Moraine: An Accumulative Software Development Environment for Software Evolution

Tetsuo Yamamoto[†] **Makoto Matsushita**[†] **Katsuro Inoue**[†] [‡]

[†] Graduate School of Engineering Science,
Osaka University
1-3 Machikaneyama, Toyonaka,
Osaka 560-8531, Japan

[‡] Graduate School of Information Science,
Nara Institute of Science and Technology
8916-5, Takayama, Ikoma,
Nara 630-0101, Japan

{t-yamamt, matusita, inoue}@ics.es.osaka-u.ac.jp

Abstract

Recently software is large-scale and complex, and is being developed in more distributed way, with lots of developers. To solve these circumstances, enabling the software evolution is now becoming the hot topic of software engineering; we think it is also important to evolve not only software itself but also software development environment. In this paper, we propose Moraine, an accumulative software development environment for software evolution. Moraine records all the changes of files as an evolution history, and trace a history by specified date or query from users' request. We also implement a prototype of Moraine, and show the performance of file recording.

1 Introduction

Recently software is large-scale and complex, and is being developed in more distributed way, with lots of developers. In this circumstance, software development and maintenance are very difficult. Researches on improving the quality and maintainability of software have been emerged. However, when a present software is executed with the new environment, you must follow it in the change in the environment. The cost of the maintenance work of the software for this is very expensive. The ultimate solution is that the software is evolutionable; however, we think that not only software itself but also the software development environment is evolutionable. Since the future software development environments is the successful extension of the past and the present time of them, one of major issues in the support efforts of software evolution is management of configurations and versions. The software configulation management solves the difficulties to identify, organize, manage software products. Managing products is required in each phase of software development. For example, elements of software component should be clearly identified and defined in a design phase. However, consistent software configulation management is not easy work. In many software development organizations, they employ specific persons who only manage configurations and versions of on-going project.

In many cases, existing configuration management tools such as RCS[7] and CVS[1] have been used. However, these version management tools need special care to use properly. For that reason education and training are essential for beginners. These tools also rely on specific proprietary configuration models. Therefore, it is difficult to change the models.

In this paper, we have devised a novel approach for supporting software evolution. We proposed here Moraine, which is an accumulative software development environment. Using Moraine, all the files automatically are recorded and users can retrieved specified file. The background concepts of Moraine is:

- Disc and CPU are becoming very cheap.
- We can record all results what we have done.
- We want to decrease management cost as low as possible.
- We do not want specific models for configuration and versioning.

Moraine is composed of two features, accumulation and reference. All activities to files are recorded automatically by Moraine as an evolution history. Accumulated histories are analysed, and used to improve the software development.

This paper is organized as follows. In Section2, we describe about concepts of Moraine. In Section3, we show an experimental implementation of a versioning file system, which is a basic technology needed for Moraine. Finally, we conclude this work and present further works in Section4.

2 Concept of Moraine

In this section, we explain about two major features of Moraine.

2.1 Accumulation

Moraine records all the files we have created as an evolution history. We do not need to care the file preservation; you can delete it from the current environment if it is not needed at all. It is easy to create a next version; you can simply edit or create file, and old versions will be simply discarded. Thus, we do not need special preparation for creating a new revision. Engineers are not requested to earn how to use it at first. The typical operations should be automatically achieved by Moraine. Operations to a file (read and write) are automatically mapped into activities of version management; engineers do not consider what should be done to manage the product versions. Consequently, there are no difference between the operations to usual file system and this file system from a viewpoint of users' processes.

2.2 Reference

Moraine retrieves the history by specified time or configuration mark given by the user. Users can tag a meaningful name to a file, because a file automatically recorded. More than one file be retrieved at the same time by tagging the same name to files. Users can also show a delta between versions and branching is easily established. We can go back to a specific version, and resume our work from that point.

Metrics for the evolution is easily obtained by Moraine. The derived metrics is used for the project management, and it is very important for the success of further project.

