The Evolution of a Source Code Control System

Alan L. Glasser

Bell Laboratories
Holmdel, New Jersey 07733

The Source Code Control System (SCCS) is a system for controlling changes to files of text (typically, the source code and documentation of software systems). It is an integral part of a software development and maintenance system known as the Programmer's Workbench (PWB).

SCCS has itself undergone considerable change. There have been nine major versions of SCCS. This paper describes the facilities provided by SCCS, and the design changes that were made to SCCS in order to provide a useful and flexible environment in which to conduct the programming process.

1. INTRODUCTION

The Source Code Control System (SCCS) is a system for controlling changes to files of text (typically, the source code and documentation of software systems). It provides facilities for storing, updating, and retrieving all versions of a file of text; for controlling updating privileges; for identifying the version of a retrieved file; and for recording who made each change, when and where it was made, and why.

SCCS, a system for controlling change, has itself undergone considerable change. Nine distinct versions of SCCS have existed. The first five versions were implemented for IBM's OS, and the last four versions were implemented as a collection of programs that run under the PWB/UNIX* time-sharing system [8]. This paper is primarily concerned with the PWB versions of SCCS.

The PWB [4, 5] differs from most program development facilities in that, for many projects, program development and execution of the resulting programs take place on two different machines: one that is best for program development (PWB), and one that is best for the execution of the production system (called a "target"). The PWB provides a single, uniform programming environment, even across projects that run on different target systems.

SCCS is used extensively within Bell Laboratories. For example, at one installation more than 3 million lines of source code are controlled with SCCS. These 3 million lines represent 5000 files and the work of approximately 500 programmers. Additionally, SCCS is used at many installations outside the Bell System.

Projects that use SCCS have more than one customer, and are usually in the process of enhancing their software products. Also, these projects usually support at least two versions of the product, the old reliable version and the new, enhanced but possibly unreliable version. When a customer discovers a bug in the old version, the project can't merely deliver the new version of the product. The bug must be fixed for all supported versions. The only acceptable method of correcting a bug is to change the source code and re-compile. Modifying an executable form of a module is not allowed. SCCS attempts to solve many of the problems that result from in working in the above described way. This paper describes the facilities provided by SCCS, and the source code maintenance problems that we have discovered and solved with SCCS.

* UNIX is a Trademark of Bell Laboratories.

2. FACILITIES

2.1 Basic Use

Examples of SCCS usage are given in [3, 5, 7]. The following is a typical example of use:

> A programmer is informed that program *monthlyrpt* has a bug. A copy of the program is requested from SCCS by executing the command:
>
> ```
> get —e s.monthlyrpt
> ```
>
> The —e indicates that the person intends to edit the retrieved version, and add the change (called a *delta*) to the SCCS master copy. The programmer then proceeds to edit the retrieved file, compile it, and test it. The edit—compile—test cycle is usually repeated a number of times. When the bug is fixed, the change is added to the SCCS master copy by executing the command:
>
> ```
> delta s.monthlyrpt
> ```
>
> This command adds the change to the SCCS master copy of the file. The change is identified by the name of the programmer, the current date and time, and a programmer-supplied reason for the change.
>
> At this point the programmer would take the necessary steps to deliver the fixed program to the appropriate customers.

The additional effort the programmer must spend in order to use SCCS is quite small. It should be noted that the above example is indeed typical of the use of SCCS.

2.2 Protection

SCCS relies on the capabilities of the PWB/UNIX time-sharing system for most of the protection mechanisms needed to prevent unauthorized changes to SCCS files. SCCS provides protection mechanisms for controlling which people may add deltas to a module, and what releases of a module may be changed. In addition to programmers who change source code, SCCS allows for an *administrator* (sometimes called a *program librarian*) who is responsible for controlling the updating privileges of each module for a given project. The degree of control employed is regulated by the administrator. The administrator has an *admin* command with which to perform these tasks. The operating system provides the protection necessary to prevent anyone but the administrator from changing this information.

