Version control

A component of software configuration management, **version control**, also known as **revision control** or **source control**,[1] is the management of changes to documents, computer programs, large web sites, and other collections of information. Changes are usually identified by a number or letter code, termed the "revision number", "revision level", or simply "revision". For example, an initial set of files is "revision 1". When the first change is made, the resulting set is "revision 2", and so on. Each revision is associated with a timestamp and the person making the change. Revisions can be compared, restored, and with some types of files, merged.

The need for a logical way to organize and control revisions has existed for almost as long as writing has existed, but revision control became much more important, and complicated, when the era of computing began. The numbering of book editions and of specification revisions are examples that date back to the print-only era. Today, the most capable (as well as complex) revision control systems are those used in software development, where a team of people may concurrently make changes to the same files.

Version control systems (**VCS**) most commonly run as stand-alone applications, but revision control is also embedded in various types of software such as word processors and spreadsheets, collaborative web docs[2] and in various content management systems, e.g., Wikipedia's page history. Revision control allows for the ability to revert a document to a previous revision, which is critical for allowing editors to track each other's edits, correct mistakes, and defend against vandalism and spamming in wikis.

Contents

External links

Overview

In computer software engineering, revision control is any kind of practice that tracks and provides control over changes to source code. Software developers sometimes use revision control software to maintain documentation and configuration files as well as source code.

As teams design, develop and deploy software, it is common for multiple versions of the same software to be deployed in different sites and for the software's developers to be working simultaneously on updates. Bugs or features of the software are often only present in certain versions (because of the fixing of some problems and the introduction of others as the program develops). Therefore, for the purposes of locating and fixing bugs, it is vitally important to be able to retrieve and run different versions of the software to determine in which version(s) the problem occurs. It may also be necessary to develop two versions of the software concurrently: for instance, where one version has bugs fixed, but no new features (branch), while the other version is where new features are worked on (trunk).

At the simplest level, developers could simply retain multiple copies of the different versions of the program, and label them appropriately. This simple approach has been used in many large software projects. While this method can work, it is inefficient as many near-identical copies of the program have to be maintained. This requires a lot of self-discipline on the part of developers and often leads to mistakes. Since the code base is the same, it also requires granting read-write-execute permission to a set of developers, and this adds the pressure of someone managing permissions so that the code base is not compromised, which adds more complexity. Consequently, systems to automate some or all of the revision control process have been developed. This ensures that the majority of management of version control steps is hidden behind the scenes.

Moreover, in software development, legal and business practice and other environments, it has become increasingly common for a single document or snippet of code to be edited by a team, the members of which may be geographically dispersed and may pursue different and even contrary interests. Sophisticated revision control that tracks and accounts for ownership of changes to documents and code may be extremely helpful or even indispensable in such situations.

Revision control may also track changes to configuration files, such as those typically stored in `/etc` or `/usr/local/etc` on Unix systems. This gives system administrators another way to easily track changes made and a way to roll back to earlier versions should the need arise.

History

Structure

Revision control manages changes to a set of data over time. These changes can be structured in various ways.

Often the data is thought of as a collection of many individual items, such as files or documents, and changes to individual files are tracked. This accords with intuitions about separate files but causes problems when identity changes, such as during renaming, splitting or merging of files. Accordingly,

some systems such as Git, instead consider changes to the data as a whole, which is less intuitive for simple changes but simplifies more complex changes.

When data that is under revision control is modified, after being retrieved by *checking out*, this is not in general immediately reflected in the revision control system (in the *repository*), but must instead be *checked in* or *committed.* A copy outside revision control is known as a "working copy". As a simple example, when editing a computer file, the data stored in memory by the editing program is the working copy, which is committed by saving. Concretely, one may print out a document, edit it by hand, and only later manually input the changes into a computer and save it. For source code control, the working copy is instead a copy of all files in a particular revision, generally stored locally on the developer's computer;[note 1] in this case saving the file only changes the working copy, and checking into the repository is a separate step.

If multiple people are working on a single data set or document, they are implicitly creating branches of the data (in their working copies), and thus issues of merging arise, as discussed below. For simple collaborative document editing, this can be prevented by using file locking or simply avoiding working on the same document that someone else is working on.

Revision control systems are often centralized, with a single authoritative data store, the *repository,* and check-outs and check-ins done with reference to this central repository. Alternatively, in distributed revision control, no single repository is authoritative, and data can be checked out and checked into any repository. When checking into a different repository, this is interpreted as a merge or patch.

