

# PRCS: The Project Revision Control System

Josh MacDonald<sup>1</sup>, Paul N. Hilfinger<sup>1</sup>, and Luigi Semenzato<sup>2</sup>

<sup>1</sup> University of California at Berkeley, Department of Electrical Engineering and Computer Sciences, Berkeley CA 94720, USA,

<sup>2</sup> National Energy Research Scientific Computing Center, Lawrence Berkeley National Laboratory, Berkeley CA 94720, USA

**Abstract.** PRCS is an attempt to provide a version-control system for collections of files with a simple operational model, a clean user interface, and high performance. PRCS is characterized by the use of project description files to input most commands, instead of a point-and-click or a line-oriented interface. It employs optimistic concurrency control and encourages operations on the entire project rather than individual files. Although its current implementation uses RCS in the back-end, the interface completely hides its presence. PRCS is free. This paper describes the advantages and disadvantages of our approach, and discusses implementation issues.

## 1 Overview

PRCS is an attempt at producing a version control system for collection of files that is competitive with existing commercial and free systems with regard to ease of use, implementation, and maintenance. PRCS borrows freely from well-established concepts and ideas in this area, but re-engineers them into a small set of orthogonal features, a clean version model, and a simple yet powerful user interface based on text editing.

The design and development of PRCS was partly motivated by the authors' dissatisfaction with existing systems. Commercial systems tend to be large, feature-laden, and expensive. They often have fancy graphical user interfaces, which can make many operations intuitive, but typically present a sharp threshold beyond which even conceptually simple operations become exceedingly cumbersome. CVS (Concurrent Version System)[1], the standard non-commercial solution, is widely used but we find that many aspects of its operation, administration, and user interface are unnecessarily complex, mostly because of its choice of version model and its explicit dependency on RCS.

CVS and several other systems are designed around the idea that version control for a group of files can be achieved by grouping together many single-file version-control operations. PRCS defines a project version as a labeled snapshot of a group of files, and provides operations on project versions as a whole. Our experience shows that focusing on this model results in a cleaner system than a model cast explicitly as groups of single-file operations.

One notable difference between PRCS and other systems is its user interface. In the PRCS model, each version of a group of files contains one distinguished file,

called the version descriptor. This file contains a description of the files included in that particular version. All user-controlled project settings are entered and manipulated through this file. Indeed, many user-interface issues become text-editing problems from the start, rather than requiring a graphical user interface or numerous command-line utilities for maintaining project state.

Additionally, PRCS includes several novel features, including an improved keyword replacement mechanism and a flexible way of incorporating PRCS into external programs.

In the remainder of this paper, we provide a small example (§3), and discuss the system design and operational model (§2), related work (§4), branch control (§5), distinguishing features (§6), implementation considerations and implications for back-end storage management (§7), and future work (§8).

## 2 Operational Model

PRCS presents the user with the abstraction of named *projects*, which are collections of *project versions* (or simply *versions*), each of which is a snapshot of a set of files arranged into a directory tree. Every project version is labeled with a *version name*, which is unique within that version's project. Finally, a PRCS *repository* contains a group of projects.

Each version contains one distinguished file in its top-level directory, known as its *project-version descriptor*, or simply *version descriptor*. This is an ordinary text file that identifies the version and contains a list of the version's constituent files. When a file is added to a project version for the first time, it is assigned a unique, internal identifier, which follows the file through its history, even as it is renamed. This identifier is known as the *internal file family*. The version descriptor contains a mapping between internal file families and file names. We give more detail in §6.1.

Users modify directory structures of files independently of the repository. We use the adjective *working* to distinguish these files and directories from those that are stored in the repository: thus *working version*, *working file*, and *working version descriptor*. When we say that a project version is a snapshot of a directory structure, we typically mean a snapshot of such a working project version.

Before any working version is *checked in* (the usual term for “taking a snapshot”) PRCS modifies its descriptor to reflect the new (repository) version. In particular, the resulting version descriptors contain a record of the identity of the versions from which they were derived, thus inducing a partial order on versions that serves to define ancestry.

Within this framework, PRCS provides the following major operations:

**Checkin** deposits a labeled snapshot of a working version in the repository.

**Checkout** reconstructs any project version, identified by project and version label.

**Diff** compares project versions.

**Merge** reconciles two project versions interactively by modifying the working project version. PRCS uses the notion of ancestry described above to

determine a common ancestor version that allows it to determine the proper handling of many inconsistencies between the versions being merged.

