Versioning Systems for Evolution Research

Romain Robbes
Faculty of Informatics
University of Lugano, Switzerland
romain.robbes@lu.unisi.ch

Michele Lanza
Faculty of Informatics
University of Lugano, Switzerland
michele.lanza@unisi.ch

Abstract

Research in evolution goes on par with the use of versioning systems by developers of the case studies. There is a great diversity of versioning systems with advantages and disadvantages both from the technical as well as from the conceptual point of view. In this paper we analyze the currently used versioning systems from the point of view of a software evolution researcher. Thus we do not focus on whether a certain versioning system is technically better than another one, but rather on what kind of information it offers for software evolution research. We present a non-exhaustive list of dimensions that are important for performing research in software evolution, do a survey on the current main-stream versioning systems and discuss what is actually needed for future versioning systems to support both software evolution and the related research field.

1 Introduction

The first software versioning systems appeared more than thirty years ago. Since then, the software industry has recognized the importance of versioning systems and is using them extensively since more than two decades now [14] [15] [10], boosted by the emergence of open source software. The importance of recording the history of a software system during its development has the advantage of allowing to reconstruct the original design intentions of the developers, as well as their subsequent variations in time. In this domain, versioning systems offers a set of functionalities that standard backup means such as tapes or CDs/DVDs can not offer, *e.g.,* the ease to quickly recover any version at any time, and the presence of meta-data which can be analysed and inspected with tools such as ViewCVS[1]. Example meta-data is the exact timestamp of a commit operation on the system, the identity of the person who performed it, *etc.*

Versioning systems also allow efficient sharing of a project, and hence ease team software development. The facilities in software management and the amount of data retrieved fostered the research field of software evolution[12], whose goal is to analyze the history of a software system and infer causes to its current problems, and possibly predict its future.

Recently, the primary source of information for software evolution research has been the open-source community, and some of its biggest software projects such as the Mozilla software suite [4], [9], or the Linux Kernel [8]. The open-source movement is indeed a very good opportunity for software evolution research, as such projects usually make their source code public, and grant anyone access to their CVS repositories. Obtaining such unrestricted access from a private software company is much more difficult. Since research in software evolution is very sensible to the quality and the quantity of information recovered, studies of such large-scale and open projects is very useful due to the amount of information contained here.

The problem we discuss in this paper is that the type and quality of evolution research that can be performed is constrained by the versioning system used by the developers of a system, as each versioning system generates only certain types of information. We analyze and compare the following versioning systems, which constitute the state of the practice:

- CVS (being the successor of SCCS[2] and RCS[3], actually the most used versioning system),
- Subversion (said to be the "next generation" CVS),
- and SourceSafe (a widely used commercial systems).

These versioning systems all have in common that the level of granularity is given by files, *i.e.,* the systems record file versions. As opposed to these we then compare them with an *entity-based* versioning system called StORE[4]:

[1] See http://viewcvs.sourceforge.net/ for more information.

[2] See http://cssc.sourceforge.net/manual/cssc.html for more details.
[3] See http://www.gnu.org/software/rcs/rcs.html.
[4] See http://www.cimcomsmalltalk.com.

StORE records versions of program entities like packages, classes, methods, *etc.*

Our comparison is from the point of view of a software evolution researcher and not from the point of view of a software developer[5]. On a sidenote, those interests should at least partially match, since researchers are ultimately working to support development, and need data from the developers to achieve their studies, so they should be able to communicate with the most widely used versioning systems. Conversely, developers might be interested in versioning systems researchers are using or recommending, as it opens the possibility to use tools developed by researchers as well as their expertise to their full extent.

The comparison between the file-based versioning systems and in respect to an entity-based versioning system leads us to deduce key aspects that a *future* versioning system should possess to not only allow for a more fine-grained history recording but also to permit high quality research in software evolution.

Structure of the paper. We start by describing the dimensions of interest for performing research in software evolution. We then compare a set of versioning systems with respect to these dimensions, and discuss their shortcomings. Afterwards, we present a versioning system which takes a quite different stance by recording the history of program entities, not mere files. We then highlight some of the future trends in versioning systems to ease software evolution and the related research. We conclude by discussing related work and give an outlook on our future work in this field.

2 Versioning Systems Dimensions

Each versioning systems has its strengths and weaknesses, as well as its technical particularities. The dimensions we compare are further divided in three groups: availability of evolution material, releveant technical features and high-level usability. We do not take into account all possible aspects, but are only interested in those that allow for performing software evolution research. Nevertheless, the following list has to be considered non-exhaustive:

2.1 Availability

Widespread usage: Since we need to analyse the evolution of real-world software, we should base ourselves on the most used versioning systems to have a wider panel of software to choose from, *i.e.*, performing evolution research based on a little known versioning system is of limited impact and allows for little generalization of the results.

