A Survey of Version Control Systems

Ali Koc, M.A.
Graduate Center – City University of New York
New York, USA

Abdullah Uz Tansel, PhD
Baruch College – City University of New York
New York, USA

***Abstract*—**Version control has been an essential aspect of any software development project since early 1980s. In the recent years, however, we see version control as a common feature embedded in many collaborative based software packages; such as word processors, spreadsheets and wikis. In this paper, we explain the common structure of version control systems, provide historical information on their development, and identify future improvements.

Keywords- version control; revision control; collaboration.

I. Introduction

Change is a vital aspect of data. Most data encounters multiple modifications over the course of its life. For certain forms of complex data, version control (aka revision control) systems are commonly used to track changes; such as software source code, documents, graphics, media, VLSI layouts, and others.

The main purpose of version control systems is collaboration. The first version control systems appeared as far back as 1970s -- with the principle of easier management of source code files continuously modified by multiple software developers (1). Since then, software developers, engineers, scientists and even artists have shown increasing interest to version control systems.These systems provide the ability to see the evolution of data over time, a snapshot of it in certain point in time, ora way to recover/restore if needed. These aspects of version control systems made them a vital component of collaborative and groupware systems.

Version control systems allow sharing of data among nodes where every node can stay up to date with the latest version of the data. If anything goes wrong, with version control you can always discard your changes and start from the last known valid version. If the mistake was done long time ago, version control system has the facility to travel through the revision tree and continue from a version that works.

Version Control systems are not just about the change, but are also about the reasoning behind the changes. Often times, we would like to see not only the evolution of the data but also the reasoning behind it. In a Version Control System, each change is associated with answers to the following questions: who, when and why. For every change, a message explaining the change, the user and the date of the change is stored. This allows group of users to see who works how much, give credit or blame mistakes – *blame* is actually a command in some version control systems (2).

This paper is organized such that in section 2 we explain the structure of version control systems and provide an abstract depiction of their basic functionalities. Section 3 traces the history of version control systems and briefly describes commonly used software systems. We discuss possible functions that can be added to version control systems in Section 4, and Section 5 concludes the paper.

II. Structure of Version Control Systems

The data, its versions, and all the information associated with each version are stored in a location called *repository*. There are four repository models commonly employed in Version Control systems. The earlier version of Version Control systems kept repository *local* – the person making the changes and the repository would be on the same machine (1) (3). There are also examples of repository located on a *shared folder* which allows users from within a local area network to collaborate (4). Later on, *client/server* models became popular where the central repository is located on a server and all clients are able to read and submit changes (2) (5). In the recent years, *distributed* repository systems are getting increasing interest as they allow collaboration without the need of a central repository (6) (7) (8).

The determination of *atomic* (smallest) *data unit* to be tracked is an important aspect of Version Control Systems. An *atomic* data unit is the smallest portion unit of data where any change to a part of this data would constitute the whole unit to be marked as changed. For example, if we are trying to version control text files, the atomic data unit could be a line, sentence or a word (or a character for that matter). These data units could be applied to any text file. For example, a source code file is essentially a text file. However, contextual knowledge about the text file may allow us to choose a more appropriate atomic data unit. For example, for a source code file, a more appropriate atomic unit may be each statement of the programming language. Most version control systems today are text based. While there are number of systems that are more informed about the context of the text – such as systems that are aware of programming language, xml etc. – using line as the atomic data unit is by far the choice of most popular version control system solutions in use today (9).

While there are variety of systems developed and in use today for variety of data types, some common functionality exist in all version control systems.

In all version control systems, the first step is to get the data from the repository. The retrieval of data from the repository with possibly the intent of changing it is called *checkout*. If this is the first checkout, than the repository is empty, hence an empty data set is retrieved. The retrieved data set is called the *local/work copy*. Some version control systems allow user to place a *lock* on the checked out data in the repository. This avoids concurrency problems, where multiple people may checkout the same parts of the data set with the intention of changing. Other systems allow multiple users to checkout and modify the same parts of the data set and deals with concurrency problems later on.