As you can see from this list, there is no operation for locking a project, version, or file. Instead, PRCS employs optimistic concurrency control. Users are expected to merge changes prior to check-in when conflicts occur, and are informed when this is necessary.

### 3 An Example

This section presents an example of basic use of PRCS to give some flavor of the interactions involved. Consider two programmers, Jack and Jill, performing maintenance on a system we'll call WaterWorks. For illustrative purposes, we'll assume that WaterWorks is not initially under source-code control. Jack therefore acquires some directory structure of files. He places it under PRCS's control with the following commands (lines beginning with a `jack%` or `jill%` prompt are user input).

```
jack% prcs checkout WaterWorks
prcs: Project not found in repository, initial checkout.
prcs: You may now edit the file 'WaterWorks.prj'
jack% prcs populate
prcs: 245 files were added.
```

Jack now has a version descriptor file for the new project, `WaterWorks.prj`, containing the names of all files in and below his current directory. Nothing has been added to the repository yet. Jack edits the version descriptor to add descriptive text about the project, if desired, and to remove any uninteresting files from the list. He then performs the command

```
jack% prcs checkin
```

which takes a snapshot of the collection of listed files, and labels it '0.1' in the repository. He performs a set of edits, adding new files to the version descriptor, if needed, and then

```
jack% prcs checkin -rJack
```

which takes a snapshot of the current state of his working version of the project, and names this version 'Jack.1'. Jack can now continue editing, performing periodic checkpoints with

```
jack% prcs checkin
```

Meanwhile, Jill gets her own copy of the WaterWorks project with

```
jill% prcs checkout -r0 WaterWorks
```

which retrieves the version Jack started with, 0.1. Jill makes modifications, creates her own branch with

```
jill% prcs checkin -rJill
```

and performs a cycle of edits and checkins, just as Jack did.

Eventually, Jack has completed a coherent set of modifications, and checks it into the main branch (labeled '0' here), with

```
jack% prcs checkin -r0
```

creating a snapshot of the project named '0.2'. When Jill attempts to do the same with, say, her version Jill.5, PRCS tells her that the latest version in the main branch is not an ancestor of the one she is checking in. She is thus alerted to look at the changes made since she branched off:

```
jill% prcs diff -r0.1 -r0.2
```

and eventually to *merge* her changes with those in 0.2, using

```
jill% prcs merge -r0
```

This commences a dialog in which PRCS asks how to deal with each conflict between changes from version 0.1 (the common ancestor of 0.2 and Jill.5) and 0.2 versus changes between 0.1 and Jill.5. When this is complete, and Jill has made whatever edits are needed to reconcile her changes with Jack's, she can perform the checkin

```
jill% prcs checkin -r0
```

Jack and Jill can continue indefinitely with this cycle of edits, checkins, and merges. Most interactions with PRCS on the command line are as simple as these.

## 4 Related Work

The list of available version control and configuration management systems is quite long. Two systems deserve special attention: the Concurrent Version System (CVS) because it is a *de-facto* standard among free systems; and the commercial system Perforce<sup>tm</sup> [4], because it shares with PRCS similar approaches to several problems. In §5, we compare these three systems with respect to two features: labeling of sets of file versions, and branch control.

Research in software evolution and CASE tools has focused on augmenting traditional entity-relation design and analysis techniques with version histories and dependencies. One such attempt is the PROTEUS Configuration Language (PCL) [5]. PCL allows for the description and control of software variability by tracking changes in the requirements and tools required to build the each variation, where variations may include different hardware platforms, releases, or customer configurations.

## 5 Branch Control

Much of the variation between SCM systems found today can be reduced to their differing notions and definitions of branches, labels, and change numbers. A branch is a division in project development, a logical split occurring when a particular ancestor has more than one descendent. Branches are typically used to manage and help reintegration of variants in a project's development. Easy and fast operations on branches are essential for exploiting a version control system to its full potential. This section defines the common approaches to branch management and discusses their limitations and implications for ease of use.