License: It is now common to study open-source software, as it allows a complete access to the application's history. Since open-source software tends to use open-source versioning tools, their license models must be taken into account. Having access to the source code or the e-mail archives of a project, which is commonplace for open-source, can also partly make up for a lack of formal documentation.

Portability: For which operating systems do the versioning systems exist? Once again this points to the applicability and the impact of evolution research, *i.e.*, performing evolution research on a single-platform versioning system is of limited impact.

Generality: What kind of information can the versioning system handle? The most prominent one, CVS, has for example problems in handling binary files such as documentation or diagrams which are usually images, PDFs or other non human-readable files. Moreover, most large systems are written in more than one language and the versioning system must allow to handle such a case as well. The most generic versioning systems can handle virtually any kind of file, so they can also be used to version design documents. Genericity, and especially programming-language agnosticity is also a factor influencing widespread use of the versioning system, but also imposes a least common denominator on the operations the system is able to perform on the program's source.

2.2 Relevant technical features

Refactoring support: The most used refactoring that poses problems to current versioning systems is the renaming of files. Is the versioning system capable of keeping track of such refactorings and can it consider the renamed file as the same item or will it consider them as two different ones? If it considers these as two different entities, we must either find a way to link them, which can be computationally expensive [16], or have the history of one entity split in two parts, resulting in a rather important loss of information.

Changeset support: Can a versioning system group related changes, so that they appear as a single logical entity, even if they affect different files? An example of this is the duality often found between .cc and .h files in C++. The developer frequently needs to modify both the header and the implementation file at the same time, so having them fitting in the same changeset groups them logically. Changeset support allows to group in the same way changes whose relationship is not obvious (*e.g.*, modification of a method signature, and of all its references for example). Furthermore, changesets allow to separate changes happening in

[5] See http://better-scm.berlios.de/comparison/comparison.html for such a study.

the same file but who are ultimately unrelated to each other (such as two distinct bug fixes). Thoroughly using changesets could help tracking such logical groups of changes, if thoroughly used by the developers.

Line-wise history: Does the versioning system allow to recover the complete evolution of one single item, *i.e.,* down to the line level? Even if most versioning system are not tracking the evolution of software entities, some can track the evolution of single lines of code. This allows to operate at a finer-grained level than files.

Release tagging: How easy is it to tag a particular set of files to identify a release? Is it just a "good practice" or is it enforced by the system? Another question here is if the information can be used for analysis, and if it is actually used by the system. Support for such kind of meta-data can be useful for evolution analysis, as we can draw different hypothesis for a "stable" part of the system, compared with an "experimental" part of it.

Branching and Merging: Is it natural when using the system under study to make a new branch of the program, and to merge changes later on? This facility has an effect on the development style of the application. Easy branching might render the navigation between versions more difficult (the path between versions becomes a graph, and no longer a line), but it might on the other hand help to pinpoint particular sets of changes in combination with tagging. For example, it is highly probable that changes to a branch tagged as "stable" are bug fixes, and that changes applied to other branches are belonging to a particular set of features, each specific to a branch (if there are several of them). Having changeset support in conjunction to branching is useful: If branching is not commonplace, most feature additions or bug fixes will be added to the main branch, making it difficult to tell them apart.

Collaboration Style: Versioning systems were developed out of the need to handle concurrent modifications of a system by several developers. Thus the collaboration policy enforced by the versioning system has an influence on the development style of applications, in the same way merging or branching does. There are two main collaboration strategies: *concurrent development* and *file locking*. One allows any developer to make changes to any file, while the other imposes the developer to first check out a file, change it, and commit it again. In the meantime, nobody else can check the same file out. Of course, these are general rules, and some systems allow both strategies but put the accent on one in particular. If a locking policy is in place, developers will have a tendency to hold back their changes, and commit all changes on a specific file at the same time, rather than contributing them to the system on the fly. Some other phenomenoms, such as code ownership, might also affect the development of the system in some way (this being good or bad is out of the scope of the paper). Moreover, code ownership is more probable in the presence of a locking policy.

2.3 High-level usability

File-based vs. Entity-based: The fundamental question here is *what* to analyze. If a versioning system only versions files (as most of them do) then we can only perform research based on the file versions. If we want to perform research on the evolution of the software artifacts, *e.g.,* classes or methods, we must reconstruct those artifacts and their evolution.

