Spring Source Code Control: Issues and Solutions

Douglas Niehaus
atools:cook.tex 1.2

April 1989

Contents

1 Introduction **2**

2 Version Control Theory **2**

2.1 Files, Units, and Unit Mapping 2
2.2 Products and Slists 5
2.3 Multi-Directory Products 6
2.4 SCCS Data Base Interactions 7
2.4.1 Get Group 8
2.4.2 Edit Group 9
2.4.3 Delta Group 10
2.4.4 Query Commands 11
2.4.5 Adding Files to the Data Base 12
2.5 Makefiles 12
2.5.1 Makefile Structure and Function 12
2.5.2 Useful Targets 14
2.6 Software Manufacturing 15

3 Specific Scenarios: The Cook Book **17**

3.1 Creating a Product 17
3.2 Multi-Directory Products and Makefiles 18
3.3 Fetching Existing Files 18
3.3.1 Getting a Specific File 19
3.3.2 Getting All Files in a Unit 19
3.3.3 Getting the Slist 20
3.3.4 Populating the Product 20
3.4 Editing Files 21
3.4.1 Editing a File 21
3.4.2 Editing the Slist 22
3.4.3 Deciding not to Edit Any More 22
3.4.4 Checking On What is Being Edited 23
3.5 Creating New Versions of a File 23
3.5.1 New Versions of Files 24
3.5.2 Unit Level Actions 25
3.5.3 New Versions of Products 26
3.5.4 Creating a Branch Line of Development 26
3.5.5 Merging Branch Lines of Development 27
3.6 Miscellaneous Commands 28
3.6.1 File and Product History 28
3.6.2 Unit Composition and Directory to Unit Name Association 29
3.6.3 Adding a File to a Unit 29
3.6.4 Extracting the SCCS Label from a File 29

4 Future Development **30**

A SCCS Command Reference Sheet **32**

B Example Makefile **33**

1 Introduction

The Spring project will require extensive development effort by a number of people over an extended period of time. The development will almost certainly be sporadic and carried on by a changing group. Further, even those people working at the same time are likely to be working on different sections of the system, needing to combine their separate lines of development only at intervals.

All of these factors indicate the need for a reliable and flexible method of controlling and tracking the evolution of the source code. This paper describes such a system, and gives detailed descriptions of particular scenarios. The first section discusses the problems of source code control at a more abstract level, and the second section gives specific commands that deal with the situations described.

2 Version Control Theory

The method of source code control described depends on the proper coordination of a number of concepts. New terms are used to describe files and procedures used to produce a program, and these terms are then related to particular tools and techniques. Any particular method described is always only a choice among many alternatives. The crucial question is whether the set of choices made cooperate with one another properly. Thus, while the reader may question a particular choice, he or she should remain aware of how that choice and its alternatives would interact with other parts of the system.

2.1 Files, Units, and Unit Mapping

The fundamental item controlled by the SCCS system is the text file. All files are considered as text, and may contain arbitrary characters. There is a limitation that lines in the text file, delimited by the newline character, may not start with control-A. This results from an implementation choice embedded in the tools. Each file is controlled individually, and each version of the file ever held by the data base is available on command. A file is uniquely identified by a label containing three fields:

unit-name:file-name SCCS-ID

The unit-name is a symbolic label for the set of files in a particular working directory. Each set of files associated with the same unit name are considered as a group by the SCCS data base tools. This is discussed in greater detail later.

REL.VER

REL.VER.BRANCH.BR_VER

Figure 1: SCCS ID formats

The file-name is precisely what it seems, the name of the file in question. The SCCS-ID is essentially a serial number uniquely identifying a particular incarnation of the file. It is a two or four field label, as illustrated in Figure 1, where each field may vary from 1 to 9999.

The two field type of SCCS-ID is by far the most commonly used. The first number is called the release number. It is the most significant part of the label, and varies only by user request. It is commonly used to differentiate generations of a system. The version number of a file is unique within a release, and is controlled by the data base tools. Each new version of a file submitted to the SCCS data base is labeled with a version number one higher than its predecessor.

The two field label is sufficient for straight line development on a file. However, development is sometimes conducted along more than one path. This can happen for several reasons; two enhancements to a system that are essentially independent of one another may be developed in parallel and then integrated with one another at a later time. Another reason is that a special version of a product may be required, whose features are not wanted in the basic version.

The four field ID is available to manage branch development. The first two fields are the release and version numbers as before. They describe the version on the *trunk* from which a branch is being created. The branch number permits more than one line of development from a given point on the trunk, and the branch version number tracks development along the branch in the same way that first one tracks it along the trunk.

Thus a particular version of a file is uniquely identified by its ID, whether two or four fields are used. Note that when parallel development takes place along branches, subsequent integration of the independent changes is required to produce a new trunk version incorporating both lines of development.

Since any practical system is composed of many files, tracking a single file is not sufficient. Since development is organized in working directories, the SCCS system also knows about these groups of files, which are called *units*. This explains the unit-name section of the SCCS label. Files in different units may have the same name, since the unit-name file-name pair is used to identify a file.

The unit name is also used to decouple the working directory from the SCCS data base. All interaction with the SCCS data base refers to the unit name. This is done for several reasons.

First, it makes interaction with the data base independent of the location of the working and storage directories in the file system. The separation of the working and storage areas avoids the clutter of a storage directory or storage files in the working space. Second, it makes parallel development much easier. Since access to the data base is independent of the working space location, all access actions are the same. Since they are all the same, interaction with the data base from more than one place becomes easier to manage. Finally, it makes the administration of the data base for large projects much simpler. The storage directories can be concentrated in one section of the file system, and then strongly protected. Other source code control schemes placed the storage files in or near the working directories, thus distributing the data base and making its administration more difficult.

Two features make the unit name easy to use. First, a global environmental symbol, "PUNIT", gives the path name of the unit mapping file. Thus it is easy to create new units, and track new file groups. Second, the SCCS tools make a number of default assumptions that make normal use of the tools convenient. In particular, they look for a file named ".unit" in the current directory, which holds the default unit name for the directory. For example, consider the unit *spring* and its mapping:

spring /usr/src/sccs/spring/s/

This is a line from the global unit name mapping file whose path name is given by the "PUNIT" symbol. The ".unit" file in the main Spring directory contains the unit name "spring". The corresponding path name specifies the directory within which SCCS stores the database of versions for all files in the unit.

This simple method for associating a default unit name with a directory has an important effect. The context of the SCCS commands changes as the developer changes directories. This is appropriate since the developer is changing context with respect to the code as he or she changes directories as well. The default unit name and file mappings will be discussed again in connection with makefiles. Note also that the **set_unit** and **unit** commands are available to establish and query the unit name to directory association.