On the checked out local copy, user works and makes changes. When the work is completed, the new modified data set needs to be sent to the repository for storage. The action of sending the changed data set to the repository is called *commit*. Each commit may contain changes to different parts of the data set – for example, a program source code may contain many source code files and multiple files may have been changed and needed to be committed. It is desirable for version control systems to perform *atomic commits*, ensuring that if any part of the data set being committed fails to be accepted by the repository, than the whole commit should fail in order to not to compromise the consistency of the data.

Most systems require users to write a message with each commit called *commit message*. This message along with the name of the user and the date of the commit will be associated with the changed data set. Each commit is assigned a unique *version number*. For most version control systems, this is a sequential number starting from version 1. For version control systems using distributed repository model, it is not possible to coordinate a sequential numbering on the versions, hence they employ other methods to assign unique version numbers to each commit. Most popular methods include pseudorandom number assignment and/or using hash of the changes as the version number (7) (8).

If a lock was not placed on the repository upon check out, than a prior user may have checked out the same data and then committed it before the user; which would put the user's local copy out of sync with the repository. If any of the atomic data units modified in the local copy of the user has been changed on the repository, the local copy of the user goes into *conflict* state. Note that, it is the local copy of the user that is in conflict with the repository. For most version control systems, conflict is not a valid state and the repository will not allow the user to commit the new modified data set until the user reads and incorporates the committed changes of the prior user (2). Once a local copy goes in conflict state, user needs to review conflicting local and repository changes and make further changes if necessary to resolve conflicts. Version Control systems allow *resolved* feature to get the local copy out of the conflict state. If same data set is changed but no changes collide on atomic data units (no conflict), the version control system would automatically incorporate the changes.

Often times, user may want to update the local working copy from the repository as there may have been possible commits since the last checkout. If the update contains changes that would conflict with the local changes, local copy goes into conflict state. The changes on the local copy needs to be reviewed and should be marked as resolved after complying with the changes of the repository.

The changes may be stored in the repository using various models. The simplest model is *snapshots*, where a complete copy of the data set is stored for each version. For certain data types, this method is still employed – such as in the version control of images, binary media where determination of atomic data unit is not necessarily well defined. For data sets that have a well defined atomic data unit, the most common method is *changesets (deltas)*, where only the modified atomic data units are stored with each version. This, in most cases, provides significant saving in repository storage space.

The *changeset* method, however, comes with the need of data set construction in order to get to a certain version of it. For example, in order to check out the latest version of the data set, a build process is required starting from the initial version of the data set (version 1), incorporating each delta, all the way to the last version, traversing the complete version tree. A typical software project repository is composed of thousands of versions. Assuming most users are interested in the latest version of the data set, this creates a significant overhead, especially for large repositories. To avoid this overhead, most version control systems employ *reverse delta* scheme. In this scheme, the last version of the data set is stored and all the changes are recorded as reverse deltas from this data set.

Atomic Data Unit	Repository
Line, Sentence, Statement, Node, etc.	Local, Shared Folder, Client/Server or Distributed
Concurrency	**Storage**
Lock, Merge or Both	Snapshot, Forward Delta or Reverse Delta

Table 1 - Classification of Version Control Systems

Sometimes it is desirable to evolve data sets in multiple directions in parallel and have the option to merge the evolutions later on into main data set. For example, a mechanical engineering firm may start with a core design of a device, and then create multiple branches out by incorporating different styles and features. At some stage during development, some branches may prove to be unsuitable, hence discontinued; while other branches may contain features they would like to incorporate to the core design. Many types of data sets go through this type of evolution making *branches* a vital feature of version control systems. The root of the repository (the main data set) is called the *trunk*. It is possible to branch out from any version of the trunk. Also, most version control systems allow branching out from other branches as well (10). *Tags* are special types of branches that mark a milestone along the evolution of a data set – ex. major release of software, an approved version of a prototype. *Merge* functionality allows changes on one branch to be applied to another branch.

Let's setup a repository system for data set x. The snapshot versions of x are represented as $x_1, x_2, x_3, x_4 \ldots x_n$, where n is the number of the latest version. Let x_0 represent the initial state of the repository. For systems using deltas, they are represented as:

$$dx_i = x_{i+1} - x_i$$

Keep in mind that these versions may have been committed to different branches of the repository.