Documentation: How much documentation is there for a given versioning system? A researcher in software evolution needs to retrieve information from the documentation to be able to exploit the available information as much as possible. The focus of the documentation should also be taken into account: a researcher needs different kinds of information than the one commonly found in user manuals.

Underlying technology: Is the versioning system using a database or plain text files? A database approach allows to perform research techniques such as data mining without requiring a preprocessing step. In the case of a file-based approach a researcher must first reconstruct the needed information and store it in an appropriate format, *i.e.,* a database. Furthermore, a database allows more complex queries and probably better performance, but is more sensible to data loss, so the two have distinct pros and cons.

Infrastructure needed: What kind of tools does the versioning system provide? We want to know what tools a researcher must use (or build) in addition to the versioning system to be able to use it effectively. The needs are vastly different than those of developers in this case. For example to use CVS a researcher might need to build his own database, or parse the source code with his own or a third-party parser. Having this provided by the versioning system would tremendously reduce the overhead faced by scientists when they have to build new tools [11]. Reduced overhead would allow in turn scientists to focus on the important aspects of their research.

Evolution information: What kind of information about the evolution (as opposed to the versions) of the items is provided by the versioning system? This varies from when a file has been modified and in which proportions, to retrieving accurate versions of each program entity, and allowing to compute or retrieve various kinds of high-level or lower level metrics of the system. These metrics can be about a specific system version, or they can characterise the evolution process itself[6, 7].

Information browsing and navigation: A comparison dimension related to the previous one is the complexity of the navigation between successive versions of the software entities under analysis. We seek to evaluate whether we have access to a lot of versions, and if some of this information if lost or needs to be processed to find history links.

If browsing is easy, we can formulate time-based queries on some specific entities and hence exploit the available information.

3 A Partial Comparison of Versioning Systems

Table 1 shows how well the three systems on the dimensions considered, alongside StORE which will be discussed later on. When appropriate, qualitative grades are used. The following paragraphs detail the findings for each dimension and each examined versioning systems.

Usage: CVS is the current standard, and is free software. It is hence very widely used, both in the open source community and in the industrial world. Subversion arrived recently (version 1.0 was issued in 2004), but it is already considered the successor of CVS, as it fixes some of CVS's most glaring technical flaws. Hence it will probably be the open source standard in a few years. SourceSafe is the solution provided by Microsoft, and is a natural choice for enterprises using Microsoft's IDEs and solutions.

License: CVS and Subversion are open-source, and are hence widely used for cost and ideological reasons. SourceSafe on the other hand is closed-source and is pay-per-use. It will thus only be used in a professional context, thus limiting its overall usage.

Portability: CVS and Subversion are multi-platform and well supported by the open-source community. For example, some people wrote GUIs for them, even if they were not designed for this at start. On the other hand, SourceSafe was mainly developed for the Windows family of operating systems. Nevertheless, several third-parties provide some support for it on Unix and Macintosh platforms, but their use is quite anecdotic.

Generality: The three systems considered can handle any kind of files, from Ascii source code to binary documents. But the quality of support varies. Subversion supports file directories, which eases file-level refactoring support. CVS is also less performant than the others at handling binary files (a manual option must be provided when such a file is added).

Refactoring support: CVS does not support basic refactorings such as moving and renaming of files. These split the history of the file. Since Subversion versions directories as well as files, it can detect when a file or directory is moved or renamed, so it supports file-level refactorings. SourceSafe is in-between, as it can support both refactorings, but not in automatic way. The procedure seems a bit complicated, and has been described by some as a "kludge", this is due to the fact that Sourcesafe does not support versioning of directories (or projects, as they are called in the SourceSafe terminology).

Changesets: Subversion provides a partial support for changesets. One changeset is implicitly produced on each commit, so that changes on separate files can be linked together, but different changes in the same file can not be separated using this approach. Neither CVS nor SourceSafe have changeset support.

Line-wise history: Both CVS and SubVersion can do this easily. It is less direct using SourceSafe and involves using a visual differentiation tool, so this may not be of use for evolution research, being difficult to use programatically.

Release tagging: All systems support tagging in some ways. Still, it seems a bit more difficult to rename a label (which is the SourceSafe term for tagging) in SourceSafe than in the other two systems. Subversion has an advantage due to changeset support (*i.e.*, tagging an entire change set is possible and useful).

Branching and Merging: CVS and Subversion support branching and merging quite well. It is based a variation of tagging for them. SourceSafe is more difficult to use with branching (SourceSafe's branching and merging support is qualified by some as "weak"), and is also unrelated to labelling. Subversion again has an advantage because of changesets, as they version a group of files rather than separated files.