2.2 Products and Slists

We now know that the SCCS data base tracks files as they evolve individually, that it associates groups of files with a unit name, and that a default unit name can be associated with a working directory. For simple programs this is enough to permit adequate source code control. However, keeping track of the development of simple programs is easy enough *without* any tools at all. More complex programs require additional measures to properly manage them.

The idea of the *product* is used to focus attention where it belongs; on what is being produced. In our case, the Spring kernel and associated tools are the products. In general, the product is whatever is being developed. This idea is deceptively simple. If the product is built in a single directory, perhaps from a single file, then the problem of control is simple. If it is built from many files in many directories, then the problem is more complex. In any case, the *real* object that requires control is the *product*. Tracking changes to individual files is necessary, but incidental, to tracking the product.

It is thus imperative to identify precisely what is required to *completely* describe a product. The source files that are used to build it are obviously required. However, it is also necessary to know how the source files should be combined. This is one reason that the **make** command is so important in the proper control of a product. It is discussed in detail in a later section, but a simple understanding of its role is required to properly explain the *slist*.

The makefile is a text file containing all the instructions required to construct the product from its component source files. As a text file it can be labeled and placed under SCCS control. It is thus possible to track the instructions for assembly of the product even as the evolution of its component pieces is recorded. The **make** command uses file modification times to determine when processing is required, and when it is not. For example, when the source file for a program has a later time than the object file produced from it, recompilation is required.

Each individual file required to build the product can be tracked, but this is not enough. Each version of the *product* is built from a particular version of each component file, but which version is completely dependent on the version of the *product*. An individual file may pass through many versions as a new version of the product is produced. Conversely, a file may remain unchanged through many versions of the product. A way to track the product itself, as distinguished from tracking its component files, is required.

Since the product is built by **make** from its component files, a version of the product is uniquely identified by the set of files used to build it. Since each component file is thus uniquely identified

in the SCCS data base by its file label, a version of the product is uniquely identified by the set of labels from the files used to build it. This list of file labels is called the *slist*, or source list. As a text file it can be tracked by SCCS. The file labels it contains are then used by SCCS to specify the versions of all other files used to build the product. Observe that the label of a version of the slist is sufficient to uniquely identify each version of the product, and to permit its reproduction. Remember also that **make** only considers the time stamps on the file, not SCCS labels. This situation is workable, and is discussed in detail later.

The SCCS tools make many default assumptions about the name and use of the slist which are designed to speed normal development. These conventions are discussed in a later section. For the moment it is important to grasp the fact that the slist is used to specify the versions of *all* the other source files used to build the product, and that it thus controls the product produced completely and reliably.

2.3 Multi-Directory Products

As products increase in size, it is often impractical or inconvenient to build them in a single directory. When this is the case, a product is built in a set of directories organized into a tree. This tree structure is important. It means that no matter how large the tree must be, it is still a self-contained unit whose location within the file system is irrelevant to the construction of the product. As with products produced in a single directory, developers can maintain and control separate copies of the product source in a natural an efficient manner.

The SCCS tools take this approach into account, and support the idea of *unit families*. A unit family is a set of units, each associated with a group of files, which are all required to produce a single product. The family consists of the *controlling* unit and *subordinate* units. A single slist holds the file labels for all files in the controlling and subordinate units. A naming convention is used to tell the SCCS tools where the controlling slist is relative to a given directory. When the subordinate units are named, the UNIX path name style is used, placing slashes between unit name components. The tools count the slashes to determine how many directories above the current one they must look for the controlling slist.

For example, the whole Spring product is contained in six directories comprising a tree of two levels. The controlling unit is "spring" with five subordinate units. The slist is a member of the unit "spring", and is named "spring.sl". The working directory structure reflects the unit names, and SCCS commands executed in any of the five subordinate directories automatically consult the controlling slist. Note that the five subordinate units happen to be children of the "spring"

directory. This is not required. The *storage* directory path is arbitrary, and each unit's files are managed separately.

unit-name	Storage Directory
spring	/usr/src/sccs/spring/s/
spring/filsys	/usr/src/sccs/spring/filsys/s/
spring/io	/usr/src/sccs/spring/io/s/
spring/k2	/usr/src/sccs/spring/k2/s/
spring/kernel	/usr/src/sccs/spring/kernel/s/
spring/trace	/usr/src/sccs/spring/trace/s/

Each directory has its own makefile which is responsible for building the portion of the product produced by the source files in that unit, and for calling makefiles in subordinate directories. Thus, the product as a whole is built by the makefile in the controlling unit processing its local files, and calling the makefiles in all subordinate directories. This modular organization is not required, but is flexible and provides some standardization across products. Standardization of makefile targets gives a developer starting work on an existing product a reliable set of clues about the product's structure.

2.4 SCCS Data Base Interactions

The discussion so far has established a conceptual framework, and introduced some of the basic techniques at our disposal in constructing a code development, control, and construction method. However, the ideas must be used in a way that will give the desired results; well organized and reproducible products. The development and control method described in this paper depends on the integration of the SCCS data base, with extensive use of **make**, into the development cycle. The development method is based on a single driving goal; to verify that the SCCS data base contains all the information required to produce a given version of a product on demand. This level of specification makes the idea of "software manufacturing" possible.

The phrase refers to the idea that the development of a product includes the specification of how it should be assembled. This specification should be so complete that the entire product can be constructed by a third party knowing only standard makefile targets. This level of specification ensures that no hidden assumptions or environmental factors influence the construction of the product. Given the ideas we have already discussed concerning the control of text files, makefile targets, and the slist; it should be easy to see that a product's specification can be preserved in the data base.

Since developers change jobs and entropy increases, it is in the developing organization's best

interest to verify that the specification contained in the data base is *complete*. Manufacture of the software by a third party is an excellent check on the correctness of the product's specification as held in the data base. The third party starts in an empty directory and uses nothing but objects extracted from the SCCS data base to build the product. This is why the *populate* target in the makefile is required to build the subordinate directory structure as needed.

Software manufacturing is discussed in the last part of this section. Before we address that, however, we must consider the interface of the SCCS data base. The second half of the paper is dedicated to discussion of specific scenarios which include examples of the commands discussed here. For the moment we will present the interface on a more general level.

The next few sections each address a particular kind of interaction with the data base. Within each group there are commands to deal with individual files, units as a whole, and slists. These commands have the corresponding suffixes "f", "u", and "sl".

