merged at all in this case, rather than the source history file being copied over the destination history file, is that administrative information (for example, flags and access lists) stored in the destination history file would be overwritten.

To accomplish the merger, the Configuring program determines where the delta histories diverge and adds (to the destination workspace) only the deltas that were created in the source workspace since they diverged. To determine where the histories diverge, the Configuring program compares the delta tables in both the parent and child history files; information used in this comparison includes comments and data such as when and who created the delta. Figure 10–1 contains an example of a Putback transaction where the Configuring program adds deltas 1.3 and 1.4 from the child workspace to the SCCS history file in the parent.

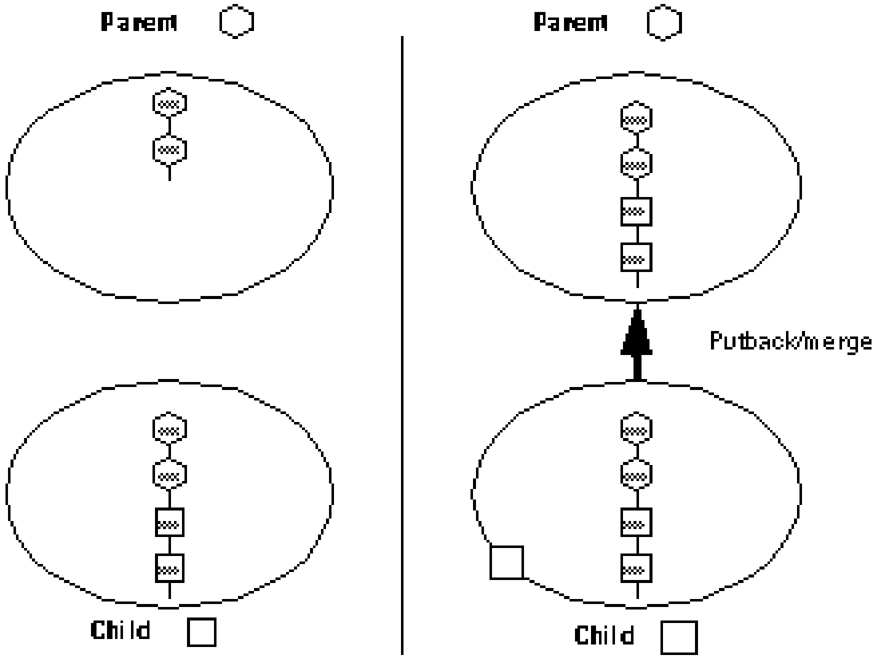


Figure 10–1 Updating a File in the Destination Workspace That Has Not Changed

Merging Files That Conflict

When you propagate files between parent and child workspaces, often both the version of the file from the parent and the version in your child changed since they were last updated. When that is the case, the parent and child versions of the file are in conflict.

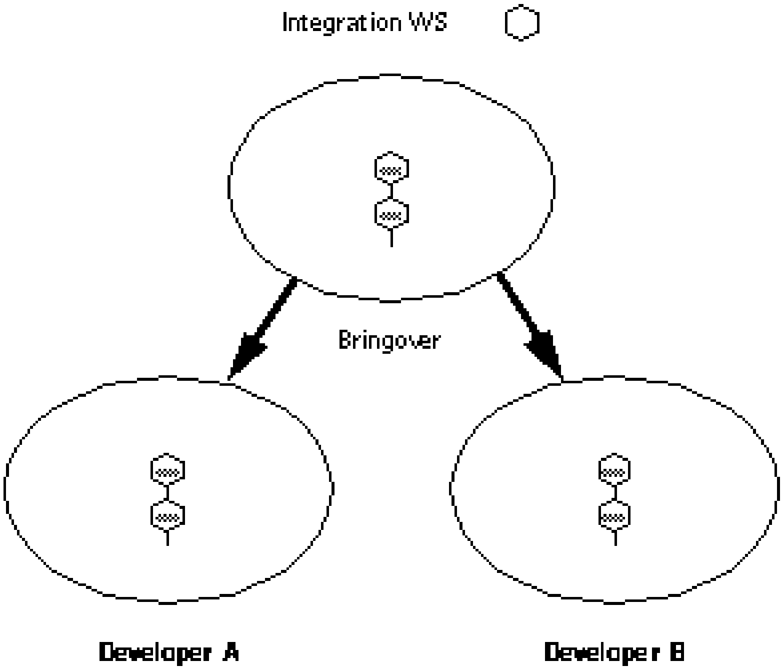
When file contents conflict, Configuring aids you in resolving the potentially conflicting changes that were made to the file, and preserves the file's delta, administrative, and comment history. To accomplish this, Configuring merges the SCCS deltas from the parent into the history file in the child. Configuring's Resolve transaction is then used to resolve the conflict in the child. See Chapter 9" for details on resolving conflicts.

An Example of Merging

This merge example involves an integration workspace and two child workspaces owned by different developers. The developers bring over copies of the same file from the integration workspace, and independently change the file. The illustrations show how the SCCS history file is manipulated when conflicts occur and when they are resolved. Some notes regarding the following figures:

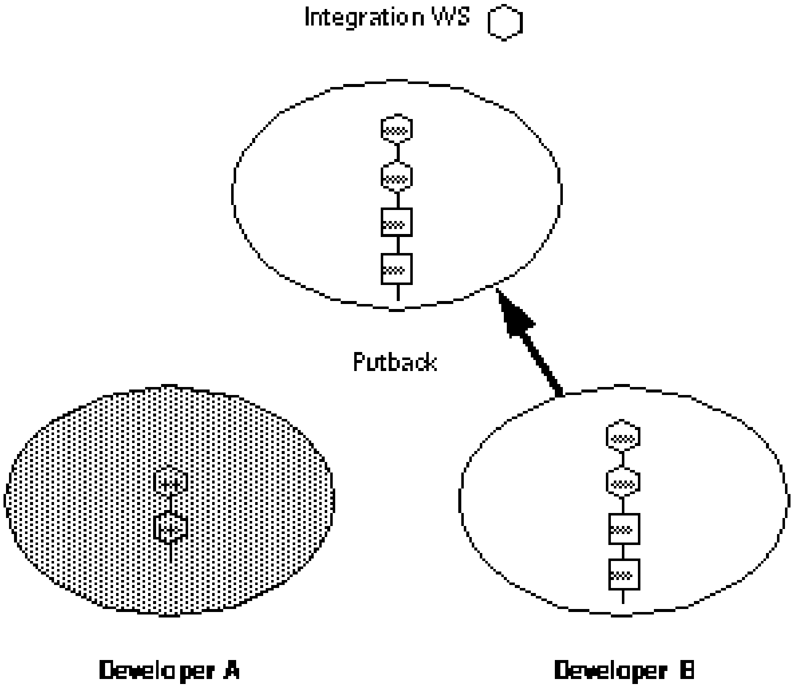
- The default delta (the point at which the next delta is added to the SCCS delta tree) is identified by an unattached descending line.
- You can use Versioning to graphically display SCCS delta trees in much the same way they are depicted here.

Both developers copy the same file from the integration workspace with the Bringover transaction. The file is new in both workspaces, so the SCCS history file is copied to both.



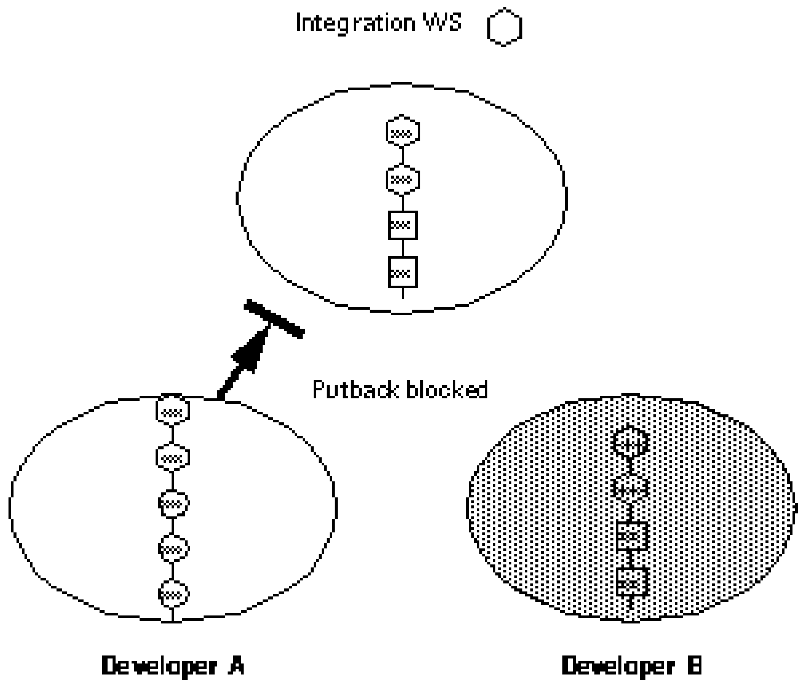
Developer B makes changes to the file, creating two new deltas: 1.3 and 1.4, and then puts the file back into the integration workspace (with the Putback transaction). Configuring appends the two new deltas to the parent SCCS delta tree.

Rather than replacing the destination workspace version of the SCCS history file with the source's version, the new deltas are added to the destination SCCS history file to preserve administrative information, such as access lists.



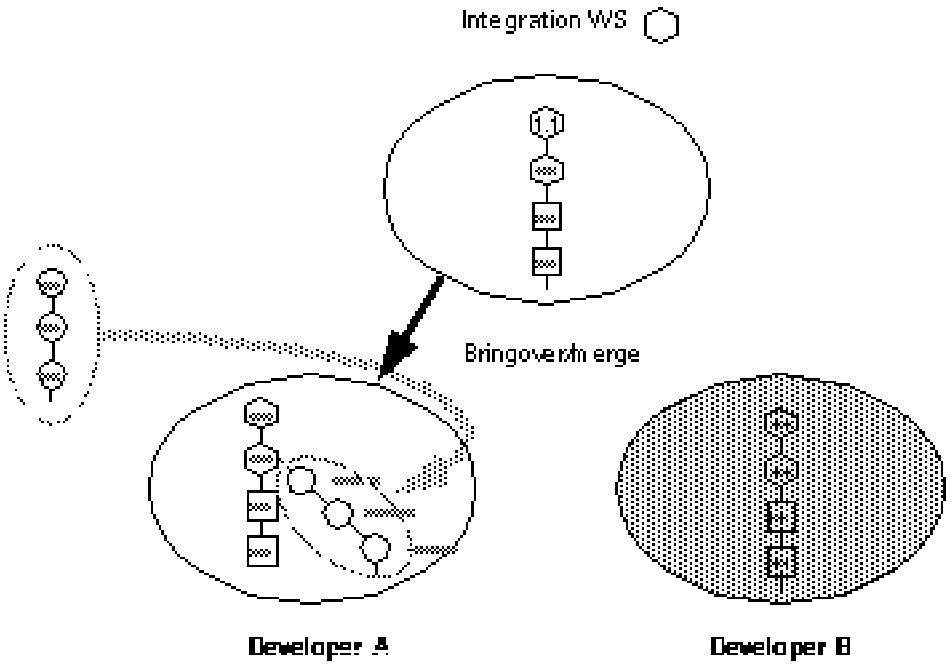
In the meantime, Developer A also changes the file (creating three new deltas: 1.3, 1.4, and 1.5) and now attempts to put back the file into the integration workspace.

Configuring blocks the Putback transaction of Developer A because the files are in conflict. The changes put back by Developer B would be overwritten. Developer A must also incorporate the changes made by Developer B into his work.



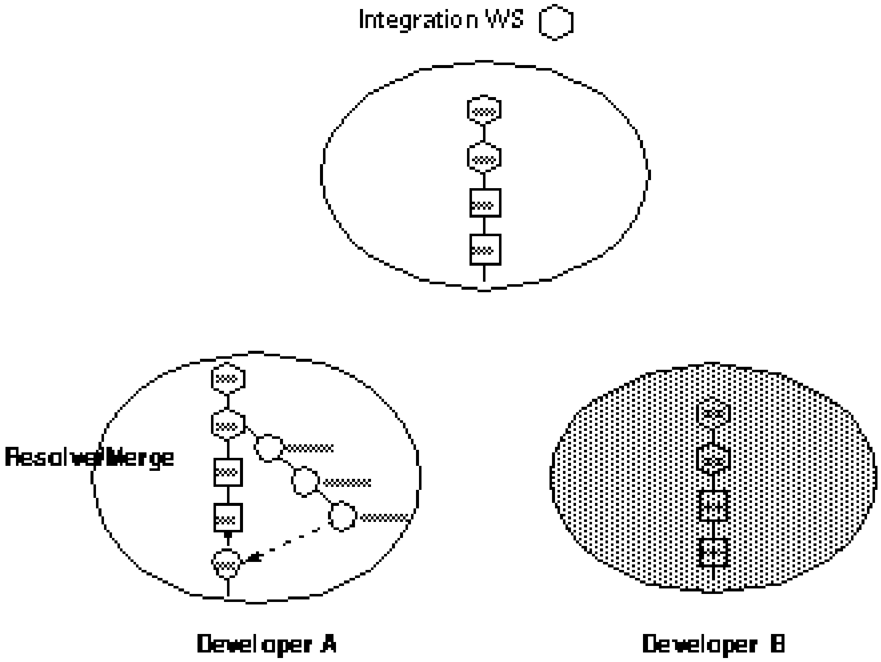
Developer A brings over the file that now contains the changes made by Developer B into his workspace from the integration workspace. The deltas created by Developer B are added into the child SCCS history file by Configuring.

The delta tree brought down from the parent is unchanged in the child. The new deltas created in the child are attached as an SCCS branch to the last delta that the child and parent had in common; the deltas from the child are assigned new SIDs accordingly. The deltas are renumbered using the SCCS branch numbering algorithm that derives the SID from the point at which it branches. In this case the branch is attached to SID 1.2; the first delta is renumbered to 1.2.1.1. The last delta created in the child (1.2.1.3, formerly 1.5) is still the default delta. Therefore, any new deltas that Developer A creates in the child before the conflict is resolved are added to the child line of work, and not the trunk (the parent line of work).

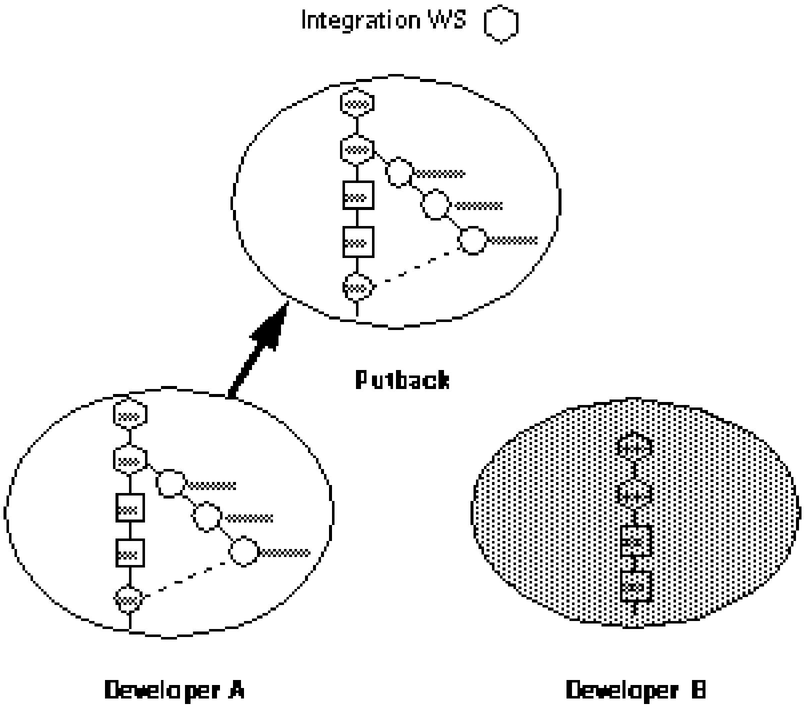


Developer A resolves the conflict in his workspace using the Resolve transaction (see Chapter 9" for details regarding conflict resolution). Developer A uses the Resolve transaction to help him decide how to merge the versions of the file represented by SIDs 1.2.1.3 and 1.4. When he commits the changes, the Resolve transaction places the newly merged contents into a new delta 1.5:

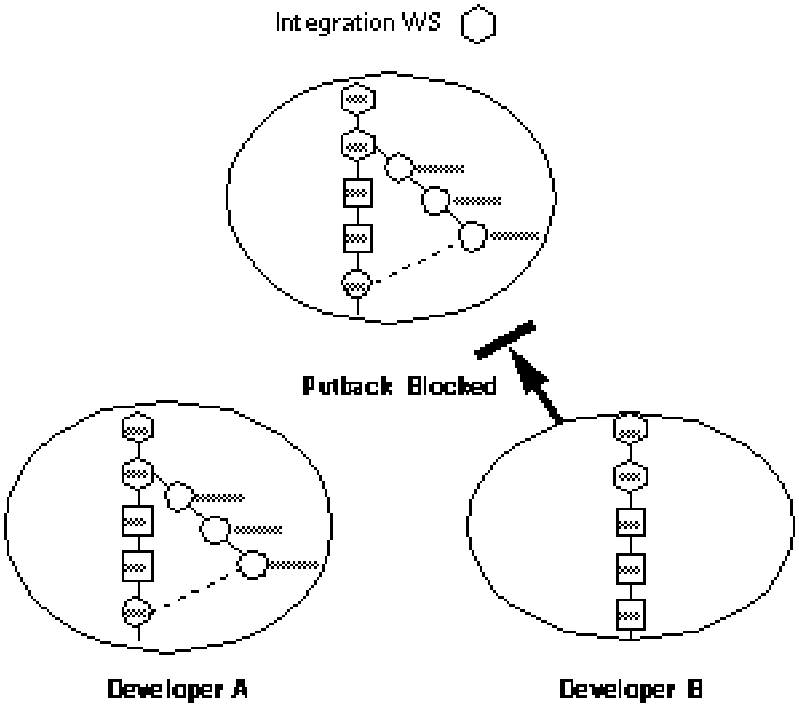
- The new delta, 1.5, is contained in a circle because it is created by Developer A.
- The newly created delta is now the default location for any new work created by Developer A.



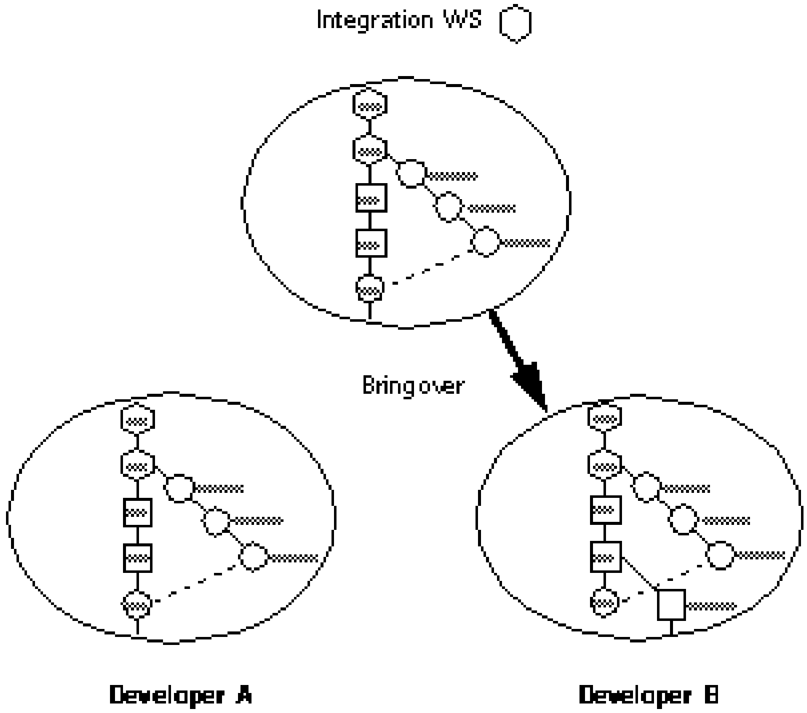
With the conflict resolved, Developer A puts back the file into the integration workspace. The branch and the newly created delta are added to the SCCS history file in the integration workspace.



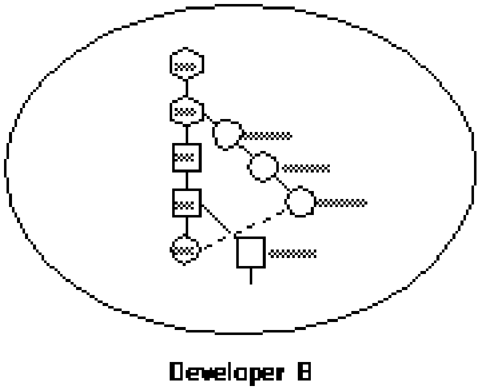
Developer B makes another change to the file in her workspace, creating delta 1.5. She attempts to put back the new work to the integration workspace, but the Putback transaction is blocked because it conflicts with the newly merged delta 1.5 that was put back by Developer A.



Developer B brings over the changed file into her workspace where its deltas are added into the child SCCS history file and renumbered by Configuring.

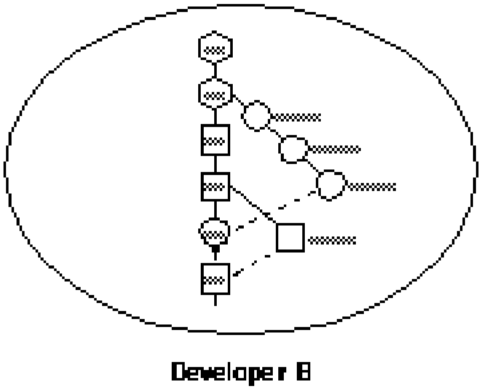


As in the previous case, Configuring appends the delta created by Developer B to the last common delta on the delta tree trunk as a branch and renumbers it appropriately. 1.5 becomes 1.4.1.1. 1.4.1.1 remains the default delta. Any new deltas created in the child before the conflict is resolved will be added to the branch.



Using the Resolve transaction, Developer B resolves the conflict merging the differences between 1.5 and 1.4.1.1 to create the new delta 1.6:

- The newly created merged contents are added as a new delta to the parent delta 1.6.
- The new delta is owned by the developer who owns the workspace.
- The new delta becomes the default delta; therefore, new work in the child will now be added beneath it.



CHAPTER **11**

Configuring Example

This chapter illustrates the basic bringover, putback, resolve cycle using an example.

The chapter contains the following topics:

- "Creating Workspaces" on page 134
- "Putting Back Changes" on page 138
- "Updating a Workspace" on page 140
- "Resolving Conflicts" on page 144

The example employs a simple case to demonstrate:

- Using the Bringover Create transaction to create two new child workspaces from a common parent
- Using the Putback transaction to put back changes from one of the child workspaces to the parent
- Using the Bringover Update transaction to update the other child workspace with those changes
- Resolving conflicts created during the Bringover Update transaction

For this example, assume two writers (Jane and Bob) are responsible for maintaining the man pages for some of the Configuring commands. The main man page workspace is named `man_pages`. The writers decide that they will each do their work in separate child workspaces and merge their work in `man_pages`. Figure 11–1 shows the file system hierarchy in the workspace `man_pages`.

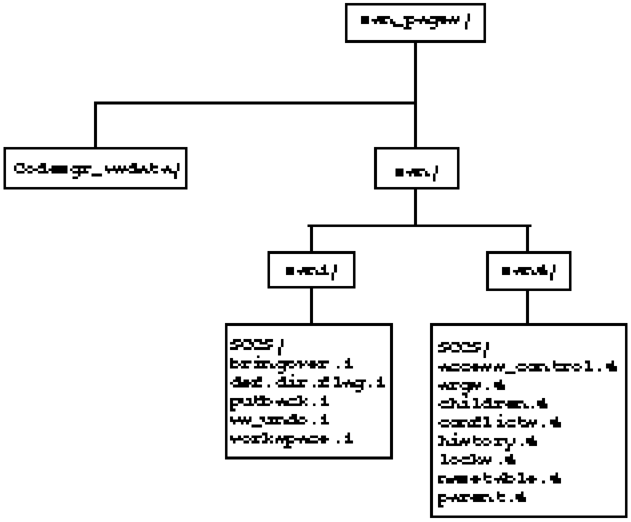


Figure 11–1 The `man_pages` Workspace

Creating Workspaces

Each writer creates his and her own child workspaces. Each child contains the same files as the parent workspace `man_pages`.

1. Jane selects the `man_pages` icon on the Workspace Graph pane and chooses Transactions > Bringover > Create (see Figure 11–2).

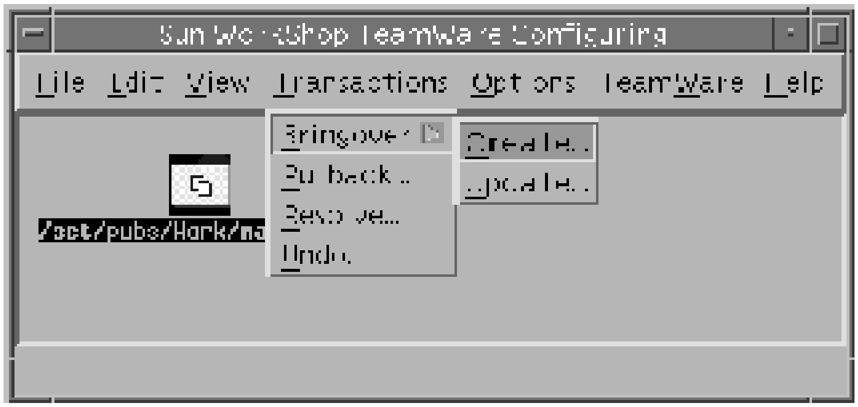
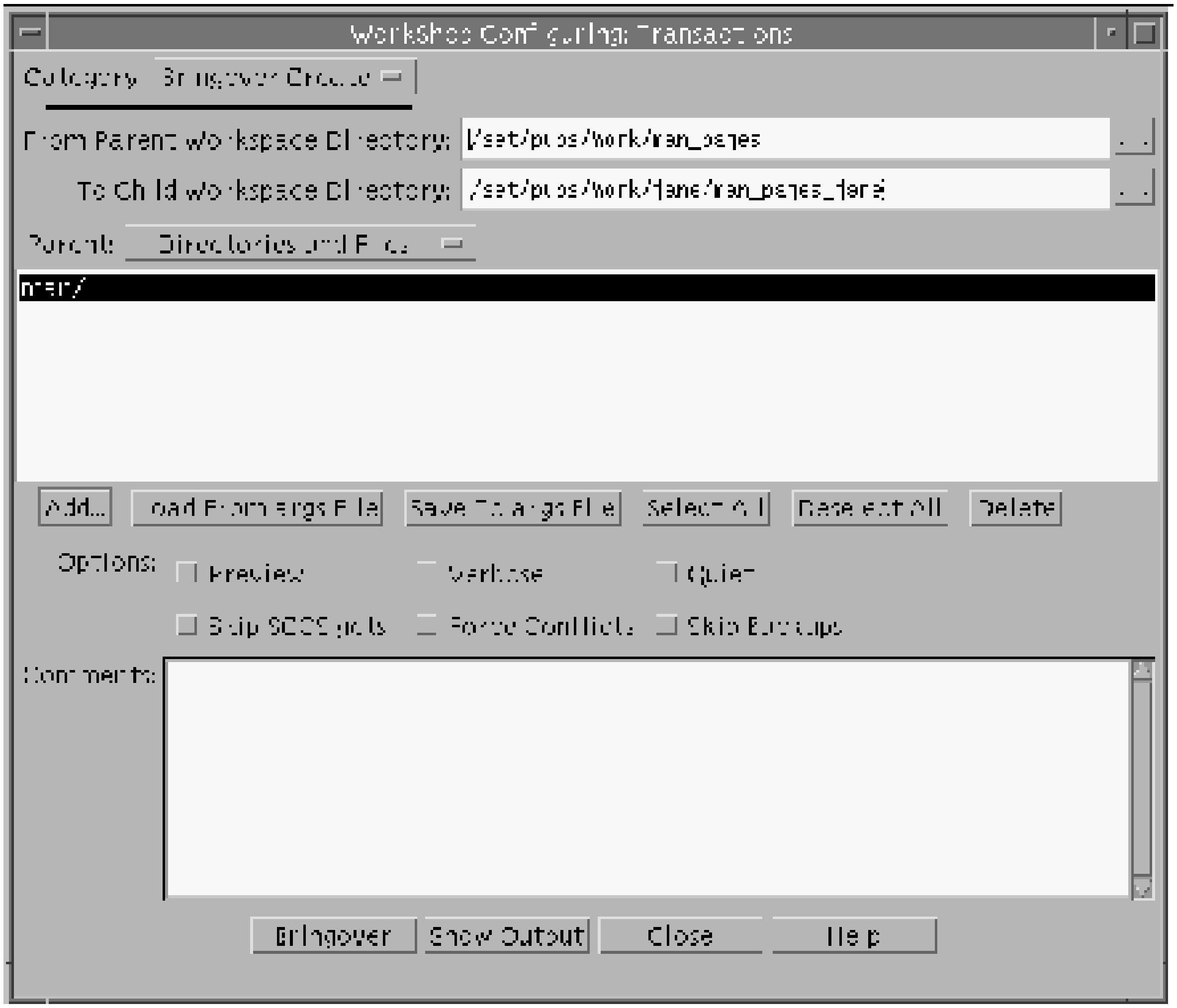


Figure 11–2 Opening the Bringover Create Transactions Window

2. The Bringover Create pane of the Transactions window opens (see Figure 11–3).



Category: Bringover Create

From Parent Workspace Directory: /set/pubs/work/man_pages

To Child Workspace Directory: /set/pubs/work/jane/man_pages_jane

Parent: Directories and Files

man/

Options: □ Preview □ Verbose □ Quiet

□ Skip SCCS gets □ Force Conflicts □ Skip Backups

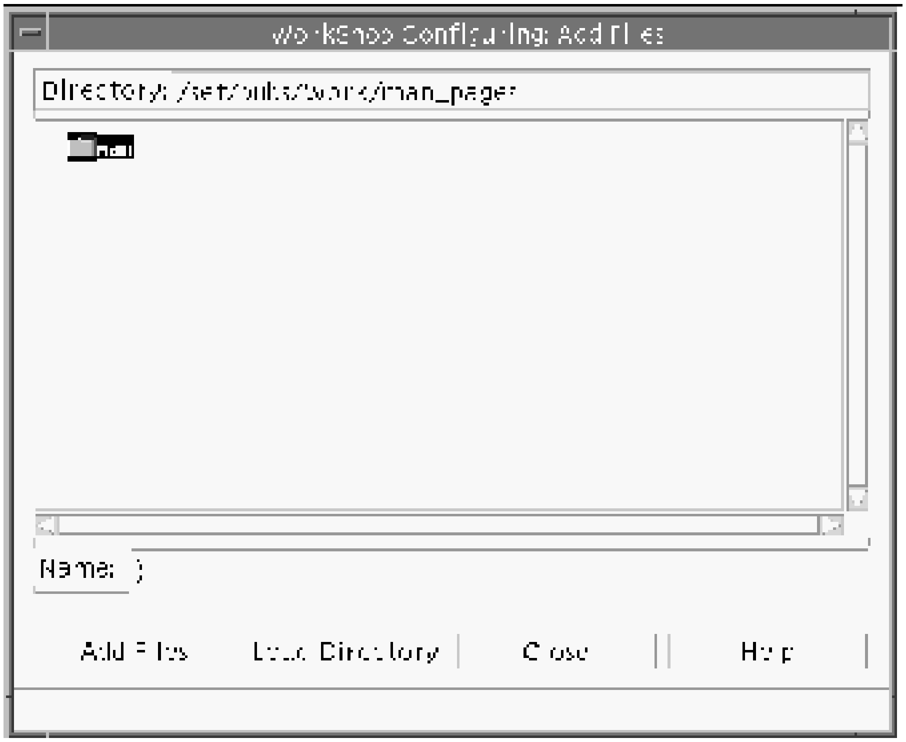
Comments:

Figure 11–3 Transactions Window — Bringover Create of `man_pages_jane`

3. Jane enters the following information:

- The path name of the child workspace in the To Child Workspace Directory text box
- The directory `man/` in the File List pane using the Add Files dialog box opened by the Add button (see Figure 11–4).

