CVS II:
Parallelizing Software Development

Brian Berliner

Prisma, Inc.
5465 Mark Dabling Blvd.
Colorado Springs, CO 80918
berliner@prisma.com

ABSTRACT

The program described in this paper fills a need in the UNIX community for a freely available tool to manage software revision and release control in a multi-developer, multi-directory, multi-group environment. This tool also addresses the increasing need for tracking third-party vendor source distributions while trying to maintain local modifications to earlier releases.

1. Background

In large software development projects, it is usually necessary for more than one software developer to be modifying (usually different) modules of the code at the same time. Some of these code modifications are done in an experimental sense, at least until the code functions correctly, and some testing of the entire program is usually necessary. Then, the modifications are returned to a master source repository so that others in the project can enjoy the new bug-fix or functionality. In order to manage such a project, some sort of revision control system is necessary.

Specifically, UNIX[1] kernel development is an excellent example of the problems that an adequate revision control system must address. The SunOS[2] kernel is composed of over a thousand files spread across a hierarchy of dozens of directories.[3] Pieces of the kernel must be edited by many software developers within an organization. While undesirable in theory, it is not uncommon to have two or more people making modifications to the same file within the kernel sources in order to facilitate a desired change. Existing revision control systems like RCS [Tichy] or SCCS [Bell] serialize file modifications by allowing only one developer to have a writable copy of a particular file at any one point in time. That developer is said to have ‘‘locked’’ the file for his exclusive use, and no other developer is allowed to check out a writable copy of the file until the locking developer has finished impeding others’ productivity. Development pressures of productivity and deadlines often force organizations to require that multiple developers be able to simultaneously edit copies of the same revision controlled file.

---

[1] UNIX is a registered trademark of AT&T.

[2] SunOS is a trademark of Sun Microsystems, Inc.

[3] Yes, the SunOS 4.0 kernel is composed of over a *thousand* files!

The necessity for multiple developers to modify the same file concurrently questions the value of serialization-based policies in traditional revision control. This paper discusses the approach that Prisma took in adapting a standard revision control system, RCS, along with an existing public-domain collection of shell scripts that sits atop RCS and provides the basic conflict-resolution algorithms. The resulting program, **cvs**, addresses not only the issue of conflict-resolution in a multi-developer open-editing environment, but also the issues of software release control and vendor source support and integration.

2. The CVS Program

cvs (Concurrent Versions System) is a front end to the RCS revision control system which extends the notion of revision control from a collection of files in a single directory to a hierarchical collection of directories each containing revision controlled files. Directories and files in the **cvs** system can be combined together in many ways to form a software release. **cvs** provides the functions necessary to manage these software releases and to control the concurrent editing of source files among multiple software developers.

The six major features of **cvs** are listed below, and will be described in more detail in the following sections:

1. Concurrent access and conflict-resolution algorithms to guarantee that source changes are not "lost."
2. Support for tracking third-party vendor source distributions while maintaining the local modifications made to those sources.
3. A flexible module database that provides a symbolic mapping of names to components of a larger software distribution. This symbolic mapping provides for location independence within the software release and, for example, allows one to check out a copy of the "diff" program without ever knowing that the sources to "diff" actually reside in the "bin/diff" directory.
4. Configurable logging support allows all "committed" source file changes to be logged using an arbitrary program to save the log messages in a file, notesfile, or news database.
5. A software release can be symbolically tagged and checked out at any time based on that tag. An exact copy of a previous software release can be checked out at any time, *regardless* of whether files or directories have been added/removed from the "current" software release. As well, a "date" can be used to check out the *exact* version of the software release as of the specified date.
6. A "patch" format file [Wall] can be produced between two software releases, even if the releases span multiple directories.

The sources maintained by **cvs** are kept within a single directory hierarchy known as the "source repository." This "source repository" holds the actual RCS ",v" files directly, as well as a special per-repository directory (CVSROOT.adm) which contains a small number of administrative files that describe the repository and how it can be accessed. See Figure 1 for a picture of the **cvs** tree.