### 5.1 Labels vs. Change Numbers

To use Perforce's terminology, both labels and change numbers are mechanisms for naming sets of revisions of files contained in a project.<sup>1</sup> A *label*—also known as a *tag* in RCS or CVS—names an arbitrary set of file revisions chosen by the user. The file revisions named by a label may change at any time, and may consist of revisions that were created at different times and by different developers. A *change number* or *labeled change* refers to a set of revisions that were committed during a single transaction. Although CVS provides allows one to group operations atomically, it not have a mechanism for naming each transaction and later retrieving the contents of the project at the moment changes were committed. One can simulate change numbers with CVS tags, but this is not directly supported. Perforce supports both mechanisms and allows the user to construct views of the project named by either label or change number.

CVS cannot reconstruct change numbers from its repository. Since CVS treats each operation as a group of operations on individual RCS files in the repository, information such as this can only be obtained by check-in-time clustering or by carefully applying an immutable label at each commit.

The issue of labeling each change to the project is of great importance. Without these labels, later reconstruction of the project's history is not as straightforward as one might think. For example, with CVS it is difficult to request an operation such as: “display all changes made by user *A* during his last commit.” One must examine the logs of all affected files, search for changes by user *A*, and then request a list of all differences in the project between some time before and some time after the commit occurred. By contrast, this operation is simple in Perforce and in PRCS: look for the last change by user *A*, request to view the changes at that change number.

PRCS only supports change numbers, which we call *version names*. All version names have the form *M.N*. Here, *M* is the *major version name*, a string chosen by the user (0, `Vendor-2.2`, and `broken-branch` are typical examples),

---

<sup>1</sup> We use the term *project* to refer to the Perforce depot, the CVS module, and the PRCS project.

and  $N$  is the *minor version name*, a positive integer numeral assigned consecutively by the system. The user is free to ascribe any desired semantics to major version names. This labeling scheme has proved satisfactory; so far, we have no evidence that a system without labels is any worse for not having this feature.

## 5.2 Branching with Labels and Change Numbers

In CVS, branching is accomplished by first creating a label for the branch point, which is then used to name the new branch. Each file in the repository must be tagged with the new label—expensive for a large repository.

In a system supporting change numbers, branching is much simpler. A branch may be created during a transaction, when the change number is assigned. The notion of the parent or predecessor of a version is straightforward to define and therefore, in contrast to CVS, a version history—a tree of project versions—is easy to construct.

In PRCS, major version names can serve to identify branches. There is no explicit notion of a hierarchy of branches; the user is free to adopt any desired scheme for reflecting relationships between branches in their names. In fact, minor versions within a branch are just as independent of each other as any arbitrary pair of versions.

# 6 Features

This section describes in more detail some distinguishing features of PRCS.

## 6.1 Version Descriptors

The syntax of the version descriptor file is a slight variant on Lisp syntax. Figure 1 displays a sample project descriptor. Most project settings are entered by editing this file before a check-in. Some fields are provided by PRCS at check-in time. For example the Project-Version, Parent-Version, Version-Log, Checkin-Time, and Checkin-Login identify the project version. Others allow users to supply arbitrary information about a version. For example, anything may be placed in the New-Version-Log field before a check-in, and it becomes the Version-Log of the next version.

The Project-Keywords field contains a list of keyword-value pairs to be added to the standard set of keywords during keyword replacement. Keyword replacement is described in §6.3.

The Files entries contain lists of files, symbolic links, and empty directories contained in the version. Their main function is to associate the name of each file (as seen by the user) with its internal identification in the repository. For example, the line

```
(src/parser.c (1_parser.c 1.3 644))
```

in Figure 1 indicates that the file named `src/parser.c` is associated in this version with data in the repository whose internal label is `(1_parser.c 1.3 644)`. The next line,

```
(src/lexer.c ())
```

indicates that the working file `src/lexer.c` is not yet in the repository, but is to be added when the current working version is checked in.

From the user's point of view, the internal file labels are simply arbitrary label strings having a somewhat unusual form and a simple semantics: each uniquely identifies a particular file snapshot. The same internal label in two different version descriptors of the same project denotes the same file contents (modulo keywords; see §6.3). Check-in automatically updates the internal labels of all modified files in the version descriptor when a check-in occurs. Files in a working directory that are not mentioned in the Files entries are not included in snapshots of the directory.

The Files entries may also attach a set of attributes to any file. The `'tag'` attribute allows arbitrary labels to be attached to files. The `'no-keywords'` attribute turns off keyword replacement for the file in question. The `'difftool'` and `'mergetool'` attributes set the programs which are used to display files differences and perform three-way file merging, respectively. As illustrated, Files lists allow files to be grouped by common attributes for brevity.