2.3 Change History

A *prt* command is used to format and print all or part(s) of an SCCS file. This command is used most often to print the change-history of a module. The following is a typical, *prt* generated, change-history:

```
s.dodelt.c:

1.9    77/10/10 14:08:46    leb   9  8   00004/00001/00126
file format checks

1.8    77/06/13 13:55:06    leb   8  7   00001/00001/00126
Changed s.h to defines.h in "includes"

1.7    77/03/18 14:44:22    alg   7  6   00001/00001/00126
sidp == 0 means don't do any rmchg stuff

1.6    77/03/17 11:22:08    leb   6  5   00003/00001/00124
indicator for first time escape to escdodelt

1.5    77/03/17 10:29:02    leb   5  4   00009/00002/00116
call escdodelt only when processing wanted delta

1.4    77/03/11 14:23:56    leb   4  3   00002/00000/00116
still more

1.3    77/03/09 15:20:24    alg   3  2   00001/00000/00115
more of the same

1.2    77/03/09 14:23:02    alg   2  1   00012/00002/00103
chged to accommodate rmdel and chghist

1.1    77/02/25 15:26:28    leb   1  0   00105/00000/00000
```

The first line of each entry contains the change level, date and time of the change, login name of the programmer who made the delta, serial number of this delta and its predecessor, and the number of lines inserted, deleted, and left unchanged by this delta. Following this is the programmer-supplied reason for the change. For many users, this automatic bookkeeping function is the most important feature of SCCS.

2.4 Change Regeneration

The *sccsdiff* command prints the differences between any two versions of an SCCS file. This program allows one to determine exactly *what* a particular change is, and *where* in the file that change was made. This capability is most useful when one must fix another person's change and that person is unavailable for consultation.

2.5 Use with Other PWB Tools

The UNIX pipe [8] is a facility to connect the output of one program to the input of another.* As most UNIX programs fit easily into multi-process pipelines [6], designing SCCS commands to be pipeline elements provides SCCS users with many powerful source code manipulation tools. Some examples follow:

2.5.1 SCCS Usage Statistics. The names of SCCS files are normally passed as arguments to an SCCS command. Alternatively, one may supply the names on the standard input. With redirection from files or pipes, one may use a prepared list of names in a file, or use a program to generate a list dynamically. The following command line will print the total number of lines of SCCS controlled text on a given PWB machine:

```
find / −print | get − −p | wc
```

The *find* command prints the names of all the files on the system. The *get* command reads the standard input for its file name arguments, and writes the generated text on the standard output. The "|" symbol is the command language syntax for a pipe [8]. Finally, the *wc* program counts the lines in a file.

* The command interpreter provides the programs that it runs with I/O channels to the user's terminal; these channels are called the *standard input*, and *standard output* [8]. Additionally, the command interpreter can redirect these channels either to or from files or pipes.

2.5.2 Report Generation. The SCCS *prt* command can print the change-history of a group of modules. If one wants this information in reverse chronological order (e.g., a report of all changes made in the past week) instead of module order, one need only connect *prt* to the *sort* program with a pipe.

2.5.3 Interface to Targets. The *send* command is used by programmers to submit batch jobs to a target machine. In addition to the RJE related functions it performs (e.g., ASCII to EBCDIC translation), *send* is a simple macroprocessor with nested file inclusion and keyword substitution. In addition to normal file inclusion, the *send* command can also include "virtual files" (i.e., the standard output of a command) by creating a pipe between itself and the command in question. The following *send* command input shows how one can include a source module in a job stream without first explicitly *get*ing the module:

```
//compprog job ...
//plcomp exec plixc
//sysin dd *
~!get −p s.module
/*
```

The "~!" tells *send* to treat the rest of the line as a command, create a pipe, pass the line to the command interpreter, and replace the line with the standard output of the command.

These three examples are representative of the ways in which SCCS can be used with other PWB tools; [3] contains more examples. Instead of one program that contains all the features used in the above examples, SCCS is a group of simple programs that can easily be connected to other simple programs when elaborate results are needed.

2.6 Source Code Change Statistics

Although the primary use of SCCS is to control changes to source code, the last few versions of SCCS have had a number of features added that allow one to gather source code change statistics in a simple manner. For example, the recording of the number of lines inserted, deleted, and left unchanged for each delta is a feature that exists only in the current version of SCCS. Histograms of the number of changes versus the number of modules having that many changes, the sizes (in lines) of modules across a period of time, and the frequency of change for a given module or a histogram for a set of modules can all be easily generated.