2.1. Software Conflict Resolution[4]

[4] The basic conflict-resolution algorithms used in the **cvs** program find their roots in the original work done by Dick Grune at Vrije Universiteit in Amsterdam and posted to **comp.sources.unix** in the volume 6 release sometime in 1986. This original version of **cvs** was a collection of shell

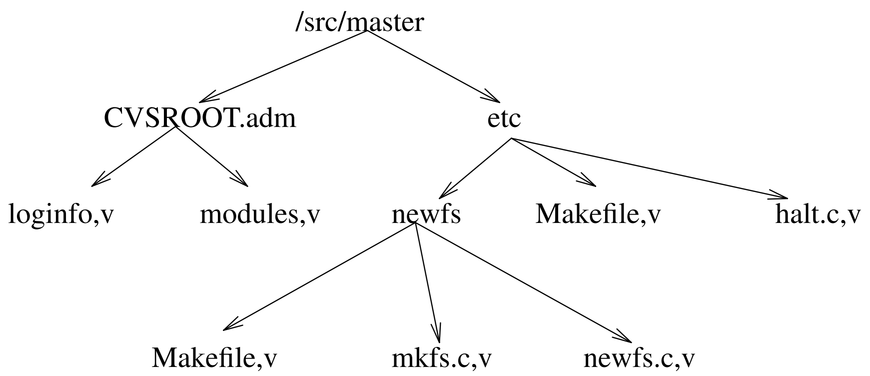

Figure 1.
cvs Source Repository

cvs allows several software developers to edit personal copies of a revision controlled file concurrently. The revision number of each checked out file is maintained independently for each user, and **cvs** forces the checked out file to be current with the ‘‘head’’ revision before it can be ‘‘committed’’ as a permanent change. A checked out file is brought up-to-date with the ‘‘head’’ revision using the ‘‘update’’ command of **cvs**. This command compares the ‘‘head’’ revision number with that of the user’s file and performs an RCS merge operation if they are not the same. The result of the merge is a file that contains the user’s modifications and those modifications that were ‘‘committed’’ after the user checked out his version of the file (as well as a backup copy of the user’s original file). **cvs** points out any conflicts during the merge. It is the user’s responsibility to resolve these conflicts and to ‘‘commit’’ his/her changes when ready.

Although the **cvs** conflict-resolution algorithm was defined in 1986, it is remarkably similar to the ‘‘Copy-Modify-Merge’’ scenario included with NSE[5] and described in [Honda] and [Courington]. The following explanation from [Honda] also applies to **cvs**:

> Simply stated, a developer copies an object without locking it, modifies the copy, and then merges the modified copy with the original. This paradigm allows developers to work in isolation from one another since changes are made to copies of objects. Because locks are not used, development is not serialized and can proceed in parallel. Developers, however, must merge objects after the changes have been made. In particular, a developer must resolve conflicts when the same object has been modified by someone else.

In practice, Prisma has found that conflicts that occur when the same object has been modified by someone else are quite rare. When they do happen, the changes made by the other developer are usually easily resolved. This practical use has shown that the ‘‘Copy-Modify-Merge’’ paradigm is a correct and useful one.

---

scripts that combined to form a front end to the RCS programs.

[5] NSE is the Network Software Environment, a product of Sun Microsystems, Inc.

2.2. Tracking Third-Party Source Distributions

Currently, a large amount of software is based on source distributions from a third-party distributor. It is often the case that local modifications are to be made to this distribution, *and* that the vendor's future releases should be tracked. Rolling your local modifications forward into the new vendor release is a time-consuming task, but **cvs** can ease this burden somewhat. The **checkin** program of **cvs** initially sets up a source repository by integrating the source modules directly from the vendor's release, preserving the directory hierarchy of the vendor's distribution. The branch support of RCS is used to build this vendor release as a branch of the main RCS trunk. Figure 2 shows how the ‘‘head’’ tracks a sample vendor branch when no local modifications have been made to the file.