The interface described here is built on top of some existing tools with a less focussed world view. Some are defined as **csh** aliases, and some are shell scripts. The simplicity of the interface also involves limitations. Ideally there are no limitations that will annoy a person developing a product in the normal course of the work. However, if such an unreasonable limitation is found, we can change the tools to eliminate it.

The commands in the simple interface are listed on a reference sheet included as an appendix.

2.4.1 Get Group

The *get* operation fetches existing versions of a file from the data base. The command **getf** accepts file names and consults the controlling slist to determine the versions of the files required. It then looks at the labels of copies in the current directory to see if the required versions are already present. If so, the files are not touched. This is important in the context of the **make** command's way of doing minimal work by checking file modification times. The file is not touched unless a new version is really required.

If the slist was not found in the correct place, it is retrieved from the data base at the last version on the trunk of the development tree. If the "-r" option is given, its argument specifies the version of the slist required.

The **getu** command does essentially the same thing at the level of the unit. It checks the controlling slist, fetching it if necessary, and compares each of the file labels against the versions specified in the slist, replacing those at incorrect versions. Note that, as before, the modification

%W%!@(#)atools:histu.sh 1.2->1.3 by niehaus on 89/02/16!

Figure 2: Editing Information

times of files at the proper version are not changed, preserving the ability of **make** to do minimal work. If the "-r" option is used, it specifies the version of the slist to use. The "-t" option specifies that no slist is consulted, but that the top version of every file should be gotten.

The **getsl** command fetches the slist either at the last version along the main trunk of the revision tree, or the version specified by the "-r" argument. Note that this command and others that fetch the slist can be executed in any directory of a multi-directory product. The commands use the unit naming convention discussed earlier to find the directory of the controlling slist.

2.4.2 Edit Group

The get group of commands operates on file versions that the data base already contains. However, new versions of files must be created on request. The method used by the SCCS commands is to require that the person creating a new version of a file inform the data base of this fact, and receive a writable copy of the file with a special label. This label gives the starting version of the file, the new version that will be created, the person making the request, and the date. This information is illustrated in Figure 2.

The "%W%" is called a *keyletter* by SCCS, and is expanded into the standard file label when a stored version is extracted from the data base. When an editing version is requested, the keyletters are not expanded, but editing information is placed next to the "%W%" keyletter between exclamation marks. This information indicates that the file "histu.sh", a member of the the unit "atools", was taken out for editing at version 1.2, and will create the new version 1.3 when finished. This editing version was requested by the user "niehaus" on the given date.

The **editf** command takes a list of files as arguments and takes copies of them from the data base for editing. This means that a copy of the file with write permission is placed in the current directory. The controlling slist is consulted for the proper base version from which the new version will be made. Thus development along a branch can take place without explicit actions from the developer since the slist is in control. This is important when parallel development is being done for later integration. Note that the *initiation* of a branch line of development *does* require a specific request. The "-r" option can be used to edit a specific version of a file, without reference to the

slist. This is only used when establishing a branch line of development.

Editf, in consulting the controlling slist, also checks to see if it is out for editing. If not, it is taken out for editing automatically. This is done under the theory that since the slist represents the product as a whole, if a new version of a file is being created, then a new version of the product is being created. Therefor, the slist must be taken out for editing. The tools make an effort to determine what slist version is already present, and to take that version from the data base for editing. Under normal circumstances everything works out well. If no copy of the slist is present, then the tools assume the end of the trunk by default. If the developer is working along a branch line of the slist, then this is wrong, and specifying the slist base version for editing is necessary.

The **editsl** command is similar to the **editf** command, but operates only on the slist. If the "-r" option is given then that version is taken out for editing. This command will also look at the current copy of the slist as does **editf**.

It should also be noted that whenever a file is taken out for editing the data base makes a note of who requested it, what version they requested, and what version will be created when they are finished editing. This information is available on request, and is also given to anyone trying to take a file out for edit that is already being edited. It is possible for two people to edit the same file in parallel, but it is generally a good idea for them to know what is happening.

Sometimes, a developer will take a file out for editing, and then decide that this was a mistake. Whatever the cause, the **uneditf** and **uneditsl** commands are available to take back the request for an editing copy of a file set or the slist.

2.4.3 Delta Group

When a file is out for editing, the developer can change it at will, to whatever extent is desired. When it has reached a state where the new version of the file should be preserved, then a command from the Delta group is needed.

The commands exist in pairs, one of each pair creates the new version of the file with the copy in the current directory, and then gets the officially labeled new version from the data base. These commands are **deltf**, **deltu**, and **deltsl**. The other half of each pair creates the version as before, but then takes the new version out for editing again. This is commonly called a check-point, and thus the commands are **chkf**, **chku**, and **chksl**.

As before, the command ending in "f" takes a list of files as arguments, those ending in "u" act on all files in the unit, and those ending in "sl" work on the slist. Thus it is quite easy to create a

new version of a single file, all files in the unit, or preserve the current state of the unit as a whole by creating a new version of the slist.

It is important to remember that as a new version of a file is created, the new version is noted in the slist. This is why the slist must be out for editing when any file it controls is being edited. The delta and check-point commands keep the controlling slist up to date. It is also important for a developer to remember to preserve his or her work in the data base periodically. The general rule of thumb is that a new version of a file should be created whenever the developer has done more work on it than could be accurately reproduced from memory if the editing copy were destroyed. This is the reason for the check-point commands. They make it convenient to preserve intermediate results. Checkpointing is also advisable at the slist level. It gives the developer a known state of the product as a whole to return to if later efforts go awry. Proper checkpoints can save a great deal of time when mistakes happen, as they inevitably do.

2.4.4 Query Commands

The commands described here are useful, but are secondary to the main interactions with the data base. They deal with inquiries made by the developer about the composition of a unit, the history of a particular file, or the files currently being edited in the unit. These commands are generally limited to the unit level at this time. It is possible to consider extending the action of the current tools to include a greater understanding of multiple unit products. The data base tools could work with the units comprising the product as a higher level entity. This is left to makefile targets at the moment. The *populate* target is an example of something presently done in a makefile that could be done by the SCCS tools.

The **lsu** command lists all the files in the current unit. The **editing** and **editing_f** commands list all the files in the current unit that are being edited. The **editing_f** version gives a complete listing of the information held by the data base about the files being edited. The **editing** version just lists the file names.

The **histf** command prints the version history of the specified file on the standard output. This information includes the version number created, the person creating it, the date it was made, and the comment from the person making the version. The **histu** command does the same for the unit as a whole by giving the history of the slist. This command might be more appropriately called **histp** to signify that it gives the history of the product.