The following table is a histogram of the number of programmers versus the number of files which have had deltas made by exactly that many different programmers. The data was extracted from a project of 65 programmers. The sample examined contains 1700 files representing 500,000 lines of source code.

Number of Programmers (X)	Number of Modules With Deltas by Exactly X Different Programmers
1	441
2	759
3	372
4	89
5	29
6	7

The next table is a histogram of the number of deltas versus the number of files containing exactly that many deltas. (For the same project as above.)

Number of Deltas (X)	Number of Modules Containing Exactly X Deltas	Number of Deltas (X)	Number of Modules Containing Exactly X Deltas
1	258	30	7
2	57	31	5
3	470	32	2
4	419	33	1
5	75	34	1
6	56	35	1
7	41	36	2
8	42	37	2
9	32	38	1
10	17	39	3
11	19	40	3
12	21	41	4
13	10	42	1
14	10	43	1
15	18	44	1
16	12	45	1
17	17	47	2
18	9	48	1
19	8	49	3
20	4	50	2
21	5	52	1
22	8	54	1
23	6	56	1
24	8	58	1
25	3	59	1
26	4	68	1
27	9	76	3
28	2	162	1
29	3		

Finally, the last two tables show the changes made to 7 modules of SCCS itself across a 2½ month period.

Total Lines of Code						
Date	3/1	3/15	4/1	4/15	5/1	5/13
Mod A	49	53	52	52	52	52
Mod B	481	489	492	492	492	740
Mod C	338	338	338	338	341	356
Mod D	119	120	120	120	120	120
Mod E	611	615	631	618	618	618
Mod F	496	497	502	502	502	502
Mod G	504	504	509	513	513	513
Totals	2598	2616	2644	2635	2638	2901

Total Number of Deltas						
Date	3/1	3/15	4/1	4/15	5/1	5/13
Mod A	1	2	4	4	4	4
Mod B	2	3	4	4	4	7
Mod C	3	3	5	5	6	7
Mod D	2	3	3	3	3	3
Mod E	2	5	9	10	10	10
Mod F	2	4	7	8	8	9
Mod G	3	5	7	9	10	10
Totals	15	25	39	43	45	50

All of the above tables were generated simply and *automatically*. No human recording of program change data was necessary. The source code change statistics that are readily available from SCCS are often difficult to obtain [1]. In fact, the lack of such data leads to much ad hoc design of support systems, and makes rigid analyses of the programming process difficult.

2.7 Summary

Many more facilities are available [3]. The facilities that have been described are among those that have led to the high level of success that SCCS has achieved with both programmers and managers. The next section examines a number of design issues that have been addressed by the various versions of SCCS.

3. ISSUES IN SOURCE CODE MANAGEMENT

Understanding the dynamics of program change has been an ongoing learning experience. Our goal has always been to make the programmer's job easier, while using storage space efficiently. In the course of implementing the nine different releases of SCCS, we have introduced (and removed) various features in attempting to meet the above goal.

3.1 Change Propagation

When a project agrees to support and maintain one version of a software system while it develops the next version of that system, one encounters the following problem in source code maintenance: when a bug is discovered in the customer's version of the program, it is desirable to fix it *once* and have all of the subsequent versions include the fix. We assume that, as is often the case, the fix for the *next* version is identical to the fix for the customer's version.

3.1.1 Propagation of Deltas.

In the original version of SCCS, deltas were identified by a release number and a delta number within that release (e.g., 2.5). When a change was made in a given release *all* deltas in all *lower numbered* releases were applied. (See [7] for details on the delta storage and accessing algorithm used.) For example, if the time order of three deltas is 1.1, 2.1, 1.2, then whenever release 2 is accessed delta 1.2 is included. This effect is called *delta propagation*. Clearly, this is not *always* what is desired, e.g., suppose delta 1.2 changes a section of the file that was completely re-written for release 2. The resulting release 2 version in this case would usually necessitate an additional delta (2.2) to *undo* any unwanted effects of delta 1.2. However, when delta 1.2 fixes a hitherto latent bug in an otherwise unchanged section of code, the delta propagation feature allows the bug to be fixed *once* and avoids the all too common situation where the same bug reappears with each new release of the program or else in alternate releases (fixed in release 1, reappears in release 2, and finally fixed in release 3). The usefulness of the delta propagation feature thus depends on which of the two above situations occur more often.

