Software Heritage SVN Loader Review

Stefan Sperling
mail@stefansperling.de

December 16, 2021

Contents

1 Introduction

The mission of Software Heritage[1] (SWH) is to collect and archive the source code of all software publicly available. A large amount of source code has already been ingested into the SWH archive. It is now important to address the long tail of the many diverse platforms that are used for developing, distributing and/or archiving publicly available source code.

Apache Subversion[2] (SVN) is an open source version control system developed as a project of the Apache Software Foundation (ASF). SVN was founded in the year 2000, with its latest stable release dating from 2020. SVN is still in use worldwide by many

[1]`https://softwareheritage.org`
[2]`https://subversion.apache.org`

companies and organizations for version control of source code and related assets. However, it has mostly been phased out by open source communities in favour of other version control systems, and some public hosting platforms based on SVN have already shut down.

SWH Loaders are components of the SWH software suite which preserve the information stored in source code repositories by translating it into the SWH archive data format. The SWH SVN Loader performs this task for SVN repositories. What follows is an analysis of the SVN Loader, discussing possible ways of improving the accuracy and completeness of information it preserves.

This work was funded by a grant from the Alfred P. Sloan Foundation[3].

2 Translation from SVN to SWH

The SVN Loader's purpose is to transfer information stored in a given SVN repository into the SWH archive. In order to achieve this, the SVN Loader needs perform a translation between two data models: From the data model used by SVN to the data model used by SWH.

An SVN repository stores versions of files and directories, arranged in a tree that looks similar to a filesystem hierarchy such as used by many computer operating systems in common use today. Each SVN repository is typically associated with either a single software project or a collection of software projects which happen to share a common repository. In any case, there is only one SVN repository per project which contains the authoritative copy of that project's version control history, apart from copies made for secondary purposes such as backup or mirrors of the central SVN server.

The SWH archive is unusual in that its ultimate purpose is to preserve all source code ever written. As such, this single archive contains many different software projects. The SWH archive retains the version history of source code (if available), such that changes which were made to source code over time can be traced and understood. Storing version history is essential to the archive's purposes and as a result its data model looks similar to data models of version control systems used to develop software.

The following sections present the SVN and SWH data models in more detail.

2.1 The SVN usage model

At its user interface, SVN presents versioned history as a series of revisions of a hierarchy of files and directories.[5] Snapshots are created as changes are made to files or directories. There is no way to modify an already existing revision. The first revision always contains an empty root directory and has revision number 0.

Users navigate an SVN repository in two dimensions: The location of a file or directory,

[3]`https://sloan.org`

identified by a path within the directory hierarchy, and a revision which was created at some point in time, identified by a revision number.

Paths look just like their counterparts in a UNIX filesystem, with the character "/" used as a path component separator: `/myproject/notes/README`

Users can address particular versions of files or directories by specifying a path and a revision number identifying the revision in which that path should be looked up. The SVN user interface usually requires the character "@" as a delimiter between the two: `/myproject/notes/README@42`

A common use case for version control systems is branching, where parallel lines of history exist which make independent modifications to the same set of files and directories over time. Parallel lines of history can be reunited at some later point in time by using the version control system's merging operation. SVN promotes the idea that branching and merging operations can be represented using path semantics. Users create new branches by copying an existing file or directory to another path with the `svn copy` command.

The `svn merge` command can be used to merge selected changes from one copy to another. In the simple case of syncing all changes between two copies, users specify a merge source path and merge target path. SVN then traces back through the history of both paths, crossing copy operations, until a common ancestor path is found in an older revision. Original file versions required as one of the three inputs of a 3-way merge[8] are obtained from this revision, and the other two files are obtained from within the two user-specified paths in the latest revision. Once a new revision containing merged changes has been created, this revision becomes the new common ancestor for use during future merges. To facilitate this, SVN stores merge-tracking information which records revision number ranges that contain already merged changes from a given merge source path. This allows future merges to filter out already merged changes once the common ancestor has been determined as usual. Merge-tracking information[4] is stored in user-visible meta-data properties associated with files or directories within the merge target path[4].