Note - The character "." (representing all of the workspace) could have been specified in the file list pane instead of `man/`. Because all SCCS files are located beneath `man/`, the two are equivalent.



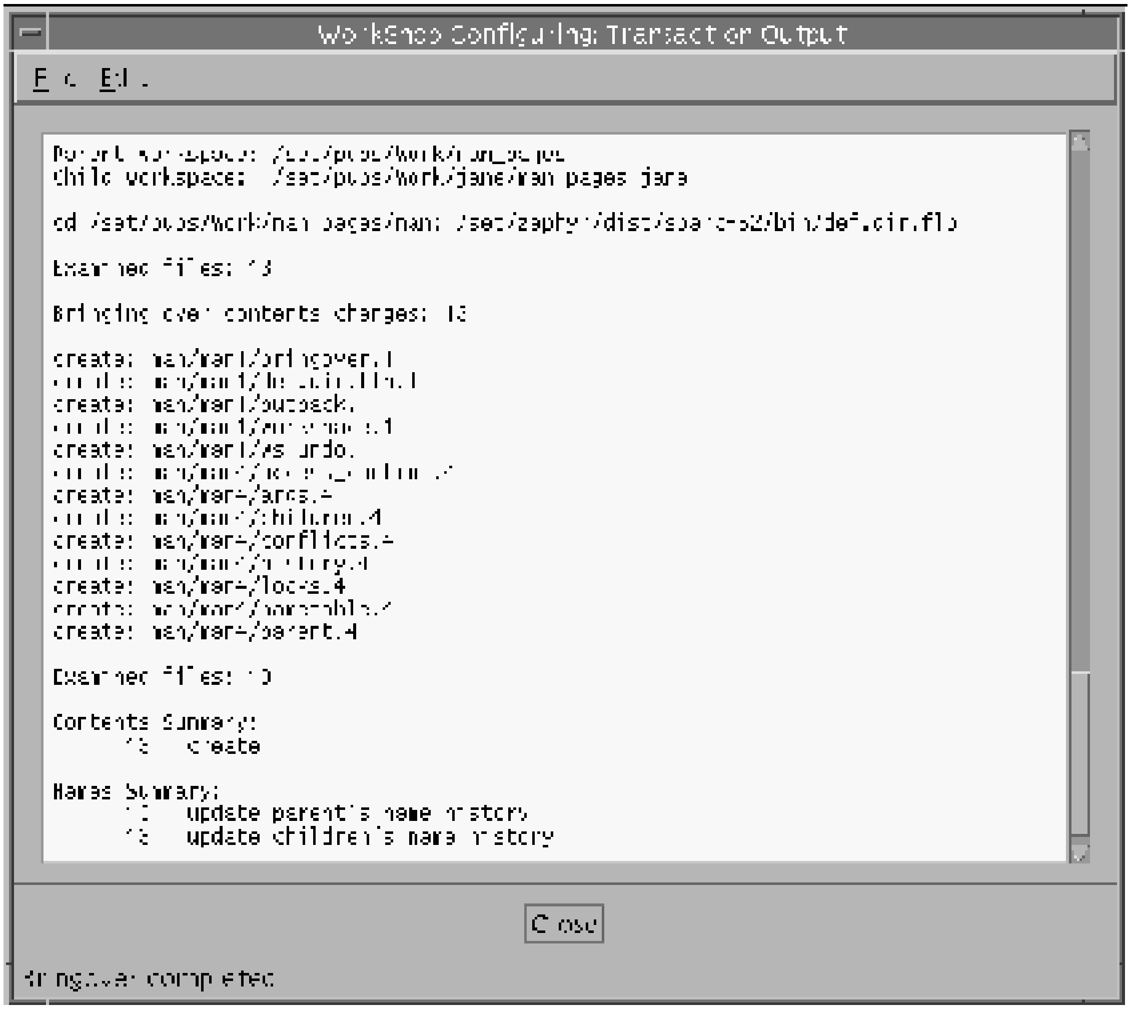
Directory: /set/pubs/work/man_pages

Name:

Figure 11–4 Add Files Dialog Box

4. Jane clicks Bringover to initiate the Bringover Create transaction.
5. Bob repeats the process to create his workspace named `man_pages_bob`.

Figure 11–5 show the output produced during the Bringover Create transaction.



```
Parent workspace: /set/pubs/work/man_pages
Child workspace:  /set/pubs/work/jane/man_pages_jane

cd /set/pubs/work/man_pages/man; /set/zaphod/dist/sparc-s2/bin/def.dir.flp

Examined files: 13

Bringing over contents changes: 13

create: man/man1/bringover.1
create: man/man1/def.dir.flp.1
create: man/man1/putback.1
create: man/man1/[illegible].1
create: man/man1/ws_undo.1
create: man/man4/access_control.4
create: man/man4/args.4
create: man/man4/children.4
create: man/man4/conflicts.4
create: man/man4/history.4
create: man/man4/locks.4
create: man/man4/notification.4
create: man/man4/parent.4

Examined files: 13

Contents Summary:
      13   create

Names Summary:
      13   update parent's name history
      13   update children's name history
```

Figure 11–5 Output From Bringover Create of `man_pages_jane`

The output includes the following important information:

- `Examined files: 13`

  During the initial examination phase of the Bringover transaction, Configuring determined that 13 files differed in the parent and child. In this case, since the child is being created and thus contains no files, all files contained in the parent are considered for the transaction.

- `Bringing over contents changes: 13`

  Configuring has determined that the 13 files in the list should be brought over from the parent to the child workspace.

- `Contents Summary: 13 Create`

This line indicates that 13 files were created (as opposed to updated) in the child. In the Bringover Create transaction, all transferred files fall into this category.

- `13 update children's name history`

  The name history is a file that assists Configuring in tracking file names. It is used to speed up the processing of renamed files.

Putting Back Changes

Bob begins work in his new workspace and makes changes to the file `bringover.1`, `putback.1`, and `args.4`. He decides that these changes are important and that Jane should have access to them. He uses the Putback transaction to copy the changes back to the common parent workspace `man_pages`.

1. **Bob selects the** `man_pages_bob` **icon on the Workspace Graph pane and chooses Transactions > Putback (see Figure 11–6).**

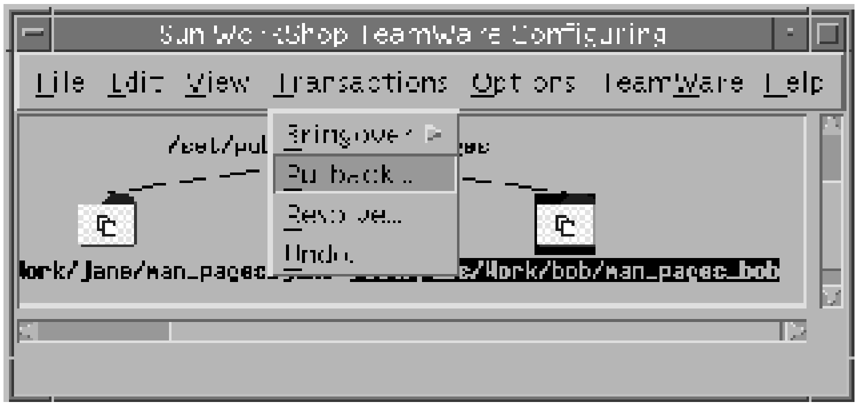
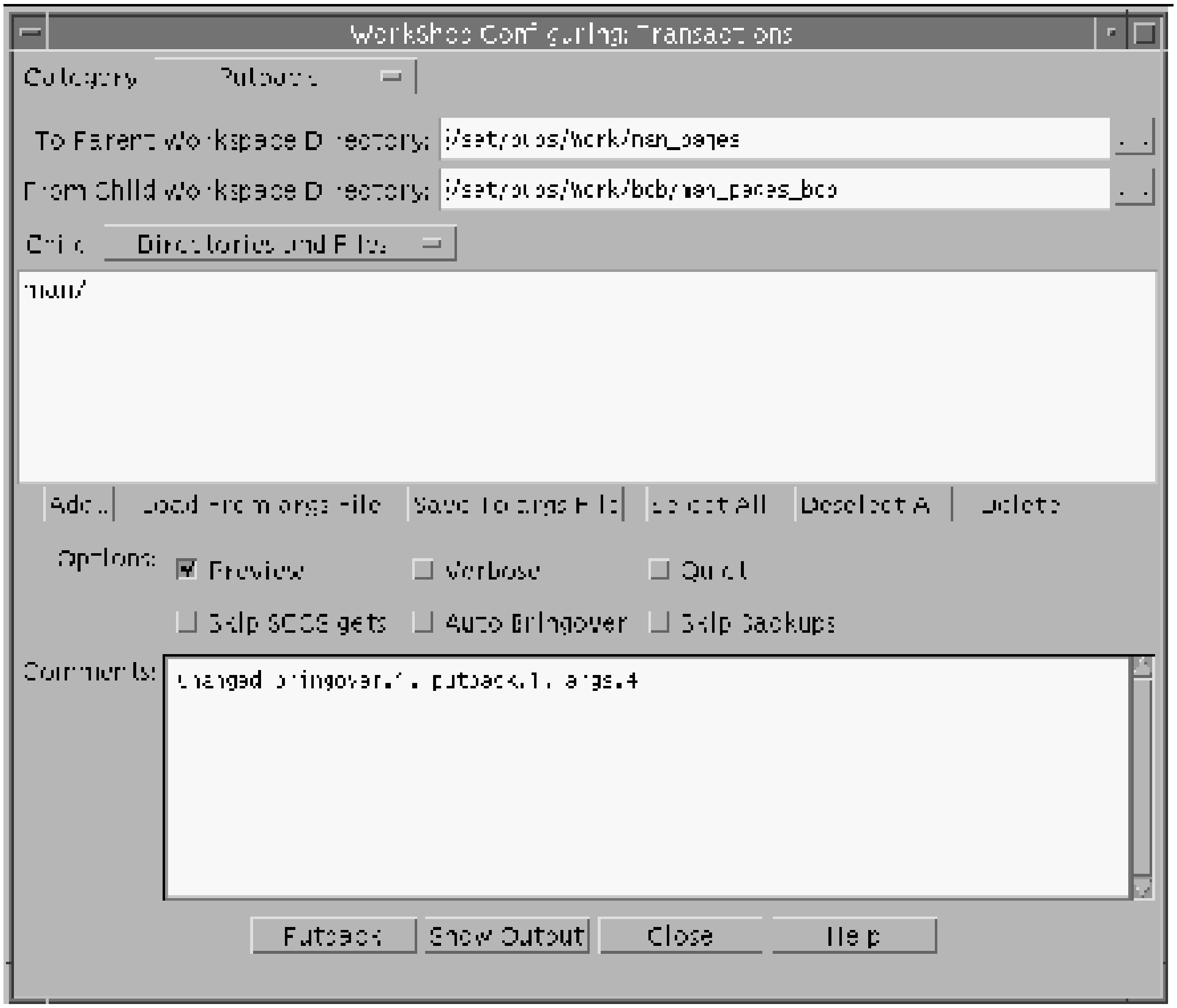


Figure 11–6 Opening the Putback Transactions Window

2. **When the Putback pane of the Transactions window opens (see Figure 11–7), Bob chooses the Preview option. By choosing this option, the transaction proceeds without actually copying files. Bob is able to view the output of the transaction without actually altering files; by using this option he is able to confirm that the transaction will proceed the way he expects.**

   Configuring automatically loads the directory `man/` into the File List pane. This is the directory that Bob specified when he created `man_pages_bob`; the value was saved in the workspace `Codemgr_wsdata/args` file. He could change the contents of the File List pane using the buttons directly below the pane.



Category: Putback

To Parent Workspace Directory: /set/pubs/work/man_pages

From Child Workspace Directory: /set/pubs/work/bob/man_pages_bob

Child: Directories and Files

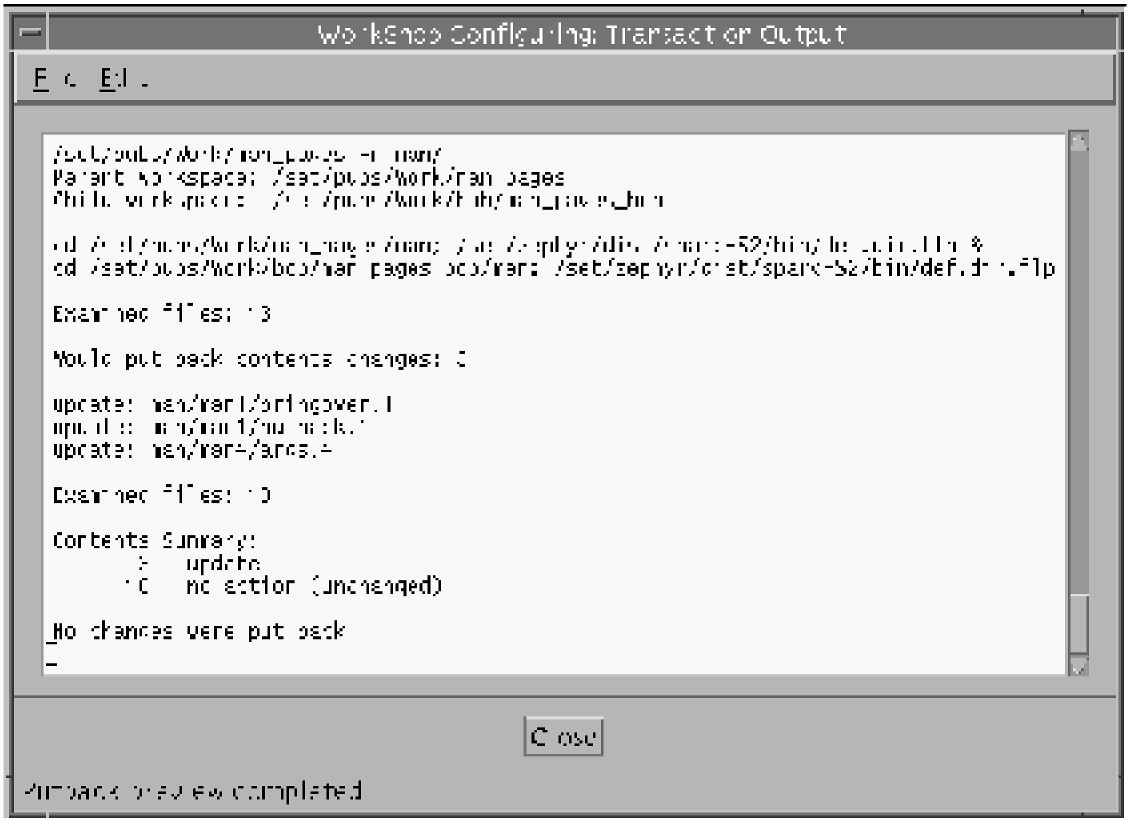
man/

Options: ☒ Preview □ Verbose □ Quiet
□ Skip SCCS gets □ Auto Bringover □ Skip Backups

Comments: changed bringover.1, putback.1, args.4

Figure 11–7 Transactions Window—Putback of `man_pages_bob`

The output is shown in Figure 11–8.


WorkShop Configuring: Transaction Output
File Edit
Close
Putback preview completed

```
/set/pubs/Work/man_pages -r man/
Parent workspace: /set/pubs/Work/man_pages
Child workspace: /set/pubs/Work/bob/man_pages_bob

cd /set/pubs/Work/man_pages/man; /set/zephyr/dist/sparc-S2/bin/def.dir.flp &
cd /set/pubs/Work/bob/man_pages_bob/man; /set/zephyr/dist/sparc-S2/bin/def.dir.flp

Examined files: 13

Would put back contents changes: 3

update: man/man1/bringover.1
update: man/man1/putback.1
update: man/man4/ares.4

Examined files: 13

Contents Summary:
       3   update
      10   no action (unchanged)

No changes were put back
```

Figure 11–8 Output from Putback of `man_pages_bob`

3. **Satisfied with the output, Bob deselects the Preview option and re-executes the Putback transaction.**

Updating a Workspace

Jane makes changes to the files `putback.1` and `locks.4` in `man_pages_jane`. Before she attempts to put the changes back to the `man_pages` workspace, she wants to update her workspace with the changes that Bob has just put back to `man_pages`. She uses the Bringover Update transaction.

1. **Jane selects the `man_pages_jane` icon on the Workspace Graph pane and chooses Transactions > Bringover > Update.**

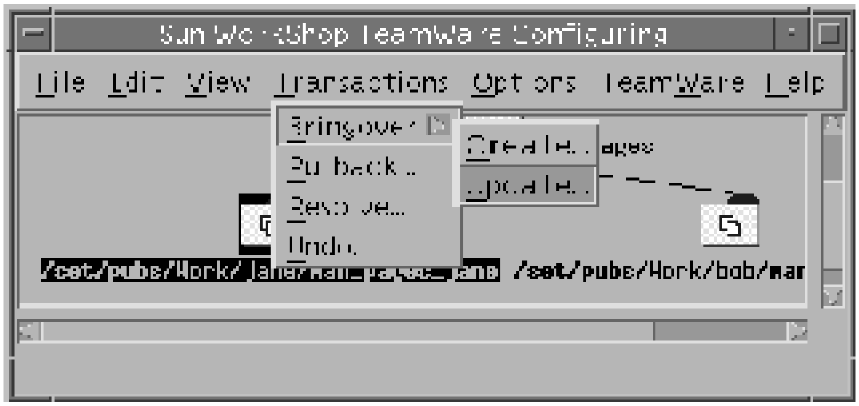
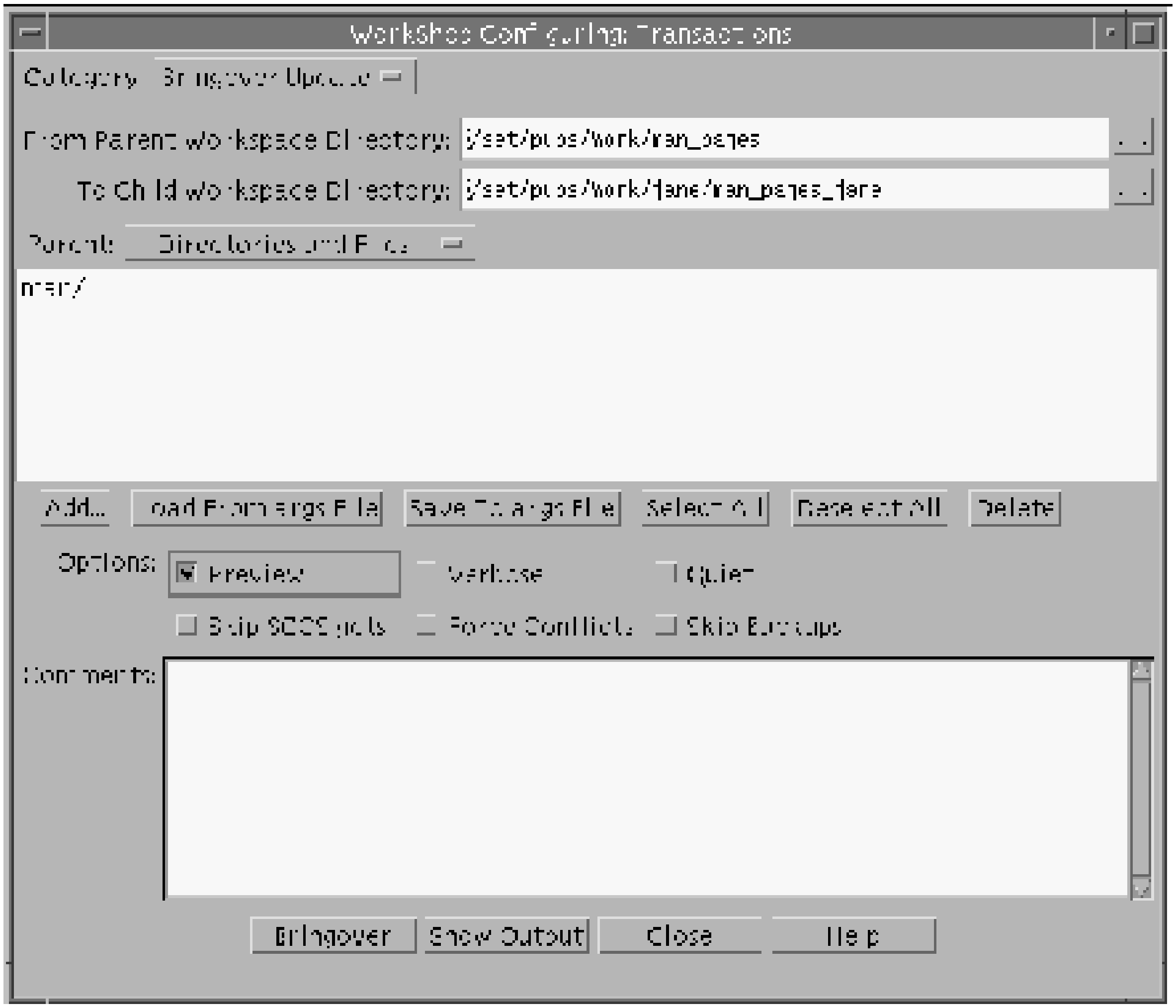


Figure 11–9 Opening the Bringover Update Transactions Window

2. **When the Bringover Update pane of the Transactions window opens, Jane chooses the Preview option. By choosing this option, the transaction proceeds without actually copying files. Jane is able to view the output of the transaction without actually altering files. By using this option she is able to determine which files have been changed prior to taking any real action.**



Category: Bringover Update

From Parent Workspace Directory: /set/pubs/work/man_pages

To Child Workspace Directory: /set/pubs/work/jane/man_pages_jane

Parent: Directories and Files

man/

Options: ☒ Preview □ Verbose □ Quiet

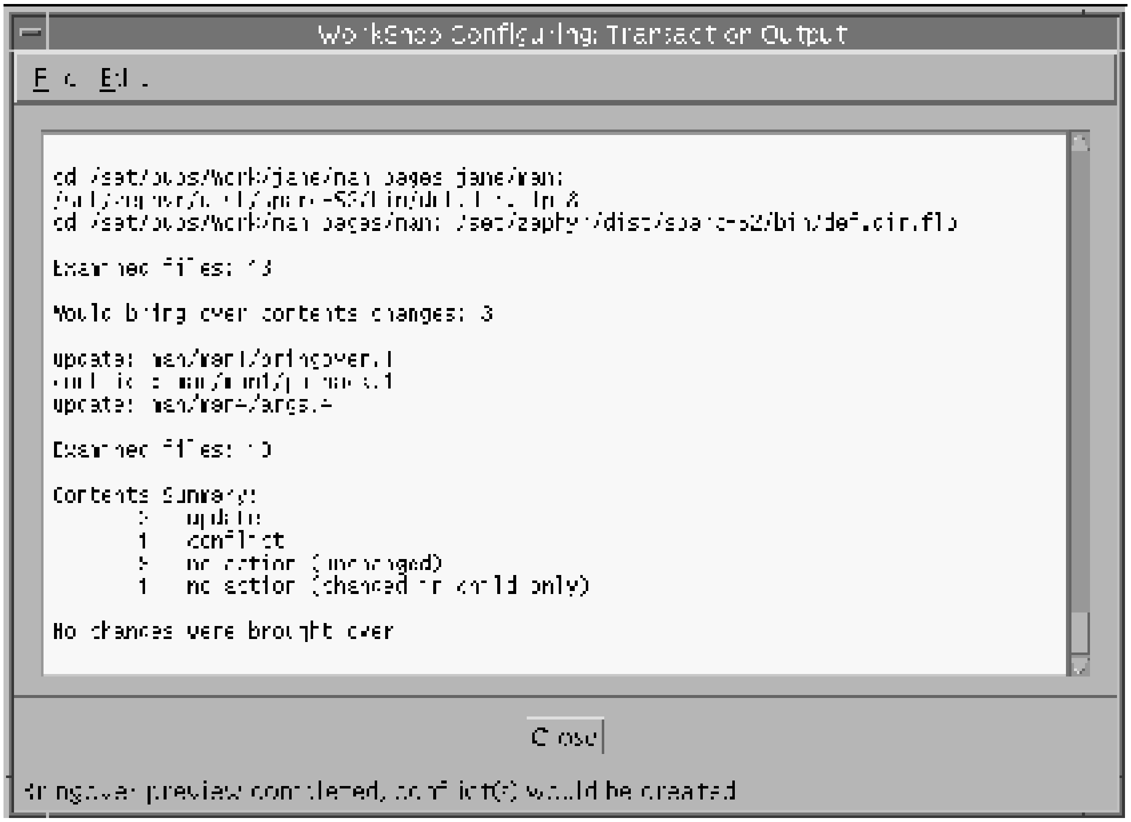
□ Skip SCCS gets □ Force Conflicts □ Skip Backups

Comments:

Figure 11–10 Transactions Window—Bringover Update of `man_pages_jane`

The output of the Bringover Update transaction (see Figure 11–11) indicates that:

- `args.5` and `bringover.1` will be updated in `man_pages_jane`.
- There will be a conflict created on `putback.1`. The conflict occurs because `putback.1` is changed both in `man_pages` by Bob and in `man_pages_jane` by Jane.
- One file (`locks.4`) is changed only in `man_pages_jane.`
- The other 11 files are unchanged.

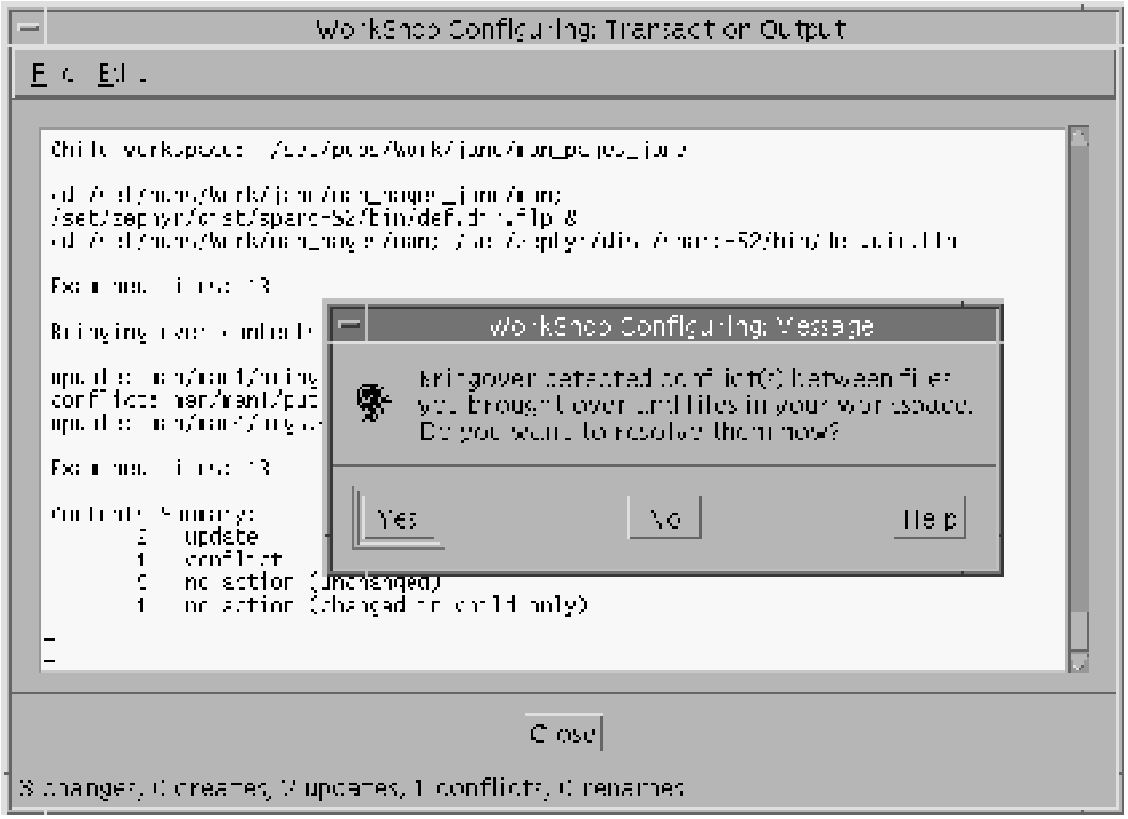


```
cd /set/pubs/work/jane/man_pages_jane/man;
[illegible]
cd /set/pubs/work/man_pages/man; /set/zephyr/dist/sparc-s2/bin/def.cir.flp

Examined files: 13

Would bring over contents changes: 3

update: man/man1/bringover.1
conflict: man/man1/putback.1
update: man/man5/args.5

Examined files: 13

Contents Summary:
       2   update
       1   conflict
       9   no action (unchanged)
       1   no action (changed in child only)

No changes were brought over
```

Bringover preview completed, conflict(s) would be created

Figure 11–11 Output From Bringover Update of `man_pages_jane`

3. **None of these changes surprises Jane, so she decides to complete the transaction by re-executing it with the Preview option deselected. After the transaction completes as expected, the Configuring program automatically presents Jane with the option to resolve the conflict created on** `putback.1` **(see Figure 11–12).**


WorkShop Configuring: Transaction Output
File
Edit
WorkShop Configuring: Message
Yes
No
Help
Close

```
Child workspace: /set/pool/work/jane/man_pages_jane

cd /set/pool/work/jane/man_pages_jane/man;
/set/zephyr/dist/sparc-S2/bin/def.dir.flp &
cd /set/pool/work/man_pages/man; /set/zephyr/dist/sparc-S2/bin/def.dir.flp

Examined files: 13

Bringing over contents

update: man/man1/...
conflict: man/man1/put
update: man/man1/try...

Examined files: 13

Contents Summary:
       2   update
       1   conflict
       9   no action (unchanged)
       1   no action (changed in child only)
```

Bringover detected conflict(s) between files you brought over and files in your workspace. Do you want to resolve them now?