The **what** command scans the named files for SCCS labels, and prints them on the standard output. It is thus useful for answering questions about what version of a file or product is sitting

before the requester.

The **unit** command prints out the unit name associated with the current directory. The **set_unit** command associates the current directory with the name given as an argument.

2.4.5 Adding Files to the Data Base

The **addf2u** command adds the named files to a unit. This involves getting them established in the data base, replacing the current copies with properly labeled ones, taking the slist out for editing, adding the new files' labels to it, and creating a new version of the slist. The command does not get the slist out for editing if it is already out, and does not create a new version. This is another potential use for **editsl**. The developer could get the slist out for editing before creating and adding several new files to a unit to avoid creating many new versions of the slist unnecessarily.

Similarly, the **addsl2u** adds an slist to a unit. This should obviously be done first when creating a new product. The **mksl** command creates the proper template for an empty slist to be added to a unit.

2.5 Makefiles

2.5.1 Makefile Structure and Function

The **make** command is commonly known as a method of automating the compilation and assembly of a software product. This is true, but is an unnecessarily limited view. The makefile is a description of dependency relationships between objects, and actions to take when certain conditions are met. **Make** is a generic system capable of a number of important activities including but not limited to product compilation. The full range of **make's** features is given in its manual page and tutorial. Here we are limited to a discussion of how it can be used to enhance software control and production.

A makefile has a simple structure. The basic item within a makefile is the target definition. The definition has the generic form:

```
target: dependency list
<tab>    action list
                ⋮
```

The **make** command assumes that every target names an object. The dependency list gives objects from which the target is made, and the action list is a user defined set of shell commands

that produce the target. Note that each line in the action list must start with a *tab* and is executed as a separate shell invocation. **Make** is picky.

Using a simple strategy, the user can create targets that function either as objects or commands. If a target is an object, then a file of that name is produced by the actions specified in the target's definition. If the target functions as a command, then the actions it specifies do not produce a file with the name of the target. A request to **make** naming a command target instructs it to perform the associated actions. This distinction is important because of how **make** decides whether or not to execute the actions associated with a particular target.

Make uses temporal relationships between target *objects* and their dependencies to decide whether to execute the targets' associated actions. When **make** considers a target, it looks to see if an object of that name exists. If it does not, then **make** executes the associated actions, which it presumes, will create the object. This is how a target functioning as a command works. Since it does not produce an object, whenever the target is specified its actions will be executed, thus functioning as a command.

If an object with the target's name *does* exist, then another feature of **make** becomes important. **Make** is designed to try and do *minimal work*. In the context of software production this means processing only those files required to produce the new version of the product. When a system is being built for the first time, this obviously requires processing all files. However, for very large systems this can require hours of computer time. When a change is made to a single file, it is possible that only that file needs to be recompiled, and then the product relinked. On the other hand, if the changed file is a global header file referenced by every code file, then all files must be recompiled.

When the **make** command is called, it is either given a makefile name explicitly, or finds one with an implicit rule. The command is also given a list of targets to process. If no target name is given, there is a rule by which it chooses a default. Now consider what **make** does when processing the requested targets.

It looks first at the dependency list for a given target, which specifies the objects from which the target object is made. When **make** considers the target, it checks the object's modification time against the modification time of each of the objects on which it is dependent. **Make** will only execute the actions associated with the target if one of the objects it depends on has a modification time later than the target object's. When the dependencies of all parts of a product are properly specified this feature can obviously save a large amount of time. However, if the dependencies are incorrect, it can lead to errors. Keeping the proper dependency lists in the makefile is a task that

is often neglected, and is thus a source of error.

The SCCS system cooperates intimately with **make** to do minimal work. As will be discussed in detail, the SCCS commands go to great lengths to change the modification times of only those files that have actually changed, thus preserving minimal work. A possible modification to **make** would be to have it use the SCCS file labels as a guide, rather than their modification time.

2.5.2 Useful Targets

It is useful to provide all makefiles with some standard targets used to invoke procedures common to all products. The precise set of targets, and the actions they take, is still project dependent. However, the general intent of the standard targets will be constant across projects.

The target set we will discuss is:

populate This target functions as a command, and calls **getu**, thus making sure every file in the current unit is at the right version. The target is also responsible for creating subordinate directories and calling their makefile's *populate* targets. This means that the controlling unit's makefile can populate the source code for the entire product.

product This target is dependent on the objects produced by the unit. Thus it will apply the rules of minimal work to its dependencies, and theirs, to determine what must be done to bring the product up to date. The answer may be that nothing needs to be done.

install When the product has been built, this target causes the constructed objects to be placed in the officially sanctioned spot. In the case of the SCCS commands, for instance, this target copies them into a *bin* directory making them publicly accessible. It is also responsible for calling the subordinate makefile's product target.

clean This target removes all the product objects. It is used as a safety or convenience measure. It is particularly useful when a developer is debugging the makefile itself. As before, it should call the same target in subordinate makefiles.

clobber This is a more aggressive version of clean, removing all but the source files.

lint This illustrates another target functioning as a command. For products made from C source files, this target would run the **lint** utility on them when activated.

editing This target is similar tot he **populate** target. It calls the *editing* command for the current units, and is responsible for calling subordinate makefile's **editing** targets. This gives the

editing state of the product as a whole when done from the controlling unit.

2.6 Software Manufacturing

The term is meant to stress the routine, predictable, and automatic nature of how finished software should be produced. Software development is still in large measure a craft, but there are many aspects of a product that can and should be reduced to routine. Creation of the product from its source files is one of them. Automation of the procedure has many beneficial effects:

- Mental effort of the developer is preserved for issues requiring judgement and thought, instead of typing commands and remembering the state of the software to decide what happens next. Automation of the procedure permits the developer to focus his or her attention elsewhere while the software is being produced.
- The **make** command can determine the minimal work required and substantially decrease the product construction time when changes are made to only a few files in a large product.
- Organization of the product into an autonomous set of files arranged in a directory or directory tree independent of its location in the file system gives flexibility. Since a product is built only within its autonomous directory structure, that structure can be located anywhere, and duplicated without having the copies interfere with one another. This permits parallel development and third party manufacture of the product.
- The transition from one developer to another is aided by a detailed product description in a standard format. This preserves and presents a great deal of information to the new developer.
- When the product makes the transition from pure development to testing and support, the manufacture of the product in a separate environment *proves* the completeness and correctness of the product description held in the data base.
- The preservation of a provably correct product description in the data base ensures that no version of a product will be unsupportable at a future date. This is very important for long lived products in either the commercial or academic environment.