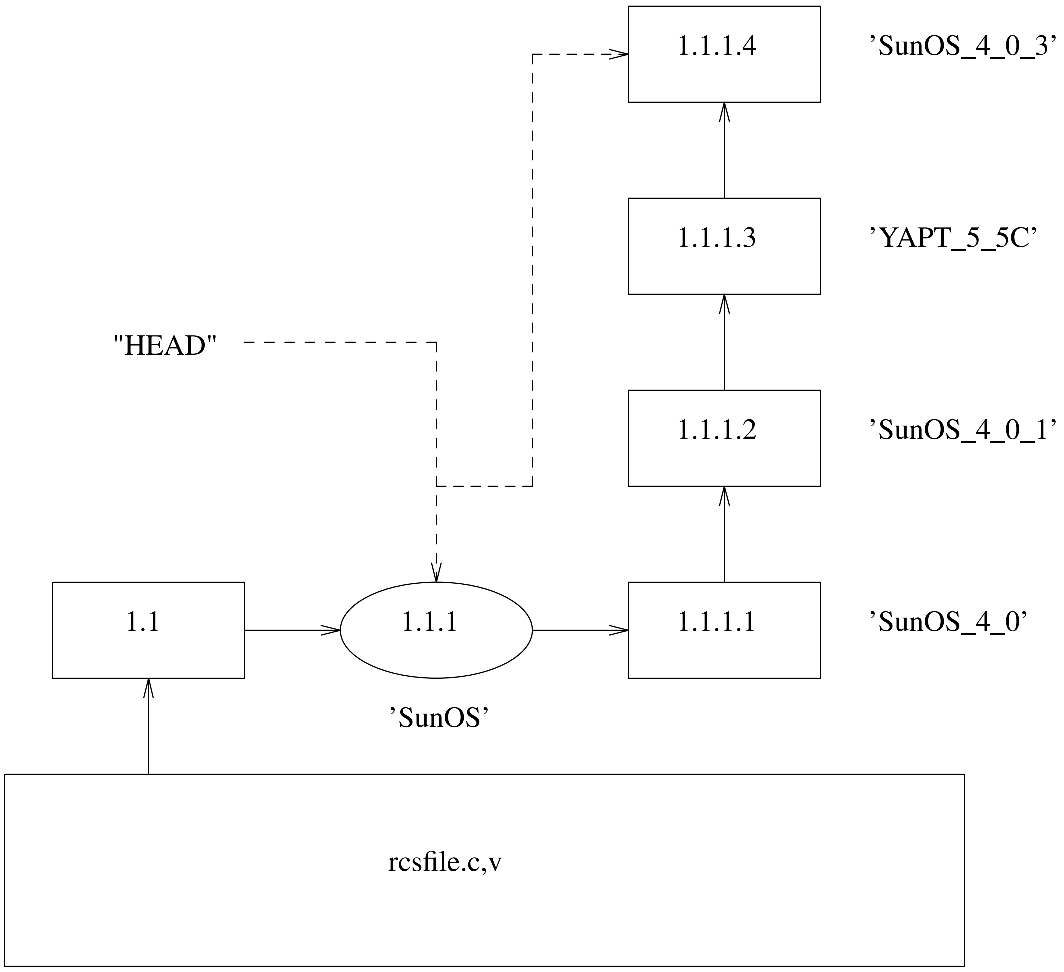

Figure 2.
cvs Vendor Branch Example

Once this is done, developers can check out files and make local changes to the vendor's source distribution. These local changes form a new branch to the tree which is then used as the source for future check outs. Figure 3 shows how the ‘‘head’’ moves to the main RCS trunk when a local modification is made.