3 Prototype of Moraine

In this section, we show an experimental implementation of a versioning file system, which is an accumulative part and basic technology needed for Moraine.

3.1 Overview

Moraine consists of two parts which are files recorded part and files referred part. files recorded part is that all the files are recorded. files referred part is that the recorded files are viewed in any way. files recorded part is the core system, which is described in the pages that follow. the system is named the version control file system(VCFS).

VCFS manages the versions of regular files (symbolic link, special file, socket, and named-pipe are out of our scope). A new version is created iff a file is created or an existing file is changed. Checking out the latest version is done with simply reading the file. VCFS also supports a file locking mechanism. Before checkin is completed, other process can only checking out the file.

VCFS employs "checkin/checkout model"[2] which was proposed by RCS, and VCFS itself does not have its own version management mechanism (subsystem); engineers can import a favorite version management sub-system which is adaptable to the model, i.e., VCFS can use RCS commands.

Current prototype of VCFS runs on FreeBSD 3.0-RELEASE[3], a BSD UNIX[4, 5] variants. VCFS is written in C and about 5000 lines total; VFS in kernel for 4500 lines, VCD for 340 lines, and 300 lines for others.

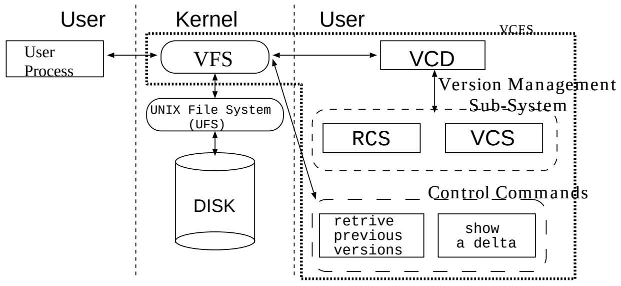

Figure 1: System overview of VCFS

3.2 System Design

VCFS is composed of file system (VFS), version control daemon (VCD), version management sub-system, and VFS control commands. Fig. 1 shows the structure of the VFS components.

The advantage of the separation of version management facility from the kernel is that switching check-in/check-out operations can be easily done. In addition, installing VCFS to a software development organization can be easily performed if RCS is already employed at the organization. VCFS also allows to work with yet another check-in/check-out style version management system.

3.2.1 File System

Since VCFS is implemented as a stackable file system, all files handled by VCFS are stored to a raw file system. Each file via VCFS is consists of two actual files; one for the latest version, and another for version repository (Fig. 2).

Usually VCFS shows the latest version of the file via file system. For example, we assume that /versiondb is mounted to /proj by VCFS and create a file /proj/foo. VCFS creates "/versiondb/foo,a" for the latest version of file foo (/proj/foo itself) and "/versiondb/foo,v", for the version repository. Note that /versiondb/foo,v is invisible via VCFS; it is only for version management and not for the ordinal operation.

VCFS always keeps the latest version of every file (/versiondb/foo,a in the previous example) for fast file read/write to the latest version which is modified in the most cases of software development. When a UNIX process opens a file in read-only mode, VCFS behaves as the same as NULL file system[6]. When a

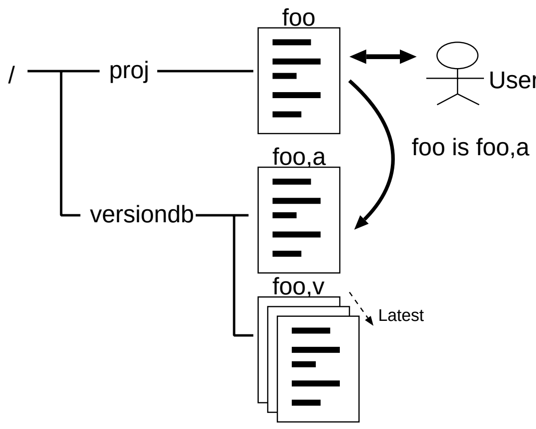

Figure 2: Mapping files between VCFS and actual file system

process opens a file in write-only or read-write mode, VCFS hooks close() system call and performs a check-in operation to a file. The check-in operation runs by the version control daemon and the daemon calls the version management sub-system which is outside of the kernel.