Graph structure

In terms of graph theory, revisions are generally thought of as a line of development (the *trunk*) with branches off of this, forming a directed tree, visualized as one or more parallel lines of development (the "mainlines" of the branches) branching off a trunk. In reality the structure is more complicated, forming a directed acyclic graph, but for many purposes "tree with merges" is an adequate approximation.

Revisions occur in sequence over time, and thus can be arranged in order, either by revision number or timestamp.[note 2] Revisions are based on past revisions, though it is possible to largely or completely replace an earlier revision, such as "delete all existing text, insert new text". In the simplest case, with no branching or undoing, each revision is based on its immediate predecessor alone, and they form a simple line, with a single latest version, the "HEAD" revision or *tip*. In graph theory terms, drawing each revision as a point and each "derived revision" relationship as an arrow (conventionally pointing from older to newer, in the same direction as time), this is a linear graph. If there is branching, so multiple future revisions are based on a past revision, or undoing, so a revision can depend on a revision older than its immediate predecessor, then the resulting graph is instead a directed tree (each node can have more than one child), and has multiple tips, corresponding to the revisions without children ("latest revision on each branch").[note 3] In principle the resulting tree need not have a preferred tip ("main" latest revision) – just various different revisions – but in practice one tip is generally identified as HEAD. When a new revision is based on HEAD, it is either identified as the new HEAD, or considered a new branch.[note 4] The list of revisions from the start to HEAD (in graph theory terms, the unique path in the tree, which forms a linear graph as before) is the *trunk* or *mainline.*[note 5] Conversely, when a revision can be based on more than one previous revision (when a node can have more than one *parent*), the resulting process is called a *merge*, and is one of the most complex aspects of revision control. This most often occurs when changes occur in multiple branches (most often two, but more are possible), which are

then merged into a single branch incorporating both changes. If these changes overlap, it may be difficult or impossible to merge, and require manual intervention or rewriting.

In the presence of merges, the resulting graph is no longer a tree, as nodes can have multiple parents, but is instead a rooted underlined directed acyclic graph (DAG). The graph is acyclic since parents are always backwards in time, and rooted because there is an oldest version. However, assuming that there is a trunk, merges from branches can be considered as "external" to the tree – the changes in the branch are packaged up as a *patch,* which is applied to HEAD (of the trunk), creating a new revision without any explicit reference to the branch, and preserving the tree structure. Thus, while the actual relations between versions form a DAG, this can be considered a tree plus merges, and the trunk itself is a line.

In distributed revision control, in the presence of multiple repositories these may be based on a single original version (a root of the tree), but there need not be an original root, and thus only a separate root (oldest revision) for each repository, for example, if two people starting working on a project separately. Similarly in the presence of multiple data sets (multiple projects) that exchange data or merge, there isn't a single root, though for simplicity one may think of one project as primary and the other as secondary, merged into the first with or without its own revision history.

Example history graph of a revision-controlled project; trunk is in green, branches in yellow, and graph is not a tree due to presence of merges (the red arrows).
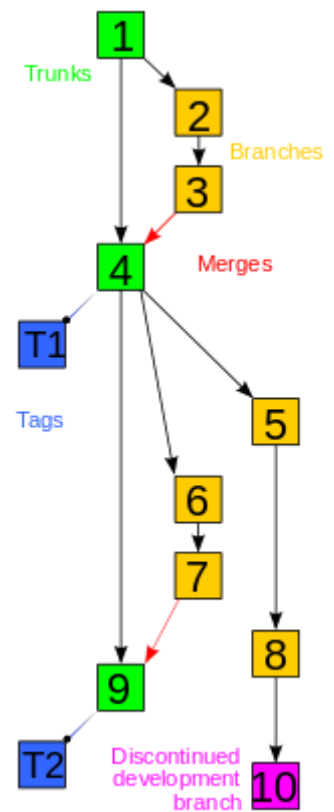
Specialized strategies

Engineering revision control developed from formalized processes based on tracking revisions of early blueprints or bluelines. This system of control implicitly allowed returning to an earlier state of the design, for cases in which an engineering dead-end was reached in the development of the design. A revision table was used to keep track of the changes made. Additionally, the modified areas of the drawing were highlighted using revision clouds.