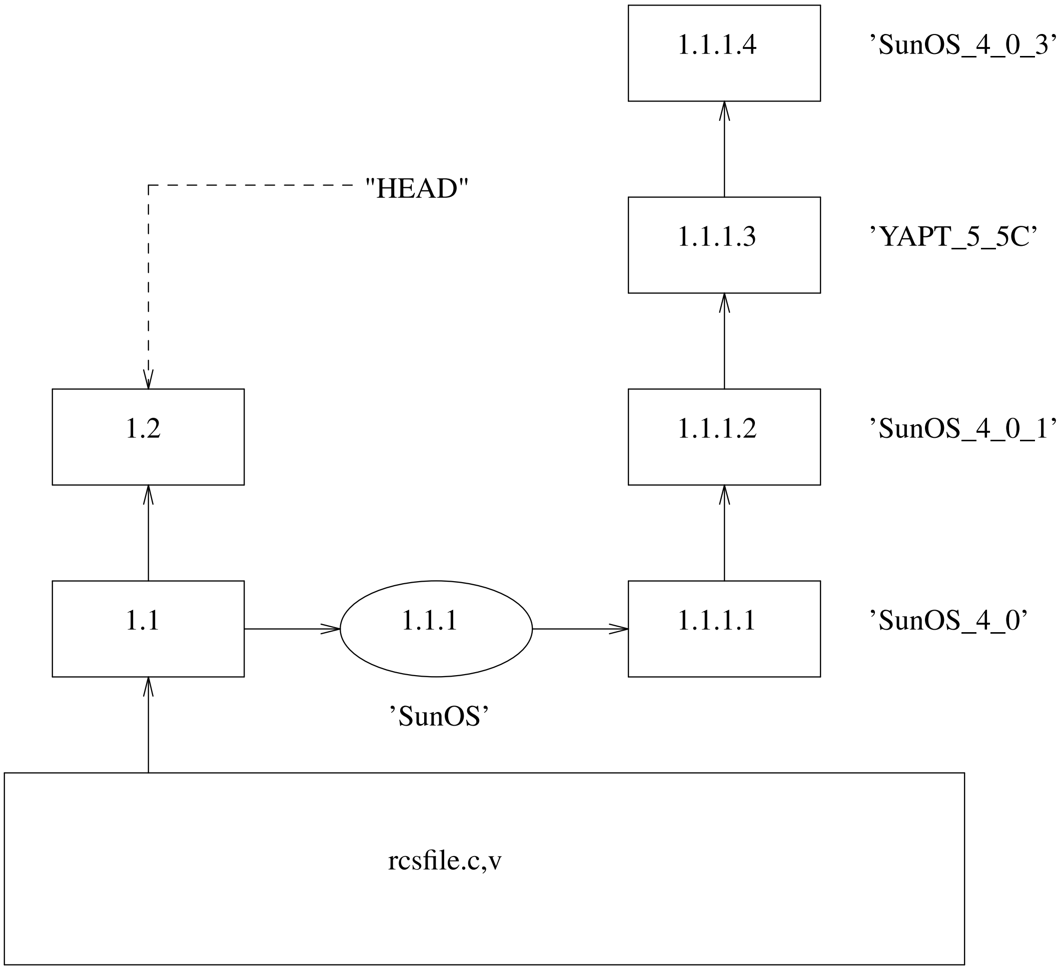

Figure 3.
cvs Local Modification to Vendor Branch

When a new version of the vendor’s source distribution arrives, the **checkin** program adds the new and changed vendor’s files to the already existing source repository. For files that have not been changed locally, the new file from the vendor becomes the current ‘‘head’’ revision. For files that have been modified locally, **checkin** warns that the file must be merged with the new vendor release. The **cvs** ‘‘join’’ command is a useful tool that aids this process by performing the necessary RCS merge, as is done above when performing an ‘‘update.’’

There is also limited support for ‘‘dual’’ derivations for source files. See Figure 4 for a sample dual-derived file. This example tracks the SunOS distribution but includes major changes from Berkeley. These BSD files are saved directly in the RCS file off a new branch.