In the final analysis, the chief virtue of the software manufacturing approach is that it provides objective evidence on the correctness of the product from a constructive point of view. If the product is manufactured using only information held in the data base, this proves the correctness of the specification held there, with respect to producing a particular version of the product. It says nothing, of course, about the correctness of the product from a functional point of view.

An even higher level of confidence can be achieved if independent manufacturing is the only interface between the developers and testers. This organization uses the SCCS data base as the information exchange medium between the development group and the outside world. It is a simple idea that automatically eliminates a wide range of standard problems.

The manufacturing group provides a routine but invaluable service. They are responsible for satisfying all requests for software. Each request asks for a particular version of the product. It should be obvious that this is the slist version. Since the slist contains a complete description of the product, it can be accurately reproduced on demand.

It is a simple matter to include the slist label in all objects that make up the product. This label is easily retrieved from the objects with the **what** command. This object labeling has two important effects. First, it means that all objects produced with this method carry with them all the information required to reproduce them *perfectly*. Not almost, pretty close, or just about; perfectly.

Second, testers and developers now have objective labels about which to engage in debate. The familiar problem of testers using the wrong versions unknowingly is now eliminated. The discussion between a tester and developer is very simple.

```
Is the version tested the proper version.
If not, what is the proper version.
If so, then either it works properly or the developer has to fix it.
```

No debate, no finger pointing, no confusion, no time wasted. The software in the tester's directory can be completely identified in seconds.

The objective reliability of this system can be maintained by following two simple rules; all products must be distributed by the manufacturing group, and all software is transmitted from development to the manufacturing group through the SCCS data base. The handoff of a new version of any product, no matter how large, can be done with a post card. The developer simply gives the unit name and slist SCCS-ID of the new version to manufacturing. They build and hand it out to the testers.

It should be noted that this approach is not limited to projects large enough to support separate groups for development, manufacturing, and testing. The same advantages can be had if separate environments are used for development and testing and only the data base is used to transfer information between them. Human nature being what it is, separate development and manufacturing machines are advisable but not required.

3 Specific Scenarios: The Cook Book

The first section of the paper described the basic theory underlying source code control, and the interface currently available. This section describes the specific commands, and gives examples for situations described more generically in the first half of the paper.

Each section discusses a particular situation, and explains how the tools can be used to solve the problems associated with it. It then gives a step by step example illustrating the subject under discussion.

Note that in the examples commentary and data base commands are freely mixed. Lines beginning with a > are commands and those beginning with a bullet or a dash are commentary.

3.1 Creating a Product

A product must be based on at least one unit. The unit name is the symbol used by the SCCS data base tools to refer to a group of files the all inhabit the same directory. Many, even most, products are built from files in a single directory. The case of a product organized in a directory tree will be discussed in the next section.

A unit name must be chosen. The global unit mapping file must have an entry placed in it, and the files must be placed under control of the slist and entered in the SCCS data base. This requires a little boot strapping.

- Choose a unit name.
- Go to the Sccs data base area and create a storage space for the unit, and add its mapping to the global mapping file.

  > cd /usr/src/sccs

  > mkunit *unit-name*

  – edit glob_unit.dat, following examples exactly. Note that this file is *itself* under SCCS control, so refer to the instructions about how to take file out for editing, and create a new version of it. The mapping line would look like.

  unit-name /usr/src/sccs/*unit-name*/s/

- Go to the directory where the source files you wish to place under control are stored.

  > cd *source-directory-name*

- Label the directory with the unit name.

  > set_unit *unit-name*

- Create an slist template.

  > mksl >*unit-name*.sl

- Add it to the unit.

  > addsl2u *unit-name*.sl

- Add all the files to the unit. Including the makefile. Note that the files should contain the %W% key letter in a comment at the start of the file. A makefile template is given in the appendix and in "/usr/src/sccs/make.tpl".

  > addf2u makefile *file-list*

- The unit is now ready for development under control.

3.2 Multi-Directory Products and Makefiles

The same principles apply to a multi-directory product as do to one in a single directory. However, there are additional factors. First, each directory must have a unit name and mapping. Second, the subordinate units should be constructed like UNIX paths to permit the tools to find the controlling slist automatically.

The controlling unit should be created first, and the slist, makefile, and source files added to it. Then the subordinate directories and files can be dealt with. Note that the *populate* and *editing* targets in the makefile should deal with the subordinate directories as discussed. Consult the makefile template for examples.

3.3 Fetching Existing Files

Here we consider the variations on taking an existing version of a file from the data base. The main choice is between operations on individual files, units, or products as a whole, for multi-unit products. The other choice is between letting the slist control the version required, or requesting one directly.

3.3.1 Getting a Specific File

- The simplest possibility is simply asking for the version specified in the slist.

  > getf *file-list*

- On the other hand, we might want a specific version of the file, regardless of what the slist says.

  > getf -r*SCCS-ID* *file-name*

- Finally, if one is comparing versions of a file with diff, the following kind of thing is useful. First get the version specified by the slist into the regular name. Then fetch the version we wish to compare it with into a scratch file. Finally, diff the two. Obviously this generalizes into two arbitrary versions each in a scratch file. The "-p" option tells the command to put the text of the file on the standard output.

  > getf *file-name*

  > getf -r*SCCS-ID* -p *file-name* >*scr-file*

  > diff *file-name* *scr-file*

3.3.2 Getting All Files in a Unit

- The simplest variation is simply getting all the files in the unit at the versions specified by the slist.

  > getu

- Sometimes we might want the latest version of every file, along the main line of development. Note this qualification, it will get the latest along the main trunk, not along branches.

  > getu -t

- Finally, when trying to do diagnostic work on a product, one sometimes wants the first version of every file. This is done because if we started our development with some source, it is customary to place it in the data base without alteration, thus making it easy to reproduce the starting point exactly.

  > getu -r1.1

  This is equivalent to, and a short way of saying, the following:

> getsl -r1.1

> getu

3.3.3 Getting the Slist

The command that gets the slist can be issued from any directory in a multi-directory product. It acts in the controlling unit in all cases. When getting the slist there are really only two choices; getting the most recent version along the trunk line of development, and getting a specific version. The latter could be for reproducing an earlier version of the product, or if the developer is working along a branch line that would not be selected by default.

- Get the latest version on r the main line of development.

  > getsl

- Get a specific version of the slist.

  > getsl -r*SCCS-ID*

3.3.4 Populating the Product

Since the SCCS commands are, at present, limited to working at the unit level or file level, the makefile must deal with some aspects of multi-unit operations. The *populate* and *editing* targets are the two standards proposed here. The *populate* target has responsibility for creating the directory structure required by the product, and then calling subordinate makefiles. It is clear that further development on the SCCS tools could include more extensive support for multi-directory products.