Collaboration Style: Sourcesafe is the only studied system which enforces file locking rather than concurrent development. Therefore projects developed using Sourcesafe will have a tendency to have a more linear style of development, especially if we consider branching abilities too.

File vs Entities: None of the versioning systems studied is entity-based. They only rely on files. This is one of the major obstacles to performing high-quality evolution research, as the entities and higher-level artifacts such as subsystems or modules must be painfully reconstructed, leading to data loss. We later on compare this three systems with an entity-based one to underline the differences. It is also worth noting that Subversion supports versioning of directories in addition to versioning files. Depending on the used language, *e.g.*, Java, this may actually directly map to entities, *e.g.*, Java packages. Note that this is only a convenience.

Documentation: CVS is wery well documented [1] [5], and there is a great body of learning material on the web. Subversion is well documented too [2], as for example an online book is available on Subversion's website. Sourcesafe's documentation [13] is less developed but still decent. Being primarily a GUI application is one of the reason why it is less needed. However, the documentation for the three versioning systems are very developer-oriented (many tutorials and user manuals, rather than information about the files formats, for example), so they are not fully useful in an evolution research context. However, CVS and Subver-

Dimension	CVS	Subversion	SourceSafe	StORE
Usage	+ +	+	+	-
License	+ +	+ +	-	+/-
Portability	+ +	+ +	+	+ +
Generality	+	+ +	+ +	- -
Refactoring support	-	+ +	+	+ +
Changeset support	-	+	-	+ +
Line-wise history	+ +	+ +	+/-	+
Release tagging	+/-	+	-	+ +
Branching and Merging	+	+ +	+/-	+ +
Collaboration Style	Concurrent	Concurrent	Locking	Concurrent
File/Entity based	file-based	file-based	file-based	entity-based
Documentation	+	+	+/-	+
Underlying technology	-	+ +	+/-	+ +
Needed infrastructure	- -	-	- -	+ +
Evolution information	-	+/-	-/?	+
History navigation	-	+/-	-	+ +

Table 1. Comparing major versioning systems according to the discussed dimensions.

sion are open-source, which facilitates access to implementations details.

Underlying Technology: CVS is still based on the RCS format, which relies on simple files scattered around the repository (typically one history file per file considered). Both Subversion and SourceSafe are using a database, but Subversion's one seems better: Based on the Berkeley DB, Subversion's database appears to be reliable, and supports *atomic commits* (meaning all the commited files are added to the repository at the same time, which improves reliability as it is impossible to have a partial commit in case of a crashing client). On the other hand, it is considered a good practice to run a repair utility regularly (analyse.exe, once per week), on SourceSafe's database. Subversion being open-source, it is at least possible to access the database programatically (Subversion and the underlying Berkeley DB provides some APIs for that [6][7][8]).

Additional Infrastructure: To perform some analysis at a finer level than the one of files, all three systems require parsing the actual source code of the provided version(s). Therefore the additional infrastructure required to use these versioning systems varies from low (if the researcher works at the file-level and the database is already built) to very high (if he works at a finer-grained conceptual level). In the latter case, he has to parse the source code of the different versions, then build a model of the system at its various states, and to relate the entities from state X to state Y if a relationship can be made. Most of this work is already done in an entity-based versioning system, but this work remains to be done with file-based versioning systems.

Evolution information recovered: As said above, if no additional work is performed, the evolution information extracted is rather poor. CVS logs can be analysed to know who did a commit on a particular file, and when. We can also know the number of added and deleted lines in the file. Note that changed lines count for one added and one deleted line. Subversion has a notion of changeset, which is more useful than keeping track of individual files and their distinct version numbers. Hence Subversion allows one to easily detect related changes on a set of files. We were not able to explore this dimension for SourceSafe as we do not have its license[9]. The other kind of information that can be easily retrieved are high-level software metrics, such as the number of lines of a file across each version, or the number of files in each directories, being an indication of the module's size. Metrics such as the number of methods per class require a parsing for each considered versioning systems.

History navigation: In this area Subversion has an advantage since it keeps the history of renamed or removed files. Therefore navigation is more complete than in the two other systems, as less history information is lost. Otherwise fine-grained entities are still not considered for each system, if no extra processing work is done on each of the versions of the program under analysis.

[6]See http://www.linuxdevcenter.com/pub/a/linux/2003/04/24/libsvn1.html for more information.

[7]See http://www.linuxdevcenter.com/pub/a/linux/2003/05/15/libsvn2.html for more information.

[8]See http://www.cs.sunysb.edu/documentation/BerkeleyDB/ for more information.

[9]Note that this problem is one of the dimensions described.