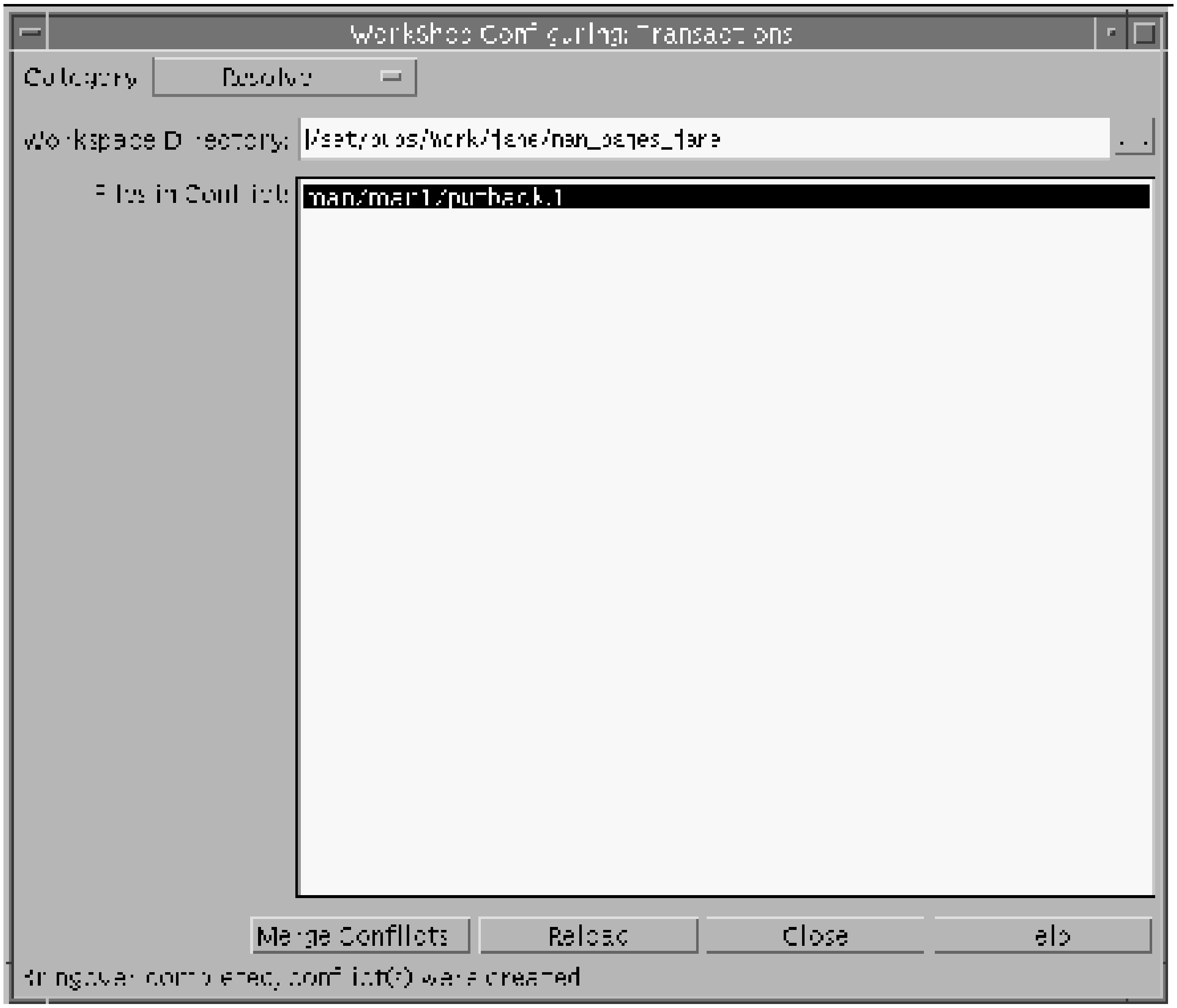
3 changes, 0 creates, 2 updates, 1 conflicts, 0 renames

Figure 11–12 Resolve Confirmation Dialog Box

Resolving Conflicts

Jane decides that she wants to resolve the conflict now and she clicks Yes. The Resolve pane of the Transactions window opens (see Figure 11–13).

Note - If the conflict is left unresolved, the Resolve transaction can be initiated later by either double-clicking the `man_pages_jane` icon, or by selecting the icon and choosing Transactions > Resolve.



/set/pubs/work/jane/man_pages_jane

man/man1/putback.1

Bringover completed; conflict(s) were created

Figure 11–13 Transactions Window—Resolve Conflict on `putback.1`

Note - If the Auto Start Merging window property is set for the Resolve pane, Merging begins execution automatically.

The file in conflict (`man/man1/putback.1`) is listed in the File List pane of the Transactions window. The file is automatically selected (highlighted) so Jane clicks Merge Conflicts. If there had been multiple files in the list, Jane could have deselected any portion of the list. If the Auto Advance property is selected (the default), the Configuring program automatically works its way down through the list of selected files.

Configuring starts Merging (see Figure 11–14) and passes it the name of `putback.1`.

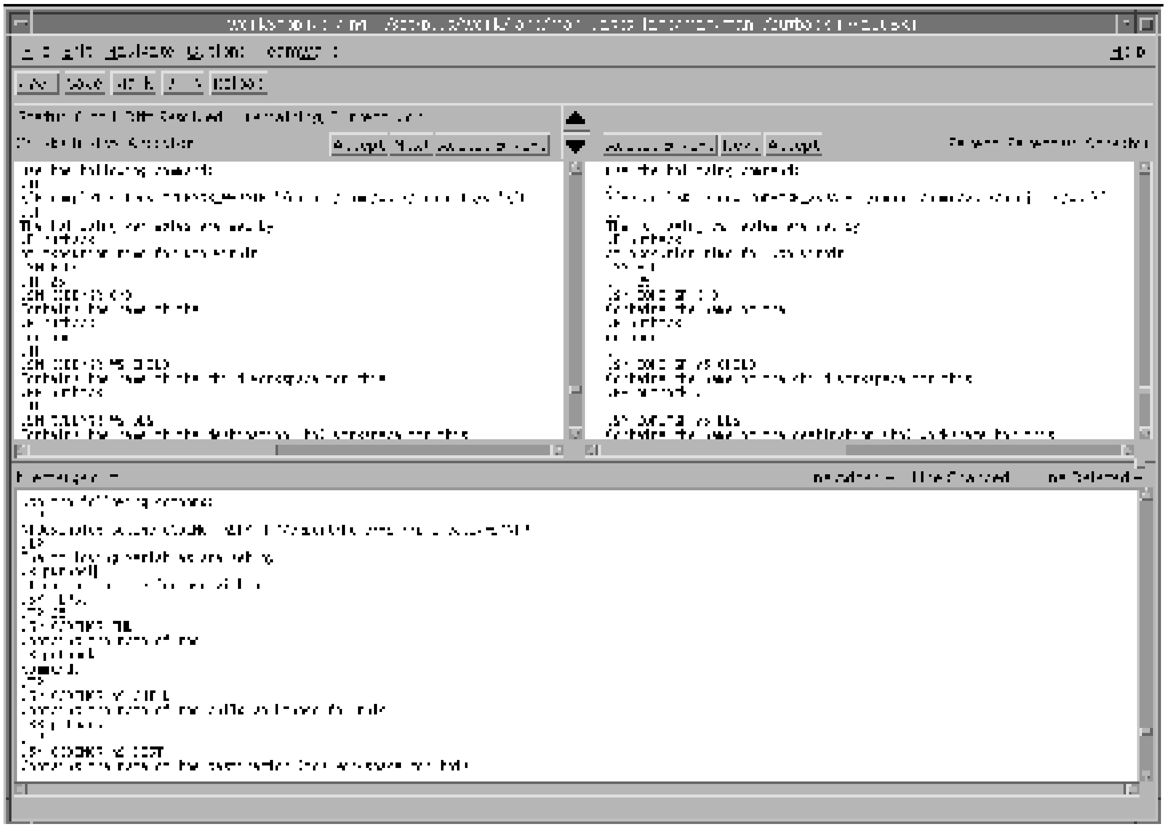


Figure 11–14 Merging Window—Merging `putback.1`

Jane works her way through the merging process, accepting Bob’s changes from the right pane and her changes from the left. When all the differences have been resolved, she saves the changes. See Chapter 13,” and Chapter 14” for more information about using Merging.

Jane can now put back her changes to the parent workspace (`man_pages`) following the same procedure that Bob used (see “Putting Back Changes” on page 138).

CHAPTER 12

Performing Basic SCCS Functions with Versioning

This chapter shows you how to perform basic SCCS functions using Versioning. It is organized into the following sections:

- "Introduction to Versioning" on page 147
- "Versioning Window" on page 149
- "History Dialog Box" on page 151
- "Typical Sessions" on page 153
- "Using Commands to Manipulate Files" on page 155
- "Changing Versioning Properties" on page 157

Introduction to Versioning

Coordinating write access to source files is important when changes will be made by several people. Maintaining a record of file updates allows you to determine when and why changes were made.

The source code control system (SCCS) allows you to control write access to source files and monitor changes made to those files. SCCS allows only one user at a time to update a file, and it records all changes in a history file.

TeamWare Versioning is a GUI to SCCS. Versioning allows you to manipulate files and perform SCCS functions without having to know SCCS commands. It provides an intuitive method for checking files in and out, as well as displaying and moving through the history branches.

With Versioning, you can:

- Check in files under SCCS
- Check out and lock a version of the file for editing
- Retrieve copies of any version of the file from SCCS history
- Visually peruse the branches of an SCCS history file
- Back out changes to a checked-out copy
- Inquire about the availability of a file for editing
- Inquire about differences between selected versions using Merging
- Display the version log summarizing executed commands

Versioning helps you perform these tasks and expedites the progress of concurrent development projects.

Branches

You can picture the deltas applied to an SCCS file as nodes of a tree with the initial version of the file as the root. The root delta is numbered 1.1 by default. These two parts of the SCCS delta ID (SID) are the release and level numbers. Successive deltas (nodes) are named 1.2, 1.3, and so forth. This structure is called the trunk of the SCCS delta tree. It represents the normal sequential development of an SCCS file.

Situations can arise, however, when it is necessary to create an alternative branch on the tree. Branches can be used, for instance, to keep track of alternate versions developed in parallel, such as for bug fixes.

The SID for a branch delta consists of four parts: the release and level numbers and the branch and sequence numbers, or release.level.branch.sequence. The branch number is assigned to each branch that is a descendant of a particular trunk delta; the first branch is 1, the next 2, and so on. The sequence number is assigned, in order, to each delta on a particular branch.

Thus, 1.3.1.1 identifies the first delta of the first branch derived from delta 1.3. A second branch to this delta would be numbered 1.3.2.1 and so on.

The concepts of branching can be extended to any delta in the tree. The branch component is assigned in the order of creation on the branch, independent of its location relative to the trunk. Thus, a branch delta can always be identified from its name. While the trunk delta can be identified from the branch delta"s name, it is not possible to determine the entire path leading from the trunk delta to the branch delta.

For example, if delta 1.3 has one branch, all deltas on that branch will be named 1.3.n. If a delta on this branch has another branch emanating from it, all deltas on the new branch will be named 1.3.2.n. The only information that can be derived from the name of delta 1.3.2.2 is that it is the second chronological delta on the second chronological branch whose trunk ancestor is delta 1.3. In particular, it is not possible to determine from the name of delta 1.3.2.2 all of the deltas between it and its trunk ancestor (1.3).

Deltas and Versions

When you check in a file, SCCS records only the line-by-line differences between the text you check in and the previous version of the file. This set of differences is known as a delta. The file version that you initially checked out was constructed from a set of accumulated deltas. The terms delta and version are often used synonymously; however, their meanings are not the same. It is possible to retrieve a version that omits selected deltas.

History Files

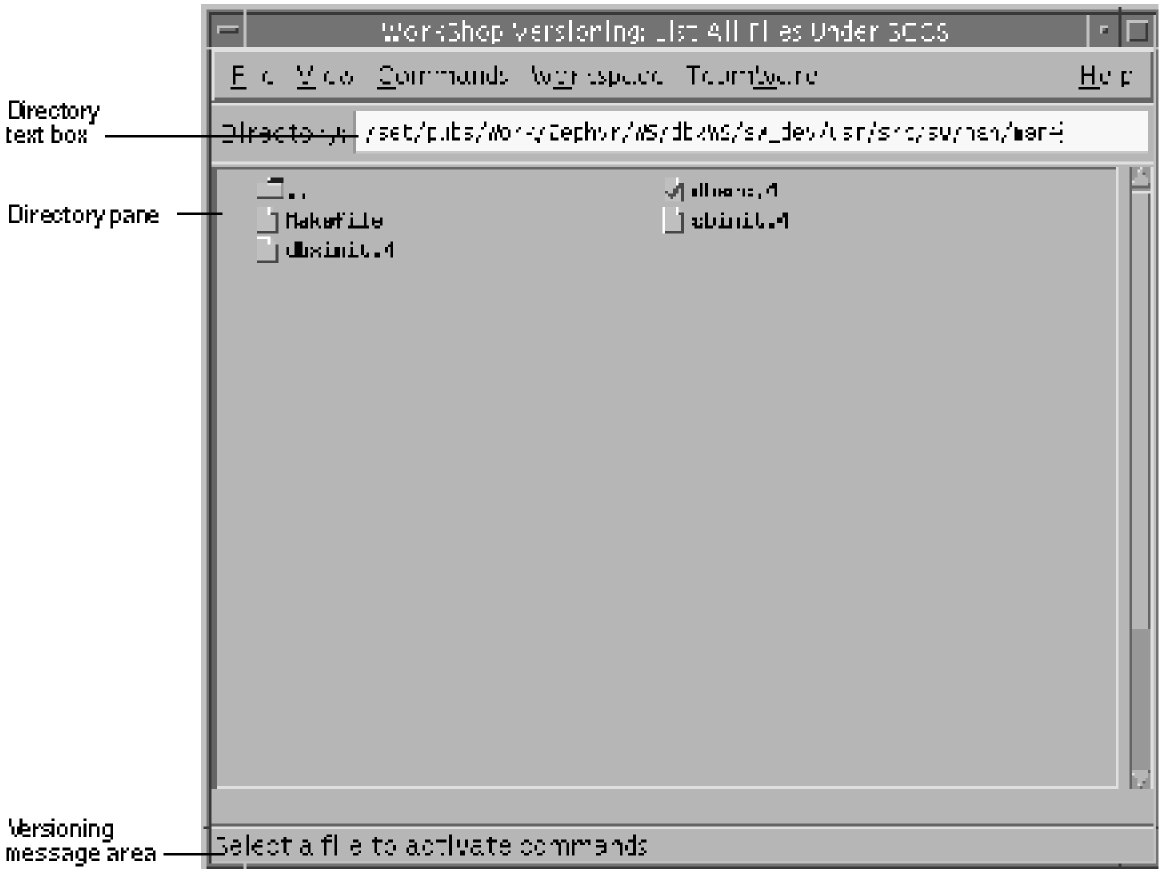
When you initially put a file under SCCS control, a history file is created for the new SCCS file. The initial version of the history file uses the complete text of the source file. The initial history file is the file that further deltas are compared to. Owing to its prefix (s.), the history file is often referred to as the s.file (s-dot-file).

SCCS Delta ID (SID)

An SCCS delta ID (SID) is the number used to represent a specific delta. This is a two-part number, with the parts separated by a dot (.). The SID of the initial delta is 1.1 by default. The first part of the SID is referred to as the release number, and the second, the level number. When you check in a delta, the level number is incremented automatically.

Versioning Window

The Versioning main window (see Figure 12–1) displays the directories and SCCS files of the loaded directory.



Directory: /set/pubs/Work/zephyr/WS/dbxWS/sw_dev/usr/src/sw/man/man4

Select a file to activate commands

Figure 12–1 Versioning Window

File menu	Provides commands for managing files
View menu	Provides commands to control and update the icons shown in the directory pane
Commands menu	Provides commands for checking files in and out
Workspace menu	Provides commands for finding, deleting, and moving files under SCCS control
TeamWare menu	Provides commands for starting other TeamWare tools
Directory text box	Specifies the directory to view

Directory pane	Displays icons for all loaded directories and files under SCCS control
Versioning message area	Displays messages about operations in the Versioning window

A file checked out from SCCS is marked with a check mark.

Directories are shown whether they contain files which are under SCCS control or not. A directory is a container for files and directories and can possibly contain SCCS files and directories further down the hierarchy. Files are only shown in the Versioning window if they are under SCCS control. To look at the non-SCCS files in a directory, use the Check in New window.

History Dialog Box

The History dialog box (see Figure 12–2) displays an illustration of SCCS delta branches for a selected file. This history graph allows you to peruse the delta structure of a file and assess associations between versions. Dashed lines are shown by default and indicate that the delta to the right of the dashed line was created by including the changes from the delta on the left. Following the dashed line provides you with a time-ordering sequence.

Using the History dialog box, you can:

- Select a delta from the history graph that will display information about the delta in the Delta Details pane.
- Select a delta from the history graph to check it in or out, depending on its current SCCS state.
- View the contents of a selected delta by choosing View > Show File Contents. This opens an editor window with the contents of the selected delta displayed.
- Select two deltas and choose View > Diff in Merging Window. This displays a Merging window in which the two selected deltas are displayed side by side for comparison.
- Select two deltas and choose View > Diff in Text Window. An editor window opens up displaying the textual differences from the SCCS `diffs` command.

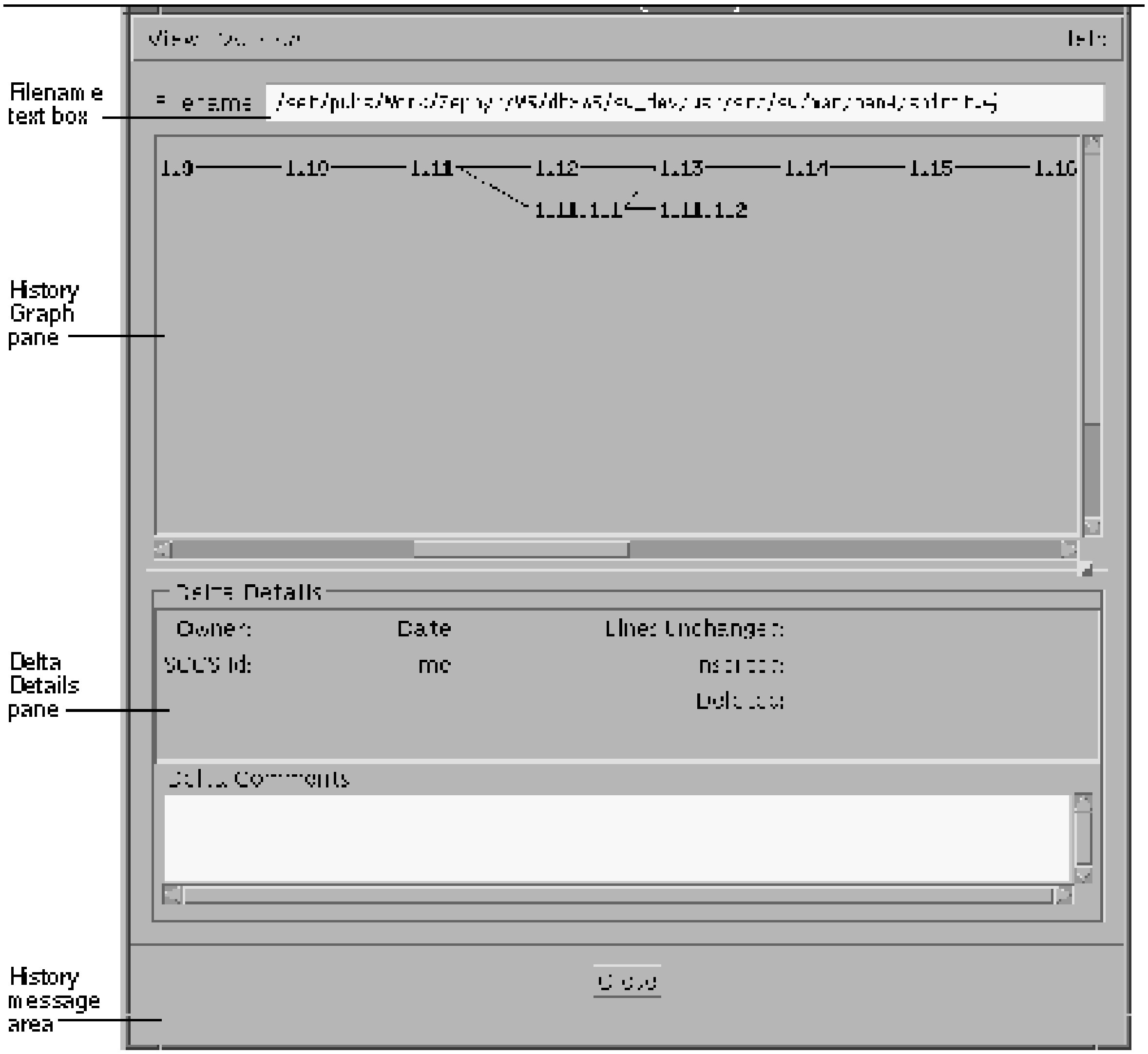


Figure 12–2 History Dialog Box

View menu	Provides commands for managing files
Version menu	Provides commands for managing icons in the History Graph pane
Filename text box	Show the file path name
History Graph pane	Shows icons for file deltas

Delta Details pane	Contains delta history information
History message area	Displays messages about operations in the History dialog box

Typical Sessions

This section gives an overview of the most common SCCS sessions using Versioning. It assumes that you are familiar with SCCS. The following scenarios are covered:

- An initial session where files are not yet under SCCS control
- A session where the project is already under SCCS control

Putting a Project Under SCCS Control

There may be instances where a project is under development before a source code control system is put in place. This scenario assumes a project is already under way and the hierarchy of the project is established. It is assumed that the project is ready to be put under SCCS control.

To put a project under SCCS control:

1. **Start Versioning in one of the following three ways:**
   - Type `twversion` at a shell command prompt, with or without a path name, followed by the ampersand symbol (&). For example:

     ```
     demo% twversion &
     ```

     ```
     demo% twversion dirname &
     ```

   - Choose TeamWare > Versioning in the Configuring window.
   - Double-click on a workspace icon in the Configuring window.
2. **Double click the project directory in the Versioning main window.**

   Versioning automatically changes to the selected directory. As there are no files yet under SCCS control, the display shows only directories.

3. **Choose Commands > Check In New.**

   The Check In New Files window displays a list of files not under SCCS control.

4. **Select the files you want to put under SCCS control and add necessary comments in the Initial Comment pane.**

5. **Click OK button to check in the selected files.**

   Once the files are under SCCS control, they will be transferred to the display in the main window.

Repeat this scenario in as many project directories as necessary. Then proceed to the scenario in "Working with a Project Under SCCS Control" on page 154" for working with files under SCCS control.

Working with a Project Under SCCS Control

Once a project is under SCCS control, you can use Versioning to perform SCCS functions. This section provides a scenario of basic SCCS tasks and how they might be applied on a project. These steps are simplified to give an overview of the process. The remaining sections of this chapter cover in-depth instructions on performing tasks.

Note - This is representative of a hypothetical session. The steps will vary according to project needs and the tasks required to fulfill them.

---

1. **Start Versioning and select the working directory.**

2. **Choose Commands > Check Out or Commands > Check Out and Edit to check out a file.**

3. **Choose File > File History to display the history graph of the file.**

4. **Select two deltas from the graph in the History window and choose View > Diff in Merging Window or View > Diff in Text Window to inspect the differences.**

5. **Make changes to the file.**

6. **Add necessary comments.**

7. **Choose Commands > Check in to check the file in.**

These steps can be repeated and varied as required by the needs of your project. The following sections of this chapter provide in-depth information on how to perform these, and other, SCCS functions with Versioning.

Using Commands to Manipulate Files

You can perform SCCS operations within the file list on a per-file basis or on a multiple-file basis. You can select a single file, or multiple files, on which to perform a function.

The Commands menu allows you to do the following:

- Check out a file
- Check in a file
- Edit a checked out file
- Check a new file under SCCS control
- Uncheckout a file

Checking Out and Checking In Files

This section covers the process of checking out and checking in files that are already under SCCS control.

Checking Out Files

You can use two methods for checking out files:

1. **Select the file(s) you want to check out from the main window display. Then, choose Commands > Check Out.**

   A check mark is displayed in the file icon(s) of the selected file(s). This method is valuable when you want to check out several files at once.

1. **Double click a file icon in the main window.**

   A check mark is displayed in the file icon and the file is checked out with you as the owner. This is *not* the default behavior. To enable this double-click action, you must change the default behavior by Choosing View > Options, selecting the Toggle SCCS State radio button in the Double Click Action section of the Options dialog box, and then selecting the Confirm Double Click Check Out checkbox (see “Changing Versioning Properties” on page 157).

Checking In Files

You can use two methods for checking in files:

1. **Select the file(s) you want to check in from the main window display and choose Commands > Check In. Enter the appropriate comments in the Check In dialog box and click OK.**

   The check mark(s) continue to be displayed on the file icon(s) until you choose Check In from the pop-up window. This method is valuable when you want to check in several files at once and the same comment can apply to each.

1. **Double click on a checked-out file icon in the main window and add the appropriate comments in the Check In dialog box and click OK.**

   The check mark continues to be displayed on the file icon until you click OK in the dialog box. This is *not* the default behavior. To enable this double-click action, you must change the default behavior by Choosing View > Options, selecting the Toggle SCCS State radio button in the Double Click Action section of the Options dialog box, and then selecting the Confirm Double Click Check Out checkbox.

Editing a Checked-Out File

This section covers the process of checking out a file that is under SCCS control and displaying it in an editor window. The Edit menu item allows you to do this in either of the following ways:

1. **To edit a file after checking it out, select the checked-out file in the Versioning window and choose File > Edit.**

1. **To check out a file and display it in an editor window, choose Commands > Check Out and Edit.**

   The default is for the file to be brought up in a cmdtool window running vi. For instructions on selecting an editor, see “Changing Versioning Properties” on page 157.

Checking in a New File

Files that are not under SCCS control are not displayed in the Versioning window. To check in a new file:

1. **Double click on the directory in the Versioning window, or type the directory path in the Directory text box.**

2. **Choose Commands > Check In New.**

The Check In New window opens displaying a list of files not yet under SCCS control.

3. **Select the file from the Check In New window display.**

4. **Enter the appropriate initial comments in the Initial Comments pane and choose Commands > Check In.**

   The checked-in files are removed from the Check In New display and are displayed in the Versioning window Directory pane.

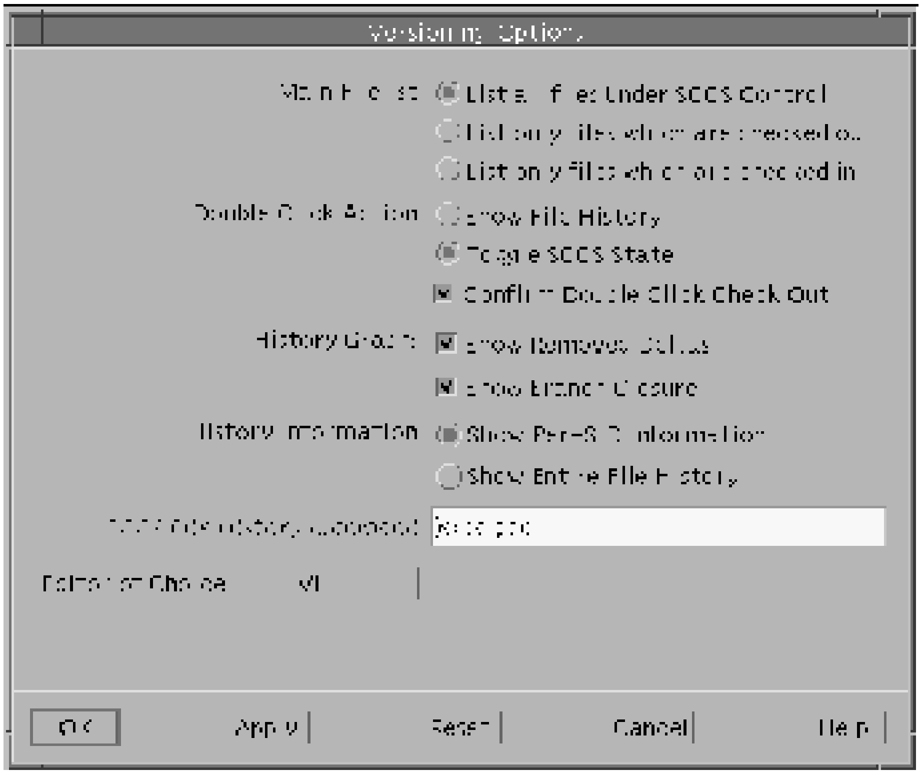
Unchecking Out a File

When you have mistakenly checked out a file and want to return the file to an unchecked out state, there is a simple way to do so without having to check in the file and add comments.

To uncheck out a file, select the checked out file and choose Commands > Uncheckout.

Changing Versioning Properties

You can use the Options dialog box (see Figure 12–3) to set the Versioning properties for the remainder of the session. To open the Options dialog box, choose View > Options.



Mail File List ● List all files Under SCCS Control

○ List only files which are checked out

○ List only files which are checked in

Double Click Action ○ Show File History

● Toggle SCCS State

☒ Confirm Double Click Check Out

History Graph ☒ Show Removed Deltas

☒ Show Branch Closure

History Information ● Show Per-SID Information

○ Show Entire File History

SCCS file history command: sccs prs

Editor of Choice vi

Figure 12–3 Versioning Options Dialog Box

Mail File List	Lets you specify the type of SCCS files displayed in the Versioning window.
Double Click Action	Lets you specify what happens when you double click in Versioning.When you select Toggle SCCS State, you can also select Confirm Double Click Check Out if you want the check mark to be displayed on the file icon until you click OK in the dialog box.
History Graph	Lets you define items for display on the history graph.
History Information	Lets you specify the extensiveness of the information displayed when you select a delta on the history graph. If you select Show Entire File History, you can also specify a command to gather the history.
Editor of Choice	Lets you specify an editor that automatically starts up when you view the contents of a delta, or open a delta to edit. If you select Other, you must type in the text box a command that will start up your editor in a separate window. The file name is appended to the supplied command.

CHAPTER 13

Starting Merging and Loading Files

This chapter explains how to start Merging, load it with files, and save the output file. The chapter is organized into the following sections:

- "Starting Merging" on page 159
- "Saving the Output File" on page 164

Starting Merging

You can start Merging from a shell command line, from the Sun Workshop main window, or from another Sun Workshop TeamWare tool.

Starting Merging from the Command Line

To start Merging from the command line without loading any input files (assuming that the Merging executable is in your search path), at a shell command prompt, type `twmerge` followed by the ampersand symbol (&):

```
demo% twmerge &
```

The `twmerge` command starts Merging (in the background) without loading any files.