Since the slist label uniquely identifies the version of the product, it must be used to fetch the makefile.

- Make a new working directory wherever required, and go into it.

  > mkdir *dir-name*

  > cd *dir-name*

- label the directory with the unit name.

  > set_unit "*unit-name*"

- Get the most recent trunk line version of the slist. Note that we could substitute a request for a specific version as explained above.

  > getsl

- Get the makefile and cause it to populate the rest of the unit, no matter how large or small.

  > getf makefile

  > make populate

- Note that the same effect for a *single directory product* could be had by substituting the call to the makefile populate with:

  > getu

3.4 Editing Files

A developer edits a file to create a new version. As discussed in the first half of the paper, the developer makes a specific request to the data base for a writable copy of the file. The data base makes a note of the base version being provided, the version ID the new version will have when created, who made the request, and the date. The next section deals with how the developer can create a new version in the data base after editing is complete.

3.4.1 Editing a File

- This is really very simple, with no significant variations. Since the decision to edit a file is made on an individual basis, the request for an editing copy is made on each individual file. Note, however, that since changing a file implies changing the product, and since the slist describes the product, editing a file implies editing the slist. Thus, the request for an editing copy of a file will check the slist, and request an editing copy of it as well, if one is not already present.

  > editf *file-list*

- A specific version of the file can be edited, without reference to the slist, by using the "-r" option. This is generally done when creating a branch line of development for a file. Note also that this use of the command only takes a single file name as an argument. The "-b" option gives the data base permission to create a branch.

  > editf -b -r*SCCS-ID file*

3.4.2 Editing the Slist

Occasionally, one would like to edit the slist as a file, and must thus request to edit it explicitly, rather than implicitly by requesting a file for editing. In that case the command is available. Note that the command will work on the controlling slist from any directory in a multi-directory product. Note also that the command will get the version already present out for editing unless a particular version is requested. This is important for developers working a long a branch line.

- The basic command, which simply gets the current version out for editing.

  > editsl

- And the request for a specific version to edit.

  > editsl -r*SCCS-ID*

3.4.3 Deciding not to Edit Any More

The best laid plans can change. Thus, while sincere at the time about wishing to edit a particular file, the developer may realize it is useless. Or, mistakes so confusing that simply starting over is the best choice may happen. In this case, commands are available to erase the editing copy and replace it with the original gotten version.

The developer should take note of an important point. The tools can automatically tell when they should take the slist out for editing. They *cannot* automatically tell when the developer no longer wishes to edit the slist. Thus, it is the *developer's* responsibility to tell the data base that the editing copy of the slist is no longer needed, since a new version of the product is not being developed.

- If we no longer need to change this file.

  > uneditf *file-list*

- On the other hand, we might just want to start over, with a fresh copy of the file we just accidently destroyed. This is not elegant, but will produce a new copy, newly out for edit, as if nothing had ever happened.

  > uneditf *file-name*

  > editf *file-name*

- The same can apply to the slist. If the developer decides not to change the product this time, or wishes to back out for any other reason, their last step should be to tell the data base that the slist is not being edited either. This command will be effective from any directory of a multi-directory product.

  > uneditsl

3.4.4 Checking On What is Being Edited

As development progresses, jobs of different sizes arise. Some are small, self-contained, and of short duration. Others are not. When the number of details grows too large, people forget what is going on. For this reason, the SCCS tools provide a way to query the data base about files currently being edited. At the present time this is a list of files for which the data base has provided editing copies *to any user*, not a list files being edited by the requesting user. This distinction is important when more than one person is working on a product, and especially when they are working along different development lines. The version of the command giving more complete information can resolve potential confusion. It displays the logname of the person editing each active file.

- The basic command provides a list of the files active in the unit associated with the current directory.

  > editing

- A full listing of the information held by the data base on each active file is available to the developer.

  > editing -f

As with the populate step, this operation is meaningful at the product level for multi-directory products. The implementation of a product wide activity inquiry must at present be implemented in the makefile. Extension of the SCCS tools to include this function could be done, with extension of the tools' awareness of directory trees and unit families.

3.5 Creating New Versions of a File

When the editing is done, or at least when enough editing is done to warrant saving the new version of the file in a safe place, the new version must be given to the data base for processing. The most common misuse of the SCCS tools is creation of new file versions at intervals that are either too

short or too long. The principle involved is simple, but subtle. The data base should contain a coherent history of the file's development. This should include each *significant* new version, without cluttering the trail with frivolous versions. The key word in that statement is "significant", and the ideal can only be approached, never reached.

A new version is obviously required when a new function is present in the code. However, a new version is also appropriate when a significant intermediate level of development has been reached on a file or product. The developer should always remember that the versions held in the data base can be recalled at will. As such, they are a significant safety measure.

Misfortune or confusion can strike at any time. A version recently stored in the data base whose properties are known will frequently be the quickest solution to a confused situation. Also, the earlier version can be compared with the current copy using the **diff** command to determine objectively and efficiently what changes have *actually* been made, not what changes the developer is *sure* were made. Any factor that automates or objectifies an aspect of the developer's task frees his or her mind for other duties.

When a new version of a file is entered into the data base, the developer's logname, the date, and the developer's comment are retained. The comment is of the greatest importance. It creates, over time, a narrative about the development of the file. This is another reason that creating new versions of a file at appropriate intervals is important. When a new feature or small set of features has been added to the file, a new version should be created. This version, as distinguished from those earlier and later, increases the resolution and richness of the information held in the data base. If at some later time another developer, or even the original, wishes to know what was required to implement a certain feature, the history in the data base, and the versions representing incremental progress tell a great deal. In contrast, if two versions in the data base are separated by thousands of changes and weeks of effort, then the differences between the first and the second tell an observer much less.

3.5.1 New Versions of Files

When a new version of a file is created, it is an individual operation, in most cases. Sometimes groups of files are, by their structure, so closely linked to one another that they are almost always edited together. The SCCS tools that create now versions provide support for all possibilities. It is important to remember, though, that the comments associated with a new version are of the greatest importance to the future. Everyone fails to make meaningful comments at some time or to some extent, but the best effort possible should be made.

Each command with the prefix "delt" has a corresponding command with the prefix "chk". The former replace the file with a labeled version, the latter take the *new* version out for editing again. This is a check-point of the file. It is useful when work deserving preservation in the data base has been reached, but editing on the file is not finished. In each of the examples below, the "delt" and "chk" versions are given together.