A particularly elegant result of this design is that because version descriptors are simply text files in a format that is intended to be readable, numerous operations needed for version control reduce to text editing. Adding, subtracting, and renaming files (including moving them across directories) is accomplished by editing the working copy of the version descriptor file before checking in the new version. For example, the file list in the version descriptor of a project consisting of the files `hello.c` and `hello.h` might look like the following.

```
(Files
  (hello.c (0_hello.c 1.1 644))
  (hello.h (1_hello.h 1.1 644))
)
```

Suppose that in our working version, we (1) rename `hello.h` to `greetings.h`, moving it to a new `include` subdirectory, (2) add a new file, `include/cards.h` and a new empty directory, `doc`, and (3) delete `hello.c`. We would modify the working version descriptor as follows.

```
(Files
  (include/greetings.h (1_hello.h 1.1 644))
  (include/cards.h ())
  (doc () :directory)
)
```

Assuming that we do not change the contents of the former `hello.h`, PRCS will, on check-in, modify this list into something like this for the new version it creates:

```

(Project-Version P Vendor 5)
(Parent-Version P Vendor 4)
(Project-Description "Sample Project")
(Version-Log "Version log for Vendor.5")
(New-Version-Log "Version log for working changes (eventually
Vendor.6).")
(Checkin-Time "Wed, 14 Jan 1998 23:24:44 -0800")
(Checkin-Login jmacd)
(Populate-Ignore ("~$" "\\..o$"))
(Project-Keywords (Release "2.2") (PatchLevel "6"))
(Files :tag=sourcefile :difftool=prcs-ediff :mergetool=prcs-emerge
  (src/main.c (0_main.c 1.5 644))
  (src/parser.c (1_parser.c 1.3 644))
  (src/lexer.c ()))
)

(Files :no-keywords :tag=image
  (img/logo.jpg (2_logo.jpg 1.2 444) :difftool=jpg-diff)
  (img/icon.gif (3_icon.gif 1.1 444) :difftool=gif-diff)
)

(Files
  (lib () :directory) ; Include a lib directory, even if empty
  (core (/dev/null) :symlink)
)

```

Fig. 1. A Sample Working Version Descriptor

```
(Files
  (include/greetings.h (1_hello.h 1.1 644))
  (include/cards.h (0_cards.h 1.1 644))
  (doc () :directory)
)
```

The internal label associated with `include/greetings.h` does not change, since it reflects a particular file contents, independent of the name.

Suppose that we later decide to re-introduce the file `hello.c` into a still-later version, renaming it to `greetings.c`. It is possible to do this by adding the line

```
(greetings.c ())
```

to the working descriptor's Files list, but PRCS also allows us to establish the association of this file with the previous `hello.c` by copying one of the latter's internal labels, as in:

```
(greetings.c (0_hello.c 1.1 644))
```

The effect of doing so is to put `greetings.c` into the same file family as `hello.c`, which means that when versions containing these files are compared or merged, PRCS can recognize the relationship between them. Old internal labels can be recovered by checking out earlier versions of the version descriptor.

Creating the first version of a project is a special case. One creates a working descriptor file by checking out a project not yet in the repository. This file has an empty Files list. One can fill this as we've done so far, with entries having '()' for their internal label. To avoid tedium and error, PRCS provides a convenience command, `populate`, that finds files under the current directory and generates new Files entries for each of them. Editing the resulting working descriptor will remove any that are not wanted.

PRCS does not, of course, implicitly trust the contents of working version descriptor, and subjects them to verification before check-ins. Indeed, it may seem dangerous to allow the user to directly modify administrative information that, in other systems, is kept hidden behind an appropriate GUI. However, our experience shows otherwise; users do not show any tendency to hang themselves, and the consistency checks suffice to prevent accidents.

The use of a text editor for project administration is admittedly controversial. However, we have found it surprisingly effective. The most common use, after all, consists of adding a log entry—a simple exercise in text insertion, as is the addition of files. Deleting files from a project or changing their names likewise correspond to standard editing operations.

## 6.2 The `execute` command

The PRCS `execute` command facilitates efficient, open-ended extension to the functions provided by PRCS. Its command syntax allows another command or script to be executed with the name of each file and directory in the project.

Regular expressions may be used to filter the list of files, allowing selection or exclusion by file name, extension, or attribute. The special ‘:tag’ file attribute has no special meaning to PRCS, and is simply included to allow grouping of files according to relevant characteristics.