2.2 The SVN repository data model

Data stored in the SVN repository is structured as a directed acyclic graph (DAG) which is hidden from users.[9] Path and revision number tuples from the user interface are internally mapped to nodes of this DAG. Nodes are identified by a node-revision identifier (ID) which is invisible to the user.

Each node represents a version of a particular file or directory. File nodes point to a particular version of file content, which may be stored verbatim or as a delta against other content. Directory nodes contain a mapping of directory entry names to node-revision IDs, facilitating traversal of the DAG in search of a node at a given path.

[4]Often referred to by the abbreviated term "mergeinfo".

Nodes may additionally refer to meta-data known as SVN properties. Some special-purpose properties exist (such as mergeinfo), and additional custom properties may be defined by users.

Node-revision IDs are tuples consisting of 3 sub-identifiers: The node ID, the copy (branch) ID, and the transaction ID.

The node ID remains constant throughout the lifetime of a node, and is shared with all nodes of the same historical lineage. In other words, when a node is copied to another path in the DAG as a result of an `svn copy` command, its node ID will be retained.

A new transaction ID is assigned when a revision is created in which the node has been altered. Changes to a node cause a bubble-up effect on parent nodes: All parent nodes are altered to point at the new node-revision ID of altered child nodes, and all parent nodes receive the same new transaction ID in this process. This implies that the root node always receives a new transaction ID when a revision is created. There is a 1:1 mapping from revision numbers to transaction IDs.

The default copy ID is zero and implies that the node did not come into existence by being copied. A new non-zero copy ID is assigned when a node is copied. Unaltered child nodes are not immediately copied along and are referred to simply by their existing node-revision IDs. If a new revision is created in which a child node of a copied node is altered, the child's copy ID will be inherited from the parent node. Already copied nodes may exist among the child nodes of a newly copied node, with a non-zero copy ID that is distinct from their parent node's copy ID. Such nodes will immediately be assigned a new copy ID while the parent node is copied, even if they are unaltered. For a more detailed explanation, see [9].

2.3 The SWH archive data model

The SWH archive is structured as a Merkle DAG,[6] similar in design to that of the Git[5] version control system. Some aspects of the Git repository format are reused in order to yield Git-compatible intrinsic object identifiers, known as Software Heritage identifiers (SWHID).

The SWH DAG is best explained from the inside out, starting with leaf nodes. Leaf nodes, known as blobs, represent the content of a particular source code file. The identifier of a blob is the hash of a header concatenated with the content of the source code file. Blob nodes correspond to Git blob objects.

Directory nodes represent the file and directory structure of a software project's collection of source code. Each directory node has a list of named entries referring to other blob or directory nodes. The identifier of a directory is the hash of a header concatenated with a manifest which lists the entries. Directory nodes correspond to Git tree objects.

Revision nodes point at a particular root directory node, and represent sets of changes

[5]`https://git-scm.com`

made to a software project over time. Each revision is linked to a set of earlier revisions which it is based on, known as the set of parent revisions. The identifier of a revision is the hash of a header concatenated with a manifest which lists the root directory ID, parent revision IDs, author and timestamp information, as well as comments about the change written by the author when the change was recorded. Revision nodes correspond to Git commit objects.

Links between revisions represent branching and merging. If a revision has more than one child revision, each child represents a distinct branch of the software project's development history. A revision which has more than one parent revision presents a merge of the set of branches represented by those parents.

Release nodes point to selected revisions which represent released versions of software. The identifier of a revision is the hash of a header concatenated with a manifest listing a revision ID, author and timestamp information, and author comments. Release nodes correspond to Git tag objects.

The SWH archive defines additional some object types that are not found in Git, all of which are also identified by SWHIDs. An origin object represents a body of source code which is imported into the SWH archive. An origin represents a type of version control or software packaging system and has a URL where source code can be fetched from. A snapshot objects links an origin to a set of available releases and revisions at a particular moment in time. A visit object links a software origin with snapshots created in the archive over time. A new visit event is recorded whenever the SWH archive imports source code from a given origin. For more details, see [6].