4 Major shortcomings of these systems

This sections outlines the major shortcomings of the three studied systems, keeping in mind that they also apply to most versioning systems available. There are two shortcomings which have major consequences, and are the cause of most of the other ones: (1) Most systems are file-based, rather than entity-based, and (2) the fact that they are snapshot-based, not change-based, *i.e.*, the program is frozen as a snapshot with a particular time stamp without recording the actual changes that happen in between two subsequent snapshots.

4.1 Entities versioned

Several shortcomings of the surveyed systems can be traced back to one in particular: All of the commonly used versioning systems are file-based. The only item they version are files, and to some extent lines. While files are too coarse-grained for detailled analysis, lines seem to be too fine-grained, as virtually nobody uses this information in evolution research because of scalability issues.

Three important problem are at least partially caused by the fact that the three systems are file-based: (1) Difficulties in browsing the extracted information, (2) loss of evolution information, and (3) high costs of infrastructure.

Browsing the information in a precise way is not possible if one is stuck at the file level, as classes themselves, and even methods, won't be part of the navigation [10]. Information about these entities is lost too, as it is most of the time not considered at all, due to parsing difficulties. The information considered is only the number of files and directories and their relationships, as well as developer information and the amount of line added, deleted or modified for each commit.

The reason for this lack of information and its sparse organization which hampers navigation is the sheer difficulty of extracting it from the software and to establish relevant links between successive version of program elements. The infrastructure needed besides the version control system is too expensive (both in time and human effort involved) to justify the cost [11]. This three reasons combined (information loss, cost of retrieval, and difficult exploitation) explains why a good part of the community is doing research at the file and directories level.

If the infrastructure to obtain more information is implemented, the results obtained are relevant and focused on different aspects of the software evolution, such as the evolution of classes or class hierarchies [6] [7].

[10]With the exception of Java for classes. However, inner classes are also lost.

An entity-based versioning system on the contrary versions the software in much finer-grained ways. It can version packages, classes, and even individual methods of a complete system, during its entire lifespan. These objects can hence be queried directly, without needing a costly parsing step and the building and definition of domain objects. Thus the infrastructure needed is much lighter, and researchers can focus on more precise relationships. So the information extracted for each entity is much more complete and precise, and the number of entities built is much higher. It becomes possible to exploit the information to the point were the researcher can evaluate the variation of the lines of code of a single method over an extended period of time without an excessive cost.

We will see in the next section an example of such an entity-based versioning system, called StORE, and the possibilities it opens for software evolution research.

4.2 Limits of the snapshot concepts

Another shortcoming that neither of the versioning systems surveyed do solve, including StORE and most other entity-based versioning systems, is at the source of an important part of the loss of the software evolution information.

The problem we are describing here is that nearly all versioning systems are based on the concept of snapshots. A snapshot is a "frozen" version of the program at a given moment of time: it can not evolve anymore. The way people use versioning systems is by commiting changes from time to time to the system, when they feel that they have completed a feature, or done some significant work towards it, which they believe to be bug-free. For example, we will probably commit this paper in our repository once the writing of this section of it is finished, but not sooner, thus leading to a loss of all intermediate changes.

What happens between two commits is a mystery. We can only make some hypothesis about it by considering the two successive states at hand. Continuing our example, although we might have written some words and sentences in a specific order, and deleted some, none of these operations will appear in any versions stored in the (CVS) repository. In this respect, all these operations are completely lost.

While preventing further evolution of a program snapshot is good to ensure stability of releases, the fact that most modifications happening between two snapshots is not recorded represents a great loss of information. Developers tend to spend quite a lot of time between commits, because they are not confortable if they commit their changes every 5 minutes, fearing having too much revisions or commiting code in an imperfect state (which is not a problem, if the changes commited are tagged appropriately). Some computations based on the cvs logs of mozilla show that the mean time between two commits by the same developer on the project is in the order of several hours. Some developers commit their changes in even larger chunks, such as several days or weeks. In these extreme cases, the history of entire classes or modules may be reduced to a handful of snapshots.

Analysing information such as CVS logs is qualified as being "software archeology"[3] , whereas what is really needed is software history: we need much more information about the software under study than what is currently offered by snapshot-based approches.

Having more evolution information at hand would allow one to be much more precise in linking the evolution of related program elements. We could infer that some method are linked if they are modified regularly in a comparatively short period of time. We feel that such a rich level of information gives us an important set of new problems to explore.

5 An example of an entity-based versioning system: StORE

StORE is the versioning system used by the Cincom VisualWorks Smalltalk programming environment. StORE uses a centralized server and a database acting as a repository for versions of programs. Developers have accounts on the server, and can publish packages as they see fit.