Various options allow many elaborate commands to be constructed. When run against a project version in the repository, PRCS can also supply a checked-out copy of each file. Instead of executing the command once per file or directory, PRCS can execute the command just once for all affected files. Finally, PRCS can supply the contents of the file to the command on the standard input. For example, the command:

```
prcs execute --pipe --match /\.cc$ P -- wc
```

pipes each file in the working version of project P with the .cc extension to the wc command.

### 6.3 Keyword Replacement

PRCS, like many other version control systems, allows keyword replacement in selected files. Abstractly, we may think of this replacement as being performed upon check-in of a file. Internally, PRCS actually does not do so, in order to avoid having to store changes of keywords as part of the change information for a file.

Within files in a version, PRCS recognizes two forms of keyword instance: simple and formatted. Simple keyword instances (as in RCS) have one of these forms:

```
$keyword$  
$keyword: value$
```

and on check-in, such instances are replaced by an instance of the second form, with the appropriate value.

Sometimes, one needs keyword replacement data without the leading “*\$keyword:*” and trailing ‘\$’. A novel feature of PRCS, the *formatted keyword instance*, allows this. When PRCS encounters the text

```
arbitrary text $Format: "format-string"$ arbitrary text
current-string
```

at check-in time, it replaces the line reading *current-string* with *format-string*, after first substituting simple keywords in the latter. The line containing the “Format” instance itself is not altered. For example, Figure 2 illustrates the contents of a file before and after formatted keyword replacement for project version 0.4.

*Before replacement:*

```
/* $Format: "char* version = \"$ProjectVersion$\"; "$ */
char* version = "x.x";
```

*After replacement:*

```
/* $Format: "char* version = \"$ProjectVersion$\"; "$ */
char* version = "0.4";
```

**Fig. 2.** Formatted Keyword Replacement

There are 13 predefined keywords that have values dependent on the project version and individual file version. Many, such as `Date` and `Author` (the date at which a modified version of a file was checked in and the identity of the user checking it in), will be familiar to users of RCS. Others are project-related, such as `ProjectDate` and `ProjectAuthor` (the date of a check-in and the identity of the user doing it). In addition, users may introduce their own keywords in the project descriptor, as was illustrated in Figure 1.

## 6.4 Subprojects

PRCS has no explicit facilities for managing *subprojects*—one or more PRCS-controlled projects within another *super-project*. Nevertheless, it is possible to get much of the effect, in a typically minimalist fashion, simply by including version descriptor files for the subprojects among the files included in the super-project. Checking in the super-project consists of first checking in the subprojects (if needed), and then checking in the superproject. On checking out this version, one gets the subproject descriptor files. These uniquely identify the constituent subproject versions, so checking out any of the subprojects based on its descriptor (a simple operation in PRCS) recovers the subproject. It is not clear to us that this simple recursive process represents something sufficiently

common and error-prone to really need explicit support. While automated support process is conceivable, furthermore, it is not clear what the best policy is for mapping operations on the super-project onto the sub-project. For example, when branching the super-project it may or may not be desirable to create a new branch in the sub-project. Questions like this and our lack of experience with sub-project management has kept automated support for sub-projects out of our current design.

## 7 Implementation

PRCS is implemented using RCS as a back-end storage mechanism. While this helped in the quick implementation of a robust prototype, it has several limitations as a long-term strategy.

- Starting a new process for a revision control operation on each file is expensive.
- RCS calling and locking conventions make interfacing a higher-level version control systems to RCS unnecessarily complicated.
- Binary files are not well supported.
- The branching mechanism of RCS has unexpected performance implications, and it is difficult to optimize its use, either automatically or with user intervention.

These and other problems make RCS a less-than-ideal back-end tool. The process startup costs would be avoided by an implementation of RCS as a set of thread-safe library routines, but this would not fix its interface problems. The current implementation of PRCS, for example, is complicated by code to manage locking and unlocking of RCS files and placing or linking files into the correct path before performing batch check-ins. Additionally, the delta mechanism of RCS is out-of-date. New delta algorithms have been developed that outperform LCS-based delta algorithms, and handle binary files as well as text files, as demonstrated by Hunt *et al* [2].

Finally, the RCS branching model is inappropriate for higher level tools. RCS forces each version to be placed in a tree. Its location on the tree greatly effects the performance of operations on it. Experience with PRCS has shown that it is difficult for a high-level version control tool to place these versions well.