Version control is widespread in business and law. Indeed, "contract redline" and "legal blackline" are some of the earliest forms of revision control,[3] and are still employed in business and law with varying degrees of sophistication. The most sophisticated techniques are beginning to be used for the electronic tracking of changes to CAD files (see product data management), supplanting the "manual" electronic implementation of traditional revision control.

Source-management models

Traditional revision control systems use a centralized model where all the revision control functions take place on a shared server. If two developers try to change the same file at the same time, without some method of managing access the developers may end up overwriting each other's work. Centralized revision control systems solve this problem in one of two different "source management models": file locking and version merging.

Atomic operations

An operation is *atomic* if the system is left in a consistent state even if the operation is interrupted. The *commit* operation is usually the most critical in this sense. Commits tell the revision control system to make a group of changes final, and available to all users. Not all revision control systems have atomic commits; notably, CVS lacks this feature.[4]

File locking

The simplest method of preventing "concurrent access" problems involves locking files so that only one developer at a time has write access to the central "repository" copies of those files. Once one developer "checks out" a file, others can read that file, but no one else may change that file until that developer "checks in" the updated version (or cancels the checkout).

File locking has both merits and drawbacks. It can provide some protection against difficult merge conflicts when a user is making radical changes to many sections of a large file (or group of files). However, if the files are left exclusively locked for too long, other developers may be tempted to bypass the revision control software and change the files locally, forcing a difficult manual merge when the other changes are finally checked in. In a large organization, files can be left "checked out" and locked and forgotten about as developers move between projects - these tools may or may not make it easy to see who has a file checked out.

Version merging

Most version control systems allow multiple developers to edit the same file at the same time. The first developer to "check in" changes to the central repository always succeeds. The system may provide facilities to merge further changes into the central repository, and preserve the changes from the first developer when other developers check in.

Merging two files can be a very delicate operation, and usually possible only if the data structure is simple, as in text files. The result of a merge of two image files might not result in an image file at all. The second developer checking in the code will need to take care with the merge, to make sure that the changes are compatible and that the merge operation does not introduce its own logic errors within the files. These problems limit the availability of automatic or semi-automatic merge operations mainly to simple text-based documents, unless a specific merge plugin is available for the file types.

The concept of a *reserved edit* can provide an optional means to explicitly lock a file for exclusive write access, even when a merging capability exists.

Baselines, labels and tags

Most revision control tools will use only one of these similar terms (baseline, label, tag) to refer to the action of identifying a snapshot ("label the project") or the record of the snapshot ("try it with baseline *X*"). Typically only one of the terms *baseline*, *label*, or *tag* is used in documentation or discussion; they can be considered synonyms.

In most projects, some snapshots are more significant than others, such as those used to indicate published releases, branches, or milestones.

When both the term *baseline* and either of *label* or *tag* are used together in the same context, *label* and *tag* usually refer to the mechanism within the tool of identifying or making the record of the snapshot, and *baseline* indicates the increased significance of any given label or tag.

Most formal discussion of configuration management uses the term *baseline*.

Distributed revision control

Distributed revision control systems (DRCS) take a peer-to-peer approach, as opposed to the client-server approach of centralized systems. Rather than a single, central repository on which clients synchronize, each peer's working copy of the codebase is a bona-fide repository.[5] Distributed revision control conducts synchronization by exchanging patches (change-sets) from peer to peer. This results in some important differences from a centralized system:

- No canonical, reference copy of the codebase exists by default; only working copies.
- Common operations (such as commits, viewing history, and reverting changes) are fast, because there is no need to communicate with a central server.[1]:7

Rather, communication is only necessary when pushing or pulling changes to or from other peers.

- Each working copy effectively functions as a remote backup of the codebase and of its change-history, providing inherent protection against data loss.[1]:4

Integration

Some of the more advanced revision-control tools offer many other facilities, allowing deeper integration with other tools and software-engineering processes. Plugins are often available for IDEs such as Oracle JDeveloper, IntelliJ IDEA, Eclipse and Visual Studio. Delphi, NetBeans IDE, Xcode, and GNU Emacs (via vc.el).