3.2.2 Version Control Daemon

VCD is a daemon process which acts as a bridge between the kernel and the version management sub-system. VCD dispatches the request from the kernel to the version management sub-system.

3.2.3 Version Management Sub-System

The version management sub-system is the actual version management part of VCFS. In general, version management sub-system consists of a set of tools. VCFS employs external version management systems as the sub-systems, and we can change the sub-systems to use. Current prototype of VCFS has two kinds of version management sub-systems, RCS and VCS.

RCS keeps only a delta of each version and the latest version. RCS allows version derivation (a tree structure of version sequence), and a derived version is called as a branch. RCS sub-system uses RCS tools to record versions.

VCS is a simple version management system as the one of VMS file system. VCS saves all versions as-is, and does not calculate the delta between versions.

No version derivation is allowed, however, registering a new version is faster than the RCS sub-system.

3.2.4 VFS Control Commands

VFS control commands help to control VFS behavior. Following commands are already available. We are now working on Web-based VCFS tools which overrides VFS control commands, to support graphical and understandable representation of version history.

- Retrieve previous versions
  Users can retrieve any one previous version (not the latest version) by specifying a version number and/or its created date.

- Make a branch
  Branching is done with control commands if the version management sub-system has branching feature.

- Show a delta between versions
  Users can check the difference between versions.

3.3 Evaluation

This section discusses the prototype of VCFS from viewpoints of the system performance and storage size. We take up UFS (actually FFS, standard file system for BSD UNIX) and NULLFS (loop-back file system, implemented as a stackable file system) to compare with our VCFS. All tests are made at our Moraine development testbsd; 166MHz Pentium PC having 48MB RAM and FreeBSD 3.0-RELEASE.

3.3.1 Performance

At first, we measured an elapsed time for a UNIX process to read distinct new files of 1M bytes size repeatedly. "An elapsed time" means time between process initiation and process termination; be aware that an elapsed time includes an overhead of typical UNIX processes (process initialization, etc).

Fig. 3 shows the results of the reading test of VCFS, UFS, and NULLFS. "VCFS(RCS)" means VCFS using RCS as the version management sub-system. The vertical axis shows an elapsed time, and the horizontal axis shows the number of file reading.

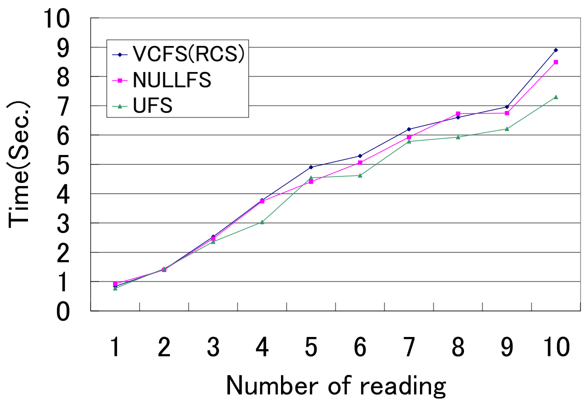

Figure 3: Read (1M bytes files)

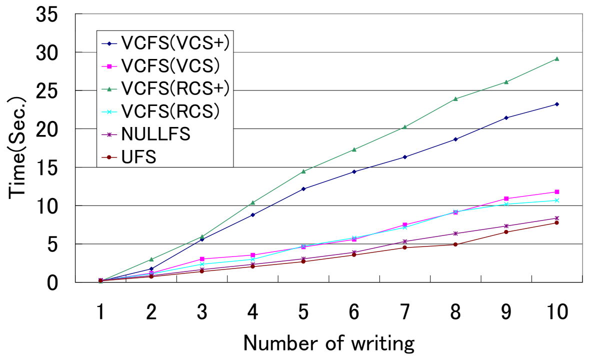

Figure 4: Write (1M bytes files)

The time for VCFS(RCS) is almost the same as one of NULLFS. With regard to file reading, it is understood that it takes almost no extra time as for the version control functions in the file system. VCFS(RCS) and NULLFS are slower than UFS, since there is an implementation overhead of a stackable file system.