StORE versions program-level entities, such as classes, methods, packages and bundles (a bundle being a kind of package able to contain packages and other bundles). It is thus easier to version programs with StORE, as most of the infrastructural needs are being taken care of by the framework. This is true for both from the researcher's and the developer's point of view. Developer can for example merge different versions of entities since they can think in terms of packages, classes and methods rather than files. As a side-note, some common cases such as formatting problems and changes in comments, which do not have a significance for the system, can be dealt with automatically, since StORE works at the programming language level. The cost for this ability is a decrease in genericity of the versioning system. The same kind of reasoning is possible for researchers: They can track down classes or methods with ease, and easily define metrics on them ignoring irrelevant changes such as formatting issues. This partly addresses the problems of infrastructure, information extraction and information navigation mentioned earlier.

Store also features blessing levels, which are special kind of tags whose use is enforced by the system. Common tags are: Broken, Work in Progress, Development, Integration-ready, Merged, and Release. They allow developer to pick particular branches and versions to base their work on, and ease merging. They can also be used as indicators by the software evolution researcher. Before commiting (publishing in the StORE terminology) some changes to the StORE repository, a developer must give it a blessing level that is recorded in the repository and can be used later on. The blessing level can be used for access control for example, and can be useful for software evolution research too. Changes to a release tagged as "Stable" should be mostly bug-fixes for example.

However, there are still some shortcomings: StORE is smalltalk-specific, as it is working only on smallltalk code. It is not widespread at all. Of course, being restricted to a language which is not really widespread is not positive either. Still, since the emergence of StORE practically all industrial-level and open-source applications written in VisualWorks are versioned with it, thus leaving a fertile ground for software evolution research.

Another shortcoming is that it is still snapshot-based, as CVS and Subversion are, which means that information loss is still inevitable, depending on how often developers publish their changes to the repository.

Table 1 shows a comparison of the three versioning systems analysed in this paper, along with StORE. We will from now on focus on the comparison of StORE as an example of an entity-based versioning system with file-based versioning systems, showing the strenghts of both kinds of systems. The reader should keep in mind that some of these items are specific to StORE and the other compared systems, and are not relevant to every file or entity-based versioning systems.

Usage: Store is used by a niche of users, so it is not widespread. This is a characteristic shared by all entity-based versioning systems, as they have to be widespread to be relevant for software evolution research.

Generality: Since all entity-based versioning systems record program entities, they are to a certain extent programming-language specific. This reduces their generality, as they

Refactoring support: Entity-based systems typically support higer-level refactorings than file and directories level refactoring such as move or renames. They can support renaming of program entities, and are thus much more focused. StORE is an extreme case here, as it does not even support file-level refactorings (this is due to the particular nature of Smalltalk, which is not a file-based programming language).

Changeset support: This is orthogonal to the notion of being file or entity-based.

Line-wise history: This level is often replaced by higher-level entities histories, such as methods and classes.

Line-level history is less important in the long term as one might lose sight of the forest because of all the trees standing in the way. can not handle any kind of file. Other versioning systems might be needed to store the documentation or the design documents living alongside the project. This is a factor which could hamper their widespread usage to a certain degree. On the other hand, being programming language specific allows for example to tell the difference between simple reformating of the code and changes at a deeper level involving modifications of the abstract syntax trees of methods.

Branching and merging: Since entity-based systems can work on a finer-grained level of abstraction, their merging and branching capabilities are higher, hence the development style of the application will be affected. [11].

Collaboration style: StORE enforces concurrency, but ENVY (another entity-based versioning system) enforces ownership of classes by developers, which means the concurrency policy is orthogonal.

Needed infrastructure: Since entity-based versioning systems store program entities directly, it is much easier to reflect on them without requiring a parsing step. Therefore they can be used with a far lighter extra infrastructure.

History navigation: For the same reason, such systems tend to link successive versions of entities at each level, making navigation between successive versions easier.

Evolution information: The infrastructure being lighter to use, less information is lost.

To sum it up, we could say that entity-based versioning systems reduce the work needed by both researchers (to analyse software systems) and developers (to maintain and develop those systems), as they operate on several levels of abstraction (class, method, package). They are most of the time language-dependent, and are more powerful for this, as they don't have to deal with the "lowest common denominator" between a great number of different files. On the other hand, they are usually much less generic, and they are currently still experimental.