The complete `twmerge` command is summarized below, with command options enclosed in square brackets.

`twmerge [-b] [-r] [-a` *ancestor*`] [-f1` *name1*`] [-f2` *name2*`] [-l` *listfile*`] [` *leftfile rightfile* `[`*outfile*`] ][-`

`-b`	Causes Merging to ignore leading blanks and tabs when comparing lines.
`-r`	Starts Merging in read-only mode. When you specify this option, only the input file text panes are displayed, and the output text pane is absent.
`-a` *ancestor*	Specifies an existing ancestor file of the two files to be merged (called *descendants* of the ancestor file). The merged file is based on this ancestor file and the changes to it that have been made in the descendants. When used with the `-l` *listfile* option, *ancestor* is a directory of files, which you can load in succession from the File menu.
`-f1` *name1*	Sets the file name displayed for the first (left) file. This option is useful when a list of files is being loaded (with the `-l` option), and you want to display a name for reference only in the Merging window. For example, if you are loading files from two directories that correspond to two different revisions of a product, you could use the `-f1` option to display the name `Rev1` above the left pane and the `-f2` option to display the name `Rev2` above the right pane.
`-f2` *name2*	Sets the file name displayed for the second (right) file.
`-l` *listfile*	Specifies a file that contains a list of individual file names. This option is useful for merging entire project directories. Merging uses the names in *listfile* to successively load files from directories you name with the *leftfile* and *rightfile* arguments, placing the output files in the directory you name with the *outfile* argument. The names in *listfile* must match file names in the *leftfile* and *rightfile* directories. When used with the `-a` *ancestor* option, the *ancestor* argument must be a directory: Merging will look in the *ancestor* directory for files that have the same names as those in *listfile* and use those with matching names as ancestor files for each merge. If you specify the character "`-`" for *listfile*, Merging reads the list of files from standard input.
leftfile	The name of the left file to be loaded for comparison. When used with the `-l` *listfile* option, *leftfile* is a directory of files, which you can load in succession from the File menu.
rightfile	The name of the right file to be loaded for comparison. When used with the `-l` *listfile* option, *rightfile* is a directory of files, which you can load in succession from the File menu.

outfile — Specifies the name of the merged output file. If you do not specify an *outfile*, the output file is given the default name `filemerge.out`. If you want to specify a different name when you save the file, use Save As from the File menu.

When used with the `-l` *listfile* option, *outfile* names the directory to be used when each merged output file is saved. Individual file names in the *outfile* directory are the same as the names listed in *listfile*.

`-V` — Does not start up Merging, but displays the version number of the application.

Note - If you use the `-l` *listfile* option, then all three input file names (*ancestor*, *leftfile*, and *rightfile*) must be directories. If you do *not* use the `-l` *listfile* option, then any two input file names can be directories, but one of the three must be a simple file name. In this case, Merging uses the file name to find a file with the same local name in each directory.

Loading Two Files at Startup

To load two files at the time you start Merging from the command line, change to the directory in which the files are stored and specify the file names on the command line. To merge two files named `file_1` and `file_2`, use the following command:

```
demo% twmerge file_1 file_2 &
```

The first file listed appears in the left text pane; the second file appears in the right pane (see Figure 4–1).

Loading Three Files at Startup

To merge the same two files and at the same time compare them to a common ancestor named `ancestor_file`, change to the directory in which the files are stored and use the following command:

```
demo% twmerge -a ancestor_file file_1 file_2 &
```

The ancestor file is not displayed, but differences between the ancestor file and the two descendants are marked, and the merged output file is based on the ancestor file.

Loading Files from a List File

You can sequentially load files into Merging from a list of file names. For example, suppose ancestor versions of a project's source files are stored in a directory named /src. You have been editing the files file_1, file_2, and file_3 in your directory /usr_1, and another developer has been simultaneously editing the same files in the directory /usr_2. You have been given the responsibility of merging the changes to both sets of files, and you want to place the merged versions in a directory named /new_src.

To merge the /src, /usr_1, and /usr_2 directories, you first create a list file that contains only the names of the three files to be merged with each name on a separate line, as follows:

```
file_1
file_2
file_3
```

Name the file sourcelist and place it in the working directory where you plan to start Merging. Change to that directory (with the cd(1) command) and start Merging with the following command:

```
demo% twmerge -a /src -1 sourcelist /usr_1 /usr_2 /new_src &
```

This command causes Merging to load /usr_1/file_1 into the left text pane, /usr_2/file_1 into the right text pane, and compare both files to the common ancestor /src/file_1. After you have resolved the differences between the two files, choose Save from the File menu to automatically save the output file as /new_src/file_1.

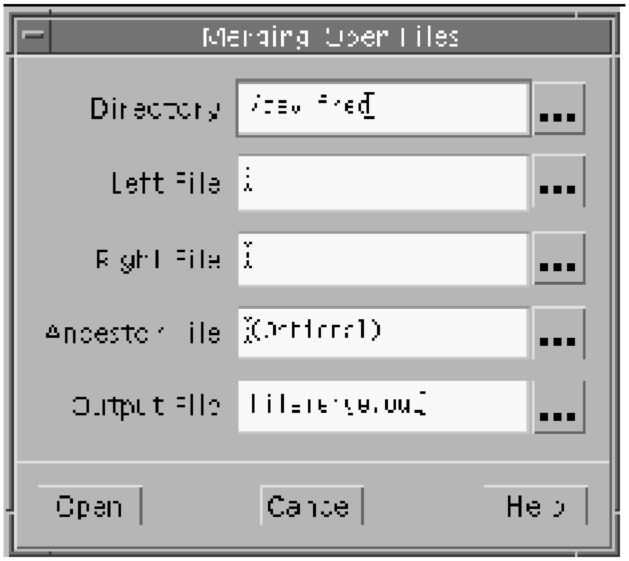
Starting Merging From Sun WorkShop or From Another TeamWare Tool

You can start Merging from tool bar or the Tools menu in the Sun WorkShop main window, or from the TeamWare menu in another TeamWare tool:

- To start Merging from the Sun WorkShop main window, click the Merging icon on the tool bar.
- To start Merging from Sun WorkShop TeamWare, select TeamWare > Merging in the main window of Configuring, Versioning, or Freezepointing.

Loading Files from the Merging Window

To load files in Merging, choose File > Open or click the Open button. The Open Files dialog box (see Figure 13–1) is displayed.



Merging Open Files

Directory /tsw/fred

Left File

Right File

Ancestor File (Optional)

Output File file.merge.out

Figure 13–1 Open Files Dialog Box

Directory text box

Shows the current working directory whenever you start Merging from Sun WorkShop or from the command line with no arguments. You can edit this field. Merging interprets the file names you specify in the window as relative to the current working directory. Therefore, you can use such constructs as *subdir/filename* to specify a file in a subdirectory and *../filename* to specify a file in a parent directory. Any file name you specify that begins with a "/" character is interpreted as an absolute path name, not as relative to the current working directory

Left File text box

Lets you specify the file to appear in the left text pane, also considered the child pane.

Right File text box

Lets you specify the file to appear in the right text pane, also considered the parent pane.

Ancestor File text box

Lets you specify the name of an ancestor file. If you type a file name in this text box, Merging compares the file to the files to be merged and identifies lines in those files that differ from the ancestor. The automerged file is based on the ancestor file, but the ancestor file itself is not displayed in any Merging window. If you do not type an ancestor file name, Merging compares only the left and right files and derives the output file from them. Automerging is not possible without an ancestor file.

Output File text box	Lets you specify the name for the merged output file. Merging uses the name `filemerge.out` unless you specify a different name, and stores the file in the current working directory.
Open button	Loads the files you have specified in the text boxes.

In a loaded Merging window the names of the left file, right file, and output file are displayed above the appropriate text panes. The name of the ancestor file (for a three-way diff only) is displayed in the window header.

Saving the Output File

Save the output file by clicking the Save button or choosing File > Save. The name of the output file is the name you specify in the Output File text box in the Open Files dialog box.

To change the name of the output file while saving, choose File > Save As and type the new file and directory names in the Save dialog box.

CHAPTER **14**

Examining Differences

This chapter further explains some features and presents an example of how to use Merging. The chapter is organized into the following sections:

- "Moving Between Differences" on page 165
- "Resolving Differences" on page 165
- "Automatic Merging" on page 166
- "Reloading Files" on page 166
- "An Example" on page 166

Moving Between Differences

You can navigate through differences between files using either the Next and Accept & Next buttons or the up arrow and down arrows.

Resolving Differences

A difference is represented by a blank line in the merged (output) file in the lower text pane. To resolve a difference, you edit the line displayed there by either:

- Accepting the line displayed and incorporating it into the merged file by clicking either the Accept or Accept & Next button over the pane you wish to accept
- Editing the line in the merged file by hand, and marking the difference as resolved by choosing Edit > Mark Selected as Resolved

Automatic Merging

If you have loaded a common ancestor file, Merging is often able to resolve differences automatically, based on the following rules:

- If a line has not been changed in either descendant (it is identical in all three files), it is placed in the merged file.
- If a line has been changed in only one of the descendants, the changed line is placed in the merged file. A change could be the addition or removal of an entire line, or an alteration to some part of a line.
- If identical changes have been made to a line in both descendants, the changed line is placed in the merged file.
- If a line has been changed differently in both descendant files so that it is different in all three files, Merging places no line in the merged file. You must then decide how to resolve the difference—either by using a line from the right or left file, or by editing the merged file by hand.

Reloading Files

If Merging is invoked by a Resolve transaction in Configuring, the Reload button is displayed in the Merging window tool bar. This button lets you ask Merging to abandon all edits that have been performed on the two files and reload them from disk, and to automatically resolve any non-conflicting differences if the Auto Merge option is selected.

An Example

This example merges two files that have a common ancestor. The files are `file_1` and `file_2`, and the ancestor file is named `matriarch`. The descendant files `file_1` and `file_2` were derived from `matriarch` by editing. The edits show all varieties of changes that could occur in the descendants: deleting lines, adding new lines, and changing lines.

The content of each line in the example helps to identify whether or not it was changed, and how. The ancestor file contains only twelve lines and is shown in Code Example 14–1.

Merging does not number lines in the files it loads; the numbers are part of the example text and were placed there for clarity.

CODE EXAMPLE 14–1 Ancestor File (`matriarch`)

```
1 This line is deleted in file_1
2 This line is in all three files
3 This line is deleted in file_2
4 This line is in all three files
5 This line is in all three files
6 This line is changed in descendants
7 This line is in all three files
8 This line is changed in descendants
9 This line is in all three files
10 This line is changed in file_2
11 This line is in all three files
12 This line is in all three files
```

Code Example 14–2 shows the contents of `file_1`. This file is identical to `matriarch` with the following exceptions:

- The line numbered 1 in the `matriarch` file was deleted in `file_1`.
- A new line was added following the line numbered 4.
- The line numbered 6 was changed (a different change was made to this line in `file_2`).
- The line numbered 8 in the `matriarch` file was changed (an identical change was made to this line in `file_2`.)

CODE EXAMPLE 14–2 Descendant File (`file_1`)

```
2 This line is in all three files
   3 This line is deleted in file_2
   4 This line is in all three files
    &&& Added to file_1 &&&
   6 This line is modified in file_1 from matriarch
   5 This line is in all three files
   7 This line is in all three files
   8 #&# Changed in file_1 and file_2 #&#
   9 This line is in all three files
   10 This line is changed in file_2
   11 This line is in all three files
   12 This line is in all three files
```

Code Example 14–3 shows the contents of `file_2`. This file is identical to `matriarch` with the following exceptions:

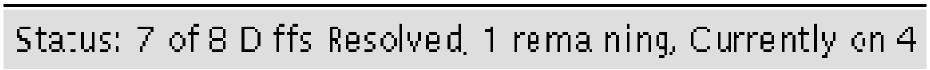
- The line numbered 3 in the `matriarch` file was deleted.
- The line numbered 6 was changed (a different change was made to this line in `file_1`).
- The line numbered 8 was changed (an identical change was made to this line in `file_1`).

- The line numbered 10 was changed (no change was made to this line in `file_1`).
- A new line was added following the line numbered 11.

CODE EXAMPLE 14–3 Descendant File (`file_`)

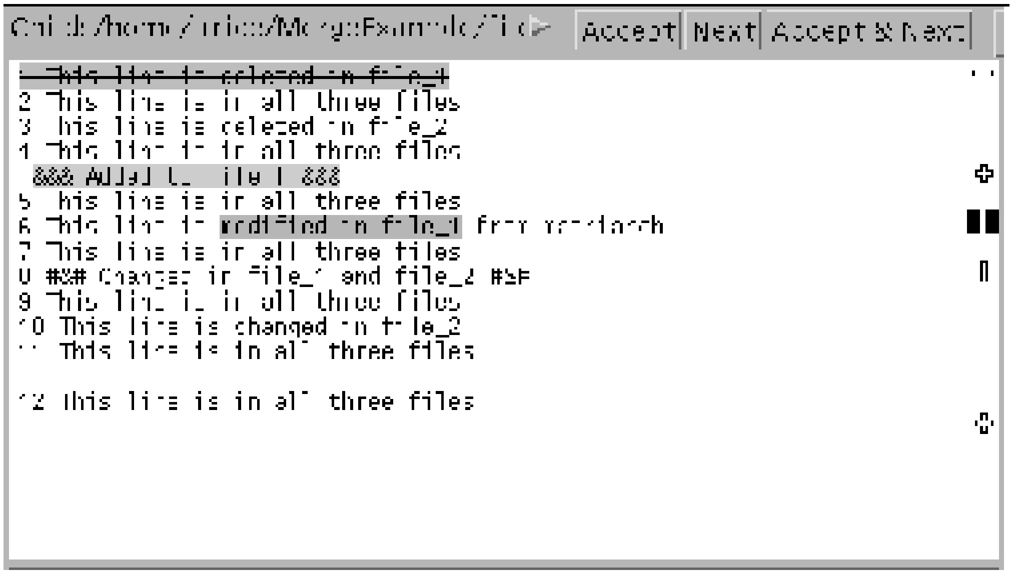
```
1 This line is deleted in file_1
2 This line is in all three files
4 This line is in all three files
5 This line is in all three files
6 This line is altered in file_2 from matriarch
7 This line is in all three files
8 #&# Changed in file_1 and file_2 #&#
9 This line is in all three files
10 ### Changed in file_2 ###
11 This line is in all three files
 ### Added to file_2 ###
12 This line is in all three files
```

In the upper left of the Merging window, Merging has reported finding seven differences, of which only one remains unresolved (see Figure 14–1)—six differences were resolved by automerging, and are marked by glyphs in outline font (see Figure 14–2 and Figure 14–3).



Status: 7 of 8 Diffs Resolved, 1 remaining, Currently on 4

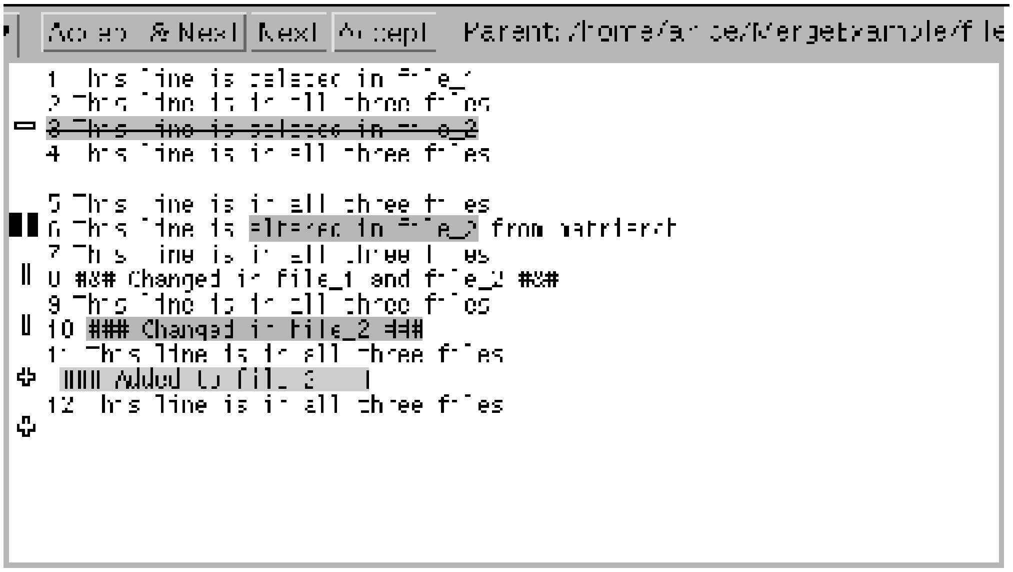
Figure 14–1 Merging Status of File_1 and File_2 After Automerging



~~1 This line is deleted in file_1~~

```
2 This line is in all three files
3 This line is deleted in file_2
4 This line is in all three files
 &&& Added to file_1 &&&
5 This line is in all three files
6 This line is modified in file_1 from matriarch
7 This line is in all three files
8 #&# Changed in file_1 and file_2 #&#
9 This line is in all three files
10 This line is changed in file_2
11 This line is in all three files

12 This line is in all three files
```

Figure 14–2 `File_1` Displayed in Child Pane After Automerging



```
1 This line is deleted in file_1
2 This line is in all three files
3 This line is deleted in file_2
4 This line is in all three files

5 This line is in all three files
6 This line is altered in file_2 from parent
7 This line is in all three files
8 ### Changed in file_1 and file_2 ###
9 This line is in all three files
10 ### Changed in file_2 ###
11 This line is in all three files
 ### Added to file_2 ###
12 This line is in all three files
```

Figure 14–3 `File_2` Displayed in Parent Pane After Automerging

The meaning of the glyphs is as follows: a vertical bar means a change in the marked line, a plus sign signifies a line added, a minus sign means a line was deleted. Unresolved states are marked by solid glyphs, unresolved by outline. These glyphs are highlighted in color except when the color map is full. The default significance is: red indicates a change, green indicates a deletion, yellow shows an addition.

The unresolved difference (line 6) is marked by a vertical bar.

Examining Differences

Merging highlights the unresolved difference, which it identifies as the line numbered 6 in `file_1` and `file_2`. When differences are being resolved with Merging, the resulting Merging window (`filemerge.out`) shows the current state of the file with automatic merging.

You could proceed to the next difference by clicking the down arrow above the appropriate file or choosing Navigate > Next. The next difference becomes the current difference.

You could proceed through the differences by clicking on the down arrow.

Automerging preserves a change that was made to one file if no change was made in the other file.

Glyphs

When a difference has not been resolved by automerging, as indicated by the solid highlighted glyph next to the lines involved in the difference, you need to resolve the difference by making a choice. The vertical line indicates that the line has been changed (as opposed to added or deleted). In this case, automerging failed because the same line was changed differently in the two files, and Merging could not decide which change was more valid.

Resolving a Difference

You could resolve this difference in one of the following ways:

- Clicking the Accept or Accept & Next button on the left to place the line from `file_1` into the output file
- Clicking the Accept or Accept & Next button on the right to place the line from `file_2` into the output file
- Editing the output file by hand

Editing the Output File

To edit the output file, you would move the pointer into the output file's text pane and place it in the line you want to change. In this example, the following line was typed in:

```
>>> This line edited by hand <<<
```

You would choose Edit > Mark Selected as Resolved. This menu item marks the difference as resolved. In this example there are no more unresolved differences, so the next difference remains the current one.

The message in the upper left part of the window now indicates that all differences have been resolved. Nevertheless, you would proceed to verify the automerged differences.

You could continue to navigate through the differences by clicking the down arrow.

The final difference results from a line that was added only to `file_2`. Merging would place the new line in the output file just as it did when a new line was added to `file_1`, which resulted in the third difference.

Saving Output File

Once all differences have been resolved and the automerged differences verified, you would save the output file. The output file takes the name shown in the Open Files dialog box, which by default is `filemerge.out.`To write the file, you would choose

File > Save or click the Save button. To save the file under another name, you would choose File > Save As.

CHAPTER **15**

Preserving Project State With Freezepointing

This chapter discusses how Freezepointing can help you preserve snapshots of your work, and the process of recreating those key points. The chapter contains the following sections:

- "Introduction to Freezepointing" on page 173
- "How Freezepointing Works" on page 174
- "Starting Freezepointing" on page 176
- "Creating a Freezepoint File" on page 178
- "Recreating (Extracting) a Source Hierarchy" on page 180

Introduction to Freezepointing

During the software development process it is often useful to create "freezepoints" of your work at key points. Those freezepoints serve as snapshots of a project that enable you to later recreate the state of the project at key development points.

One way to preserve the state of the project is to make a copy of the project hierarchy using the `tar` or `cpio` utilities. This method is very effective, but it requires a large amount of storage resources and time.

With Freezepointing, you preserve freezepoints quickly and simply, using a small amount of storage resource.

You can use Freezepointing through two functionally equivalent user interfaces. You can access the user interfaces with the following commands:

- `twfreeze`—for the GUI

- `freezept`—for the CLI

Note - Freezepointing is a companion tool to the Configuring tool. Therefore, Freezepointing assumes that you are creating freezepoints of Configuring workspace hierarchies. You can also use Freezepointing to preserve nonworkspace directories that contain SCCS files. If you specify a directory that is not a workspace, a cautionary warning is issued.

This chapter refers primarily to the GUI. For information about the CLI, see the `freezept`(1) man page. The GUI is documented online. You can access the online help from any TeamWare GUI by opening the pull-down menu from the Help button, and clicking on Help Contents.

How Freezepointing Works

Freezepointing enables you to create freezepoint files from Configuring workspaces.

Note - Nonworkspace directory hierarchies that contain SCCS history files can also be preserved using Freezepointing. Freezepointing issues a warning if the directory is not a workspace.

At a later time you can use the freezepoint files to recreate the directory hierarchies contained in the workspaces.

Note - Freezepointing gives you the option of recreating a workspace containing the SCCS histories for all extracted files. Alternatively, you may choose to extract the derived files only; that is, retrieve the default delta using the SCCS `get` command with no options. In this case, SCCS histories are not recreated and TeamWare does not create a workspace.

Freezepoint Files

A freezepoint file is a text file that lists the default deltas from the SCCS history files contained in the workspace hierarchy being preserved. When you later recreate the hierarchy, Freezepointing uses those entries as pointers back to the original history files and to the delta that was the default at the time the freezepoint file was created.

The deltas are *not* identified by their standard SCCS delta ID (SID). Instead, a new means of identification called an SCCS Mergeable ID (SMID) is used. Using the SMID enables Freezepointing to work properly with files in which SIDS have been renumbered as part of a Configuring Bringover Update transaction. For more information, see Appendix A.”

The freezepoint file contains the following information:

- The path name of the workspace from which the list of deltas was created
- The date and time that the file was created
- The login name of the user who created the freezepoint
- A group of hex digits that identifies the most recent SCCS deltas found in each file’s corresponding SCCS history file
- A group of hex digits that identifies the root delta in each file’s corresponding SCCS history file
- An optional user-supplied comment

The following example shows three entries from a freezepoint file.

```
filemerge.1 (previously 1.5) 92/03/19 14:09:08 jon a6f4fe81 89b4632b 418e7950 5510740e cf9ab4e1 95627c33 2
putback.1 (previously 1.40)92/06/02 16:36:16 george 5b791c60 2b827cfd f0cc9a73 46ac975 24d9b3ec f87d1975 9
resolve.1 (previously 1.19) 92/06/10 16:38:07 paul f21fa6e6 668bf818 e4964f36 240d825c f1d3f57 8cc4c31c 9f
```

You can view the contents of a freezepoint file using a standard text editor.

Creation

When you create a freezepoint file, you specify directories and files to Freezepointing in the Directories and Files pane of the Freezepointing window (see Figure 15–1). Freezepointing recursively descends the directory hierarchies and identifies the most recently checked-in deltas in each SCCS history file. Freezepointing then creates a freezepoint file that consists of a list of those files and unique numerical identifiers for each delta.

Extraction

You can later use Freezepointing to recreate the source hierarchy. You specify the name of the freezepoint file, the path name of the directory hierarchy from which the deltas are to be extracted (if different from the hierarchy from which it was derived), and the directory where you want the source hierarchy recreated.

The extract operation consists of creating a new directory hierarchy based on the information contained in the freezepoint file. The new hierarchy is comprised of g-files defined by the default deltas in the original SCCS history files; *the history files themselves are not recreated* unless you ask FreezePointing to create a workspace while performing the extraction. Deltas are extracted from SCCS history files located in the original source workspace.

Source Workspace

The source workspace is the directory hierarchy that contains the SCCS history files from which the freezepoint file is created. Usually, the source workspace is also the directory hierarchy from which g-files are later extracted to recreate the hierarchy.

Note - You can specify an alternate source directory at the time you perform the extract operation.

Destination Directory

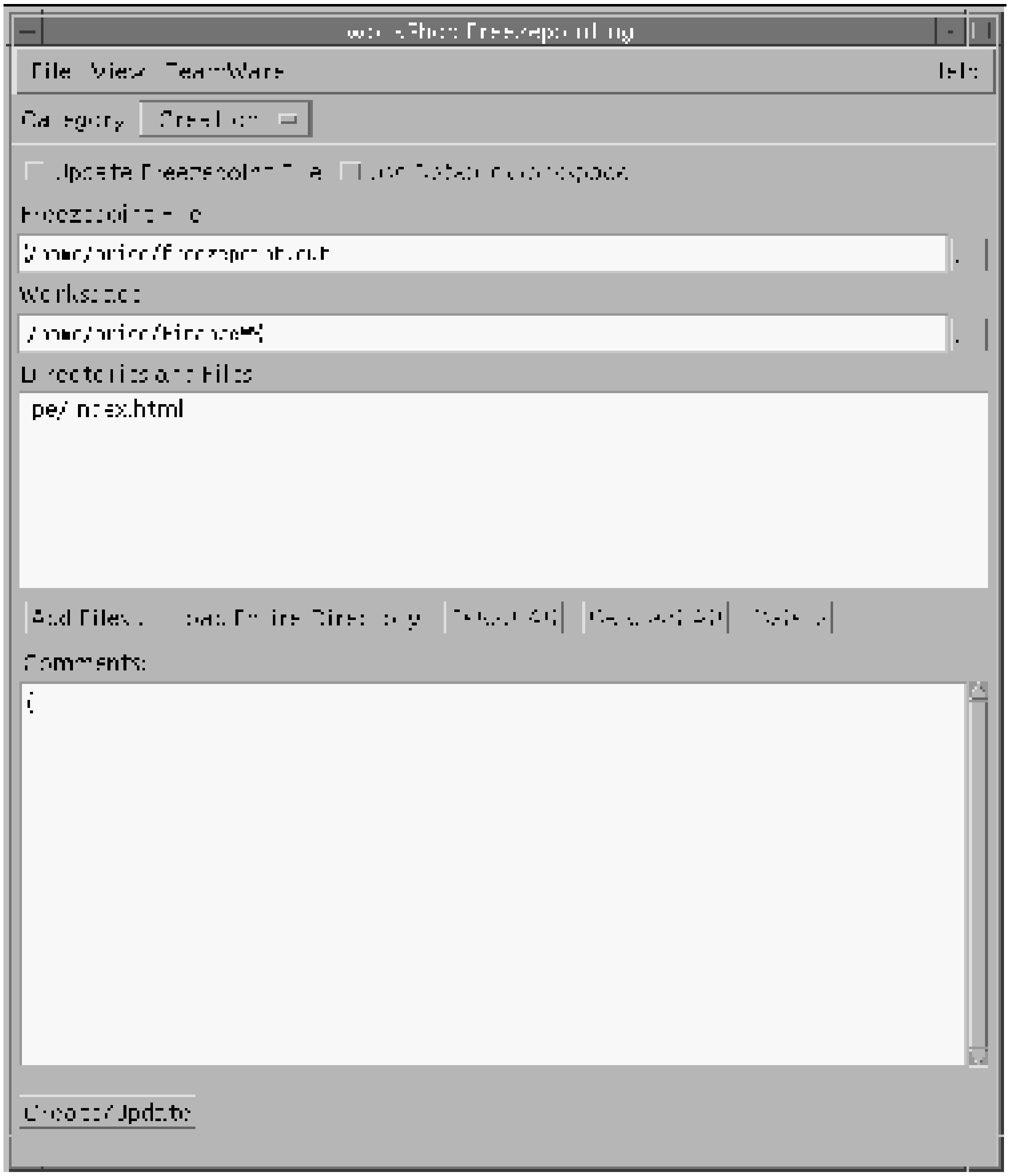
The destination directory is the top-level directory into which the files listed in the freezepoint file are extracted. You specify the path name of this directory in the Extract pane of the Freezepointing window.

Starting Freezepointing

You can start Freezepointing by:

- Typing `twfreeze` at a shell command prompt, followed by the ampersand symbol (&)
- Choosing TeamWare > Freezepointing in the Configuring, Merging, or Versioning window

The Freezepointing window opens in Creation mode (see Figure 15–1).



Freezepoint File: /home/brian/freezepoint.out

Workspace: [illegible]

Directories and Files: pe/index.html

Figure 15–1 Freezepointing Window in Creation Mode

File menu	Provides a command to exit Freezepointing
View menu	Provides the Show Output command
TeamWare menu	Provides commands for starting other TeamWare tools

Category list box	Lets you switch between the Creation and Extraction panes of the Freezepointing window
Update Freezepoint File checkbox	Lets you choose to update an existing freezepoint file rather than create a new one
Use Default Workspace checkbox	Lets you use the workspace named in the freezepoint file
Freezepoint File text box	Lets you type the absolute path name of the freezepoint file
Workspace text box	Lets you specify the source workspace.
Directories and Files pane	Contains a list of files and directories that will be preserved in the freezepoint file
Add Files button	Opens the Add Files dialog box, where you can select files to add to the Directories and Files pane
Load Entire Directory button	Loads the entire workspace directory
Select All	Selects all the files and directories listed in the pane
Deselect All	Deselects all the files and directories listed in the pane
Delete	Deletes the selected items from the pane

Creating a Freezepoint File

To create a Freezepoint file:

1. **Select Creation from the Category list box.**

   The Creation pane is displayed.

2. **If you want to update an existing Freezepoint file, select the Update Freezepoint File checkbox.**

   The Use Default Workspace checkbox is enabled.

3. **To freezepoint the workspace named in the existing Freezepoint File, select the Use Default Workspace checkbox.**

4. **To create a new Freezepoint file, type the name for the new file in the Freezepoint File text box.**

   When you start Freezepointing, the Freezepoint File text box is automatically set to contain the file `freezepoint.out` appended to the path name of the directory from which Freezepointing was started. Delete `freezepoint.out` and type the path name of your freezepoint file in the Freezepoint File text box.

   Path names that are not absolute are assumed to be relative to the directory in which Freezepointing is started.

5. **Type the name of the source workspace in the Workspace text box.**

   When you start Freezepointing, the Workspace text box is automatically set to the workspace you have specified through the `CODEMGR_WS` environment variable. If the variable is not set, and the directory from which Freezepointing is started is hierarchically within a workspace, the Workspace text box is initialized with the path name of that workspace.