We also measured an elapsed time to write files, similar to file reading test described before. Fig. 4 shows the results of writing test. "VCFS(VCS)" means VCFS using VCS as a version management sub-system. "VCFS(VCS+)" and "VCFS(RCS+)" includes a time of synchronization between VCD and sub-system. In general, VCD should not wait the termination of sub-system as usual; however we measured these for comparison purpose.

NULLFS is about 10% slower than UFS, and it is the same of reading test. Writing a file in VCFS

Table 1: A sample data

	All lines of codes	The number of files	The number of versions	Compilation success number of times
data1	9339	45	311	222
data2	4067	20	147	92
data3	2543	18	247	110

Table 2: Total file size (K bytes)

	UFS	VCFS(RCS)	VCFS(VCS)
data1	225	1388	3149
data2	117	546	1377
data3	73	604	1501

requires a new version registration which is not required by the reading, so VCFS consumes 30% or more time compared with UFS. Actual time for a new version registration is twice as of UFS, comparing VCFS(RCS+)/VCFS(VCS+) with UFS. Actually, VCD and sub-system work asynchronously so VCFS is only 30% slower than UFS, and we assume that it is reasonable.

3.3.2 File Size

We applied a sample data taken from a programming seminar of Osaka University (Table 1), and measures the total file size saved into a filesystem. The test creates/updates files, and then compile them. Table 2 shows the result of file size test.

The result of UFS shows an actual size of final version of products. The result of VCFS shows there are more disk spaces to save the whole data; VCFS(RCS) requires several times, and VCFS(VCS) requires ten times as much as of UFS. However, the result of VCFS(VCS) shows that there are *only ten times* as much as the final version, if we have saved *all* version of files; we think it is reasonable size for supporting software evolution. Note that the total file size may vary; it depends on what sub-system is used with VCFS.

4 Conclusion

In this paper, we proposed "Moraine", an accumulative software development environment for software evolution; Moraine records all the files we have created, and shows the history of them. We also implemented and evaluated VCFS which is a basic technology for Moraine.

As a further work, user interface in which users can retrieve files easily should be implemented. Also, supporting distributed software development environment and more usability evaluation are required.

References

[1] Berliner, B.: CVS II: Parallelizing Software Development, *Proceedings of 1990 Winter USENIX Conference*, Washington, D.C. (1990).

[2] Feiler, P. H.: Configuration Management Models in Commercial Environments, Technical Report CMU/SEI-91-TR-7, Software Engineering Institute, Carnegie-Mellon University, Pittsburgh, Pennsylvania 15213 (1991).

[3] Hubbard, J. K.: RELEASE NOTES FreeBSD Release 3.0-RELEASE. This document is available on the World-Wide Web at the URL "http://www.freebsd.org/releases/3.0R/notes.html".

[4] Leffler, S., McKusick, M., karels, M. and Quarterman, J.: *The Design and Implementation of the 4.3BSD UNIX Operating System*, Addison-Wesley (1989).

[5] McKusick, M., Bostic, K., karels, M. and Quarterman, J.: *The Design and Implementation of the 4.4BSD UNIX Operating System*, Addison-Wesley (1996).

[6] Pendry, J.-S. and McKusick, M.: Union Mounts in 4.4BSD-Lite, *Proceedings of the USENIX 1995 Technical Conference*, New Orleans, LA, USA, pp. 25–33 (1995).

[7] Tichy, W. F.: RCS – A System for Version Control, *Software–Practice and Experience*, Vol. 15, No. 7, pp. 637–654 (1985).