- Creation of a new version of a file can be requested quite simply. If more than one file name is given, new version of each will be made. The SCCS commands will request comments about the files individually. When the new versions have been created, labeled copies of the files will be placed in the current directory, or new versions taken out for editing as indicated by the command used.

  > deltf *file-list*

  > chkf *file-list*

- If the developer does not with to wait for the comment prompt, or if the same comment is *truly* appropriate to all the files in the list, then the comment can be given on the command line.

  > deltf -y"*comment*" *file-list*

  > chkf -y"*comment*" *file-list*

3.5.2 Unit Level Actions

As mentioned before, sometimes a whole group of files is ready at the same time. If development and testing are finished, then all the files in the unit may be ready at once. As above, the developer can make comments about the files individually or give them all the same comment with a command line argument. As before, a labeled copy of each file is placed in the current directory or a new version taken out for editing.

Note that

- This will prompt for comments on a file by file basis.

  > deltu

  > chku

- This will give every file the same comment.

> deltu -y"*comment*"

> chku -y"*comment*"

3.5.3 New Versions of Products

When the changes made to files that are preserved in the data base collectively represent a new and meaningful state of the product as a whole, then a new version of the product should be preserved. As before, this could be an intermediate stage representing work that must be safeguarded, though it is not complete, or it could be the final product. In any case, since the slist represents the product, a new version of the slist is created as with other files. As before, a labeled copy of the new version is placed in the controlling directory, or a new copy taken out for editing.

Note, however, that it is *not* possible to tell the SCCS tools to create a new version of every file currently being edited in a multi-directory product. This is a possible extension to the tools, but must await further consideration and development.

- This will prompt for a comment.

  > deltsl

  > chksl

- This will use the argument given as the comment.

  > deltsl -y"*comment*"

  > chksl -y"*comment*"

3.5.4 Creating a Branch Line of Development

Branch line development is most commonly done when two or more people wish to develop new features for a product independently. When this is the case, they can each create their own branch from the common base version. They then develop their own enhancements in independent efforts, which will have to be merged later. Further, since all enhancements are not perfectly independent, they may wish to cooperate to some degree.

The branch lines of development are easily established by first creating a branch for the slist, and then for each source file the first time it is edited during branch line development. This requires some care from the developer, but is not difficult. If two developers edit the same file, the SCCS

tools will inform them of this when the file is edited by the second person. This is often a good time for them to talk.

The branch for the slist is established by taking it out for editing an specifying a special flag. It is then most often immediately checked back in to anchor the branch, and note who is establishing it.

> editsl -b

This assumes that the last member of the trunk of the development for the slist should be the root of the branch. If an earlier version is required then the user must specify the version of the slist.

> editsl -b -r*file-ver*

Branches for source files are a little complicated. For reasons that are a little obscure, the slist must be out for editing when the file is taken out. This scenario would go something like this:

> editsl -b -r*file-ver*

> editf -b *file-list*

> chkf -y"establish branches" *file-list*

> chksl -y"establish branche"

The **editsl** command announces the intention of the user to create a branch line of development for the product. The **editf** command then creates a branch for each of the named files. The **chkf** command then establishes the branch for each of the files in the data base and in the slist. The **chksl** command then establishes the slist branch in the data base. From then on the developer would have to keep track of the slist version at most. Note also that the "-b" option is required whenever a file is edited for the first time along the branch line of development.

3.5.5 Merging Branch Lines of Development

When two or more developers have finished their new system features independently, they must be merged into a new composite version of the product. There is no automatic way to do this since a comparison of the text changes along each line of development gives no clue as to the proper way

to merge them. Text based merging schemes assume that the two lines of development on the same file are completely orthogonal, an assumption which is clearly unreliable. Thus there is no realistic alternative to a code review. However, the SCCS tools can help by specifying accurately what files have been changed, and by whom. A list of files edited by each line of development is obtained by using the **diff** command to compare the slist at the base of the branches to the slists at the end of each of the branches.

This specifies the files that changed along each branch. The text changes made could be found by using **diff** to compare the text of each changed file. One file the version at the base of the branche, and the other the version at the end of each branch.

3.6 Miscellaneous Commands

The commands described in this section fill out the list of data base interaction functions. They are those left over after describing the commands central to getting using the data base to get existing versions, and create new ones. They mostly fall into the category of inquiry operations, but not all.

3.6.1 File and Product History

When a developer is starting new work on an existing product, or resuming work after a long break, a review of file and product histories is in order. The information stored on each new version created in the data base is available to the user on request. The history of the product is of course the history of the slist. The information of the greatest use is the set of comments narrating changes to the files, each supplied by the developer as new versions were created.

- We can get the history of a single file with a single command. The output can be piped through more, or redirected into a file. This is advisable for files with a long history, as the output can be quite large.

  > histf *file-name*

- The history of the slist can be requested directly as the history of the file, or with the unit history command.

  > histu

3.6.2 Unit Composition and Directory to Unit Name Association

Usually the association between a directory and a unit name will be obvious. However, two commands make it easy to check and change the association:

- Print the unit name associated with the current directory:

  > unit

- Set the unit name associated with the the current unit:

  > set_unit *unit_name*

Occasionally a developer is curious to know the names of all the files under stored in the data base under the current unit. Extracting all the entries in the controlling slist that have the unit name in question will answer the question for the current version. However, sometimes the set of files used for a product changes.

- The list of all files under control, even if they do not appear in the current version of the slist.

  > lsu

3.6.3 Adding a File to a Unit

The **addf2u** command adds the named files to a unit. This involves getting them established in the data base, replacing the current copies with properly labeled ones, taking the slist out for editing, adding the new files' labels to it, and creating a new version of the slist. The command does not get the slist out for editing if it is already out, and does not create a new version. This is another potential use for **editsl**. The developer could get the slist out for editing before creating and adding several new files to a unit to avoid creating many new versions of the slist unnecessarily.

- The command takes a list of file names. Note that each file must contain the key letter %W%. This is expanded into the SCCS file label when a particular version of the file is retrieved.

  > addf2u *file-list*

3.6.4 Extracting the SCCS Label from a File

When a developer has a question about the version of a file, there are usually two options. They can look inside the file and read the label, or they can use the **what** command. When the file in

question is not a source file, but is a product that has been labeled with the file label of the slist, then **what** is the only option. It echoes the file label onto the standard output.

- The command takes a list of file names.

  > what *file-list*

- It can also be asked to echo only the file labels for each file in the list, and not the file name as well.

  > what -i *file-list*

- It can also be asked to echo only the SCCS-ID of the files on the list. This can be useful if the developer is using the command in a script. Note that this option only makes sense when applied to a single file.