Each node of the DAG may be associated with arbitrary extrinsic meta-data which is not stored in the DAG itself.[1]

2.4 What is lost in translation?

The main differences between SVN's DAG and SWH's DAG are the semantics represented by nodes in the graph, the nature of links between nodes, and the ways identifiers are assigned to nodes.

At present, the SVN Loader maps the SVN DAG onto the SWH DAG as follows:

- All SVN file and SVN directory nodes are mapped to SWH blobs and SWH directories at the same path as occupied by the node in the SVN DAG.
- SVN revisions are mapped to SWH revisions.

This mapping preserves the content of all source code files stored in an SVN repository, and thus meets the minimum requirements of the SWH archive. However, the mapping is imperfect for the reasons listed below:

- There is no equivalent of an intrinsic node ID in the SVN DAG, and an SVN node-revision ID cannot be accurately mapped to the SWH DAG. The SVN transaction ID can be mapped to an SWH revision ID. The closest equivalent of the SVN node

ID is the path of the node in the SWH DAG, which is inaccurate since nodes at different paths in the SVN DAG can share a node ID. The SVN copy ID is lost entirely after mapping a node into the SWH DAG.

- The main line of SVN development history is usually represented by an SVN directory node which is not the root node of the SVN DAG. Whereas only root directory nodes of the SWH DAG are tied to a line of history via an SWH revision object.
- SVN directories which were copied and then changed may represent branches. Mergeinfo is a possible differentiator between the user-level concepts of branches and regular copies.[6] For an accurate representation, changesets for SVN directories which represent branches would need to be mapped to SWH revision objects.
- SVN files which were copied and then changed may represent branches of a file. The SWH model does not represent branches of a single file. This leaves extrinsic meta data as a possible place to store such information about files.
- A copied SVN directory may contain arbitrarily nested other copies of nodes from within or outside this directory. This can result in a structure where nested branches exist in the SVN DAG path space. The SWH DAG cannot represent nested branches as it would require something akin to SWH directory objects pointing at SWH revisions, with a risk of creating cycles in the DAG.
- Each SVN file or directory node can store mergeinfo which cannot be accurately represented in the SWH DAG, unless in the very specific case where an SVN revision merges all changes from one SVN directory copy to another, and only if these directories do not contain nested branches as described above. This leaves extrinsic meta data as a possible place to store mergeinfo.
- It is possible that a single SVN revision modifies multiple copied files or directories at once. If any of those copies represent branches then the SWH DAG is unable to represent the SVN changeset accurately. Such a changeset would need to be split into multiple per-branch SWH changesets.

These conversion inaccuracies have several consequences:

- Software projects which share a large SVN repository[7] by using a subdirectory per project to host source code end up being mapped to the same SWH origin, even when the SWH archive intends to map them to a distinct origin each.
- When browsing the archive, SWH origins loaded from an SVN repository present branches in the path space, rather than the revision space. Users of the archive need to be aware of this in order to make accurate sense of the archived data.
- While it is possible to export data from the SWH archive back into the Git repos-

[6]The presence of mergeinfo on a node implies a branch, but a copy may still represent a branch even in the absence of mergeinfo. This is because Subversion 1.4 and older did not create mergeinfo, requiring users to track merges manually.

[7]See `https://svn.apache.org/repos/asf/` for example.

itory format, it is impossible to export data back into the original SVN repository format. This makes it difficult to revive development of archived SVN projects from the SWH archive alone without loss of prior version control history.

- Archived data does not accurately preserve information about the workflow of SVN-based projects, which is implicitly encoded in the branching/merging structure of the repository. This information can be considered part of the culture of software development and thus worthy of being preserved. The SWH archive does preserve such information accurately in case of Git-based projects.

3 Improving the SVN Loader

Unfortunately, resolving all incompatibilities between the SVN and SWH data models is not trivial. Full compatibility to SVN would require substantial changes to the data model used by the SWH archive, some of which might break existing assumptions already made during the design phase of SWH.

Nevertheless, this section suggests some changes that could be made to the design of the SVN Loader in order to address some of the problems outlined in previous the section.