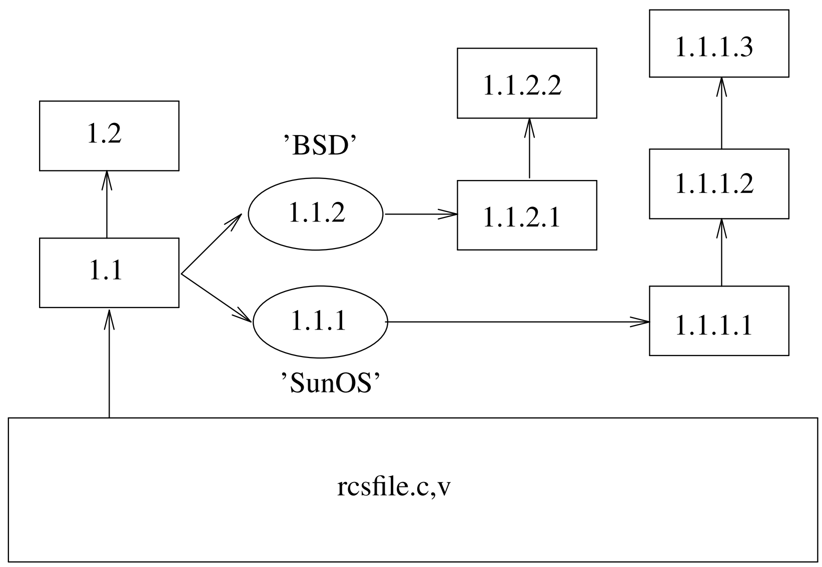

Figure 4.
cvs Support For "Dual" Derivations

2.3. Location Independent Module Database

cvs contains support for a simple, yet powerful, "module" database. For reasons of efficiency, this database is stored in **ndbm** (3) format. The module database is used to apply names to collections of directories and files as a matter of convenience for checking out pieces of a large software distribution. The database records the physical location of the sources as a form of information hiding, allowing one to check out whole directory hierarchies or individual files without regard for their actual location within the global source distribution.

Consider the following small sample of a module database, which must be tailored manually to each specific source repository environment:

```
#key      [-option argument] directory [files...]
diff      bin/diff
libc      lib/libc
sys       -o sys/tools/make_links sys
modules   -i mkmodules CVSROOT.adm modules
kernel    -a sys lang/adb
ps        bin Makefile ps.c
```

The "diff" and "libc" modules refer to whole directory hierarchies that are extracted on check out. The "sys" module extracts the "sys" hierarchy, and runs the "make_links" program at the end of the check out process (the *-o* option specifies a program to run on check*o*ut). The "modules" module allows one to edit the module database file and runs the "mkmodules" program on check*i*n to regenerate the **ndbm** database that **cvs** uses. The "kernel" module is an alias (as the *-a* option specifies) which causes the remaining arguments after the *-a* to be interpreted exactly as if they had been specified on the command line. This is useful for objects that require shared pieces of code from far away places to be compiled (as is the case with the kernel debugger, **kadb**, which shares code with the standard **adb** debugger). The "ps" module shows that the source for "ps" lives in the "bin" directory, but only *Makefile* and *ps.c* are required to build the object.

The module database at Prisma is now populated for the entire UNIX distribution and thereby allows us to issue the following convenient commands to check out components of the UNIX distribution without regard for their actual location within the master source repository:

```
example% cvs checkout diff
example% cvs checkout libc ps
example% cd diff; make
```

In building the module database file, it is quite possible to have name conflicts within a global software distribution. For example, SunOS provides two **cat** programs: one for the standard environment, */bin/cat*, and one for the System V environment, */usr/5bin/cat*. We resolved this conflict by naming the standard **cat** module "cat", and the System V **cat** module "5cat". Similar name modifications must be applied to other conflicting names, as might be found between a utility program and a library function, though Prisma chose not to include individual library functions within the module database at this time.

2.4. Configurable Logging Support

The **cvs** "commit" command is used to make a permanent change to the master source repository (where the RCS ",v" files live). Whenever a "commit" is done, the log message for the change is carefully logged by an arbitrary program (in a file, notesfile, news database, or mail). For example, a collection of these updates can be used to produce release notices. **cvs** can be configured to send log updates through one or more filter programs, based on a regular expression match on the directory that is being changed. This allows multiple related or unrelated projects to exist within a single **cvs** source repository tree, with each different project sending its "commit" reports to a unique log device.