	SNAPSHOT	FORWARD DELTA	REVERSE DELTA
Checkout	$y = x_n$	$y = \sum_{i=1}^{n} dx_i$	$y = \sum_{i=n}^{1} dx_i$
Commit	$x_{n+1} = x_n + dy$	$dx_{n+1} = dy$	$x_{n+1} = x_n + dy$ $x_n = x_{n+1} + x_n$
Update	$y' = x_n + (y' - y)$	$y' = y' + \sum_{i=k}^{n} dx_i$	$y' = y' + \sum_{i=n}^{k} dx_i$

Table 2 - Basic operations of a version control system

A typical use scenario of a version control system would follow these steps:

Checkout: $y = x_n$; where y is the new local working copy

Modify: $y' = f(y)$; where y' is the modified copy, and f is the work on y.

Delta: $dy = y' - y$;

Commit: $x_{n+1} = x_n + dy$

Above operations are for a snapshot based system. In the case of a version control system where changeset method is employed:

Checkout: $y = \sum_{i=1}^{n} dx_i$; where y is the new local working copy

Modify: $y' = f(y)$; where y' is the modified copy, and f is the work on y.

Delta: $dy = y' - y$;

Commit: $dx_{n+1} = dy$

III. The History of Version Control Systems

A. VMS

The history of version control can be traced back to DEC's VMS operating system, which employed a natural technique of tracking revisions of files by never deleting them. The operating system simply created a new file with the same name but attaching a different sequence (version) number. The system's storage requirement was expensive and it created just too many files that were hard to distinguish by users. The system versioned files but it did not provide version control (11).

B. SCSS

The first real version control system goes back to 1972 when a Source Code Control System (SCCS) started getting developed by Marc J. Rochkind at Bell laboratories as a set of commands developed for OS/MVT, and later on UNIX. Though, he did not use the exact version control terms that we use today, his paper laid out the first clear version control system based on forward deltas, incorporating checkout, commit, and locking system to avoid conflicts (1).

SCCS was primarily developed to version control source code changes in software development environments. Each source code module is assumed to be in its own file and each file is version controlled independently. The system generates deltas using two primitives: insert line and delete line. Changing a line (even only one character change) is basically treated as the combination of deleting the current version of the whole line and then inserting the new version. Also movement of a block lines from one position to another within the module also treated as deleting the block of lines from its existing location and inserting them into their new position.

All the deltas are stored within the file in a special section called body. The body contains the text deltas of insertions (text records) and an extra record enclosing the text delta specifying the effect of it on the document (control records). For example, a new line inserted in a module would be represented inside the module file with the new text inserted into its position and enclosed with insertion control record and end control record codes indicating the version number and the extend of the new data.

While the version tree of SCCS looks linear, the sense of branching is still there through the use of version numbers and optional deltas. For example, version 1.5 could be interpreted as branch 1 version 5. Programmers can post deltas (commit) to version 1 (only to end of it), while development is going on for version 2. In the earlier version of SCCS, the new delta applied to version 1 (let's say v1.6) would be automatically carried to version 2, hence the linear version structure. The support for optional deltas allows certain deltas to be ignored providing a sense of branching and version tree type structure. For example, a new delta applied to version 1 (ex. v1.6) would have the flag/tag/option-letter so that it is ignored for version 2. This feature, as explained in the paper, is also used to incorporate temporary fixes only for certain customers (1).

Marc explains the motivation behind the development of SCCS as the common challenges faced in software development life cycle. As he puts it, as soon as something goes wrong, the first question an experienced programmer asks is "What changed?", and SCCS was designed to address just that. SCCS was not only used for source code control, it was also used for documentation versioning. In fact, in the paper, Marc says that it would be appropriate to call the system "text control" rather than "source control" (1).

Some UNIX distributions included SCCS as part of their standard command set. SCCS remained the dominant version control system until the release of Revision Control System (RCS) (10).

C. RCS

Revision Control System (RCS) was developed in 1980s by Walter F. Tichy, as the successor of SCCS with significant improvements. RCS organizes revisions into ancestral tree where the initial revision is the root of this tree. The edges of the tree indicate from which revision a leaf is evolved. RCS introduced the concept of merging (mergediff) by loosening the controls and implementing access controls to detect and prevent conflicts. The terms check-in (ci command), check-out (co command), branching (based on version numbering system), update are used similar to the way we use today (3).