Common terminology

Terminology can vary from system to system, but some terms in common usage include:[6]

Baseline
    An approved revision of a document or source file from which subsequent changes can be made. See baselines, labels and tags.
Branch
    A set of files under version control may be *branched* or *forked* at a point in time so that, from that time forward, two copies of those files may develop at different speeds or in different ways independently of each other.
Change
    A *change* (or *diff*, or *delta*) represents a specific modification to a document under version control. The granularity of the modification considered a change varies between version control systems.
Change list
    On many version control systems with atomic multi-change commits, a *change list* (or *CL*), *change set*, *update*, or *patch* identifies the set of *changes* made in a single commit. This can also represent a sequential view of the source code, allowing the examination of source as of any particular changelist ID.
Checkout

To *check out* (or *co*) is to create a local working copy from the repository. A user may specify a specific revision or obtain the latest. The term 'checkout' can also be used as a noun to describe the working copy. When a file has been checked out from a shared file server, it cannot be edited by other users. Think of it like a hotel, when you check out, you no longer have access to its amenities.

Clone

Cloning means creating a repository containing the revisions from another repository. This is equivalent to *push*ing or *pull*ing into an empty (newly initialized) repository. As a noun, two repositories can be said to be *clone*s if they are kept synchronized, and contain the same revisions.

Commit

To *commit* (*check in*, *ci* or, more rarely, *install*, *submit* or *record*) is to write or merge the changes made in the working copy back to the repository. The terms 'commit' and 'checkin' can also be used as nouns to describe the new revision that is created as a result of committing.

Conflict

A conflict occurs when different parties make changes to the same document, and the system is unable to reconcile the changes. A user must *resolve* the conflict by combining the changes, or by selecting one change in favour of the other.

Delta compression

Most revision control software uses delta compression, which retains only the differences between successive versions of files. This allows for more efficient storage of many different versions of files.

Dynamic stream

A stream in which some or all file versions are mirrors of the parent stream's versions.

Export

exporting is the act of obtaining the files from the repository. It is similar to *checking out* except that it creates a clean directory tree without the version-control metadata used in a working copy. This is often used prior to publishing the contents, for example.

Fetch

See *pull*.

Forward integration

The process of merging changes made in the main *trunk* into a development (feature or team) branch.

Head

Also sometimes called *tip*, this refers to the most recent commit, either to the trunk or to a branch. The trunk and each branch have their own head, though HEAD is sometimes loosely used to refer to the trunk.[7]

Import

importing is the act of copying a local directory tree (that is not currently a working copy) into the repository for the first time.

Initialize

to create a new, empty repository.

Interleaved deltas

some revision control software uses Interleaved deltas, a method that allows storing the history of text based files in a more efficient way than by using Delta compression.

Label

See *tag*.

Mainline

Similar to *trunk*, but there can be a mainline for each branch.

Merge

A *merge* or *integration* is an operation in which two sets of changes are applied to a file or set of files. Some sample scenarios are as follows:

- A user, working on a set of files, *updates* or *syncs* their working copy with changes made, and checked into the repository, by other users.[8]
- A user tries to *check in* files that have been updated by others since the files were *checked out*, and the *revision control software* automatically merges the files (typically, after prompting the user if it should proceed with the automatic merge, and in some cases only doing so if the merge can be clearly and reasonably resolved).
- A *branch* is created, the code in the files is independently edited, and the updated branch is later incorporated into a single, unified *trunk*.
- A set of files is *branched*, a problem that existed before the branching is fixed in one branch, and the fix is then merged into the other branch. (This type of selective merge is sometimes known as a *cherry pick* to distinguish it from the complete merge in the previous case.)

Promote
The act of copying file content from a less controlled location into a more controlled location. For example, from a user's workspace into a repository, or from a stream to its parent.[9]

Pull, push
Copy revisions from one repository into another. *Pull* is initiated by the receiving repository, while *push* is initiated by the source. *Fetch* is sometimes used as a synonym for *pull*, or to mean a *pull* followed by an *update*.

Repository
The *repository* is where files' current and historical data are stored, often on a server. Sometimes also called a *depot*.

Resolve
The act of user intervention to address a conflict between different changes to the same document.

Reverse integration
The process of merging different team branches into the main trunk of the versioning system.

Revision
Also *version*: A version is any change in form. In SVK, a Revision is the state at a point in time of the entire tree in the repository.

Share
The act of making one file or folder available in multiple branches at the same time. When a shared file is changed in one branch, it is changed in other branches.

Stream
A container for branched files that has a known relationship to other such containers. Streams form a hierarchy; each stream can inherit various properties (like versions, namespace, workflow rules, subscribers, etc.) from its parent stream.