A sample logging configuration file might look as follows:

```
#regex      filter-program
DEFAULT     /usr/local/bin/nfpipe -t %s utils.updates
^diag       /usr/local/bin/nfpipe -t %s diag.updates
^local      /usr/local/bin/nfpipe -t %s local.updates
^perf       /usr/local/bin/nfpipe -t %s perf.updates
^sys        /usr/local/bin/nfpipe -t %s kernel.updates
```

This sample allows the diagnostics and performance groups to share the same source repository with the kernel and utilities groups. Changes that they make are sent directly to their own notesfile [Essick] through the "nfpipe" program. A sufficiently simple title is substituted for the "%s" argument before the filter program is executed. This logging configuration file is tailored manually to each specific source repository environment.

2.5. Tagged Releases and Dates

Any release can be given a symbolic tag name that is stored directly in the RCS files. This tag can be used at any time to get an exact copy of any previous release. With equal ease, one can also extract an exact copy of the source files as of any arbitrary date in the past as well. Thus, all that's required to tag the current kernel, and to tag the kernel as of the Fourth of July is:

```
example% cvs tag TEST_KERNEL kernel
example% cvs tag -D 'July 4' PATRIOTIC_KERNEL kernel
```

The following command would retrieve an exact copy of the test kernel at some later date:

```
example% cvs checkout -fp -rTEST_KERNEL kernel
```

The *-f* option causes only files that match the specified tag to be extracted, while the *-p* option automatically prunes empty directories. Consequently, directories added to the kernel after the test kernel was tagged are not included in the newly extracted copy of the test kernel.

The **cvs** date support has exactly the same interface as that provided with RCS, however **cvs** must process the ",v" files directly due to the special handling required by the vendor branch support. The standard RCS date handling only processes one branch (or the main trunk) when checking out based on a date specification. **cvs** must instead process the current "head" branch and, if a match is not found, proceed to look for a match on the vendor branch. This, combined with reasons of performance, is why **cvs** processes revision (symbolic and numeric) and date specifications directly from the ",v" files.

2.6. Building "patch" Source Distributions

cvs can produce a "patch" format [Wall] output file which can be used to bring a previously released software distribution current with the newest release. This patch file supports an entire directory hierarchy within a single patch, as well as being able to add whole new files to the previous release. One can combine symbolic revisions and dates together to display changes in a very generic way:

```
example% cvs patch -D 'December 1, 1988' \
                   -D 'January 1, 1989' sys
```

This example displays the kernel changes made in the month of December, 1988. To release a patch file, for example, to take the **cvs** distribution from version 1.0 to version 1.4 might be done as follows:

```
example% cvs patch -rCVS_1_0 -rCVS_1_4 cvs
```

3. CVS Experience

3.1. Statistics

A quick summary of the scale that **cvs** is addressing today can be found in Table 1. Table 2 shows the history of files changed or added and the number of source lines affected by the change at Prisma. Only changes made to the kernel sources are included. The large number of source file changes made in September are the result of merging the SunOS 4.0.3 sources into the kernel. This merge process is described in section 3.3.

3.2. Performance

The performance of **cvs** is currently quite reasonable. Little effort has been expended on tuning **cvs**, although performance related decisions were made during the **cvs** design. For example, **cvs** parses the RCS ",v" files directly instead of running an RCS process. This includes following branches as well as integrating with the vendor source branches and the main trunk when

Revision Control Statistics at Prisma as of 11/11/89	
How Many...	Total
Files	17243
Directories	1005
Lines of code	3927255
Removed files	131
Software developers	14
Software groups	6
Megabytes of source	128

Table 1.

cvs Statistics

Prisma Kernel Source File Changes By Month, 1988-1989				
Month	# Changed Files	# Lines Changed	# Added Files	# Lines Added
Dec	87	3619	68	9266
Jan	39	4324	0	0
Feb	73	1578	5	3550
Mar	99	5301	18	11461
Apr	112	7333	11	5759
May	138	5371	17	13986
Jun	65	2261	27	12875
Jul	34	2000	1	58
Aug	65	6378	8	4724
Sep	266	23410	113	39965
Oct	22	621	1	155
Total	1000	62196	269	101799

Table 2.

cvs Usage History for the Kernel

checking out files based on a date.