6 Future trends in versioning systems easing software evolution

The following discussion is both from the point of view of a developer as well as of a researcher in software evolution. The main driver of evolution research is the exploitation of the history of a software system to help assessing its current state and predict its future. To do so, we must be able to perform research at a conceptual level: The systems as a whole is evolving, *i.e.*, *all* of its components down to the finest-grained level are changed. Therefore we must be able to track the evolution of entities such as classes, methods, variables, packages, *etc.*

Even more than that we must be able to track the lifetime of these entities even if they have undergone ”identity” changes, *i.e.*, renaming or moving.

The trends we would like to see would be to adress the two major shortcomings mentioned in the Section 4 and some other ones that we will list briefly.

We need to see entity-based versioning systems being more mainstream to reduce the work involved by both the developer and the researcher. We also think versioning systems experimenting with a change-based and not a snapshot-based approach would help in solving the second issue, which causes an important loss of the software's history.

Entity-based versioning systems are already a good basis for addressing the first concern, but they need to be widely used. Work still needs to be done in the area of change capture, which is the second major shortcoming, to minimize the loss of history information. The way to solve this problem could be to store each modification to the program as it happens, thus enforcing automatic "commits" in opposition to manual and irregular ones.

Another area where progress could be made is in increasing support for changesets, as they allow to group changes in a logical way. A set of changes can indeed span several physical entities, and is harder (if not impossible) to track down if entities are considered separately. This can be combined with a better support by the system of labelling, *e.g.*, set of changes can be tagged as bug fixes for example.

Increasing support for refactorings is also an important trend (Subversion supporting refactorings at the file level is a sign of this). Having a better support for more advanced refactorings would help to locate and characterize precisely when a given software system enters distinct parts of its life such as feature addition, bug fixing and refactoring.

7 Related Work

Many versioning systems are in existence, particularly in the open-source world. Since we were studying the most widespread ones, we did not consider all of them. Still, some of them are worth mentioning, such as Arch [12], Darcs

[11] Another entity-based versioning system named Monticello uses a simple yet effective merging algorithm which takes only methods in account.

[12] See http://www.gnu.org/software/gnu-arch/ for more information.

[13] Codeville [14] and Monotone [15].

Arch is entirely based on changesets, so it provides an interesting case for software evolution researchers. Its use is also growing, and may very well attain the critical mass to become really significant.

Darcs is based on a formal patch theory, which seems interesting from the technical point of view, but is rarely used.

Codeville uses a distincts and rather powerfull merge algorithm, and uses changesets, whereas Monotone does not use version numbers but rather hashes of the file contents to version them.

All these versioning systems use distributed repositories, which are useful to developers (a developer can work offline and commit to a local repository, and then merge his changes later on into another repository). Distributed repositories might make it harder to track down all versions of a software system, and hence be a cause of information loss if some things are not taken care of.

Other entity-based versioning systems exists, such as Monticello [16] and ENVY.

Monticello is the de-facto standard for the Squeak smalltalk community, whereas ENVY is now considered as "dead technology".

ENVY was originally designed for Visual Age Smalltalk, and later on ported for Visual Age Java[17].

The concept of a change-based (as opposed to version or snapshot-based) versioning system is partly implemented by Smalltalk changesets, which were the way to manage source code in Smalltalk from the start. A Smalltalk changeset records every compilation of methods, and every evaluation of code in the Smaltalk environment (as evaluated code must be compiled first). Since addition or modification of classes is ultimately done by evalutiong code, changesets can effectively reconstruct the state of a program in a very fine-grained way.

The problem is that Smalltalk changesets are low-level information, being merely a succession of commands to execute (commands who as a side effect are compiling methods or modifying classes), and can not really be used outside of their particular area. It is not really possible to use them as a tool for program comprehension, for example, as this involves computationally heavy processing.

Smalltalk changesets have now been superceded by StORE (in the VisualWorks Smalltalk environment) for packaging, storing, versioning, sharing and distributing program code, but they are still used alongside StORE as of today. Their main use now is crash recovery (in the form of change lists and change files, who can be executed to rebuild the program after a crash). A Smalltalk changeset could be seen as an incremental regular versioning system. The problem here is that every program was more or less one single changeset, or several changesets, but semantically unrelated. Managing them was done in a manual way, as well as merging of several changesets.

Last but not least, the main problem of the change sets is that they represent a niche technology with limited spread.

[13]See http://abridgegame.org/darcs/ for more information.

[14]http://codeville.org

[15]See http://www.venge.net/monotone/ for more information.

[16]See http://www.wiresong.ca/Monticello/ for more information.

[17]Is is interesting to notice that most entity-based versioning systems were pioneered in the Smalltalk community.