Tag
A *tag* or *label* refers to an important snapshot in time, consistent across many files. These files at that point may all be tagged with a user-friendly, meaningful name or revision number. See baselines, labels and tags.

Trunk
The unique line of development that is not a branch (sometimes also called Baseline, Mainline or Master)

Update
An *update* (or *sync*, but *sync* can also mean a combined *push* and *pull*) merges changes made in the repository (by other people, for example) into the local *working copy*. *Update* is also the term used by some CM tools (CM+, PLS, SMS) for the change package concept (see *changelist*). Synonymous with *checkout* in revision control systems that require each repository to have exactly one working copy (common in distributed systems)

Working copy
> The *working copy* is the local copy of files from a repository, at a specific time or revision. All work done to the files in a repository is initially done on a working copy, hence the name. Conceptually, it is a *sandbox*.

Pull request
> A developer tellling others to review his "pushed" changes

See also

- Change control
- Changelog
- Comparison of version-control software
- Distributed version control
- List of version-control software
- Non-linear editing system
- Software configuration management
- Software versioning
- Versioning file system

Notes

1. In this case, edit buffers are a secondary form of working copy, and not referred to as such.
2. In principle two revisions can have identical timestamp, and thus cannot be ordered on a line. This is generally the case for separate repositories, though is also possible for simultaneous changes to several branches in a single repository. In these cases, the revisions can be thought of as a set of separate lines, one per repository or branch (or branch within a repository).
3. The revision or repository "tree" should not be confused with the directory tree of files in a working copy.
4. Note that if a new branch is based on HEAD, then topologically HEAD is no longer a tip, since it has a child.
5. "Mainline" can also refer to the main path in a separate branch.

References

1. O'Sullivan, Bryan (2009). *Mercurial: the Definitive Guide* (http://hgbook.red-bean.com/read/). Sebastopol: O'Reilly Media, Inc. ISBN 9780596555474. Retrieved 4 September 2015.
2. "Google Docs", *See what's changed in a file* (https://support.google.com/docs/answer/190843), Google.
3. For Engineering drawings, see Whiteprint#Document control, for some of the manual systems in place in the twentieth century, for example, the *Engineering Procedures* of Hughes Aircraft, each revision of which required approval by Lawrence A. Hyland; see also the approval procedures instituted by the U.S. government.
4. Smart, John Ferguson (2008). *Java Power Tools* (https://books.google.com/books?id=YoTvBpKEx5EC&lpg=PA165&dq=cvs%20atomic%20commit&hl=en&pg=PA301#v=onepage&q=cvs%20doesn't%20support%20atomic%20commit&f=false). "O'Reilly Media, Inc.". p. 301. ISBN 9781491954546. Retrieved 20 July 2019.

5. Wheeler, David. "Comments on Open Source Software / Free Software (OSS/FS) Software Configuration Management (SCM) Systems" (http://www.dwheeler.com/essays/scm.html). Retrieved May 8, 2007.
6. Wingerd, Laura (2005). *Practical Perforce* (http://safari.oreilly.com/0596101856). O'Reilly. ISBN 0-596-10185-6.
7. Gregory, Gary (February 3, 2011). "Trunk vs. HEAD in Version Control Systems" (http://gary gregory.wordpress.com/2011/02/03/trunk-vs-head-in-version-control-systems/). *Java, Eclipse, and other tech tidbits*. Retrieved 2012-12-16.
8. Collins-Sussman, Fitzpatrick & Pilato 2004, 1.5: SVN tour cycle resolve (http://svnbook.red-bean.com/en/1.5/svn.tour.cycle.html#svn.tour.cycle.resolve): 'The G stands for merGed, which means that the file had local changes to begin with, but the changes coming from the repository didn't overlap with the local changes.'
9. *Concepts Manual* (Version 4.7 ed.). Accurev. July 2008.

Bibliography

- Collins-Sussman, Ben; Fitzpatrick, BW; Pilato, CM (2004), *Version Control with Subversion* (https://archive.org/details/versioncontrolwi00coll), O'Reilly, ISBN 0-596-00448-6

External links

- "Visual Guide to Version Control", *Better explained* (http://betterexplained.com/articles/a-visual-guide-to-version-control/).
- Sink, Eric, "Source Control", *SCM* (http://www.ericsink.com/scm/source_control.html) (how-to). The basics of version control.