6. **Create a list of the directories and files that you wish to preserve in the Directories and Files pane.**

   - You can use the Add Files dialog box to conveniently add directories and files to the Directories and Files pane. Open the Add Files dialog box by clicking Add Files.
     - You can select entire groups of files by clicking and holding the left mouse button in an empty portion of the dialog box and dragging the bounding box to surround any number of icons. When you release the button, all the files within the bounding box are selected.
     - Select files and directories by moving the pointer over any file or directory icon and clicking. You can extend the selection to include any number of additional files and directories by moving the pointer over them and clicking the middle mouse button.
     - You can also type the path name of a directory or file into the Name text box.
     - Click Add Files to List to add the file(s) to the Directory and Files pane.
   - The Load Entire Directory button inserts the "`./`" characters into the Directories and Files pane indicating that the entire workspace hierarchy be recursively preserved.

7. **Type an optional comment in the Comments text pane.**

   The comment is stored in the freezepoint file for future reference.

8. **Select the Create/Update button to create the freezepoint file.**

A counter on the bottom left corner of the Freezepointing window displays the progress of the freezepoint operation.

Note - Use the Delete button to delete selections from the Directories and Files pane. Use the Select All and Delete All buttons to select or delete large numbers of directories and files.

Recreating (Extracting) a Source Hierarchy

To extract a new source hierarchy described by a freezepoint file:

1. **Select Extraction from the Category list box.**

   The Extraction pane is displayed (see Figure 15–2).

2. **Type the path name of your freezepoint file in the Freezepointing File text box.**

   Path names that are not absolute are assumed to be relative to the directory in which Freezepointing is started.

3. **Select the Full Extract radio button or the Partial Extract radio button.**

   Full Extract extracts the complete set of frozen files. Partial Extract extracts a subset that you identify.

   You must choose either a full or partial extract whether or not you choose to create a workspace.

4. **Select the Create Workspace checkbox if you want to create a workspace that contains the SCCS histories of the frozen files.**

5. **Select one of the three Extract FreezePoint Sources radio buttons to specify the source workspace.**

   - The Use default from freezepoint files radio button uses the path name of the source workspace as it is in the freezepoint file.
   - The You specify radio button lets you type a workspace path name.
   - The Show default radio button displays the path name of the source workspace in the Workspace text box.

If you wish to specify a source workspace hierarchy other than the one contained in the freezepoint file, select the You specify radio button and type the path name of the alternate source workspace in the Workspace text box.

6. **Type the path name of the directory in which you want the new (extracted) hierarchy to be located in the Destination Directory text box.**

   Path names that are not absolute are assumed to be relative to the directory in which Freezepointing is started.

   The destination directory that you specify can be new or existing. If you extract the hierarchy to an existing directory, you receive a warning message and must confirm the operation.

7. **Click the Extract button to begin the extraction.**

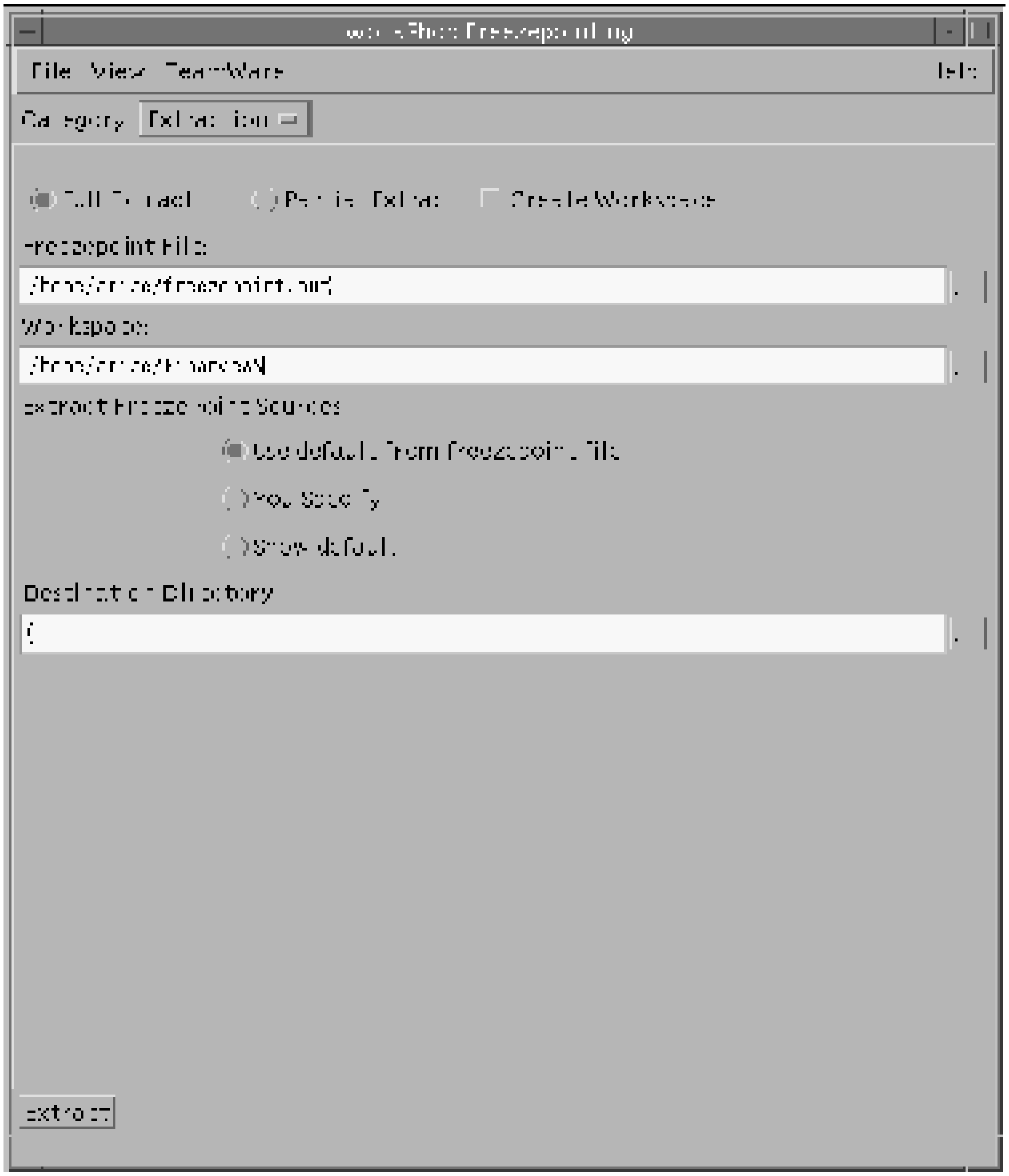
   If you selected Partial Extract, Freezepointing opens a dialog box listing the source files in the freezepoint file. Select the files you want to extract.

   Clicking the Extract button causes a series of `sccs get` operations to be performed on the source files listed in the freezepoint file. The version of each file extracted is the version specified by the SMID in the freezepoint file. The extracted g-files are written to destination directory. If you have selected Create Workspace, SCCS histories are also written to the destination directory.

A counter on the bottom left corner of the Freezepointing window displays the progress of the extract operation

---

Note - If, during an extraction, Freezepointing cannot locate a file that has been renamed or deleted, the extraction is aborted and the offending entry is named. You must edit the freezepoint file to remove the entry. Refer to the `freezepointfile`(4) man page for information that enables you to determine the new name of a renamed file.



Category: Extraction

● Full Extract ○ Partial Extract □ Create Workspace

Freezepoint File:
/home/janice/freezepoint.out

Workspace:
/home/janice/framework

Extract Freezepoint Sources

● Use default from freezepoint file
○ You Specify
○ Show default

Destination Directory
/

Figure 15–2 Freezepointing Window in Extraction Mode

File menu	Provides a command to exit Freezepointing
View menu	Provides the Show Output command
TeamWare menu	Provides commands for starting other TeamWare tools

Category list box	Lets you switch between the Creation and Extraction panes of the Freezepointing window	
Full Extract radio button	Lets you extract the complete set of frozen files	
Partial Extract radio button	Lets you extract a subset of the frozen files	
Create Workspace checkbox	Lets you create a TeamWare workspace from the freezepoint file	
Freezepoint File text box	Lets you type the absolute path name of the freezepoint file	
Workspace text box	Lets you specify the source workspace	
Extract Freezepoint radio buttons	Use default from	Uses the path name of the source workspace
	freezepoint file:	Lets you type a workspace path name
	You Select:	Uses the path name of the source workspace
	Show default:	
Destination Directory text box	Leta you specify the path name of the directory in which you want the new hierarchy to be located	

CHAPTER **16**

Troubleshooting Versioning and FreezePointing

This chapter describes some of the most common problems in Versioning and FreezePointing. It indicates where to look for information on how to overcome the problem. It is organized into the following sections:

- "Troubleshooting Checklist" on page 185
- "Reporting Problems" on page 186
- "Error Messages" on page 186
- "Display Problems" on page 187

Troubleshooting Checklist

If you are having problems using Versioning or FreezePointing, use the following checklist to rule out some of the most common reasons for the problem:

- Is the tool installed correctly?

  If not, contact your system administrator. You can also read Sun WorkShop installation documentation.

- Is `/opt/bin` in your `PATH`?

  If not, see Sun WorkShop installation documentation for information on how to add `/opt/bin` to your `PATH`.

- Is `/usr/lang` in your `PATH`?

  If not, see Sun WorkShop installation documentation for information on how to add `/usr/lang` to your `PATH`.

- Is the HELPPATH environment variable set?

  Versioning relies on finding the vertool.info file in or near the directory that contains vertool. Freezepointing relies on finding the freezepoint.info file in or near the directory that contains freezepoint. On-line help is available for each control, window, pane, and error message displayed on the screen. See "Error Messages" on page 186" for a list of the Versioning and Freezepointing error messages and instructions on what to do next.

- Do you have enough swap space?

  If you receive a message stating "Request for *xxx* bytes of memory failed," you have run out of swap space. Use the `mkfile`(8) and `swapon`(8) commands to create more swap space or abort some existing processes (windows) to free up swap space. To determine which processes occupy significant swap space, use the `ps uagx` command and look in the SZ column. To determine how much swap space you have, use the `pstat -s` command.

- Does your window system have enough resources?

  If Versioning or Freezepointing cannot activate a pop-up window, your window system may be running out of resources. Contact your system administrator for help.

Reporting Problems

If you have gone through the checklist and are still having problems, call your local service office. Have the version number of the tool ready to give to the dispatcher. To display the version number of any TeamWare component, type `toolname -V` at the prompt. For example:

```
twversion -V
```

Error Messages

Versioning and Freezepointing display messages to provide you with information or tell you about an error.

Display Problems

If you experience difficulties in color display in any of the services, you can copy the pertinent part of the resource files in the `app-defaults` directory. These files adjust color, fonts, mnemonics, and several other attributes for each TeamWare component.

These files are under the directory in which you have TeamWare installed, according to the language you are using. If you use CDE the files are in: `TW2.1/lib/locale/<lang>/app-defaults/CDE`. The resource files are listed according to component, preceded by X. For example:

```
...TW2.1/lib/locale/C/app-defaults/CDE/XCodeManager
```

If you use OpenWindows or any other non-CDE windowing system, the files are in `...TW2.1/lib/locale/C/app-defaults/non-CDE`.

To make changes to your display:

1. **Look at the resource file you want to change, and copy the appropriate lines.**

   For example, if you want to change the foreground color of the Configuring window, copy the line:

   ```
   XCodeManager*foreground: black
   ```

2. **Append the pertinent lines to your own `~/.Xdefaults` file.**

   Change it, for example, to:

   ```
   XCodeManager*foreground: green
   ```

3. **Type `xrdb < ~/.Xdefaults`.**

4. **Restart the application.**

The window foreground color will be set.

CHAPTER **17**

Building Programs in Sun Workshop TeamWare

Sun WorkShop TeamWare provides you with the ability to run one build job at a time or several build jobs concurrently. This chapter shows you how to quickly build a single application, how to customize a build, and how to fix build errors using the Building window and the Sun WorkShop editor of your choice.

This chapter is organized into the following sections:

Building a TeamWare Target

When you build a program in TeamWare, you are actually building a *TeamWare target*, which is an object derived from the following:

- *Build directory*—The directory from which the build process is invoked and also the default directory for the makefile.

- *Build command*—The command that invokes the make utility, which reads the makefile and builds the make targets.
- *Makefile*—A file that contains entries that describe how to bring a make target up to date with respect to those files on which it depends (called *dependencies*). Since each dependency is a make target, it may have dependencies of its own. Targets and file dependencies and subdependencies form a tree structure that make traces when deciding whether or not to rebuild a make target.
- *Make target*—An object that make knows how to build from the directions (rules) contained in a particular makefile. For example, a make target could be all or clean. Makefiles are generally designed so that the default target (the one you get when you do not specify a target) is the most commonly built target.

When a TeamWare target is built, it is added to the list of TeamWare targets in the Build menu and in the Edit Target command. When you begin a build, TeamWare looks for the first target in the teamWare target list and builds it.

For information on the make utility, makefiles, and make targets, see Appendix B in *Using Sun WorkShop*.

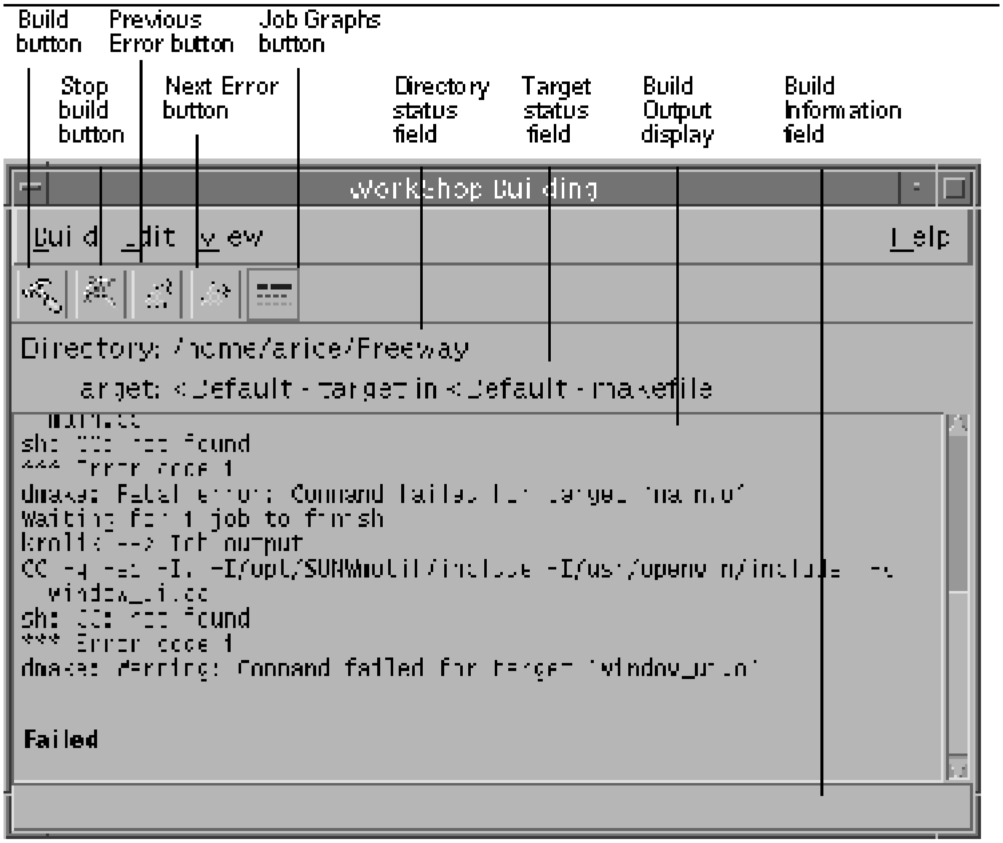
Building Window

The Building window displays information on program compilation. You can open the window by choosing TeamWare > Building in the Configuring, Versioning, or Freezepointing window.

From the Building window, you can:

- Stop a build in progress
- Edit build parameters
- Save the build output to another file
- View build errors

Figure 17–1 shows the Building window.



Directory: /home/arice/Freeway

Target: <Default - target in <Default - makefile

```
  main.cc
sh: CC: not found
*** Error code 1
dmake: Fatal error: Command failed for target `main.o'
Waiting for 1 job to finish
krolik --> Job output
CC -g -c -I. -I/opt/SUNWmotif/include -I/usr/openwin/include -c
  window_ui.cc
sh: CC: not found
*** Error code 1
dmake: Warning: Command failed for target `window_ui.o'


Failed
```

Figure 17–1 Building Window

Build menu	Provides commands for common build operations
Edit menu	Provides commands to accumulate data and to clear the Build Output Display pane
View menu	Provides commands for navigating build errors in the Build Output Display pane and for viewing information on multiple build processes when running a distributed make
Build button	Begins a build of the current TeamWare target
Stop Build button	Stops the current build in progress
Previous Error button	Moves the cursor to the previous build error in the Build Output Display pane and shows that error location in the text editor

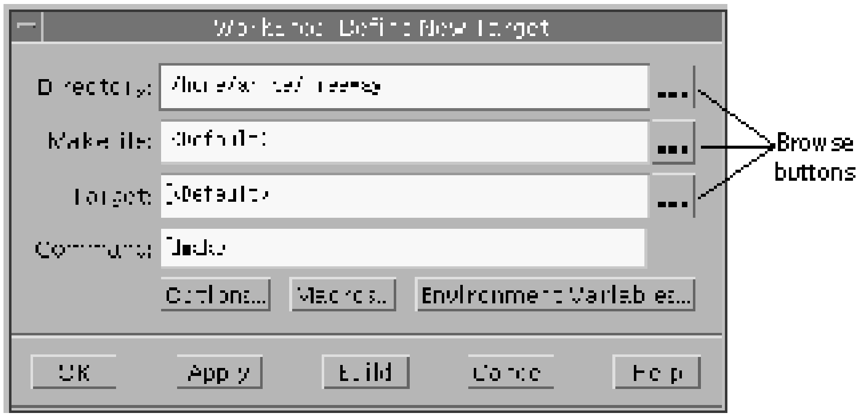
Next Error button	Moves the cursor to the next build error in the Build Output Display pane and shows that error location in the text editor
Jobs Graph button	Opens the Dmake Jobs Graph window
Directory status field	Displays the path name of the current build directory
Target status field	Displays the name of the current make target
Build Output Display pane	Displays output for the current build operation
Build Information field	Displays information about the current build

Building a Program

You can begin a build without having to specify a build command, makefile, or target. Or you can specify one or all of these. You can also customize a build by specifying make options, specifying a build mode, overriding makefile macros, or editing environment variables (see “Customizing a Build” on page 197).

Define New Target and Edit Target Dialog Boxes

You specify build parameters using the Define New Target and Edit Target dialog boxes, which are basically identical. You use the Define New Target dialog box to specify a new TeamWare target and the Edit Target dialog box to modify an existing TeamWare target. Figure 17–2 shows the Define New Target dialog box.



Directory: [illegible]

Makefile: <Default>

Target: <Default>

Command: dmake

Figure 17–2 Define New Target Dialog Box

Directory text box

Lets you type a build directory path. You can also select a directory by clicking on the browse button.

Makefile text box

Lets you specify a makefile (the default file name is `makefile` or `Makefile`. You can also select a makefile by clicking on the browse button.

Target text box

Lets you specify a make target. You can also select a target by clicking on the browse button.

Browse buttons

Let you display dialog boxes in which you can choose a build directory, makefile, or make target

Command text box

Lets you type a make command; the default command is `dmake` (described in “Running a Distributed Build” on page 211).

Options button

Opens the Options dialog box (see “Specifying Make Options” on page 197). The Options dialog box allows you to modify the parameters of a build using the options provided.

Macros button

Opens the Make Macros dialog box (see “Using Makefile Macros” on page 202), which allows you to add, change, or delete macros to be passed into the build.

Environment Variables button

Opens the Environment Variables dialog box (see “Using Environment Variables” on page 205), which allows you to add, change, or delete environment variables to be passed into the build.

OK button	Applies the build parameters and closes the dialog box.
Apply button	Applies the build parameters.
Build button	Applies the build parameters and builds the target.
Cancel button	Closes the dialog box without applying changes.
Help button	Displays online help for the dialog box.

Building With Default Values

You can begin a build without having to specify a build command, makefile, or target. TeamWare provides a default makefile name (`makefile`), a default make target, and a default make command, `dmake` (see “Running a Distributed Build” on page 211). All you need to supply is the path name for the build directory.

Default Makefile and Make Target

The Define New Target dialog box contains the value `Default` in the Makefile and Target text boxes. If you do not specify a particular makefile or make target, TeamWare looks for a file named `makefile` in the build directory and uses the first make target in that makefile. However, if make finds an SCCS history file (`s.makefile`) that is newer than the file named `makefile`, TeamWare uses the most recent version of `s.makefile`. If `makefile` does not exist, TeamWare searches for a file named `Makefile`. Again, if an SCCS history file (`s.Makefile`) exists that is newer than Makefile, TeamWare uses the most recent version of `s.Makefile`.

Using Default Values

To build a program using default build values:

1. **Look in the Directory status field in the Building window to be sure you have the correct build directory set.**

2. **If the current build directory is correct, click the Build button, choose Build > Build in the Building window, or select a TeamWare target from the list at the bottom of the Build menu.**

3. **If no build directory is displayed in the Directory status field or you want to change build directories, choose Build > New Target to open the Define New Target dialog box. Type the build path in the Directory text box.**

You can also click the browse button to open a directory chooser. Choose a directory in the list and click OK to load it into the Directory text box. Then click Build at the bottom of the dialog box.

The build output is displayed in the Build Output display pane in the Building window. Click the Stop Build button in the Building window or choose Build Stop Build to stop the build process.

Note - The next time you open the Building window, the build directory is set to the last directory in which you ran a build job. You can see the path name in the Directory status field.

Building With Nondefault Values

If you have a makefile with a unique name, a certain make target, or a specific build command, specify it in the Define New Target dialog box or Edit Target dialog box by clicking on the appropriate browse button (see Figure 17–2). Each browse button displays a dialog box.

1. **Type the name of the directory in which you want to build and click Apply to apply the change.**

   You can also select another directory from the Set Build Directory dialog box. If you have not specified a build directory, TeamWare either tries to build in the directory currently displayed in the build directory field or, if no directory is displayed, displays an error message pop-up window.

2. **Type the name of the makefile you want in the Makefile text box.**

   If you want to choose another makefile from the current build directory, type the name of the makefile in the Makefile text box, or choose a makefile from the list in the Set Makefile dialog box, and click OK.

   You can run your build in a directory that is different from the one the makefile is in. Just specify the full path name of the makefile in the Makefile text box.

3. **Type the name of the make target you want in the Target text box.**

   Type the name of the make target in the Target text box, or choose another make target in the current makefile in the Target Chooser dialog box, and click OK.

4. **Type the name of the build command you want in the Command text box.**

   If the build command you specify is something other than `make` or `dmake`, you can specify the command and any of its arguments in the Command text box. The build command is formed by prepending `setenv` commands for any environment variables specified through the Environment Variables dialog box and by appending any of the make options specified through the Options and Make Macros dialog boxes.

Note - If the path to the build command is not in your PATH environment variable, you might have to specify the full command path.

---

5. **Click Build in the dialog box to start a build with the settings you supplied.**

The build output is displayed in the Build Output display pane in the Building window. Click the Stop Build button in the Building window or choose Build > Stop Build to stop the build process.

Collecting Build Output

Build output is cleared from the Build Output display pane each time you run a build job. You can keep the build output from previous builds by setting the Accumulate Output switch to on.

To collect build output, choose Edit > Accumulate Output.

The Accumulate Output command toggles the switch on and off. When a build is performed, output for that build is displayed below the output for the previous build. You can scroll through the pane to see the output for each build. To identify specific build jobs, each build output begins with the build path and the name of the build target.

To clear the build output log, choose Edit > Clear Results.

Saving Build Output

You can maintain a history of build output information for one or more build jobs by saving the output to a file.

To save build output:

1. **Choose Build > Save Output As.**

2. **Choose or create a file in which to save the output using the Save Build Output dialog box.**

The build output log is saved as a text file.

Modifying a TeamWare Target

To edit an existing TeamWare target, choose Build > Edit Target and choose a TeamWare target from the list. The Edit Target dialog box opens, displaying the current settings for the build directory, makefile, make target, and build command. Edit any of these fields, as described in "Building With Nondefault Values" on page 195. Click Build to rebuild the TeamWare target with your new settings.

Removing a TeamWare Target

You can remove targets from the TeamWare target list and the Edit Target list in the Build menu.

To remove a target:

1. **Choose Build > Remove targets from menu in the Building window.**

2. **Select one or more targets from the list in the Remove targets from menu dialog box.**

   Hold down the Control key and click to select more than one target name.

3. **Click OK.**

Customizing a Build

You can customize a build by changing make options, specifying a build mode, using makefile macros, or using environment variables. To customize a build, choose Build > Edit Target and choose a TeamWare target from the list. The Edit Target dialog box opens. After making your changes, click Build to rebuild the TeamWare target with your new settings.

Specifying Make Options

You can specify make options using the Options dialog box, shown in Figure 17–3.

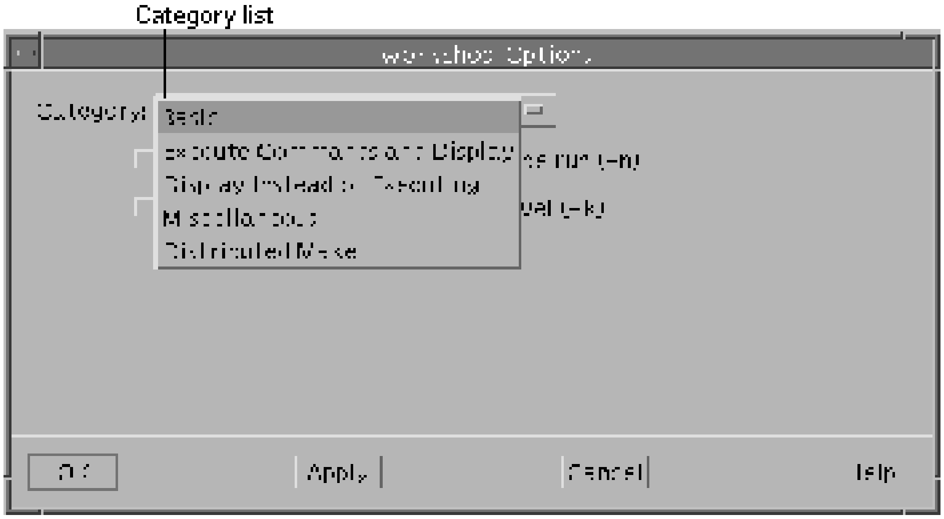


Figure 17–3 Options Dialog Box

Category list	Allows you to select a category of make options.
OK button	Applies the changes and closes the dialog box
Apply button	Applies the changes but leaves the dialog box open
Cancel button	Closes the dialog box without applying changes
Help button	Displays online help for the dialog box

Table 17–1 describes the make options you can set using the dialog box.

TABLE 17–1 Options That You Can Set in the Options Dialog Box

Category	**Option**
Basic	Echo command lines but do not execute them (`-n`)
	When an error occurs, continue with dependency branches that do not depend on the target (`-k`).

TABLE 17–1 Options That You Can Set in the Options Dialog Box *(continued)*

Category	**Option**
Execute Commands and Display	Display reasons why make chooses to rebuild a target; make displays any and all dependencies that are newer. The make displays options are also read in from the MAKEFLAGS environment variable (-d).
	Display detailed information on the dependency check and processing (-ss)
	Display the text of the makefiles read in (-D).
	Display the text of the makefiles, make.rules file, the state file, and all hidden-dependency reports (-DD).
	Silent mode. Do not echo command lines before executing them. Equivalent to the special-function target .SILENT: (-s).
Display instead of executing	Print the complete set of macro definitions and target descriptions (-p).
	Report dependencies only, do not build them (-P).
	Question mode. make returns a zero or nonzero status code depending on whether or not the target file is up to date (-q).
Miscellaneous	Touch the target files (making them appear up to date) instead of performing their rules. This procedure can be dangerous when files are maintained by more than one person. When the .KEEP_STATE: target appears in the makefile, this option updates the state file just as if the rules had been performed (-t).
	Do not use the default rules in the default makefile /usr/share/lib/make/make.rules (-r).
	Let environment variables override macro definitions within makefiles (-e).
	Ignore error codes returned by commands. Equivalent to the special-function target .IGNORE: (-i).
Distributed Make	Mode: Choose the type of make process to run: serial, parallel, or distributed (see "Specifying a Build Mode" on page 200) (-m).
	Maximum jobs: Specify the maximum numbers of jobs that are distributed to the build servers (-j).
	Runtime configuration file: Specify a runtime configuration file.

TABLE 17–1 Options That You Can Set in the Options Dialog Box *(continued)*

Category	Option
	Build server group: Specify the name of the server group to which jobs are distributed. Temporary output directory: Specify the name of the directory to which temporary output is to be written.

To specify a make option:

1. **Click Options to open the Options dialog box.**

2. **Select the options you want from the Options dialog box.**

   Click the Category list to select a category of make options. The Options dialog box gives you access to all the options to `make` and `dmake`. For information on the distributed make options, see "Specifying a Build Mode" on page 200."

3. **Click Build to apply the options and start the build.**

You can also specify make options through the Define New Target dialog box available from the New Target command in the Build menu.

Specifying a Build Mode

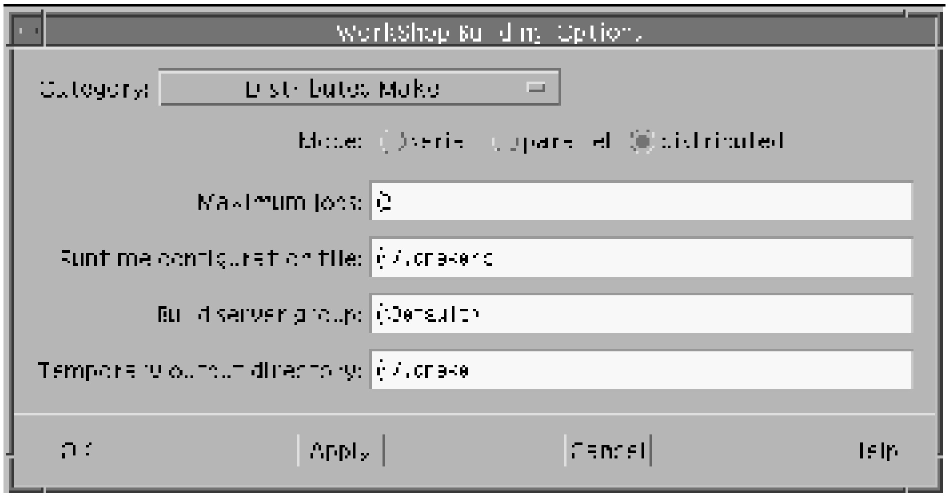
To change the default build mode of `dmake` from serial to parallel or distributed:

1. **Click Options to open the Options dialog box.**

2. **Select the Distributed Make category from the Category list.**

3. **Click the build mode you want and fill in any required text boxes in the Options dialog box, as shown in Figure 17–4.**

4. **Click Build to set the options and start the build.**



Category: Distributed Make

Mode: ○ serial ○ parallel ● distributed

Maximum jobs: 2

Runtime configuration file: ~/.dmakerc

Build server group: (Default)

Temporary output directory: ~/.dmake

Figure 17–4 Options Dialog Box for Build Mode

Serial Mode

To build in serial mode, you simply click the serial radio button. There are no text boxes for you to fill in.