Unlike SCCS, RCS used reverse delta method for storage where the most recent revision on the trunk is stored intact and all the other deltas basically describes how to go backwards from the most recent version. This, of course, has the advantage of making the checkout of the most recent copy simple and fast. It is also simpler to commit as the operation of adding a new revision is now composed of placing the document being checked in directly as the most recent version and replace the previous version with a reverse delta. Branching, however, is handled with forward deltas, thus reaching to the tip of braches can be costly. Performance gain can be achieved by implementing caching for the most recently accessed revisions (3).

RCS has been used to version control wide variety of data, such as source text of drawings, VLSI layouts, documentation, specifications, test data, form letters and articles.

The main disadvantage of version control systems up until this point, however, was that they operated only on single files and did not provide capability to handle projects consisted of multiple files.

D. CVS

CVS (Concurrent Versions System) was developed by Dick Grune as Unix shell scripts around RCS with the motivation to work collaboratively with his students on the development of a C compiler called ACK. If it is used for the version control of a single file CVS could be considered as a wrapper to RCS. The real advantage and usefulness of CVS comes from the fact that it lets users treat a whole set of files as if it is a single file, making version control (commands) simpler for multi file projects. Using symbolic mapping, CVS keeps a database of symbolic names to a set of directories and files. A single command can manipulate an entire collection of directories and files (5).

CVS also introduced distributed use by providing a client/server model allowing multiple developers at remote locations to interact with the version control system as a team. Version history of the project is stored on a central server while users worked on their local work copy on their client machines. This made certain operations of the version control system network availability dependent. However, in his paper, Dick Grune explains this as slight inconvenience under normal operation; and later on, as network systems became more dependable, it became an even less of a concern.

As its name suggests (Concurrent), another big advantage of CVS from its predecessors was that it allowed checkout of files without locking. That means users could work on the same file at the same time.

CVS made its mark to 1990s as the choice of version control system. However, CVS's lack of will to provide certain features to keep up with the changing trends, new clones of CVS started to popup: CVSNT, EVS, Open CVS and Subversion. Subversion gained traction and became the popular version control system of 2000s. In early to mid-2000s, most CVS users began to move to Subversion (10).

E. Subversion

Subversion was developed by CollabNet as the "better CVS". They wanted to create a version control systems that is free of the limitations of its predecessor (CVS) and is more suited for the changing trend of the development projects (2).

Perhaps the most important innovation of Subversion was its support for atomic commits. The word atomic is used in the context similar to the way it is used in transactional database systems. If a commit of a set of files at some point fails to complete (by perhaps crashing the system or simply being rejected by the system due to conflicts), the central database holding the changes should be resilient and have the ability to reject all the changes of the commit process, keeping the system consistent and free of corruption.

Another important capability Subversion provided over CVS was the moving and renaming of files and directories. CVS did not version file names and directories. As refactoring became more popular in software development projects, the need for this feature became apparent. Refactoring is the act of restructuring an existing set of code modules, altering their structure without changing the behavior. Today refactoring is used to improve code quality, reliability, and maintainability throughout the software lifecycle. The process of restructuring commonly requires module (file) names to be changed and/or moved. *Move* and *Rename* are standard operations in Subversion (2).

Subversion also introduced full support for Unicode and non-ASCII file names, full support for binary files, branching and tagging as cheap operations, all of which lacked by CVS system.

F. *Distributed Version Control Systems (Git, Mercurial, Bazaar, etc)*

In the late 2000s, version control discovered the advantages of distributed computing and started to move away from central repository and simple client/server model. Today, they are considered to be the future of version control systems.
Distributed version control systems keep entire repository on user's local computer, making it better suited for large projects with more independent developers. Distributed version control systems such as Git, Darcs, BitKeeper, Mercurial, Bazaar, SVK and Monotone provide significant advantages of central version control systems by allowing users to work and use full version control feature set even when there is no network connection -- a limitation inherited by central systems starting from CVS. Since dependability to network is less, then those operations that did require network connection are now much faster. In central version control systems, the changes are version tracked only when the changes are committed to the server, however, distributed systems allowed version control of changes done locally allowing early drafts of work to be revisioned without requiring it to be published to others.