Checking out the entire kernel source tree (1223 files/59 directories) currently takes 16 wall clock minutes on a Sun-4/280. However, bringing the tree up-to-date with the current kernel sources, once it has been checked out, takes only 1.5 wall clock minutes. Updating the *complete* 128 MByte source tree under **cvs** control (17243 files/1005 directories) takes roughly 28 wall clock minutes and utilizes one-third of the machine. For now this is entirely acceptable; improvements on these numbers will possibly be made in the future.

3.3. The SunOS 4.0.3 Merge

The true test of the **cvs** vendor branch support came with the arrival of the SunOS 4.0.3 source upgrade tape. As described above, the **checkin** program was used to install the new sources and the resulting output file listed the files that had been locally modified, needing to be merged manually. For the kernel, there were 94 files in conflict. The **cvs** ‘‘join’’ command was used on each of the 94 conflicting files, and the remaining conflicts were resolved.

The "join" command performs an **rcsmerge** operation. This in turn uses */usr/lib/diff3* to produce a three-way diff file. As it happens, the **diff3** program has a hard-coded limit of 200 source-file changes maximum. This proved to be too small for a few of the kernel files that needed merging by hand, due to the large number of local changes that Prisma had made. The **diff3** problem was solved by increasing the hard-coded limit by an order of magnitude.

The SunOS 4.0.3 kernel source upgrade distribution contained 346 files, 233 of which were modifications to previously released files, and 113 of which were newly added files. **checkin** added the 113 new files to the source repository without intervention. Of the 233 modified files, 139 dropped in cleanly by **checkin**, since Prisma had not made any local changes to them, and 94 required manual merging due to local modifications. The 233 modified files consisted of 20,766 lines of differences. It took one developer two days to manually merge the 94 files using the "join" command and resolving conflicts manually. An additional day was required for kernel debugging. The entire process of merging over 20,000 lines of differences was completed in less than a week. This one time-savings alone was justification enough for the **cvs** development effort; we expect to gain even more when tracking future SunOS releases.

4. Future Enhancements and Current Bugs

Since **cvs** was designed to be incomplete, for reasons of design simplicity, there are naturally a good number of enhancements that can be made to make it more useful. As well, some nuisances exist in the current implementation.

- **cvs** does not currently "remember" who has a checked out a copy of a module. As a result, it is impossible to know who might be working on the same module that you are. A simple-minded database that is updated nightly would likely suffice.
- Signal processing, keyboard interrupt handling in particular, is currently somewhat weak. This is due to the heavy use of the **system** (3) library function to execute RCS programs like **co** and **ci**. It sometimes takes multiple interrupts to make **cvs** quit. This can be fixed by using a home-grown **system** () replacement.
- Security of the source repository is currently not dealt with directly. The usual UNIX approach of user-group-other security permissions through the file system is utilized, but nothing else. **cvs** could likely be a set-group-id executable that checks a protected database to verify user access permissions for particular objects before allowing any operations to affect those objects.
- With every checked-out directory, **cvs** maintains some administrative files that record the current revision numbers of the checked-out files as well as the location of the respective source repository. **cvs** does not recover nicely at all if these administrative files are removed.
- The source code for **cvs** has been tested extensively on Sun-3 and Sun-4 systems, all running SunOS 4.0 or later versions of the operating system. Since the code has not yet been compiled under other platforms, the overall portability of the code is still questionable.
- As witnessed in the previous section, the **cvs** method for tracking third party vendor source distributions can work quite nicely. However, if the vendor changes the directory structure or the file names within the source distribution, **cvs** has no way of matching the old release with the new one. It is currently unclear as to how to solve this, though it is certain to happen in practice.