8 Conclusion and Future Work

Versioning systems currently used by software developers are not plainly satisfactory for evolution research, as they restrain the amount of evolution information we can work with. There is hence a trade-of to make between using more advanced versioning systems, and being able to make convincing case studies.

Our advice to researchers willing to study evolving systems is to base themselves on open-source products versioned using CVS, are they are the most common, and to anticipate at the same time that projects will eventually move to Subversion, which is very certainly the next standard to come, at least in the open-source world. Several major open source projects have already done that [18]. It fixes some of CVS's flaws, such as properly handling file and directory-level refactorings, and provides some useful information by its use of changesets. Keeping an eye on versioning systems such as Arch could be worthwhile too. Closed-source versioning systems such as SourceSafe will probably not be used by open-source developer anytime soon, especially if we consider what happened to the Linux kernel, which was using BitKeeper recently [19]. Since open-source software is as of now the biggest and most information-rich systems that we can analyse, working with open-source versioning software seems to be the most sensible choice.

Our claim that better versioning systems are needed still hold, and we think that an entity-based versioning system for a widespread language would be a great step forward for both software developers and researchers, as these systems handles a lot of parsing task for themselves. They hence ease the merging of conflict for developer as they abstract away the file level, allowing developers to concentrate on conflicts at the package, class and method level. They also allow researcher to find information at those levels, uncovering relations which might not be findable using other ways.

[18]For instance Apache (the most widely used web server has more than 190,000 revisions in its subversion repository), the KDE project (which has more than 400,000 revisions), and a great number of projects from the Debian distribution

[19]BitKeeper is a proprietary system which had a free license for the linux kernel, but decided to cancel it.

Our future works lies in devising a system going away from the snapshot metaphor, to adopt the change metaphor, in an attempt to move from software archeology to real software history, by attempting to lose as less information as possible. We also wish to adress other shortcoming mentioned in the "future trends" section, such as being able to track phases of refactoring, debugging and feature addition in the software life cycle.

References

[1] P. Cederqvist. Version management with CVS. Technical report.

[2] B. Collins-Sussman, B. W. Fitzpatrick, and C. M. Pilato. *Version Control with Subversion*. O'Reilly Media.

[3] M. D'Ambros. Software archaeology - reconstructing the evolution of software systems. Master Thesis, Politecnico di Milano, Apr. 2005.

[4] M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. In *Proceedings of the International Conference on Software Maintenance (ICSM 2003)*, pages 23–32, Sept. 2003.

[5] K. F. Fogel and M. Bar. *Open Source Development with CVS*. Coriolis Group Books, Scottsdale, AZ, USA, 2001.

[6] T. Gîrba, S. Ducasse, and M. Lanza. Yesterday's Weather: Guiding Early Reverse Engineering Efforts by Summarizing the Evolution of Changes. In *Proceedings of ICSM '04 (International Conference on Software Maintenance)*, pages 40–49. IEEE Computer Society Press, 2004.

[7] T. Gîrba, M. Lanza, and S. Ducasse. Characterizing the evolution of class hierarchies. In *Proceedings of European Conference on Software Maintenance (CSMR 2005)*, 2005.

[8] M. Godfrey and Q. Tu. Evolution in open source software: A case study. In *Proceedings of the International Conference on Software Maintenance (ICSM 2000)*, pages 131–142. IEEE Computer Society, 2000.

[9] M. W. Godfrey and E. H. S. Lee. Secrets from the monster: Extracting Mozilla's software architecture. In *Proc. of the Second Intl. Symposium on Constructing Software Engineering Tools (CoSET-00)*, June 2000.

[10] D. Grune. Concurrent Versions system, a method for independent cooperation. Technical report, Vrije Universiteit, Amsterdam, Netherlands, 1986.

[11] A. Hassan and R. Holt. Studying the evolution of software systems using evolutionary code extractors. In *IEEE International Workshop on Principles of Software Evolution (IWPSE04)*, pages 76–81, Sept. 2004.

[12] M. M. Lehman and L. Belady. *Program Evolution – Processes of Software Change*. London Academic Press, 1985.

[13] T. Roche. *Essential SourceSafe*. Hentzenerke, Whitefish Bay, Wis, 2001.

[14] M. J. Rochkind. The Source Code Control System. *Transactions on Software Engineering*, 1(4):364–370, 1975.

[15] W. F. Tichy. RCS — a system for version control. *Software — Practice and Experience*, 15(7):637–654, 1985.

[16] Q. Tu and M. W. Godfrey. An integrated approach for studying architectural evolution. In *10th International Workshop on Program Comprehension (IWPC'02)*, pages 127–136. IEEE Computer Society Press, June 2002.