The main disadvantage of distributed systems is that they are less intuitive from the user's point of view. They lack understandable version numbering system (as there is no central server to assign versions). They usually use hashes of the changes or unique GUIDs (12).

Lack of a central server also made system backups harder. In a central client/server model backups are taken on the server repository ensuring that all committed changes are taken into account during the backup process. In the case of the distributed model, every client has their own repository and there is no guarantee that any one of the nodes has all the changes. That means backup operation needs to be done independently by each node.

IV. Enhancing Version Control Systems

Distributed version control systems are getting wide acceptance, especially among software developers. One of the foreseeable future improvements would be to incorporate features of social networking into version control systems. For example, if we take a software engineering project and a set of developers, each developer would benefit from knowing which developer is working on which parts of the project, even ahead of the actual commit action. We propose a new feature to version control systems, called "anticipate", that provides a look ahead for developers where they would know which users are online/changing files, and which files are currently being worked on.

Ability to see real time work of team members, and perhaps communicate via instant messaging would greatly benefit collaboration. It would also avoid future conflicts. Usually conflicts occur when two developers work on the same part of the project and try to commit their changes. The "anticipate" feature would allow developers to communicate ahead of time, alleviating the complication of merging processes; perhaps even avoid conflicts, or at the very least, it would assist in conflict resolution process.

Currently, version control systems are mostly available as standalone application (CVS, SubVersion, GIT etc.). However, the recent trend has been to incorporate version control into various types of software. Word processors such as Microsoft Word, OpenOffice Writer, KWord, Pages built in version control systems. Some spreadsheet applications Microsoft Excel, Open Office Calc, KSpread, Numbers also include version control features (13). Autodesk Vault, a data management tool for revision management, is part of many Autodesk products, used mostly (but not limited to) version management of CAD documents for architectural, mechanical, civil and electrical engineering (14) (15) (16). In fact, there is number of research around version control of VLSI (Very-Large-Scale Integration) processes dating back to 1980s (17) (18). Version control on hypermedia systems was the focus of research during 1990s (19) (20). More recent research exists on version control of Simulink models (21), version control of journal articles and so on. One research area could be exploring the possibility of integrating version control directly into development languages, or perhaps supplementing the development environments in the form of a framework, such that developers could easily create applications with version control capabilities. This could be considered as bringing version control to the context of the application.

V. Conclusion

Version control is an essential part of collaboration systems that is becoming widespread in every sphere of life. We have examined the version control systems by considering repository structure, concurrency, storage, and data unit. We have also covered the history and basic functionality of version control systems.

There are several directions for research in version control systems. One area of research could be making version control available to wide variety of domains by exploring the possibility of integrating version control feature directly into development languages, or perhaps supplementing the development environments in the form of a framework, such that developers could easily create applications with built-in version control capabilities. This could be considered as bringing version control to the context of the application.

Another foreseeable future improvement would be to incorporate features of social networking, allowing users to be able to interact beyond the set operation of the version control systems.

VI. Bibliography

1. The Source Code Control System. Rochkind, Marc J. 1975. IEEE Transactions on Software Engineering.

2. Pilato, C, Collins-Sussman, Ben and Fitzpatrick, Brian. Version Control with Subversion. s.l. : OReilly Media, Inc.,, 2008.

3. RCS - A system for version control. Tichy, Walter F. 7, s.l. : John Wiley & Sons, Inc., July 1985, Softw. Pract. Exper., Vol. 15, pp. 637--654. 0038-0644.

4. MacDonald, Josh, Hilfinger, Paul and Semenzato, Luigi. PRCS: The project revision control system. Lecture Notes in Computer Science. 1998, Vol. 1439, pp. 33-45.

5. Grune, Dick. Concurrent Versions System, A Method for Independent Cooperation. 1986. techreport.

6. Loeliger, Jon. Version Control with Git: Powerful Tools and Techniques for Collaborative Software Development. s.l. : O'Reilly Media, 2009.