5. Availability

The **cvs** program sources can be found in a recent posting to the **comp.sources.unix** newsgroup. It is also currently available via anonymous ftp from ‘‘prisma.com’’. Copying rights for **cvs** will be covered by the GNU General Public License.

6. Summary

Prisma has used **cvs** since December, 1988. It has evolved to meet our specific needs of revision and release control. We will make our code freely available so that others can benefit from our work, and can enhance **cvs** to meet broader needs yet.

Many of the other software release and revision control systems, like the one described in [Glew], appear to use a collection of tools that are geared toward specific environments — one set of tools for the kernel, one set for ‘‘generic’’ software, one set for utilities, and one set for kernel and utilities. Each of these tool sets apparently handle some specific aspect of the problem uniquely. **cvs** took a somewhat different approach. File sharing through symbolic or hard links is not addressed; instead, the disk space is simply burned since it is ‘‘cheap.’’ Support for producing objects for multiple architectures is not addressed; instead, a parallel checked-out source tree must be used for each architecture, again wasting disk space to simplify complexity and ease of use — punting on this issue allowed *Makefile*s to remain unchanged, unlike the approach taken in [Mahler], thereby maintaining closer compatibility with the third-party vendor sources. **cvs** is essentially a source-file server, making no assumptions or special handling of the sources that it controls. To **cvs**:

A source is a source, of course, of course, unless of course the source is Mr. Ed.[6]

Sources are maintained, saved, and retrievable at any time based on symbolic or numeric revision or date in the past. It is entirely up to **cvs** wrapper programs to provide for release environments and such.

The major advantage of **cvs** over the many other similar systems that have already been designed is the simplicity of **cvs**. **cvs** contains only three programs that do all the work of release and revision control, and two manually-maintained administrative files for each source repository. Of course, the deciding factor of any tool is whether people use it, and if they even *like* to use it. At Prisma, **cvs** prevented members of the kernel group from killing each other.

7. Acknowledgements

Many thanks to Dick Grune at Vrije Universiteit in Amsterdam for his work on the original version of **cvs** and for making it available to the world. Thanks to Jeff Polk of Prisma for helping with the design of the module database, vendor branch support, and for writing the **checkin** shell script. Thanks also to the entire software group at Prisma for taking the time to review the paper and correct my grammar.

8. References

[Bell] Bell Telephone Laboratories. ‘‘Source Code Control System User’s Guide.’’ *UNIX System III Programmer’s Manual*, October 1981.

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

[6] **cvs**, of course, does not really discriminate against Mr. Ed.[7]

[7] Yet.

[Essick] Essick, Raymond B. and Robert Bruce Kolstad. *Notesfile Reference Manual*, Department of Computer Science Technical Report #1081, University of Illinois at Urbana-Champaign, Urbana, Illinois, 1982, p. 26.

[Glew] Glew, Andy. “Boxes, Links, and Parallel Trees: Elements of a Configuration Management System.” *Workshop Proceedings of the Software Management Conference*, USENIX, New Orleans, April 1989.

[Grune] Grune, Dick. Distributed the original shell script version of **cvs** in the **comp.sources.unix** volume 6 release in 1986.

[Honda] Honda, Masahiro and Terrence Miller. “Software Management Using a CASE Environment.” *Workshop Proceedings of the Software Management Conference*, USENIX, New Orleans, April 1989.

[Mahler] Mahler, Alex and Andreas Lampen. “An Integrated Toolset for Engineering Software Configurations.” *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, ACM, Boston, November 1988. Described is the **shape** toolkit posted to the **comp.sources.unix** newsgroup in the volume 19 release.

[Tichy] Tichy, Walter F. “Design, Implementation, and Evaluation of a Revision Control System.” *Proceedings of the 6th International Conference on Software Engineering*, IEEE, Tokyo, September 1982.

[Wall] Wall, Larry. The **patch** program is an indispensable tool for applying a diff file to an original. Can be found on uunet.uu.net in ~ftp/pub/patch.tar.