A problem of propagation was that the programmers would be unaware of any new release 1 deltas, and then proceed to access 2.1 and discover it to be different from an older (pre-1.2) 2.1. To solve this problem we added a *cutoff date* feature. This allowed an additional criterion for delta application; namely, it permitted delta application only if the delta was made before a specified date. It is interesting to note that the cutoff date feature, which is still available, has had almost no use.

3.1.2 Optional Deltas.

Finding unrestricted propagation harmful, an *optional delta* capability was added to allow the creation of *non-propagating deltas* in lower numbered releases. There were 26 sets of optional deltas, each identified by a single alphabetic character. An optional delta could belong to only one of the 26 sets. When accessing a module a single option letter could be specified. Deltas were applied as before, but all optional deltas *not* in the set specified were excluded. The original intent of optional deltas was to help solve the problem of making "temporary fixes" without causing the temporary fix code be applied in higher numbered releases. Of course, no more than 26 temporary fixes could be made to a single module. Additionally, the bookkeeping of knowing what option letter corresponded to which release was, unfortunately, left to the programmer. An interesting observation has been made on the use of optional deltas: programmers attempted to use optional deltas as a "conditional compilation" mechanism. Unfortunately, due to certain implementation details, optional deltas could not be used in this way. To save the programmers from unnecessary confusion and frustration, the optional delta feature was removed.

3.1.3 Non-propagation of Deltas.

After some experience with delta propagation, it was decided that propagation was bad and should be eliminated. Eliminating propagation solves the problem of deltas in lower numbered releases disturbing higher numbered releases. It re-introduced the problem of having to fix the same bug in the *next* version of the program. The programmer must make more than one delta even if the original (and fixed) code is identical in all releases. Our experience, thus far, has led us to believe that not propagating deltas is better than propagating them. A mechanism is provided to force the application of an arbitrary delta and a complementary mechanism is provided to force a delta *not* to be applied. This mechanism is used to *explicitly* apply a *non-propagating* delta in higher releases.

3.1.4 Eliminating Bug Recurrence.

A message was produced whenever the programmer accessed a version of a module and any deltas in lower numbered releases were not applied. This mechanism helped keep the "bug" from being forgotten, as often occurs in many source maintenance systems. The message could be turned off once one became certain that it would *never* be necessary to be reminded of the bug in the future. Again, the programmer assumed the bookkeeping job of tracking which versions had been accounted for. Unfortunately, the messages proved to be very annoying, and were turned off prematurely. This feature has also been removed. Instead, an audit capability was built to generate an exception report of deltas unaccounted for. The audit can be run as frequently as is necessary (e.g., before delivery of a new release); it solves the problem of forgetting a bug and doesn't disturb the programmer.

3.2 Delta Trees

It is convenient to view the deltas of an SCCS file as the nodes of a tree, where the root of the tree is the initial delta. In all previous versions of SCCS, the only nodes that had more than one successor were those where one successor had a higher release number. Also, no delta had more than two successors. In the discussion above, delta 1.1 has two successors: 2.1 and 1.2. The current version of SCCS is notably different from its predecessors in that it allows the deltas to form an arbitrary tree. Temporary fixes are now added as a new "branch" to the delta tree, instead of as optional deltas. This feature is most useful in a situation where work on different bugs in the same version must proceed independently and in parallel—a situation that is not infrequent when many different customers are supported.