  > what -r *file-name*

4 Future Development

Work on the source code control tools is not the purpose of the Spring project. This being the case, I have tried to limit myself to work that was essential to constructing an interface that other Spring developers could learn and use with a minimum of study. I had to balance simplicity against functional requirements, and only actual use will tell how well I have done. Changes will be made as the user community discovers flaws in the interface described here.

Despite the utilitarian purpose of this development effort and document, there are some software engineering issues of interest. This section describes extensions to the present tools that would be of more theoretical interest, than of practical value to the Spring project. They are described both for general edification and as a note to the future.

The first enhancement is fairly small, but would make the tools more elegant. I would propose that the tools provide a way to associate a directory with a default unit name without having to create a file in the directory. This could be done by the creation of a new command for the purpose. It would add and change entries in a system wide table that made the pairing much as the unit name mapping file pairs unit names and storage areas. One problem with this would be knowing when a directory was destroyed and an entry no longer required. Lacking this, the table would grow without bound, slowly perhaps, but without bound nonetheless. One way to fix this might be to add an entry in the system *cron* table to execute a pruning command every day or so. The

pruning command would go through the table and verify that the directory existed. Even this could get messy, though, if the directory was in a remotely mounted file system that was temporarily unaccessible, but would return. The individual commands would obviously check the table armed with the current directory path to see what the default unit name was.

A more ambitious suggestion is to add a way to store information in the data base about the directory structure of multi-directory products. This would obviously include information about the unit family and the "spatial" relationships between them. This would enable us to take the multi-directory SCCS commands out of the makefile and add them to the command set. It would also permit the creation of some new multi-directory, product-wide commands. Population of the product could easily include creation of the associated directory structure. The product-wide editing activity would be another command. An option for listing all active files, or only those requested by the person executing the command, or a specified user name could be added. Commands to manipulate the products directory structure as represented in the data base would obviously be needed.

The most ambitious suggestion would be to rewrite the **make** command to be an integral part of the SCCS command set. There are a number of useful enhancements to **make** that are unrelated to SCCS. These are mentioned elsewhere. The SCCS issues fall into two categories; product population on demand, and dependency calculations using SCCS IDs rather than file modification times. This has a very complex set of implications. Among them are the need to propagate the file labels of all source files into the object they are used to produce. Then, when **make** does the dependency calculation to see if an object is out of date with respect to those files used to produce it, the command would be comparing the present file label of each source file to the label propagated into the derived object. This would be relatively easy for some types of files, and difficult or impossible for others. The only method that would probably serve for all files would be to alter the file system as supported by the OS, and figure out a way for it to associate a list of file labels, of arbitrary length, with any type of file.

Other efficiency issues come to mind. The present character based method of representing file labels is quite wasteful. It is necessary for the human interface, but more efficient schemes could be imagined. N-bit fields for unit, file, and ID would be efficient. The data base would then have to expand them when presenting them to a user. One subtle point here is that the character version of the slist label should be place *in the body* of products distributed to the field so they can be comprehensible to those extracing them in the field. This may not be strictly necessary, but this innovation does have widespread implications and effects.

A SCCS Command Reference Sheet

Command	Options	Function
addsl2u		Adds an slist to a unit
addf2u		Adds a file to a unit, and the slist
chkf	-y“comment”	Check-point a file
chku	-y“comment”	Check-point each edited file in the unit
chksl	-y“comment”	Check-point the slist
deltf	-y“comment”	Create new file version
deltu	-y“comment”	Create new version of each edited file in unit
deltsl	-y“comment”	Create new slist version
editf	-b -rSID	Take a file list out for editing
editsl	-b	Take the slist out for editing
editing	-f	List files in unit being edited, short or full form
histf		Give modification history of the file
histu		Give modification history of the slist
getf	-rSID -p	Get file, at specific version, or onto stdout
getu	-t	Get each file in the unit at the top or slist version
getsl	-rSID	Get the slist at the top or specified version
lsu		List all files in the unit
uneditf		Erase editing record of files
uneditsl		Erase editing record of slist
unit		print unit name associated with current directory
set_unit		set unit associated with current directory

B Example Makefile

```
#
# %W%
#
# This is the makefile example
# It is no substitute for reading the manual page and
# the make tutorial, but it is a start
#
# This file makes three programs, all built from
# a single source file and linked to a library
# which is built in a subordinate directory
#
# First we create some Macros
#

UNIT = foo
LIB_UNIT = foo/lib
INSDIR = /usr/local/bin

#
# the all target serves to group p1 p2 and p3, which could
# be made separately
#
all: p1 p2 p3

#
# These three targets make the programs
#
p1: p1.c p1.h common.h libdir/lib.a id.o
        cc -o p1 p1.c libdir/lib.a id.o

p2: p2.c p2.h common.h libdir/lib.a id.o
        cc -o p2 p2.c libdir/lib.a id.o

p3: p3.c p3.h common.h libdir/lib.a id.o
        cc -o p3 p3.c libdir/lib.a id.o

#
# Here we let the subordinate makefile
# do the work
#
libdir/lib.a:
        cd libdir; make lib.a

#
# Here we make a compilable form for the id
# of the slist, and make it available to be linked
```

```
# into the products
#
id.c:
	mkid $(UNIT).sl > id.c

id.o: id.c
	cc -c id.c

#
# this is a command target that runs a test
#
test:
	p3 ¡test_input

#
# populate populates the current directory, then goes into
# the library directory. There it makes sure the unit name
# is in the .unit file, gets the makefile, and runs its
# populate step
#
populate: libdir
	echo "$(UNIT)" >.unit; getu
	cd libdir; echo "$(LIB_UNIT)" >.unit;
		getf Makefile;
		make populate

#
# Note that the populate step depends on the existence of
# the directory libdir, if it does not exist, this will create it
#
libdir:
	mkdir libdir

#
# this is easy to understand. The @ signs keep the command lines
# from being echoed, and messing up the format of the editing
# summary. A good trick
#
editing:
	@echo $(UNIT)
	@editing
	@cd libdir; make editing

clean:
	-rm -f p1 p2 p3

clobber: clean
	-rm -f *.o
```

```
	cd $(LIBDIR); make clobber
#
# The install step moves the manufactured stuff to the default
# installation directory

install:
	rm -f $(INSDIR)/p1
	cp p1 $(INSDIR)/p1
	chmod 555 $(INSDIR)/p1

	rm -f $(INSDIR)/p2
	cp p2 $(INSDIR)/p2
	chmod 555 $(INSDIR)/p2

	rm -f $(INSDIR)/p3
	cp p3 $(INSDIR)/p3
	chmod 555 $(INSDIR)/p3
```