Parallel Mode

To build in parallel mode, click the parallel radio button. Then specify the maximum number of build jobs to be run in the Maximum jobs text box. If you do not specify a maximum number of jobs, `dmake` uses two as the default.

Distributed Mode

To build in distributed mode, click the distributed radio button. Then specify the maximum number of build jobs to be run in the Maximum jobs text box. If you do not specify a maximum number of jobs, `dmake` uses the sum of the jobs specified for the servers in the group.

If you choose not to use the default name and location provided in the Runtime configuration file text box, type the name or path of your `.dmakerc` file (see "`.dmakerc` File" on page 212) in this text box.

If you do not type the name of a group in the Build server group text box, `dmake` uses the first group listed in the `.dmakerc` file. When running in distributed mode, `dmake` distributes jobs to the following groups in order of precedence):

1. The group specified on the command line as an argument to the `-g` option
2. The group specified by the `DMAKE_GROUP` makefile macro
3. The group specified by the `DMAKE_GROUP` environment variable
4. The first group specified in the runtime configuration file

If you do not choose to use the default name and location provided in the Temporary output directory text box, type the name of an output directory in this field.

For more information on distributed builds, see "Running a Distributed Build" on page 211.

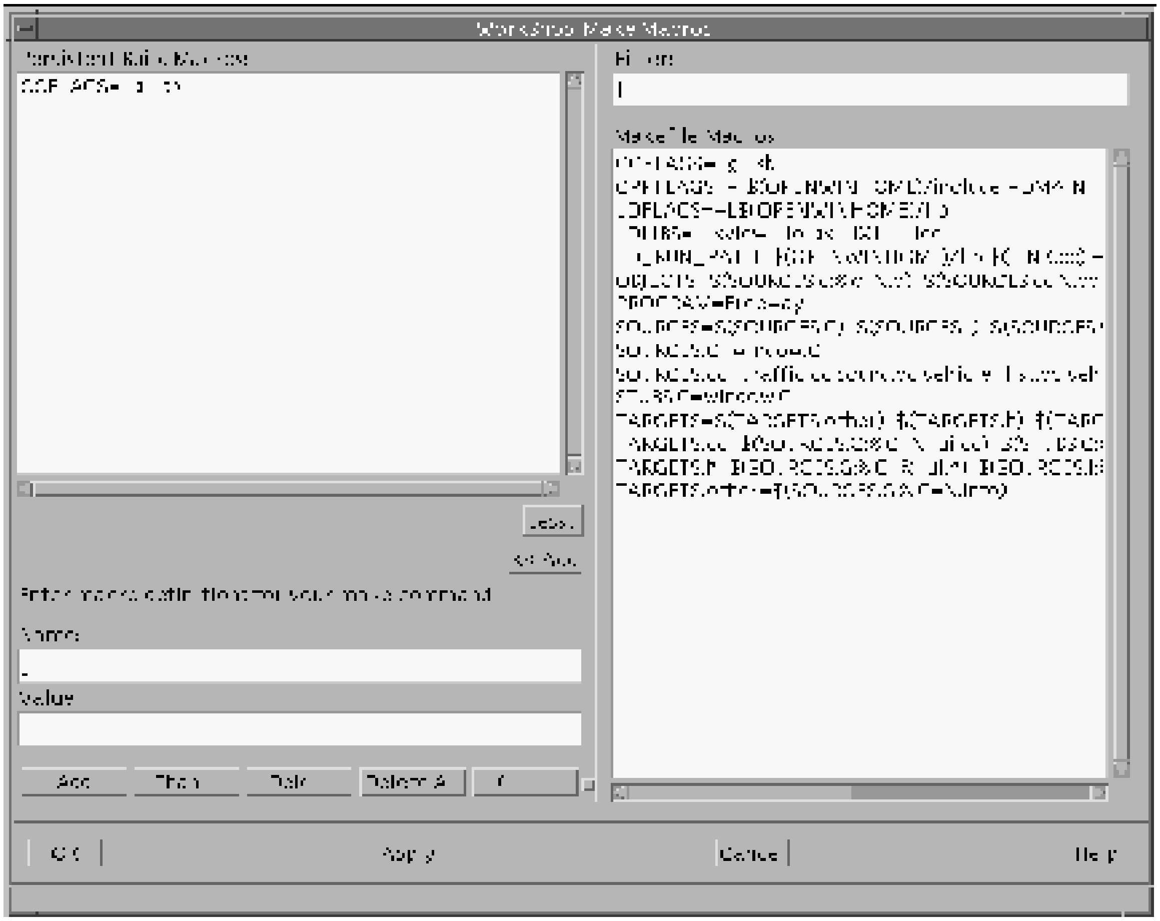
Using Makefile Macros

Makefile macros let you refer conveniently to files or command options that appear more than once in the description file. (For information on defining macros, see Appendix B in *Using Sun WorkShop*.)

Using the Make Macros dialog box (see Figure 17–5), you can add makefile macros to or delete them from the Persistent Build Macros list in your TeamWare target, and then reassign values for makefile macros in the list. You can also add macros currently defined in the makefile to the list and override their values.

All macros in the Persistent Build Macros list are saved with your WorkSet.

To open the Make Macros dialog box, click Macros in the Edit Target dialog box.



```
CCFLAGS=-g -sb
```

Enter macro definition for your make command

```
CCFLAGS=-g -sb
CPPFLAGS = -I$(OPENWINHOME)/include -DMAIN
LDFLAGS=-L$(OPENWINHOME)/lib
LDLIBS= -lxview -lolgx -lX11 -lce
LD_RUN_PATH=$(OPENWINHOME)/lib:$(CONDS)
OBJECTS= $(SOURCES.c:%.c=%.o) $(SOURCES.cc:%.cc
PROGRAM=Freeway
SOURCES=$(SOURCES.c) $(SOURCES.h) $(SOURCES.
SOURCES.c = freeway.c
SOURCES.cc = traffic.cc sports.cc vehicle.cc truck.cc
STUBS.G=window.G
TARGETS=$(TARGETS.other) $(TARGETS.h) $(TARG
TARGETS.cc= $(SOURCES.G:%.G=%_stubs.cc) $(SOURCES.G:%
TARGETS.h= $(SOURCES.G:%.G=%_ui.h) $(SOURCES.G:%
TARGETS.other=$(SOURCES.G:%.G=%.info)
```

Figure 17–5 Make Macros Dialog Box

Persistent Build Macros pane	Lists macros that will be saved with your WorkSet
More/Less button	More opens the Filter text box and Makefile Macros list pane; the button toggles to Less, which closes the pane
<<Add button	Lets you add a macro in the Makefile Macros list to the Persistent Build Macros list
Name text box	Lets you assign a name to a new macro or change the name of the selected macro on the Persistent Build Macros list
Value text box	Lets you assign a value to the macro named in the Name text box
Add button	Adds the macro defined in the Name and Value text boxes to the Persistent Build Macros list

Change button	Applies the values in the Name and Value text boxes to the selected macro on the Persistent Build Macros list
Delete button	Deletes the selected macro on the Persistent Build Macros list
Delete All button	Deletes all macros on the Persistent Build Macros list
Clear button	Clears the Name and Value text boxes
Filter text box	Lets you type a search pattern to filter the Makefile Macros list
Makefile Macros list	Lists the macros defined in the makefile in the current WorkSet
OK button	Applies the changes and closes the dialog box
Apply button	Applies the changes but leaves the dialog box open
Cancel button	Closes the dialog box without applying changes
Help button	Displays online help for the dialog box

Adding a Macro

To add a macro to the Persistent Build Macros list:

1. **Type the name of a macro in the Name text box.**

2. **Type a value for the macro in the Value text box.**

   If you make a mistake, click Clear to remove entries in the Name and Value text boxes.

3. **Click Add to add the new macro to the list.**

4. **Repeat the previous three steps to add other macros.**

5. **Click OK to close the dialog box.**

Deleting a Macro

To delete a macro from the Persistent Build Macros list:

1. **Select a macro in the list.**

2. **Click Delete (Delete All removes all macros in the list).**

3. **Click OK to establish the change and close the dialog box.**

Changing a Macro

To change the value of a macro in the Persistent Build Macros list:

1. **Select a macro in the list.**

2. **Type a new value in the Value text box and click Change.**

3. **Click OK to establish the change and close the dialog box.**

4. **Click Build in the Edit Target dialog box to start the build with the new values.**

Reviewing and Overriding Makefile Macros

A macro definition that appears in the Persistent Build Macros list overrides any macro with the same name that appears in the makefile.

To review the current macro definitions, click More to open the Makefile Macros list, which displays all the macros that are defined in the makefile associated with the build target. You can filter the list using the Filter text box.

To override the value of a makefile macro:

1. **Select a macro in the Makefile Macros list.**

2. **Click <<Add to add the macro to the Persistent Build Macros list.**

3. **Type a new value in the Value text box and click Change.**

4. **Click OK to establish the change and close the dialog box.**

   The macro definition in the Persistent Build Macros list overrides the macro definition in the makefile, and is saved with your WorkSet.

5. **Click Build in the Edit Target dialog box to start the build with the new values.**

Using Environment Variables

You can specify environment variables for your build. When you start the build, `setenv` commands for these environment variables are prepended to the build command.

Using the Environment Variables dialog box, you can add environment variables to or delete them from the Persistent Environment Variables list in your TeamWare target, and reassign values for environment variables in the list.

All macros in the Persistent Environment Variables list are saved with your WorkSet.

Note - The Persistent Environment Variables list for building your program is not the same as the Persistent Environment Variables list for running your program described in Chapter 5 of *Using Sun WorkShop.*

To open the Environment Variables dialog box, click Environment Variables in the Edit Target dialog box.

Adding an Environment Variable

To add an environment variable to the Persistent Environment Variables list:

1. **Type the name of an environment variable in the Name text box.**

2. **Type a value for the variable in the Value text box.**

   If you make a mistake, click Clear to remove entries in the Name and Value text boxes.

3. **Click Add to add the environment variable to the Persistent Environment Variables list.**

4. **Repeat the previous three steps to add other environment variables.**

5. **Click OK to close the dialog box.**

Deleting an Environment Variable

To delete a variable from the Persistent Environment Variables list:

1. **Select a variable from the list.**

2. **Click Delete (Delete All removes all environment variables in the list).**

3. **Click OK to establish the change and close the dialog box.**

Changing the Value of an Environment Variable

To change the value of an environment variable in the Persistent Environment Variables list:

1. **Select an environment variable in the list.**

2. **Type a new value in the Value text box and click Change.**

3. **Click OK to establish the change and close the dialog box.**

4. **Click Build to start the build with the new build environment.**

Reviewing and Overriding Environment Variables

An environment variable definition that appears in the Persistent Environment Variables list overrides any environment variable with the same name that appears in the current TeamWare process environment.

To review the current TeamWare process environment variable definitions, click More to open the Current Environment list, which includes all the environment variables that are currently defined in the TeamWare process environment. You can filter the list using the Filter text box.

To override the value of an environment variable:

1. **Select an environment variable in the Current Environment list.**

2. **Click <<Add to add the environment variable to the Persistent Environment Variables list.**

3. **Type a new value in the Value text box and click Change.**

4. **Click OK to establish the change and close the dialog box.**

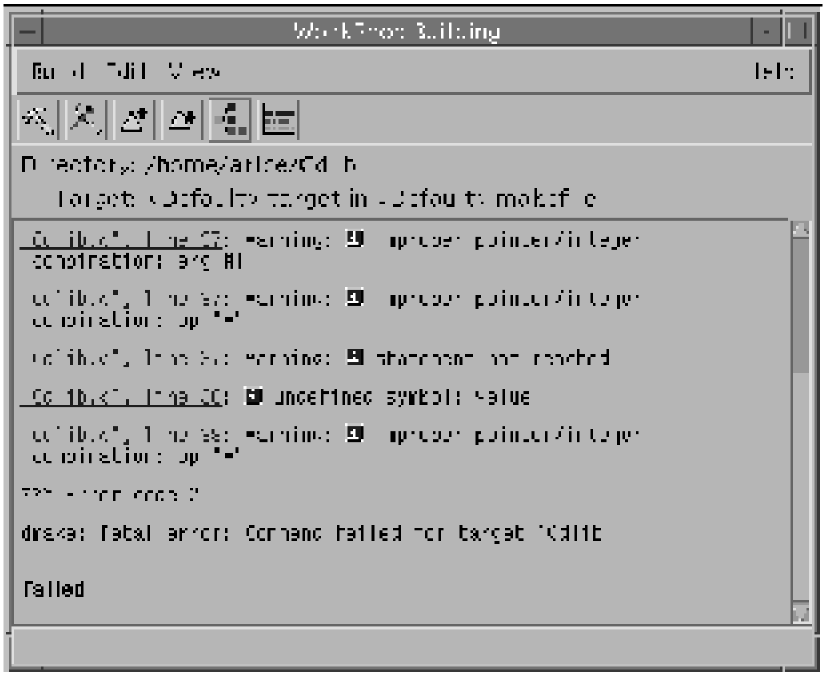
   The environment variable definition in the Persistent Environment Variables list overrides the environment variable definition in the current TeamWare process environment, and is saved with your WorkSet.

5. **Click Build in the Edit Target dialog box to start the build with the new values.**

Fixing Build Errors

The process of fixing build errors is simplified by the integration of the text editor with the build process. When a build fails, the build errors are displayed in the Build Output display pane of the Building window, as shown in Figure 17–6. Build errors that have links to the source files containing the errors are highlighted and underscored. Unrecognized errors are displayed with the rest of the build output without highlighting or underscoring.

Note - Do not run a build job and a fix (recompilation of edited source files) concurrently. The output for both jobs intermingles in the Build Output display pane in the Building window. It can be difficult to discern the output of one job from the other.



```
Directory: /home/arlice/Cdlib
   Target: <Default> target in Default makefile

"Cdlib.c", line [illegible]: warning: i  improper pointer/integer
combination: arg #1

"Cdlib.c", line [illegible]: warning: i  improper pointer/integer
combination: op "="

"Cdlib.c", line [illegible]: warning: i statement not reached

"Cdlib.c", line [illegible]: i undefined symbol: value

"Cdlib.c", line [illegible]: warning: i  improper pointer/integer
combination: op "="

*** Error code 2

dmake: Fatal error: Command failed for target `Cdlib`

Failed
```

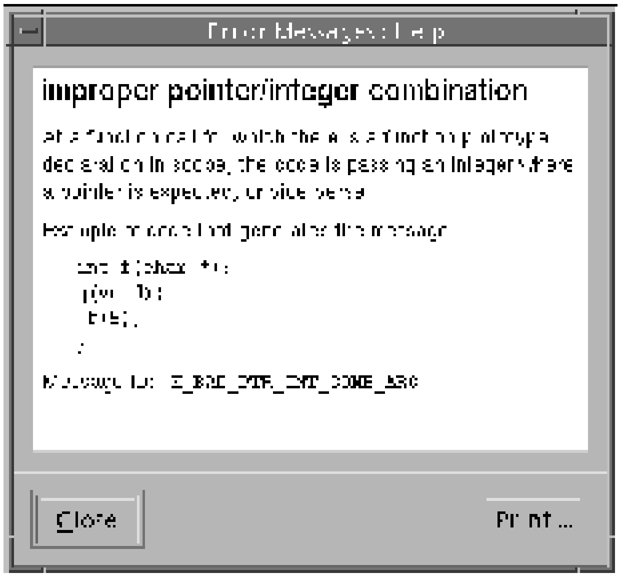
Figure 17–6 Build Errors in the Build Output Display Pane

Each error line gives the name of the file containing the error, the line number on which the error occurs, and the error message.

Error messages issued by the C compiler include an additional glyph (

i

) in the build error message. Clicking on the glyph opens a pop-up window that defines the associated error message.



improper pointer/integer combination

At a function call for which there is a function prototype declaration in scope, the code is passing an integer where a pointer is expected, or vice versa.

Example of code that generates this message:

```
int f(char *);
main(void) {
  f(5);
}
```

Message ID: E_BAD_PTR_INT_COMB_ARG

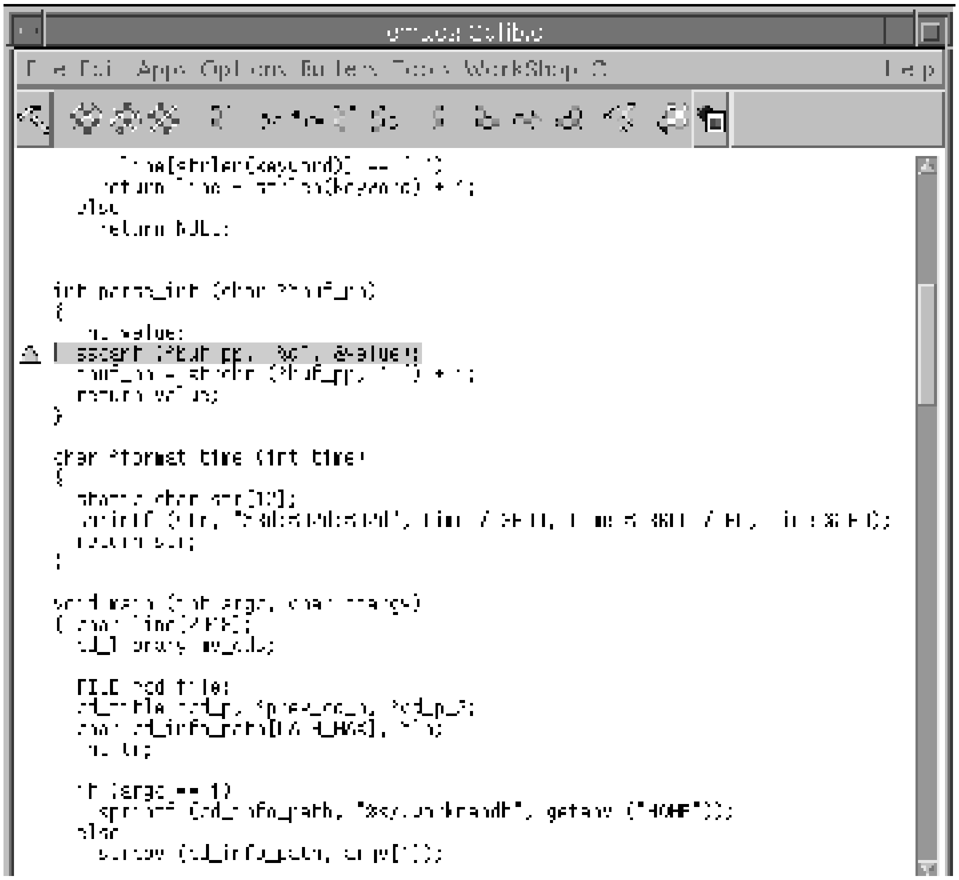
Figure 17–7 Error Message Pop-up Window

Note - Only Sun compilers produce output that can be converted to hypertext links. If you use a build command that does not call Sun compilers, you will not have links to the source files from the build errors listed in the Building window.

Displaying the Source of an Error

Clicking on the underscored error immediately starts a text editor that displays the source file containing the error. The source file is shown with the error line highlighted and an error glyph appears to the left of the line (see Figure 17–8).

Note - By using the keyboard shortcuts F4 (next error) and Shift+F4 (previous error) to navigate through the build errors, you can keep focus on the text editor window.



```
      line[strlen(keyword)] == ' ')
    return line + strlen(keyword) + 1;
  else
    return NULL;


int parse_int (char **buf_pp)
{
  int value;
  sscanf (*buf_pp, "%d", &value);
  *buf_pp = strchr (*buf_pp, ' ') + 1;
  return value;
}

char *format_time (int time)
{
  static char str[10];
  sprintf (str, "%d:%02d:%02d", time / 3600, time % 3600 / 60, time % 60);
  return str;
}

void main (int argc, char **argv)
{ char line[256];
  sd_library my_sds;

  FILE *sd_file;
  sd_profile *sd_p, *prev_sd_p, *sd_p2;
  char sd_info_path[MAXPATHLEN], *s;
  int i;

  if (argc == 1)
    sprintf (sd_info_path, "%s/[illegible]", getenv ("HOME"));
  else
    strcpy (sd_info_path, argv[1]);
```

Figure 17–8 Text Editor Window Displaying Source File With Error

Fixing an Error

The following steps show how you can use the Building window and the text editor to quickly fix build errors:

1. **Click a highlighted error in the Build Output display pane.**

   The editor window opens, displaying the source file containing the error. You do not have to search for the line containing the error—the error line is highlighted in the editor and the cursor is already positioned at the line. The error message is repeated in the footer of the text editor.

2. **In the text editor, make sure the source file can be edited.**

   If the file is under SCCS control, check it out using the appropriate menu commands in the text editor:

   - In the vi editor, choose Version > Checkout.
   - In the XEmacs editor, choose Tools > VC > Check out File *file*.
   - In the GNU Emacs editor, choose Tools > Version Control > Check Out.

3. **Edit the source file containing the error.**

4. **In the Building window, click the Next Error button in the tool bar (or use the keyboard shortcut F4) to go to the location of the next build error in the text editor.**

   As you click Next Error, notice how each successive error in the build output is highlighted and how the corresponding source line in the text editor is also highlighted.

5. **Save the edited file.**

   - In the vi editor, choose File > Save.
   - In the XEmacs editor, choose File > Save *file*.
   - In the GNU Emacs editor, choose Files > Save buffer.

6. **If the file is under SCCS control, check it in using the appropriate menu commands in the text editor:**

   - In the vi editor, choose Version > Checkin.
   - In the XEmacs editor, choose Tools > VC > Check in File *file*.
   - In the GNU Emacs editor, choose Tools > Version Control > Check In.

7. **Click the Build button in the text editor's tool bar to rebuild.**

   You can also build by clicking on the Build button in the Building window's tool bar or using the keyboard shortcut F3.

   You can watch the Build Output display pane to follow the progress of the build.

Running a Distributed Build

Distributed make (`dmake`) allows you to concurrently distribute the process of building large projects, consisting of many programs, over a number of workstations and, in the case of multiprocessor systems, over multiple CPUs. For a full description of distributed make, see "Using the `dmake` Utility" on page 226Chapter 18."

The default Sun TeamWare build command (dmake) provides three different build modes:

- *Serial mode* - dmake executes one job at a time on the local host (similar to make)
- *Parallel mode* - dmake executes multiple jobs concurrently on the local host
- *Distributed mode* - dmake executes multiple jobs over several build servers

In distributed mode, you can concurrently distribute over several servers the process of building large projects that consist of many programs. dmake parses your makefiles, determines which targets can be built concurrently, and distributes the build for those targets over a number of build servers designated by you.

By default, dmake runs in serial mode. You can set it to run in parallel or distributed mode in the Options dialog box (see “Specifying Make Options” on page 197).

Preparing for a Distributed Build

Before running a distributed build for the first time, you must create a configuration file that specifies which machines are to participate as dmake build servers. In addition, before a machine can be used as a build server, it must be configured to allows jobs to be distributed to it.

A build server should be of the same architecture and running the same operating system version as the dmake host. Be default, it is assumed that the path to the dmake executables is the same for the dmake host as it is for the build server. If it is not, you must customize the path attribute for that server (for further details see the dmake(1) man page).

.dmakerc File

The .dmakerc file is a runtime configuration file. You must set up a runtime configuration file to run a distributed build. The file contains groups (lists) of build servers and the number of jobs distributed to each build server. The dmake utility searches for this file on the dmake host to know where to distribute jobs. Generally, this file is in your home directory.

You may enclose the names of groups and hosts in the .dmakerc file in double quotes. Doing so allows more flexibility with respect to the character sequences that may be part of the group and host names. For example, if the name of a group starts with a digit it should be double-quoted:

```
group "123_sparc"
```

The dmake utility searches for a runtime configuration file in the following locations and in the following order:

- The path name specified on the command line using the -c option

- The path name specified with the `DMAKE_RCFILE` makefile macro
- The path name specified with the `DMAKE_RCFILE` environment variable
- `$(HOME)&/.dmakerc`

If `dmake` does not find a runtime configuration file, it distributes two jobs to the local host.

For information on setting up a runtime configuration file, see the `dmake` man page.

The following is a sample of a simple runtime configuration file where `jupiter`, `venus`, `saturn`, `mercury`, and `pluto` are listed as build servers:

```
# My machine. This entry causes dmake to distribute to it.
jupiter { jobs = 1 }
venus
# Manager"s machine. She"s usually at meetings.
mercury { jobs = 4 }
pluto
```

The following runtime configuration file contains groups:

```
earth { jobs = 2 }
mars { jobs = 3 }
group sunos4.x {
host parasol
host summer
}
group lab1 {
host falcon { jobs = 3 }
host hawk
host eagle { jobs = 3 }
}
group lab2 {
host heron
host avocet { jobs = 3 }
host stilt { jobs = 2 }
}
group labs {
group lab1
group lab2
}
group sunos5.x
    group labs
host jupiter
host venus { jobs = 2 }
host pluto { jobs = 3 }
}
```

`dmake.conf` File

To set up a machine to be used as a build server, you must create a configuration file called `/etc/opt/SPROdmake/dmake.conf` file on the server's file system. Without this file, `dmake` refuses to distribute jobs to that machine.

In the `dmake.conf` file, you specify the maximum number of jobs (from all users) that can run concurrently on that build server. In addition, you may specify the “nice” priority under which all dmake jobs should run. The following is an example of a `dmake.conf` file:

```
max_jobs: 8
nice_prio: 5
```

Examining Multiple Build Jobs

If you are running distributed make (`dmake` in any mode), you can use the Jobs Graph window to monitor the progress of the `dmake` run and to view the state of each build job.

The graph identifies each build server. Build jobs are graphed in clusters per server. The graph shows the length of time each build takes. Each job is indicated in the graph by a line. The appearance of the line indicates whether the build is in progress (series of dots), or whether it completed (solid green), or failed (solid red).

To open the Jobs Graph window from the Building window, click the Jobs Graph button (see Figure 17–1) or choose View > Dmake Jobs Graph.

You can select a segment of one of the jobs in the graph to see its build output in the Selected Job Output display at the bottom of the window.

Exiting Building

To kill the current build process and close all build windows, choose Build > Exit Building in the Building window.

If you want to close the building windows without killing the current build process, choose Build > Close.

CHAPTER 18

Using the dmake Utility

This appendix describes the way the distributed make (dmake) utility distributes builds over several hosts to build programs concurrently over a number of workstations or multiple CPUs.

Basic Concepts

Distributed make (dmake) allows you to concurrently distribute the process of building large projects, consisting of many programs, over a number of workstations and, in the case of multiprocessor systems, over multiple CPUs. The dmake utility parses your makefiles and:

- Determines which targets can be built concurrently
- Distributes the build of those targets over a number of hosts designated by you

The dmake utility is a superset of the make utility.

To understand dmake, you should know about:

- Configuration files (runtime and build server)
- The dmake host
- The build server

Configuration Files

The dmake utility consults two files to determine to which build servers jobs are distributed and how many jobs can be distributed to each.

Runtime Configuration File

The dmake utility searches for a runtime configuration file on the dmake host to know where to distribute jobs. Generally, this file is in your home directory on the dmake host and is named .dmakerc. It consists of a list of build servers and the number of jobs to be distributed to each build server. See "The dmake Host" on page 218 for more information.

Build Server Configuration File

The /etc/opt/SPROdmake/dmake.conf file is in the file system of build servers. It specifies the maximum total number of dmake jobs that can be distributed to each build server by all dmake users. In addition, it may specify the "nice" priority under which all dmake jobs should run.

See "The Build Server" on page 220 for more information.

The dmake Host

The dmake utility searches for a runtime configuration file to determine where to distribute jobs. Generally, this file must be in your home directory on the dmake host and is named .dmakerc. The dmake utility searches for the runtime configuration file in these locations and in the following order:

1. The path name you specify on the command line using the -c option
2. The path name you specify using the DMAKE_RCFILE makefile macro
3. The path name you specify using the DMAKE_RCFILE environment variable
4. $(HOME)/.dmakerc

If a runtime configuration file is not found, the dmake utility distributes two jobs to the dmake host.

You edit the runtime configuration file so that it consists of a list of build servers and the number of jobs you want distributed to each build server. The following is an example of a .dmakerc file:

```
# My machine. This entry causes dmake to distribute to it.
falcon   { jobs = 1 }
hawk
eagle   { jobs = 3 }
# Manager's machine. She's usually at meetings
heron   { jobs = 4 }
```

```
avocet
```

- The entries: `falcon`, `hawk`, `eagle`, `heron`, and `avocet` are listed build servers.
- You can specify the number of jobs you want distributed to each build server. The default number of jobs is two.
- Any line that begins with the # character is interpreted as a comment.

Note - This list of build servers includes `falcon` which is also the dmake host. The dmake host can also be specified as a build server. If you do not include it in the runtime configuration file, no dmake jobs are distributed to it.

You can also construct groups of build servers in the runtime configuration file. This provides you with the flexibility of easily switching between different groups of build servers as circumstances warrant. For instance, you may define groups of build servers for builds under different operating systems, or you may define groups of build servers that have special software installed on them.

The following is an example of a runtime configuration file that contains groups of build servers:

```
earth   { jobs = 2 }
mars   { jobs = 3 }

group lab1 {
   host falcon { jobs = 3 }
   host hawk
   host eagle     { jobs = 3 }
}

group lab2 {
   host heron
   host avocet     { jobs = 3 }
   host stilt     { jobs = 2 }
}

group labs {
   group lab1
   group lab2
}

group sunos5.x {
   group labs
   host jupiter
   host venus { jobs = 2 }
   host pluto  { jobs = 3 }
}
```

- Formal groups are specified by the `group` directive and lists of their members are delimited by braces ({}).

- Build servers that are members of groups are specified by the optional host directive.
- Groups can be members of other groups.
- Individual build servers can be listed in runtime configuration files that also contain groups of build servers; in this case, dmake treats these build servers as members of the *unnamed* group.