3.3 Identification

Identifying the version of a source file and identifying the version of a source file used to construct a load module are both important capabilities of a source code control system. Without these capabilities it is impossible to determine the version of a source module used to construct a failing load module. SCCS provides these capabilities in a simple, yet powerful, manner. Load modules are identified by placing identifying information in the source code so that it will appear in the load module. The identification information remains with the load module forever. Copying the load module, renaming it, or archiving it to tape have no effect on the identification information. For many systems, the identification information also appears in a memory dump of the program. Source modules are identified by embedded *identification keywords*. An identification keyword is an upper-case letter enclosed by percent (%) signs. The *get* command will replace identification keywords by appropriate values wherever they occur in the generated text. For example, "%I%" is replaced by the change level, and "%M%" is replaced by the module name. Thus, executing *get* on an SCCS file that contains the line:

```
    DCL ID CHAR(100) VAR INIT('%M% %I%');
```

gives, for example, the following:

```
    DCL ID CHAR(100) VAR INIT('MODNAME 2.3');
```

The "%W%" identification keyword is replaced by the string "@(#)", followed by the module name, a tab, and the change level. The *what* command is available to search files for the string "@(#)", and print what follows that string on the standard output. The "@(#)" string was chosen to be unlikely to appear in a load module by chance. This command thus automates the extraction of identification information.

3.4 Efficiency Considerations

Software system designers are always concerned with efficiency, and those who design source code control systems are no exception. However, two efficiency considerations in the implementation of early versions of SCCS later proved to be of dubious value.

Most PWB/UNIX time-sharing system files are text files. However, the first PWB implementation of SCCS used non-text (i.e., binary) files. This allowed change level numbers, dates, and other various numerical data to be represented as binary numbers, instead of as ASCII digit strings. The inability to use the many PWB *text file* manipulation tools proved to be far more costly than the conversion time saved.

When *get*ing an SCCS file with the intent of making a delta, the *get* command creates an auxiliary file that contains the change level requested, the change level of the delta to be made, and the login name of the programmer who made the delta. Again, in the first PWB implementation of SCCS this file contained binary data. The contents of the file were a copy of a complex data structure computed by *get*. *Delta*, which also needed this data structure, didn't have to re-compute it. However, this made it quite difficult to examine these files (to determine who is making the delta, and at what change level), and even more difficult for an administrator to change this information.

The current version of SCCS uses text files for both auxiliary files and SCCS files. These changes have increased the utility of SCCS with no noticeable degradation of performance.

4. CONCLUDING REMARKS

The current version of SCCS has been operational since early 1977. We are still learning more about the nature of program change. However, no new major versions are planned; we feel that the current version of SCCS is flexible enough for most situations.

Program maintenance is expensive [2]. It is also, at best, tedious without a tool like SCCS. Moreover, there is a severe lack of literature dealing with source code maintenance. It is hoped that this description of a successful and widely accepted source code control system improves an unfortunate situation.

5. ACKNOWLEDGEMENTS

The PWB concept was first suggested by E. L. Ivie [5]. The original design of SCCS is due to M. J. Rochkind. The author worked with L. E. Bonanni and M. J. Rochkind on the design and implementation of PWB SCCS.

6. REFERENCES

[1] Belady, L. A., and Lehman, M. M. The Characteristics of Large Systems. *Proc. Conf. on Research Directions in Software Technology*, Oct. 10-12, 1977.

[2] Boehm, B. A. Software and its impact: a quantitative assessment. *Datamation 19*, 3 (May 1973), 48-59.

[3] Bonanni, L. E., and Glasser, A. L. SCCS/PWB User's Manual. Bell Laboratories, November 1977.

[4] Dolotta, T. A., and Mashey J. R. An Introduction to the Programmer's Workbench. *Proc. Second Int. Conf. on Software Engineering*, Oct. 13-15, 1976.

[5] Ivie, E. L. The Programmer's Workbench—A Machine for Software Development. *Comm. ACM 20*, 10 (October 1977), 746-53.

[6] Kernighan, B. W., and Plaugher, P. J. Software Tools. *Proc. First National Conference on Software Engineering*, Sept. 11-12, 1975.

[7] Rochkind, M. J. The Source Code Control System. *IEEE Trans. on Software Engineering SE-1*, 4 (Dec. 1975), 364-70.

[8] Ritchie, D. M., and Thompson, K. The UNIX Time-Sharing System. *Comm. ACM 17*, 7 (July 1974), 364-75.