PRCS makes extensive use of timestamps and MD5 checksums to minimize file I/O and RCS invocations. The MD5 optimizations, though complicated by keyword replacement, reduce I/O greatly in situations where timestamps alone do not prevent a comparison.

## 8 Future Work

Upon completing the system, we immediately identified the need for distributed, multi-user version control, involving some sort of client/server structure. A design for distributed version control architecture can make many reductions in

data transfer by carefully caching data and transmitting only changes. The problem of efficiently distributing version-control repositories has been nicely addressed by Polstra's CVSup (CVS Update Protocol) [3]. Work on a client/server implementation is currently underway.

As we introduce distributed PRCS repositories, we are planning to allow users to distinguish between *local* and *network* check-ins. Often, users of a global, shared repository feel uncomfortable checkpointing their progress with extra, possibly off-branch versions because of the shared nature of the repository. Personal, possibly local modifications do not belong in the global repository. Users are forced to use multiple repositories, one for their local work and one for shared, group work.

We plan to address this problem by allowing certain branches to be marked local. Local branches are not shared with the global repository and are invisible to other user of the repository. Operations on local versions in the network context treat other local versions as transparent, following their ancestry until a network version is found. This allows, for example, a user to check out the latest version on the network branch, check in several versions on a local branch for intermediate checkpoints, then return his work to the network repository by checking in the next version, possibly after a merge against the head of the network branch. A merge between the head of the network branch and the local working version would treat original network version as its immediate parent.

We have also investigated replacement delta algorithms similar to the Vdelta algorithm and improvements on RCS that avoid the branching problems mentioned above. Conceptually, to get rid of RCS-like branching and retain the "deltas are computed between a file version and its parent" aspect of RCS, we have devised a version-control library that computes a reverse delta between the new file and the entire version file. The most recently checked in version is always the "head", in RCS terms. It is always the quickest and easiest to extract. For a repository with  $N$  versions, version  $I$  requires the application of  $(N - I)$  deltas, regardless of how the set of versions are related. This type of construction leads to a versioned storage format without the performance problems associated with branches, yet with similar storage efficiency. Branches are avoided by using the content of previous deltas in addition to the most recently checked-in file.

## 9 Conclusion

PRCS began as a study in engineering a conceptually simple solution to the version control problem, making good reuse of an existing tool—the text editor—to simplify operations from the user's point of view. It has evolved into a system that, we believe, is easy to use, presents a simple, abstract model to the user, and whose design is uncompromised by the ease or difficulty of a particular implementation.

We have observed that the snapshot, labeled-version model of project versions works well and is simpler than mechanisms that treat version control on groups of files as grouped operations on individual files. We conclude that a system

with only one version labeling paradigm—change numbering—is adequate and sufficiently powerful to meet the needs of a sophisticated version control tool, yet is simpler than many competing approaches.

*Availability.* Information about PRCS and its source and binary distributions are available at <http://www.xcf.berkeley.edu/~jmacd/prcs.html>. The experimental RCS replacement and delta algorithm implementation described in (§8) is available at <http://www.xcf.berkeley.edu/~jmacd/xdelta.html>.

*Acknowledgement.* We wish to thank the National Automated Highway System Consortium, which helped to fund initial development of PRCS under Cooperative Agreement DTFH61-94-X-00001 with the Federal Highway Administration.

## References

1. BERLINER, B. CVS II: Parallelizing software development. In *Proceedings of the Winter 1990 USENIX Conference, January 22-26, 1990, Washington, DC, USA* (Berkeley, CA, USA, Jan. 1990), USENIX Association, Ed., USENIX, pp. 341-352.
2. HUNT, J. J., VO, K.-P., AND TICHY, W. F. An empirical study of delta algorithms. *Lecture Notes in Computer Science 1167* (July 1996), 49-66.
3. POLSTRA, J. Program source for CVSup. <ftp://ftp.cvsup.freebsd.org/pub/CVSup>, 1996.
4. SEIWALD, C. Inter-file branching — A practical method for representing variants. *Lecture Notes in Computer Science 1167* (July 1996), 67-76.
5. TRYGGESETH, E., GULLA, B., AND CONRADI, R. Modelling systems with variability using the PROTEUS configuration language. *Lecture Notes in Computer Science 1005* (1995), 216-240.