In order of precedence, the dmake utility distributes jobs to the following:

1. The formal group specified on the command-line as an argument to the -g option
2. The formal group specified by the DMAKE_GROUP makefile macro
3. The formal group specified by the DMAKE_GROUP environment variable
4. The first group specified in the runtime configuration file

The dmake utility allows you to specify a different execution path for each build server. By default dmake looks for the dmake support binaries on the build server in the same logical path as on the dmake host. You can specify alternate paths for build servers as a host attribute in the .dmakerc file. For example:

```
group lab1 {
   host falcon { jobs = 10 , path = "/set/dist/sparc-S2/bin" }
   host hawk { path = "/opt/SUNWspro/bin"                    }
}
```

You can use double quotation marks to enclose the names of groups and hosts in the .dmakerc file. This allows you more flexibility in the characters that you can use in group names. Digits are allowed, as well as alphabetic characters. Names that start with digits should be enclosed in double quotes. For example:

```
group "123_lab" {
   host "456_hawk" { path = "/opt/SUNWspro/bin"                 }
}
```

The Build Server

The /etc/opt/SPROdmake/dmake.conf file is in the file system of build servers. Use this file to limit the maximum number of dmake jobs (from all users) that can run concurrently on a build server and to specify the “nice” priority under which all dmake jobs should run. The following is an example of an /etc/opt/SPROdmake/dmake.conf file. This file sets the maximum number of dmake jobs permitted to run on a build server (from all dmake users) to be eight.

```
max_jobs: 8
nice_prio: 5
```

Note - If the `/etc/opt/SPROdmake/dmake.conf` file does not exist on a build server, no dmake jobs will be allowed to run on that server.

Understanding the `dmake` Utility

To run a distributed make, use the executable file `dmake` in place of the standard `make` utility. You should understand the Solaris `make` utility before you use `dmake`. If you need to read more about the `make` utility, see the Solaris *Programming Utilities Guide*. If you use the `make` utility, the transition to `dmake` requires little if any alteration.

Impact of the `dmake` Utility on Makefiles

The methods and examples shown in this section present the kinds of problems that lend themselves to solution with `dmake`. This section does not suggest that any one approach or example is the best. Compromises between clarity and functionality were made in many of the examples.

As procedures become more complicated, so do the makefiles that implement them. You must know which approach will yield a reasonable makefile that works. The examples in this section illustrate common code-development predicaments and some straightforward methods to simplify them using `dmake`.

Using Makefile Templates

If you use a makefile template from the outset of your project, custom makefiles that evolve from the makefile templates will be:

- More familiar
- Easier to understand
- Easier to integrate
- Easier to maintain
- Easier to reuse

The less time you spend editing makefiles, the more time you have to develop your program or project.

Building Targets Concurrently

Large software projects typically consist of multiple independent modules that can be built concurrently. The dmake utility supports concurrent processing of targets on multiple machines over a network. This concurrency can markedly reduce the time required to build a large project.

When given a target to build, dmake checks the dependencies associated with that target, and builds those that are out of date. Building those dependencies may, in turn, entail building some of their dependencies. When distributing jobs, dmake starts every target that it can. As these targets complete, dmake starts other targets. Nested invocations of dmake are not run concurrently by default, but this can be changed (see "Restricting Parallelism " on page 225 for more information).

Since dmake builds multiple targets concurrently, the output of each build is produced simultaneously. To avoid intermixing the output of various commands, dmake collects output from each build separately. The dmake utility displays the commands before they are executed. If an executed command generates any output, warnings, or errors, dmake displays the entire output for that command. Since commands started later may finish earlier, this output may be displayed in an unexpected order.

Limitations on Makefiles

Concurrent building of multiple targets places some restrictions on makefiles. Makefiles that depend on the implicit ordering of dependencies may fail when built concurrently. Targets in makefiles that modify the same files may fail if those files are modified concurrently by two different targets. Some examples of possible problems are discussed in this section.

Dependency Lists

When building targets concurrently, it is important that dependency lists be accurate. For example, if two executables use the same object file but only one specifies the dependency, then the build may cause errors when done concurrently. For example, consider the following makefile fragment:

```
all: prog1 prog2
prog1: prog1.o aux.o
 $(LINK.c) prog1.o aux.o -o prog1
prog2: prog2.o
 $(LINK.c) prog2.o aux.o -o prog2
```

When built serially, the target aux.o is built as a dependent of prog1 and is up-to-date for the build of prog2. If built in parallel, the link of prog2 may begin before aux.o is built, and is therefore incorrect. The .KEEP_STATE feature of make detects some dependencies, but not the one shown above.

Explicit Ordering of Dependency Lists

Other examples of implicit ordering dependencies are more difficult to fix. For example, if all of the headers for a system must be constructed before anything else is built, then everything must be dependent on this construction. This causes the makefile to be more complex and increases the potential for error when new targets are added to the makefile. The user can specify the special target .WAIT in a makefile to indicate this implicit ordering of dependents. When dmake encounters the .WAIT target in a dependency list, it finishes processing all prior dependents before proceeding with the following dependents. More than one .WAIT target can be used in a dependency list. The following example shows how to use .WAIT to indicate that the headers must be constructed before anything else.

```
all: hdrs .WAIT libs functions
```

You can add an empty rule for the .WAIT target to the makefile so that the makefile is backward-compatible.

Concurrent File Modification

You must make sure that targets built concurrently do not attempt to modify the same files at the same time. This can happen in a variety of ways. If a new suffix rule is defined that must use a temporary file, the temporary file name must be different for each target. You can accomplish this by using the dynamic macros $@ or $*. For example, a .c.o rule that performs some modification of the .c file before compiling it might be defined as:

```
.c.o:
 awk -f modify.awk $*.c > $*.mod.c
 $(COMPILE.c) $*.mod.c -o $*.o
 $(RM) $*.mod.c
```

Concurrent Library Update

Another potential concurrency problem is the default rule for creating libraries that also modifies a fixed file, that is, the library. The inappropriate .c.a rule causes dmake to build each object file and then archive that object file. When dmake archives two object files in parallel, the concurrent updates will corrupt the archive file.

```
.c.a:
 $(COMPILE.c) -o $% $<
 $(AR) $(ARFLAGS) $@ $%
 $(RM) $%
```

A better method is to build each object file and then archive all the object files after completion of the builds. An appropriate suffix rule and the corresponding library rule are:

```
.c.a:
 $(COMPILE.c) -o $% $<
 $(COMPILE.c) -o $% $<
lib.a: lib.a($(OBJECTS))
 $(AR) $(ARFLAGS) $(OBJECTS)
 $(RM) $(OBJECTS)
```

Multiple Targets

Another form of concurrent file update occurs when the same rule is defined for multiple targets. An example is a yacc(1) program that builds both a program and a header for use with lex(1). When a rule builds several target files, it is important to specify them as a group using the + notation. This is especially so in the case of a parallel build.

```
y.tab.c y.tab.h: parser.y
 $(YACC.y) parser.y
```

This rule is actually equivalent to the two rules:

```
y.tab.c: parser.y
 $(YACC.y) parser.y
y.tab.h: parser.y
 $(YACC.y) parser.y
```

The serial version of make builds the first rule to produce y.tab.c and then determines that y.tab.h is up-to-date and need not be built. When building in parallel, dmake checks y.tab.h before yacc has finished building y.tab.c and notices that y.tab.h *does* need to be built, it then starts another yacc in parallel with the first one. Since both yacc invocations are writing to the same files (y.tab.c and y.tab.h), these files are apt to be corrupted and incorrect. The correct rule uses the + construct to indicate that both targets are built simultaneously by the same rule. For example:

```
y.tab.c + y.tab.h: parser.y
 $(YACC.y) parser.y
```

Restricting Parallelism

Sometimes file collisions cannot be avoided in a makefile. An example is xstr(1), which extracts strings from a C program to implement shared strings. The xstr command writes the modified C program to the fixed file x.c and appends the strings to the fixed file strings. Since xstr must be run over each C file, the following new .c.o rule is commonly defined:

```
.c.o:
 $(CC) $(CPPFLAGS) -E $*.c | xstr -c -
 $(CC) $(CFLAGS) $(TARGET_ARCH) -c x.c
 mv x.o $*.o
```

The dmake utility cannot concurrently build targets using this rule since the build of each target writes to the same x.c and strings files. Nor is it possible to change the files used. You can use the special target .NO_PARALLEL: to tell dmake not to build these targets concurrently. For example, if the objects being built using the .c.o rule were defined by the OBJECTS macro, the following entry would force dmake to build those targets serially:

```
.NO_PARALLEL: $(OBJECTS)
```

If most of the objects must be built serially, it is easier and safer to force all objects to default to serial processing by including the .NO_PARALLEL: target without any dependents. Any targets that can be built in parallel can be listed as dependencies of the .PARALLEL: target:

```
.NO_PARALLEL:
.PARALLEL: $(LIB_OBJECT)
```

Nested Invocations of Distributed Make

When dmake encounters a target that invokes another dmake command, it builds that target serially, rather than concurrently. This prevents problems where two different dmake invocations attempt to build the same targets in the same directory. Such a problem might occur when two different programs are built concurrently, and each must access the same library. The only way for each dmake invocation to be sure that the library is up-to-date is for each to invoke dmake recursively to build that library. The dmake utility recognizes a nested invocation only when the $(MAKE) macro is used in the command line.

If you nest commands that you know will not collide, you can force them to be done in parallel by using the .PARALLEL: construct.

When a makefile contains many nested commands that run concurrently, the load-balancing algorithm may force too many builds to be assigned to the local machine. This may cause high loads and possibly other problems, such as running out of swap space. If such problems occur, allow the nested commands to run serially.

Using the dmake Utility

You execute dmake on a *dmake host* and distribute jobs to *build servers*. You can also distribute jobs to the dmake host, in which case it is also considered to be a build server. The dmake utility distributes jobs based on makefile targets that dmake determines (based on your makefiles) can be built concurrently. You can use a machine as a build server if it meets the following requirements:

- From the dmake host (the machine you are using) you must be able to use rsh, without being prompted for a password, to remotely execute commands on the build server. See man rsh(1) or the system AnswerBook documentation for more information about the rsh command. For example:

```
demo% rsh build_server which dmake
   /opt/SUNWspro/bin/dmake
```

- The bin directory in which the dmake software is installed must be accessible from the build server. See the share(1M) and mount(1M) man pages or the system AnswerBook documentation for more information about creating shared filesystems.
- By default, dmake assumes that the logical path to the dmake executables on the build server is the same as on the dmake host. You can override this assumption by specifying a path name as an attribute of the host entry in the runtime configuration file. For example:

```
group sparc-cluster {
     host wren   { jobs = 10 , path = ‘‘/export/SUNWspro/bin’’}
     host stimpy { path = ‘‘/opt/SUNWspro/bin’’            }
```

- The source hierarchy you are building must be:
  - Accessible from the build server
  - Mounted under the same name

From the dmake host you can control which build servers are used and how many dmake jobs are allotted to each build server. The number of dmake jobs that can run on a given build server can also be limited on that server.

Note - If you specify the `-m` option with the `parallel` argument, or set the `DMAKE_MODE` variable or macro to the value `parallel`, `dmake` does not scan your runtime configuration file. Therefore, you must specify the number of jobs using the `-j` option or the `DMAKE_MAX_JOBS` variable or macro. If you do not specify a value this way, a default of two jobs is used.

Note - If you modify the maximum number of jobs using the `-j` option, or the `DMAKE_MAX_JOBS` variable or macro when using `dmake` in distributed mode, the value you specify overrides the values listed in the runtime configuration file. The value you specify is used as the total number of jobs that can be distributed to all build servers.

Controlling `dmake` Jobs

The distribution of `dmake` jobs is controlled in two ways:

1. A dmake user on a dmake host can specify the machines to use as build servers and the number of jobs to distribute to each build server.
2. The “owner” on a build server can control the maximum total number of dmake jobs that can be distributed to that build server. The owner is a user who can alter the `/etc/opt/SPROdmake/dmake.conf` file.

Note - If you access `dmake` from the GUI (Building window), use the online help to know how to specify your build servers and jobs. If you access `dmake` from the command line, see the `dmake` man page (`dmake.1`).

CHAPTER **19**

Error and Warning Messages

This chapter describes the error and warning messages displayed by TeamWare Configuring.

The chapter contains the following sections:

- "Error Messages" on page 229
- "Warnings Messages" on page 250

All Configuring messages are numbered and are listed in numerical order. For each message, the meaning of the message and a possible remedy for the error are provided.

Error Messages

Table 19–1 describes error messages, their meanings, and possible remedies.

TABLE 19–1 Configuring Error Messages

1000 - 1999	System Errors Error messages between 1000 and 1999 report errors from operating system calls made by Configuring commands. They consist of a short Configuring message and an appended system error message and number. Refer to operating system documentation for information regarding these errors.
2000	`Line too long or unexpected end of file in` *file_name*

	Meaning: While reading the *file_name*, a line was encountered that contained too many characters for a Configuring command to buffer. The maximum line length is 1024 characters. Remedy: Reduce the size of the long line and re-execute the command.
2001	`Must specify a [child]* workspace either with the -w option or via the CODEMGR_WS environment variable` Meaning: The Configuring command could not determine the workspace on which to act. Configuring commands attempt to acquire the workspace path name in the following order: ■ As specified by the command's -w option ■ As specified by the value of the environment variable `CODEMGR_WS` ■ The current directory, if it is hierarchically within a workspace *When the error is reported by Bringover and Putback the word child is included, when reported by Undo and Resolve it is not included. Remedy: Specify the workspace path name using one of the methods listed above.
2002	`Cannot use the -p option to reparent the child of an NSE environment` Meaning: You cannot use the –`-p` option with the CLI `bringover` and `putback` commands to reparent a workspace that has an NSE environment as a parent. Remedy: Use the workspace reparent command to reparent a workspace whose parent is an NSE environment to a Configuring workspace. You cannot reparent such a workspace to another NSE environment.
2003	*directory_name* `is not a workspace` Meaning: The directory specified in the command is not a Configuring workspace. Configuring workspaces are distinguished by the presence of the `Codemgr_wsdata` directory in the top level directory. Remedy: Specify a different workspace name or use the CLI `workspace create` command or the GUI File Create Workspace command to convert the directory into a workspace.
2004	`Workspace` *workspace_name* `doesn't have a parent workspace`

TABLE 19–1 Configuring Error Messages *(continued)*

	Meaning: A Configuring command (Bringover or Putback) could not complete execution because a parent workspace could not be found for workspace workspace_name. Remedy: Use the CLI `workspace parent` command or the GUI Edit Parent menu item to reparent the orphaned workspace.
2005	`Parent workspace` *workspace_name* `is not visible as it is not mounted on` *machine_name* Meaning: The file system that contains the parent workspace is not currently mounted on machine machine_name. Remedy: Mount the file system that contains the parent workspace and re-issue the command.
2006	`Filename` *file_name* `has too many ".." path components in it` Meaning: Relative file names specified to Configuring commands are interpreted as being relative to the root directory of the workspace. If a file name contains “`..`” components, it is possible for one of the “`..`” components to reach a directory that is hierarchically above the workspace root. Remedy: Specify the path name with fewer (or no) “`..`” path name components
2007	`Could not get username for uid` *uid_number* Meaning: The uid could not be found in the NIS maps or in `/etc/passwd` Remedy: Check NIS server and maps.
2008	`No version number in file` *file_name* Meaning: When a Configuring command accesses a metadata file (a file in the `Codemgr_wsdata` directory) it checks the version number written in the file when it was created (for example, VERSION 1). The metadata file file_name does not contain the version string. Remedy: Check the integrity of *file_name*. The version string may have been removed when the file was edited. If the version string is missing, and the file is not otherwise corrupted, use the CLI `workspace create` command or the GUI File Create Workspace menu item to create a new workspace. Check the value of the version string for the analogous file in the new workspace and edit that string into *file_name*.

TABLE 19–1 Configuring Error Messages *(continued)*

2009	`Command` *command_name* `failed, /bin/sh killed by signal` *signal* Meaning: A Configuring command attempted to execute command_name and was unable to because the shell was killed by signal. Remedy: Re-execute the Configuring command.
2010	`Command` *command_name* `failed, could not execute the shell, /bin/sh` Meaning: A Configuring command could not start a shell. This indicates that some system resource, such as swap space or memory was insufficient. Remedy: Check system resources.
2011	`Command` *command_name* `killed by signal` signal Meaning: A command started by a Configuring command received signal signal. Remedy: Re-execute the command. If the error re-occurs, refer to the Solaris documentation for information about the signal.
2012	`Command` *command_name* `exited with status` *status* Meaning: Configuring expects commands it executes to exit with a status of zero indicating successful completion. Configuring considers it an error if a command exits with a non-zero status. Remedy: Refer to the documentation for command_name to determine the meaning of status.
2013	`FLP` *FLP_name* `does not exist in the parent or child workspace` Meaning: The file list program (FLP) FLP_name specified for the Bringover or Putback transaction, could not be found in either the parent or child workspace Remedy: Check the path name of the intended FLP and re-execute the transaction.
2014	`Could not execute` *program_name* Meaning: A Configuring command attempted to execute another program and was unable to do so.

TABLE 19–1 Configuring Error Messages *(continued)*

	Remedy: Ensure that your installation is correct. Ensure that the program is in your search path and that its permissions are set correctly.
2015	`Workspace` *workspace_name* `already exists` Meaning: An attempt was made to create a workspace that already exists. Remedy: Re-execute the command using a different workspace name.
2016	`Workspace` *name* `does not exist` Meaning: The workspace name specified as an argument for a Configuring command could not be found. Remedy: Ensure that the path name was specified correctly.
2017	`Can't open file` *file_name* `so can't get comments for check in` Meaning: Configuring stored checkin comments in a temporary file and was unable to open that file to read the comments. Remedy: Check file permissions and other file system problems that would prohibit opening the file.
2018	`Can't reparent a workspace to itself` Meaning: An attempt was made (either as part of a transaction, or by using an explicit reparent command) to make a workspace its own new parent. Remedy: Re-execute the command, specifying a different parent.
2019	`Internal error: unknown locktype` *lock* Meaning: The workspace lock file (`Codemgr_wsdata/locks`) is corrupted. An unknown lock value was found. Remedy: Edit the lock file to repair the damage. For more information, see the `locks(4)` man page or “Ensuring Consistency Through Workspace Locking” on page 83.
2020	`You must specify a workspace name`

TABLE 19–1 Configuring Error Messages *(continued)*

	Meaning: The Configuring command could not determine the workspace on which to act. Configuring commands attempt to acquire the workspace path name in the following order: ■ As specified by the command's –`-w option` ■ As specified by the value of the environment variable `CODEMGR_WS` ■ The current directory, if it is hierarchically within a workspace Remedy: Specify the workspace path name using one of the methods listed above.
2021	`Cannot obtain a type lock in workspace workspace_name because it has the following locks: Command:` *command (pid)*, `user:` *user*, `machine:` *machine*, `time:` *time*
	Meaning: To ensure consistency, Configuring interworkspace commands lock workspaces while they read and write data in them. The command you issued could not obtain a lock because the workspace is already locked. While Configuring is reading and examining files in the parent workspace during a Bringover transaction, it obtains a *read-lock* for that workspace. When it is manipulating files in the child workspace, it obtains a *write-lock*. Read-locks may be obtained concurrently by multiple Configuring commands that read files in the workspace. No commands may write to a workspace while any read-locks are in force. Only a single write-lock can be in force at any time; no Configuring command may write to a workspace while a write-lock is in force. Lock status is controlled by the `Codemgr_wsdata/locks` file in each workspace. Remedy: If the system is running normally, wait until the command that is locking the workspace releases its lock. If the workspace is stuck in a locked state (for example, the system crashed while a command had a lock in force), use the GUI Options Workspace menu item and selecting Locks from the Category list box in the Workspace Properties dialog box, or use the `workspace locks` command, to remove the lock.
2022	`Invalid subcommand -` *command_name* Meaning: An attempt was made to obtain help on a subcommand of the `resolve`, `workspace`, or `codemgr` command and the name of a non-existent subcommand was specified. Remedy: For the list of valid subcommands for each command, type the command and specify the help subcommand.
2023	Not used.
2024	`File` *file_name* `has no deltas`

TABLE 19–1 Configuring Error Messages *(continued)*

	Meaning: The SCCS history file *file_name* contains no deltas, therefore it cannot be processed. Remedy: Perhaps the history file was mistakenly overwritten.
2025	`Could not find the` *command_name* `command.Executable does not exist:` *name* `Also could not find the` *name* `command in PATH` *PATH_contents* Meaning: A Configuring command attempted to execute another program and was not able to find it. Remedy: Ensure that your installation is correct. Include the directory that contains the missing program.
2026	`Unknown SCCS control character (char) in file` *file_name* `at line` *line_number* Meaning: A Configuring command expected *file_name* to be an SCCS history file; based on the character it encountered, it is either not a history file, or it has been corrupted. Refer to the Solaris SCCS documentation regarding SCCS history file format.
2027	`Corrupted file -` *file_name*`, line` *line_number* Meaning: A Configuring command was unable to read a workspace metadata file (a file in the Codemgr_wsdata directory). Illegal characters were found in line *line_number*.
	Remedy: Check and repair the file. All Configuring metadata files are ASCII text files and can be edited. See the *file_name*(4) man page or Chapter 7” for more information on its format.
2028	`Could not find the` *command_name* `command in PATH` *path_name* Meaning: A Configuring command attempted to execute another program and was not able to find it. Remedy: Ensure that your installation is correct. Include the directory that contains the missing program.
2029	`The file has unresolved conflicts. Run ‘edit m’ and search for ^<<<<<<<`

TABLE 19–1 Configuring Error Messages *(continued)*

	Meaning: This error is issued by the resolve command. An attempt was made to save the file while it still contained unresolved conflicts. Remedy: Use the `edit m` subcommand (edit the merged result) to resolve the conflicts and then save the file. Conflicts are marked with ^<<<<<<<.
2030	`No file with number` *file_number* Meaning: The resolve command creates a numbered list of files that contain conflicts. The *file_number* chosen does not exist in this list. Remedy: Use the `list` subcommand to list the files and determine the correct number of the file you wish to specify.
2031	`Can’t find home directory so can’t write to file` *file_name* Meaning: A Configuring command was unable to find the user’s home directory and cannot locate file *file_name*. This usually indicates a problem with NIS maps. Remedy: Check NIS server and appropriate NIS maps.
2032	`Can’t parse line in file` *file_name*`:` *line* Meaning: Upon startup, the resolve command reads the `~/.codemgr_resrc` file to obtain user defined properties. The line *line* could not be interpreted correctly by the program. Remedy: Correct the file `~/.codemgr_resrc` file so that it includes only valid entries. For information regarding these entries, see the resolve(1) manual page.
2033	`Must specify a directory list either as arguments or via the CODEMGR_WSPATH variable` Meaning: This message is reported by the `workspace list` command when a directory (or list of directories) was not specified in a way in which `workspace list` can search for workspaces to list. Directories can be specified as the standard argument to the command, or by defining the `CODEMGR_WSPATH` variable to contain the path name of a directory. Remedy: Re-execute the command specifying a directory, or set the `CODEMGR_WSPATH` directory to contain a directory path.
2034	`internal error: Access control operation operation_name does not have a Ibuilt-in default`

TABLE 19–1 Configuring Error Messages *(continued)*

	Meaning: A Configuring command attempted to verify access permission for a workspace operation (for example: bringover-from, putback-to, reparent-to). An internal consistency check failed. Remedy: Contact your local service representative.
2035	`Access control file does not exist` Meaning: A Configuring command attempted to verify access permission for a workspace operation (for example: bringover-from, putback-to, reparent-to). The access control file (`Codemgr_wsdata/access_control`) in the affected workspace was not found. Remedy: If the access control file has been deleted from the workspace, copy a new one from another workspace and edit it so that the access permissions are correct. If no other workspaces are available, create a new workspace using the CLI `workspace create` command or the GUI File Create Workspace menu item and copy the file from the newly created workspace. For more information refer to the `access_control`(4) man page or “Controlling Access to Workspaces” on page 73.
2036	`Cannot specify common ancestor file; there is no common ancestor delta` Meaning: The ancestor (`a`) was specified as an argument to a `resolve` subcommand (`diff`, `edit`, `more`). The files that are being resolved do not have an ancestor in common. This occurs most commonly in cases where files with the same name are created concurrently in both the child and the parent. They have the same name but are not descended from a common ancestor. For more information about ancestors and their role in resolving conflicts, see Chapter 9.” Remedy: Proceed with the conflict resolution process without specifying the ancestor (`a`) as an argument to the `diff`, `edit`, and `more` subcommands.
2037	`Invalid argument -` *character* Meaning: An invalid argument was specified to one of the `resolve` subcommands. The command expected one of the following characters: a (ancestor), c (child), p (parent), m (merged result). Remedy: Specify one of the valid arguments: a, c, p, m. See the `resolve`(1) man page for more information.
2038	`Parent workspace is an NSE environment. Use the nseputback command`

TABLE 19–1 Configuring Error Messages *(continued)*

	Meaning: The “parent” in a putback transaction is an NSE environment and not a Configuring workspace. Remedy: Use the `nseputback` command to putback changes from the workspace to the environment.
2039	`File` *file_name* `is probably not an s-file on line` *line_number* `expected ^A, but got` *char* Meaning: A Configuring command expected file_name to be an SCCS history file; based on the format it is either not a history file, or it was corrupted. Remedy: Refer to Solaris SCCS documentation regarding SCCS history file format.
2040	`File` *file_name* `has not been merged. Use the twmerge subcommand first or the filemerge subcommand` Meaning: An attempt was made to commit (save) a file that had not yet been merged. Remedy: Merge the file using either the `twmerge` subcommand, or the `filemerge` subcommand (which executes the Merging GUI). For more information about the `resolve` command see the `resolve`(1) man page.
2041	*path_name* `is not a workspace or a directory` Meaning: The string path_name specified in the Bringover Create transaction is not a Configuring workspace or a directory. Remedy: Specify a different workspace or directory name.
2042	`Can’t create ToolTalk message, error = TT_error_code` Meaning: The resolve command communicates with the Merging program via the ToolTalk service. ToolTalk is an interapplication communication service distributed with the Solaris OpenWindows windowing system. In this case the `resolve` command called a ToolTalk routine to create a ToolTalk message for Merging. The ToolTalk routine could not create the message and passed back TT_error_code. Remedy: Refer to the OpenWindows ToolTalk documentation for information about the error.
2043	`SCCS file` *file_name* `is corrupted`

TABLE 19–1 Configuring Error Messages *(continued)*

	Meaning: The SCCS admin -h command reports that the newly computed check-sum does not compare with the one stored in the first line of the file. Remedy: See the Solaris SCCS documentation for more information.
2044	`Unable to create a temporary name from template` *temp_file_name* Meaning: A Configuring command was unable to create a temporary file for its use. This is a Configuring internal error. Remedy: Check for any system-level reasons why the command could not write this file (for example, file permission restrictions or incorrect command ownership).
2045	`Fprintf of` *file_name* `failed` Meaning: A command was unable to write to the file *file_name*. Remedy: Check file permissions and other such file system problems that would prohibit writing in the file system.
2046	`Version mismatch in file` *file_name*`, expected version` *expected_number*`, but found` *actual_number* Meaning: Each Configuring metadata file (`Codemgr_wsdata/*`) contains a string that includes a version number (for release 1.0 the version number string is VERSION 1). As a consistency check, when Configuring commands read and write to these files, they check to determine whether the file contains the version that the command expects. In this case the command expected to find expected_number but found actual_number instead. This may indicate that old binaries are being used with new metadata files and could cause the file to be corrupted.
	Remedy: Make sure that the most current versions of the Configuring binaries are being accessed.
2047	`Do not know how to convert file` *file_name* `from version` *found_version_number* `to` *current_version_number*

TABLE 19–1 Configuring Error Messages *(continued)*

	Meaning: Each Configuring file (`Codemgr_wsdata/*`) contains a string that includes a version number (for release 1.0 the version number string is `VERSION 1`). As a consistency check, when Configuring commands read and write to these files, they check to determine whether the file contains the version that the command expects. It is anticipated that when new versions of Configuring binaries and metadata files are released, the formats of some of these files may change. Commands contain code to make this conversion. A command found a metadata file with a version number earlier than 1. Remedy: Since this is the first release of Configuring, the version string in the metadata file must have been inadvertently changed during editing. Check the file and make sure that the first line reads “VERSION 1”.
2048	`Must specify at least one file, directory or -f argument to a bringover that creates a child workspace` Meaning: The command-line for a Bringover transaction was not constructed properly. An argument that specifies at least one file, directory or FLP must be included. If this argument is omitted, Configuring attempts to take the arguments from the workspace’s `Codemgr_wsdata/args` file. Remedy: Re-enter the command and ensure that you’ve included the correct number of arguments.
2049	`Could not determine where` *file_name* `is mounted from` Meaning: Configuring commands convert path names of NFS mounted directories to the *machine_name*:*path_name* format to do much of their work. This message indicates that the mount entry contained in `/etc/mnttab` is no longer present. Remedy: Remount the file system that contains file_name.
2050	`Could not determine the absolute pathname for` *file_name* Meaning: A Configuring command was unable to read a directory. This indicates some corruption in the file system; for example, incorrect directory permissions. Remedy: Check the file system, especially directory and file permissions in the path of file_name.
2051	`Can’t rename to` *file_name*`; it exists`

TABLE 19–1 Configuring Error Messages *(continued)*

	Meaning: During a Bringover, Putback, or Undo transaction, a file was found that was renamed in the source workspace to a name already in use in the destination workspace. Remedy: Change the name in one of the directories.
2052	`Corrupted file -` *file_name*`, text after BEGIN,` *line number*`. Can’t send notification` Meaning: A Configuring command encountered an error when reading the workspace notification file `Codemgr_wsdata/notification`. The BEGIN statement that delimits the list of files / directories for which notification is requested must be the only text on the line, other text was encountered. The Configuring command cannot correctly parse the request; if the file contains a notification request, it cannot be sent. Remedy: Edit the notification file and enter the appropriate BEGIN statement. See the `notification`(4) man page or “How to Notify Users of Changes to Workspaces” on page 77 for more information on its format.
2053	`Corrupted file -` *file_name*`, text after END,` *line number*`. Can’t send notification` Meaning: A Configuring command encountered an error when reading the workspace notification file `Codemgr_wsdata/notification`. The END statement that delimits the list of files / directories for which notification is requested must be the only text on the line, other text was encountered. The Configuring command cannot correctly parse the request; if the file contains a notification request, it cannot be sent. Remedy: Edit the notification file and enter the appropriate END statement. See the `notification`(4) manual page or “How to Notify Users of Changes to Workspaces” on page 77 for more information on its format.
2054	`Corrupted file -` *file_name*`, missing BEGIN,` *line number*`. Can’t send notification` Meaning: A Configuring command encountered an error when reading the workspace notification file `Codemgr_wsdata/notification`. The BEGIN statement that delimits the list of files / directories for which notification is requested, is missing. The Configuring command cannot correctly parse the request; if the file contains a notification request, it cannot be sent. Remedy: Edit the notification file and enter the appropriate BEGIN statement. See the `notification`(4) man page or “How to Notify Users of Changes to Workspaces” on page 77 for more information on its format.