7. OSullivan, Bryan and Bryan, OSullivan. Mercurial: The Definitive Guide. s.l. : OReilly Media, Inc.,, 2009.

8. Darcs: distributed version management in haskell. Roundy, David. s.l. : ACM, 2005. pp. 1--4.

9. Version Control. Grogg, Jill E. and Weddle, Jeff. 2010. Encyclopedia of Library and Information Sciences 3rd ed.

10. The history of version control. Ruparelia, Nayan B. 1, s.l. : ACM, January 2010, SIGSOFT Softw. Eng. Notes, Vol. 35, pp. 5--9. 0163-5948.

11. Sharick, Paula. The essential guide to VMS utilities and commands: VMS version 5. s.l. : Van Nostrand Reinhold Co., 1990.

12. Efficient Use of GUIDs. Lutteroth, Christof and Weber, Gerald. s.l. : IEEE Computer Society, 2008. pp. 115--120.

13. Towards XML version control of office documents. Ronnau, Sebastian, Scheffczyk, Jan and Borghoff, Uwe M. s.l. : ACM, 2005. pp. 10--19.

14. A Unifying Framework for Version Control in a CAD Environment. Chou, Hong-Tai and Kim, Won. s.l. : Morgan Kaufmann Publishers Inc., 1986. pp. 336--344.

15. Computer-Aided Software Engineering in a distributed workstation environment. Leblang, David B and Chase, Jr. 3, s.l. : ACM, April 1984, SIGSOFT Softw. Eng. Notes, Vol. 9, pp. 104--112. 0163-5948.

16. A study of version control for collaborative CAD. Chang, Zhiyong, Zhao, Jie and Mo, Rong. s.l. : Springer-Verlag, 2007. pp. 140--148.

17. Modeling concepts for VLSI CAD objects. Batory, D S and Kim, Won. 3, s.l. : ACM, September 1985, ACM Trans. Database Syst., Vol. 10, pp. 322--346. 0362-5915.

18. Batory, Don S and Kim, Won. Support for Versions of VLSI CAD Objects. s.l. : University of Texas at Austin, 1985. techreport.

19. Nested composite nodes and version control in an open hypermedia system. Fernando, Luiz, et al. 6, s.l. : Elsevier Science Ltd., September 1995, Inf. Syst., Vol. 20, pp. 501--519. 0306-4379.

20. Deep hypertext with embedded revision control implemented in regular expressions. Grishchenko, Victor. s.l. : ACM, 2010. pp. 3:1--3:10.

21. Modern Revision Control and Configuration Management of Simulink Models. Sauceda, Jeremias and Kothari, Suraj. Detroit, MI : SAE 2010 World Congress & Exhibition, 2010. Model-Based Design of Embedded Systems.

22. Sint, Rolf, Schaffert, Sebastian and Stroka, Stephanie. Combining Unstructured, Fully Structured and Semi-Structured Information in Semantic Wikis. Heraklion, Greece : 4th Workshop on Semantic Wikis, June 2009.

23. Supporting 3D City Modelling, Collaboration and Maintenance through an Open-Source Revision Control System. Roupé, Mattias and Johansson, Mikael. 2010. CAADRIA 2010 NEW FRONTIERS. pp. 347-356.

24. A survey and comparison of CSCW groupware applications. Rama, Jiten and Bishop, Judith. s.l. : South African Institute for Computer Scientists and Information Technologists, 2006. pp. 198--205.

25. A database approach for managing VLSI design data. Katz, Randy H. s.l. : IEEE Press, 1982. pp. 274--282.

26. CSCW: history and focus. Grudin, Jonathan. 1994, IEEE, pp. 19-26.

27. Using text animated transitions to support navigation in document histories. Chevalier, Fanny, et al. s.l. : ACM, 2010. pp. 683--692.

28. Version management of composite objects in CAD databases. Ahmed, Rafi and Navathe, Shamkant B. 2, s.l. : ACM, April 1991, SIGMOD Rec., Vol. 20, pp. 218--227. 0163-5808.

29. Supporting distributed collaboration through multidimensional software configuration management. Chu-Carroll, Mark C and Wright, James. s.l. : Springer-Verlag, 2003. pp. 40--53.

30. A fine-grained and flexible version control for software artifacts. Junqueira, Daniel C, Bittar, Thiago J and Fortes, Renata P. s.l. : ACM, 2008. pp. 185--192.