TABLE 19–1 Configuring Error Messages *(continued)*

2055	`File` *file_name* `has incomplete delta table` Meaning: The delta table in the SCCS history file file_name is incomplete. This indicates that the file was corrupted. Remedy: Fix the file, or copy in a new version.
2056	`Badly formatted line in` *file_name*`:` *line_number* Meaning: A Configuring command was reading a temporary log file left over from an aborted Bringover or Putback operation and encountered a malformed line. This indicates that the file was corrupted. Remedy: Execute the `workspace updatenames` command to rebuild the nametable and then re-execute the command.
2057	`Zero-length SCCS file,` *file_name* Meaning: An SCCS history file was encountered that contained no data. Remedy: Remove the SCCS history file.
2058	`Can't get a version of the child file until it is checked in` Meaning: During a Resolve transaction a file was encountered that is not checked in to SCCS. Files must be checked in before conflicts can be resolved. Remedy: Check the file in and re-start the transaction.
2059	`Name history serial number` *number* `out of order in file` *file_name* Meaning: Rename information in the SCCS history file file_name is corrupted. The name history records in this SCCS file are not in numerically descending order. Remedy: Reorder the name history records, or copy in a new version of the file using the Bringover or Putback transaction.
2060	`Delta serial number` *number* `out of order in file` *file_name* Meaning: Delta numbers are not in numerically descending order in the SCCS history file file_name. This indicates that the file is corrupted.

TABLE 19–1 Configuring Error Messages *(continued)*

	Remedy: Reorder the delta numbers, or copy in a new version of the file using the Bringover or Putback transaction.
2061	`Must have DISPLAY environment variable set to invoke twmerge or filemerge` Meaning: Merging is an XWindows program. The `DISPLAY` environment variable is is used to determine where the program displays its windows. If you are running CDE or OpenWindows, this variable is usually set for you. For tother XWindow display servers, you may need to set this variable manually before starting Merging. Remedy: Determine the correct setting for your display server and set the `DISPLAY` environment variable appropriately.
2062	`Can't resolve file` *file_name* `because it is writable` Meaning: The file file_name is not checked out from SCCS but its file permissions indicate that it is writable. Resolving this conflict will result in writing to a file that is not checked out. Remedy: Reconcile the file permissions (for example, check the file out and then check it back in) and then re-execute the Resolve transaction.
2063	`Cannot create workspace` *name* `because it would be nested within workspace` *name* Meaning: An attempt was made to create a workspace hierarchically beneath an existing workspace. Remedy: Create the new workspace hierarchically outside of any existing workspaces.
2064	`Cannot delete a workspace that is a symbolic link. Run workspace delete` *workspace_name* Meaning: Configuring commands will not delete directories or files that are symbolic links. You must delete the physical copy of the file. The appropriate command line is provided. Remedy: Use the `workspace delete` command to delete *workspace_name*.

TABLE 19–1 Configuring Error Messages *(continued)*

2065	This error message may be issued in any of the following forms: `User` *user_name* `does not have access to bringover from workspace` *workspace_name* `User` *user_name* `does not have access to bringover to workspace` *workspace_name* `User` *user_name* `does not have access to putback from workspace` *workspace_name* `User` *user_name* `does not have access to putback to workspace` *workspace_name* `User` *user_name* `does not have access to undo workspace` *workspace_name* `User` *user_name* `does not have access to delete workspace` *workspace_name* `User` *user_name* `does not have access to move workspace` *workspace_name* `User` *user_name* `does not have access to change the parent of workspace` *workspace_name* `User` *user_name* `does not have access to change the parent to workspace` *workspace_name* Meaning: The user *user_name* attempted an operation that affected the workspace *workspace_name*; access permissions in *workspace_name* do not permit *user_name* access to execute that operation. Remedy: The file *workspace_name*`/Codemgr_wsdata/access_control` is a text file that specifies access permissions for various workspace operations. The owner of the workspace must change the permissions to include user_name in order for the operation to proceed. Permissions can be changed using the Options Workspace menu item and selecting Access Control from the Category list box in the Workspace Properties dialog box, or by editing the `access_control` file directly. See the `access_control`(4) man page or Chapter 7” for more information.
2066	`Corrupted file -` *file_name*`, whitespace in` pathname`, line` *line_number*`. Can’t send notification` Meaning: A Configuring command encountered an error when reading the workspace notification file `Codemgr_wsdata/notification`. A whitespace character was encountered in a line where a single path name was expected.

TABLE 19–1 Configuring Error Messages *(continued)*

	Remedy: Edit the `Codemgr_wsdata/notification` file to remove the whitespace characters from the line. See the `notification`(4) man page or “How to Notify Users of Changes to Workspaces” on page 77 for more information on its format.
2067	`Corrupted file -` *file_name*`, missing notification event, line` *line_number*`. Can’t send notification` Meaning: A Configuring command encountered an error when reading the workspace notification file `Codemgr_wsdata/notification`. The Configuring event (for example, bringover-to) was not specified. Remedy: Edit the `Codemgr_wsdata/notification` file to add the correct event. See “How to Notify Users of Changes to Workspaces” on page 77 for a list of valid events.
2068	`putback: sccs error for file` *file* `can not create lock file` Meaning: Someone else is updating the SCCS file or the p-file, or you do not have write permission for the directory in which the SCCS file resides.
	Remedy: If someone else is updating the SCCS file or the p-file, wait until they release the lock and then try again to access the file. If you do not have write permission in the directory, you cannot create a lock file in that directory until you obtain write permission.
2069	Not used
2070	Not used
2071	Not used
2072	Not used
2073	Not used
2074	`Workspace` *workspace_name* `has no locks` Meaning: An attempt was made to remove locks from a workspace that had no active locks.
2075	`Lock` *lock_name* `does not exist for workspace` *workspace_name* Meaning: While using the workspace locks `-r` command, a lock number was specified that is out of range of the lock list.

TABLE 19–1 Configuring Error Messages *(continued)*

	Remedy: Check the lock numbers for the workspace using the `workspace locks` command and enter a valid number.
2076	`Internal error: Cannot find the directory in which command` *command_name* `is located because avo_find_dir_init() has not been called` Meaning: This is an internal error. Remedy: Please contact your local service representative.
2077	*number* `is not a valid number` Meaning: While using the `resolve` command, a number was referenced that is outside of the listed values. Remedy: List the values to determine the valid number for your selection.
2078	`Cannot access workspace` *workspace_name* Meaning: File permissions for workspace_name prohibit access by the Configuring command. Remedy: Default permissions for workspace directories are 777.
2079	`Could not parse name history for file` *file_name*`, contains:` *text* Meaning: There is a format error in the name history record in the SCCS history file file_name. The troublesome text is displayed. Remedy: If possible, fix the record; otherwise copy a new version of the file using the Bringover or Putback transaction.
2080	`Could not remove or rename backup directory` *directory_name* Meaning: Configuring attempted to clear the backup area directory_name so that it could backup a new transaction. Configuring was not able to delete or rename the directory out of the way. The most likely cause is that file permissions have been changed for the directory. Remedy: Check directory permissions for directory_name. Default Configuring permissions for this directory are 777.
2081	`build_workspace_list:` *path_name* `does not start with a /`

	The Configuring GUI program was expecting a fully qualified path name to be returned from a subprocess. This is an internal error. Please contact your local service representative.
2082	`Workspace workspace_name's parent does not exist in the filesystem` Meaning: The parent workspace is not mounted or visible on this machine. Remedy: Mount the parent workspace on the executing machine.
2083	`Workspace` *workspace_name*`'s child does not exist in filesystem` Meaning: The child workspace is not mounted or visible on this machine. Remedy: Mount the child workspace on the executing machine.
2084	`codemgrtool: internal error in args_strlist_from_wsname() : NULL args_list` Meaning: Internal error. Remedy: Contact your local service representative.
2085	`codemgrtool: internal error in undo_strlist_from_wsname() : NULL undo_list` Meaning: Internal error. Remedy: Contact your local service representative.
2086	`codemgrtool: path_name doesn't start with a /` Meaning: Internal error. Remedy: Please contact your local service representative.
2087	Not used.

TABLE 19–1 Configuring Error Messages *(continued)*

2088	`Nametable in workspace` *workspace_name* `cannot be read because the following SCCS files have identical root deltas`
	file_name
	file_name
	`Run the following command and then re-execute the` *command_name* `command:`
	`path_name/workspace updatenames` *workspace_name*
	Meaning: An SCCS history file was copied within a workspace using the `cp` command. As a result, the two files contain the identical root delta. Configuring uses the root delta to distinguish between files. The `workspace updatenames` command enables Configuring to distinguish between the files.

	Remedy: Execute the `workspace updatenames` command and then re-execute the command that spawned the error.

2089	`Cannot move workspace` *workspace_name*
	`Because it is a symlink to` *directory_name*`.`
	`Use a workspace name that is not a symlink.`
	Meaning: Configuring commands will not move directories or files that are symbolic links.
	Remedy: Move the workspace to a name that is not a symbolic link.

2090	`Nametable in workspace` *workspace_name* `not written because the following SCCS files have identical root deltas`
	file_name
	file_name
	`Run the following command and then re-execute the` *command_name* `command:`
	`path_name/workspace updatenames` *workspace_name*

TABLE 19–1 Configuring Error Messages *(continued)*

	Meaning: An SCCS history file was copied within a workspace using the `cp` command. As a result the two files contain the identical root delta. Configuring uses the root delta to distinguish between files. The `workspace updatenames` command enables Configuring to distinguish between the files. Remedy: Execute the `workspace updatenames` command and then re-execute the command that spawned the error.
2091	`Internal error: hash table missing entry` Meaning: Internal error. Remedy: Contact your local service representative.

2092	`An SCCS file (A) was copied (to file B). The original SCCS file (A) cannot be found.`
	`Run the following command and then re-execute the` *command_name* `command:`
	`path_name/workspace updatenames` *workspace_name*
	Meaning: An SCCS history file was copied within a workspace using the `cp` command. The original file (A) was subsequently renamed or removed from the workspace. Configuring is unable to determine whether the files has been renamed (and to what name) or removed from the workspace. The workspace updatenames command interactively displays the possible names to which the file could have been renamed, and asks you to determine the file's current state: its new name, or its absence from the workspace. Configuring can then correctly propagate the changes throughout the workspace hierarchy.
	Remedy: Execute the `workspace updatenames` command and then re-execute the command that spawned the error.

2093	`Internal error: SmIDs not equivalent` Meaning: Internal error. Remedy: Contact your local service representative.
2094	`Internal error: SmID not found` Meaning: Internal error. Remedy: Contact your local service representative.

TABLE 19–1 Configuring Error Messages *(continued)*

2095 - 2499	Not used
2500 - 2600	Internal errors Meaning: Error numbers 2500 through 2600 are Configuring programs internal errors. These errors indicate problems that users cannot correct. If you encounter any of these errors, contact your local service representative.

Warnings Messages

Table 19–2 describes warning messages, their meanings, and possible remedies

TABLE 19–2 Configuring Warning Messages

2601	`Could not remove backup directory old_dir_name, so it was renamed to` *new_dir_name* Meaning: Configuring attempted to clear the backup area *old_dir_name* so that it could backup a new transaction. Configuring was not able to clear the backup directory by deleting it, but it was able to rename it out of the way to the name *new_dir_name*. The most likely cause is that file permissions were changed for the directory. Remedy: Check directory permissions for *old_dir_name*. Default Configuring permissions for this directory are 777. Delete the contents of *new_dir_name*.
2602	`File` *file_name* `is not under SCCS in either workspace - ignored` Meaning: Configuring could not find an SCCS history file in either workspace for *file_name*. Remedy: The file name was probably entered incorrectly, re-execute the command.
2603	`Zero length filename - ignored`

TABLE 19–2 Configuring Warning Messages *(continued)*

	Meaning: A file name specified as an argument on the command-line (or in the `Codemgr_wsdata/args` file) contained no characters (““). Remedy: Re-execute the command and re-specify the file name argument. If the problem persists, check the arguments listed in the `args` file.
2604	`Filename` *file_name* `has whitespace characters in it - ignored` Meaning: A file name specified as an argument on the command-line (or in the `Codemgr_wsdata/args` file) contained whitespace characters. Configuring commands do not accept file names that contain whitespace characters. Remedy: Re-execute the command and re-specify the file name argument. If the problem persists, check the arguments listed in the `args` file.
2605	Not used
2606	`File` *file_name* `not brought over because it is a` *file_type* `in workspace` *workspace_name* `and a` *file_type* `in workspace` *workspace_name* Meaning: A file name has a different file type (regular file vs. directory vs. symbolic link) in the parent and child workspaces. Remedy: Take whatever action is appropriate to make the listed files the same type, or change one of the names.
2607	Not used
2608	`Workspace` *child_ws_name* `is a child of` *parent_ws_name*`. Could not update its parent file` Meaning: During a `workspace delete` or `workspace move` operation involving *child_ws_name*, the command found that the `children` file in the workspace’s parent (*parent_ws_name*) did not contain an entry specifying *child_ws_name* as a child of that parent. Remedy: Advisory only: the command corrects the discrepancy, however, this could indicate that the parent’s `children` file was corrupted.
2609	Not used
2610	*directory_name* `is not a workspace`

TABLE 19–2 Configuring Warning Messages *(continued)*

	Meaning: The directory specified in the command is not a Configuring workspace. Configuring workspaces are distinguished by the presence of the `Codemgr_wsdata` directory in the top level directory. Remedy: Specify a different workspace name or use the CLI `workspace create` command or GUI File Create Workspace menu item to convert the directory into a workspace.
2611	*file_name* `does not exist in either workspace - ignored` Meaning: The file *file_name* was not found in either the parent or child workspace. Remedy: Check to be sure the name was specified correctly.
2612	Not used
2613	`Filename` *file_name* `has too many ".." path components in it - ignored` Meaning: Possible causes include that the Configuring command cannot resolve the path name into a: ■ workspace-relative file name ■ fully qualified workspace name Remedy: Specify the path name with fewer (or no) "`..`" path name components
2614	`Line` *line_number* `too long or unexpected end of file in` *file_name* Meaning: While reading the `Codemgr_wsdata/nametable` file, a line was encountered that contained too many characters for a Configuring command to buffer. The maximum line length is 1024 characters. This indicates that `nametable` has been corrupted. Remedy: Configuring automatically rebuilds the nametable. This takes some time.
2615	`Line` *line_number* `has bad format in` *file_name* Meaning: This indicates that the `Codemgr_wsdata/nametable` file has been corrupted. Remedy: Configuring automatically rebuilds the nametable. This takes some time.

TABLE 19–2 Configuring Warning Messages *(continued)*

2616	Not used
2617	`Unexpected name table editlog record type` *type_number* `- ignored` Meaning: A Configuring command was reading a temporary log file left over from an aborted Bringover or Putback operation and encountered a malformed record. This indicates that the file has been corrupted. Remedy: Execute the `workspace updatenames` command to rebuild `nametable` and then re-execute the command.
2618	`Can't open` *file_name* `- can't send mail notification` Meaning: The Configuring notification facility failed to open the file *file_name*. As a result, notification mail is not sent for the current operation. Remedy: Check file permissions for *file_name*.
2619	Not used
2620	`Can't fork process to send notification` Meaning: Lack of system resources (memory, swap space) prevented the Configuring notification facility from sending notification mail. Remedy: Check system resources.
2621	Not used
2622	`Filename` *file_name* `contains a comment character (#) - ignored` Meaning: A file name specified as an argument to a command (or in the `Codemgr_wsdata/args` file) contains the # character. Configuring reserves this character to denote comments. Remedy: Change the name of the file so that its file name does not contain the # character. If the problem persists, check the arguments listed in the `args` file.
2623	`Read-lock left in workspace` or `Write-lock left in workspace` Meaning: A Configuring command was unable to remove locks in *workspace_name*. This may indicate that there is insufficient disk space, or that permissions on the file `Codemgr_wsdata/locks` were changed since the lock was originally written.

TABLE 19–2 Configuring Warning Messages *(continued)*

	Remedy: Remove the locks by using the GUI Options Workspace menu item and selecting Locks in Category list box in the Workspace Properties dialog box, or by using the CLI `workspace locks` command.
2624	`File` *file_name* `is checked out in workspace` *workspace_name*`. The changes in the checked out file will not be brought over` Meaning: The file *file_name* is checked out in the parent workspace. You are being advised that any changes in the g-file were not brought over as part of the Bringover transaction. Remedy: Not applicable.
2625	`File` *file_name* `is not in conflict according to the SCCS file. Removing it from the conflict file` Meaning: The information in the SCCS history file indicates that the file contains no unresolved conflicts, however, the `Codemgr_wsdata/conflicts` file in the workspace lists it as being in conflict. The command removed it from the `conflicts` file. Remedy: Not applicable.
2626	`File` *file_name* `not brought over because it is unresolved in workspace` *workspace_name* Meaning: The file *file_name* was not brought over because it contains an unresolved conflict in *workspace_name.* Remedy: Use the GUI Resolve transaction or the CLI `resolve` command to resolve the conflict and then re-execute the Bringover transaction.
2627	`Directory` *directory_name* `is mounted read-only.` Meaning: Before beginning Bringover and Putback transactions, Configuring checks to determine whether the destination workspace root (top-level) directory is accessible for writing. This is not treated as an error condition because lower level directories within the workspace could be mounted from different areas and they may be accessible for writing. This warning is issued as an early warning that directory permissions might be set incorrectly. Remedy: If write access is not intentionally denied, change the root directory permissions.

TABLE 19–2 Configuring Warning Messages *(continued)*

2628	`Not updating g-files because get command couldn’t be found in PATH search_path` Meaning: The g-files could not be updated as part of a Bringover or Putback transaction because the SCCS `get` command could not be executed; it was not found in your search path. Remedy: If you want g-files to be updated as part of transactions, include the `get` command in your search path.
2629	`Will not be able to run twmerge or filemerge` Meaning: The `resolve` command was not able to connect with the ToolTalk message service. The ToolTalk service is used by the resolve command to communicate with Merging. Remedy: The ToolTalk service is normally installed as part of CDE or OpenWindows. Check the CDE or OpenWindows documentation to determine why the ToolTalk service is not present or responding.
2630	`This workspace is being created over an existing directory` Meaning: You are converting an already existing directory into a Configuring workspace. Creating a workspace from an existing directory hierarchy consists of creating the `Codemgr_wsdata` metadata directory in the top-level directory. Once the directory becomes a workspace, its contents can be deleted using the Configuring `workspace delete` command. Remedy: Not applicable.
2631	`File` *file_name* `not brought over because it is checked out and not writable in workspace` *workspace_name* Meaning: The file *file_name* was not brought over as part of the Bringover transaction because it is checked out (p-file exists) and writable in the child workspace *workspace_name*. The unusual state of this file indicates that it is safer not to process the file. Remedy: Reconcile the write permissions with its SCCS status.
2632	`Omitting contents change to file` *file_name* `because of rename error`

TABLE 19–2 Configuring Warning Messages *(continued)*

	Meaning: An error was encountered while processing the name of *file_name*. As a result, the change in the file from the source workspace could not be propagated to the destination workspace. Remedy: Correct the rename problem (see the rename error text) and re-execute the Configuring transaction.

APPENDIX **A**

About SCCS Mergeable IDs

This appendix explains why SCCS Mergeable IDs (SMIDs) are necessary, how to translate SCCS delta IDS (SIDs) to SMIDs, and how to translate SMIDs to SIDs. It contains the following sections:

- "Why SMIDs Are Necessary" on page 257
- "SMID/SID Translation" on page 258

  The use of SCCS Mergeable IDs (SMIDs) ensures that every delta is uniquely identifiable, even if its SCCS delta ID (SID) is changed. A SMID is a number generated using the Xerox Secure Hash Function. When you use Freezepointing to create a freezepoint file, it calculates the SMID for both the current delta and the root delta in the SCCS history file. Using both of these values, Freezepointing can identify a delta in a file even if its SID has been changed.

Why SMIDs Are Necessary

Note - This section briefly discusses how Configuring merges SCCS history files. For more information, see Chapter 10."

When Configuring encounters a file conflict during a Bringover Update transaction (file is changed in both the parent and child workspaces), it merges the new deltas from the parent workspace into the SCCS history file in the child. When this merge occurs, the deltas that were created in the child are moved to an SCCS branch off of the delta that both deltas have in common (common ancestor).

When Configuring relocates the child deltas to a branch, it changes their SID. If SIDS were used in freezepoint files to identify deltas, this relocation would invalidate the information contained in the freezepoint file. For that reason, SIDs cannot be used to identify deltas after conflicting SCCS histories have been merged.

SMID/SID Translation

SMID/SID translation is available only through the Freezepointing CLI.

The `freezept` command `sid` and `smid` subcommands enable you to translate specified SIDs into SMIDs, and to translate specified SMIDs into SIDs. The ability to make these translations is useful if you wish to write your own scripts or programs to track deltas.

Translating SIDs to SMIDs

Use the `freezept smid` command to translate SIDs to SMIDs. The syntax is:

`freezept smid` [`-w` *workspace*] [`-r` *SID*] [`-a`] *file*

- Use the `-r` option to specify the SID (in file *file*) for which you wish to calculate a SMID.
- Use the `-a` option to calculate a SMID for all of the SIDS in *file*.
- For convenience you can use the `-s` option to specify a directory from which *file* is relative.

Examples

```
example% freezept smid -r 1.38 module.c

SID 1.38 = SMID “f5b67794 705f0768 a89b1f4 588de104”
```

```
example% freezept smid -a bringover.1
SID 1.1 = SMID “b05b0a2f 1db5246e 1a466014 707e38f5”
SID 1.2 = SMID “d6a5c61f 5634f0ef 9847a080 d0d7b212”
SID 1.2 = SMID “e31acdd5 6c1232e2 9e81c287 1edb2f41”
SID 1.3 = SMID “c34c91b4 a818622a 2457356a 489b2728”
SID 1.4 = SMID “98c0fd8d 889563fb cf722c2b 6afc9636”
SID 1.5 = SMID “b1e24be3 752fec3e df2d2717 a9b3f1fa”
SID 1.6 = SMID “2b93d39 1ea2f6ba 9814320c bc609acb”
SID 1.7 = SMID “1db7d640 42b0f009 35c60d7b b230bd85”
SID 1.8 = SMID “906dfe9a ca7e2d6c a64da5be 4baef254”
```

Translating SMIDS to SIDS

Use the `freezept sid` command to translate SMIDs to SIDs. The syntax is:

`freezept sid` [`-w` *workspace*] [`-m` “*SMID*”] [`-a`] *file*

- Use the `-m` option to specify the SMID (in file *file*) for which you wish to calculate a SID.
- Use the `-a` option to calculate a SID for all of the deltas in *file*.
- For convenience you can use the `-s` option to specify a directory from which *file* is relative.

Note - Because the SMID contains white space, you must enclose it within quotation marks.

Examples

```
example% freezept sid -m "64fdd0df de9d7dd de75812 23da96aa" module.c
SMID “64fdd0df de9d7dd de75812 23da96aa” = SID 1.36
```

```
example% freezept sid -a bringover.1
SMID “b05b0a2f 1db5246e 1a466014 707e38f5” = SID 1.1
SMID “d6a5c61f 5634f0ef 9847a080 d0d7b212” = SID 1.2
SMID “e31acdd5 6c1232e2 9e81c287 1edb2f41” = SID 1.2
SMID “c34c91b4 a818622a 2457356a 489b2728” = SID 1.3
SMID “98c0fd8d 889563fb cf722c2b 6afc9636” = SID 1.4
SMID “b1e24be3 752fec3e df2d2717 a9b3f1fa” = SID 1.5
SMID “2b93d39 1ea2f6ba 9814320c bc609acb” = SID 1.6
SMID “1db7d640 42b0f009 35c60d7b b230bd85” = SID 1.7
SMID “906dfe9a ca7e2d6c a64da5be 4baef254” = SID 1.8
SMID “77481e8a 61542339 cc28f532 e5fc6389” = SID 1.9
SMID “cb97c9a6 d0342cf6 19b7b743 2436ca1c” = SID 1.10
SMID “46de4131 b95b9973 93958a07 b960074c” = SID 1.11
```

Glossary

Access control
The Configuring facility by which users can control access to workspaces by Configuring commands.

Branch (SCCS)
A delta or series of deltas that are placed off of the main line of deltas in an SCCS history file.

Bringover Create
The transaction used to copy groups of files from a parent workspace to a nonexistent child workspace. The new child workspace is created as a result of the transaction. All Configuring transfer transactions are performed from the perspective of the child workspace; hence the Bringover Create transaction “brings over” files to the child from the parent workspace. See also *Bringover Update, Workspace* and *Putback.*

Bringover Update
The transaction used to update an existing child workspace with respect to files contained in its parent workspace. All Configuring transfer transactions are performed from the perspective of the child workspace; the Bringover Update transaction “brings over” files to the child from the parent workspace. See also *Bringover Create, Workspace*, and *Putback.*

Child workspace
A workspace that has a parent workspace listed in its `Codemgr_wsdata/parents` file. Development work is typically done in child workspaces and put back to parent workspaces after it has been tested. The Configuring transfer transactions are viewed from the child workspace perspective, and all conflicts are resolved in the child workspace.

`Codemgr_wsdata` **directory**
Every TeamWare workspace contains a “metadata” directory in its root directory named `Codemgr_wsdata`. Configuring stores data about the workspace in `Codemgr_wsdata`. The presence of this directory is the sole factor that defines it as a TeamWare workspace (as opposed to a normal directory). Configuring commands use the

presence or absence of this directory to determine whether a directory is a workspace. See “The Workspace Metadata Directory ” on page 65 for more information.

Conflict

The condition that exists when a file has changed in both the child and parent workspace. Conflicts are identified by the Bringover Update transaction and are resolved by using the Resolve transaction.

Copy-Modify-Merge

The concurrent development model upon which Configuring is based. Using this model, multiple developers concurrently *copy* sources from a common area, *modify* the source in isolation, and then *merge* those changes with changes made by other developers.

Create

Used in Configuring transaction output. Files are said to be created if they exist in the source workspace and not in the destination workspace, and are copied into the destination workspace as part of a Bringover or Putback transaction.

Default line of work

The branch in an SCCS history file upon which the next delta will be added.

`def.dir.flp`

The default FLP shipped with Configuring is `def.dir.flp`; this FLP recursively descends directory hierarchies and lists all files for which SCCS history files exist. See *FLP*.

Delta

The set of differences between two versions of a file checked into SCCS.When you check in a file, SCCS records only the line-by-line differences between the text you check in and the previous version of the file. This set of differences is known as a delta. The file version that you initially checked out was constructed from a set of accumulated deltas. The terms delta and version are often used synonymously; however, their meanings are not the same. It is possible to retrieve a version that omits selected deltas. See *Version*.

Merging

The TeamWare utility used to merge deltas during Resolve transactions. See Chapter 9” and Chapter 10.

FLP

An FLP or *FIle List Program* is a program or script that generates a list of files to `stdout` that Configuring then processes during Bringover and Putback transactions. See *def.dir.flp*.

Freezepointing

The TeamWare utility used to make snapshots of workspaces (or portions of them) at important junctures or “freezepoints.”

g-file (SCCS) The working copy of a file retrieved from an SCCS history file by the `sccs-get` command.

History files When you initially put a file under SCCS control, a history file is created for the new SCCS file. The initial version of the history file uses the complete text of the source file. The initial history file is the file that further deltas are compared to. Owing to its prefix (s.), the history file is often referred to as the s.file (s-dot-file).

Integration workspace A workspace to which multiple developers put back (merge) their work.

Lock To assure consistency, the Configuring file transfer transactions Bringover and Putback lock workspaces while they are working in them. Locks are recorded in the `Codemgr_wsdata/lock` file in each workspace; the Configuring commands consult that file before acting in a workspace. See *read-lock, write-lock.*

Merge To produce a single version of a file from two conflicting files (deltas). Usually accomplished with the assistance of the Merging program.

Notification A Configuring facility that mails notice of events, such as changes to files or directories, to users.

Parent workspace A workspace that has a child workspace(s) listed in its `Codemgr_wsdata/children` file. Parent workspaces are typically used as integration areas, since development, testing, and conflict resolution occur in child workspaces.

Putback The transaction used to update a parent workspace with respect to files contained in its child workspace. All Configuring transfer transactions are performed from the perspective of the child workspace; the Putback transaction “puts back” files to the parent from the child workspace. See also *Bringover Create, Bringover Update,* and *Workspace.*

Read-lock A lock that is obtained by a Configuring command while it examines the contents of a workspace. A read-lock assures that the workspace does not change while the command is examining files in a workspace. Read-locks may be obtained concurrently by a number of commands; no Configuring command may write to the workspace while a read-lock is in force. See *lock, write-lock.*

Reparent To change the parent of a child workspace.

Resolve

To produce a new delta of a file from two conflicting deltas. See *merged, conflict*.

Root directory

The top-level directory of a Configuring workspace. This directory's path name is the name by which the workspace is referred.

SCCS Delta ID (SID)

A SID is the number used to represent a specific delta. This is a two-part number, with the parts separated by a dot (.). The SID of the initial delta is 1.1 by default. The first part of the SID is referred to as the release number, and the second, the level number. When you check in a delta, the level number is incremented automatically.

SCCS Mergeable ID (SMID)

A SMID is a a number generated using the Xerox Secure Hash Function that ensures that every delta is uniquely identifiable, even if its SID is changed

SCCS history file

The file that contains a given file's delta history; also referred to as an "s-dot-file." All SCCS history files must be located in a directory named SCCS, which is located in the same directory as the g-file. See *g-file*.

SID

SCCS delta ID—The number used to represent a specific SCCS delta.

SMID

SCCS Mergeable ID—The number that ensures that a specific SCCS delta is uniquely identified.

Undo

To return a workspace to the state it was in before the most recent Bringover or Putback transaction, thereby "undoing" the action of the transaction.

Update

Files are said to be updated during a Bringover or Putback transaction if they exist in both the source workspace and in the destination workspace, and have changed in the source workspace. The SCCS history file in the destination workspace is updated with new deltas from the source workspace.

Version

When you check in a file, SCCS records only the line-by-line differences between the text you check in and the previous version of the file. This set of differences is known as a delta. The file version that you initially checked out was constructed from a set of accumulated deltas. The terms delta and version are often used synonymously; however, their meanings are not the same. It is possible to retrieve a version that omits selected deltas. See Delta.

Versioning

The TeamWare program that provides a graphical interface to SCCS. See the section on Versioning.

Workspace

A workspace is a specially designated (but standard) directory and its subdirectory hierarchy. Usually each developer on a project works in their own isolated workspace concurrently with other developers programming in other workspaces. Configuring provides utilities to "intelligently" copy files from workspace to workspace.

Workspace hierarchy

A hierarchy of parent and child workspaces in which programmers and release engineers can develop, test, share, and release software products.

Write-lock

A lock that is obtained by a Configuring command that changes data in a workspace. Only one write-lock may be obtained for a workspace at any time. When a write-lock is in force, only the Configuring command that owns the lock may write to the workspace; other commands cannot obtain read-locks from the workspace. See *lock*, *read-lock*.

Index