3.1 Improving translation accuracy

The current SVN Loader design operates above the SVN repository layer and hence lacks direct access to the SVN DAG. However, such direct access should not be necessary because all information stored in the SVN DAG is represented by corresponding user-visible concepts which the SVN Loader is exposed to. The same level of abstraction is also used by regular SVN client implementations. As such, there is nothing an SVN client could do that the SVN Loader could not do. The SVN Loader should take full advantage of this.

3.1.1 Preserving copyfrom information

The SVN Loader currently ignores copy source information which is available via the SVN repository access (RA) editor, a set of SVN client data structures and callbacks which apply changes to a given target tree[7]. Because copy source information cannot be encoded in the SWH DAG the SVN Loader is forced to model the creation of a copy destination as a plain addition to the SWH DAG's path space. However, the SVN Loader could additionally store information about copy source paths and revision numbers as extrinsic meta-data in order to preserve information which links together the history of a copy source and destination.

This would allow tracing back through the history of copied paths imported from SVN repositories just like a regular SVN client could do, and it would help with the recognition of possible nested branches. A way to query the SWH archive for a given SVN path and revision number could be added, in order to yield a corresponding SWH snapshot, path, and revision. This would facilitate queries for artifacts which represent a given SVN copy source.

3.1.2 Supporting multi-origin SVN repositories

The SVN Loader's design should take into account that SVN repositories may carry an organizational structure that resembles a software forge which hosts multiple software projects. While SVN repositories which map to a single SWH origin exist, such repositories should be treated as a special case, not as the regular case.

Obtaining a list of origins from an SVN repositories is a task which may be more suitable for an SWH Lister. SWH Listers are components of the SWH software suite which create an index of origins available at a given location, and are usually applied to software forges or collections of source code packaged by Linux distributions. A new SVN Lister could be designed and implemented. This SVN Lister would need to use heuristics in order to infer the organizational structure of a given SVN repository. The SVN Lister would yield paths to individual projects in an SVN repository, paths to individual development branches within each project, and a list of remaining paths which could not be classified using existing heuristics. Human intervention would likely be required in order to obtain accurate results, though the heuristics could be improved over time. One example of a suitable heuristic would be an existence check for subdirectories called `trunk`, `tags`, and `branches`, a common directory naming scheme used extensively by the official SVN documentation[5]. As another example, the SVN Lister could attempt to match top-level directories in the SVN repository against names of projects known to use the SVN repository, in order to categorize multi-project repositories into several origins.

To treat an SVN repository as a source of multiple SWH origins, the SVN Loader would need to accept a URL which points to an arbitrary directory node at some path in a given revision of the SVN repository. The SVN Loader could then import revisions from the history of this node without assuming that SVN revision numbers of imported revisions will be sequential. Revision numbers will still descend as history is traversed, but certain revision number ranges will be inoperative on the directory node of interest and should simply be skipped. In cases where a path is copied from elsewhere into the directory of interest, the SVN Loader could store information about the copy source path and revision number as extrinsic meta-data, even though the copy source path might not (yet) exist in the SWH archive.

3.1.3 Preserving SVN properties

SVN properties presented on an SVN file or directory node should generally be preserved as extrinsic metadata. SVN Properties which modify file or directory contents such as `svn:eol-style`[8], `svn:executable`[9] and `svn:special`[10] are already interpreted by the SVN Loader in order to apply corresponding state changes to files or directories imported into the SWH archive. However, the original values of SVN properties which caused such state changes are not retained in the SWH archive.

Because SVN properties are not stored the SVN Loader has to do some unnecessary work. For example, in order to synchronize the initial state of an `svn:eol-style` property during a visit, the SVN Loader processes revisions from previous visits again. This would not be needed if the most recent value of `svn:eol-style` was available in metadata.

The `svn:mergeinfo` property is of particular interest. This property is an important indicator when determining whether a copied SVN node does in fact represent a branch. In some cases mergeinfo could even be used to infer a set of multiple parent revisions to use for an imported SWH revision, such that the branching and merging structure of a project is preserved. However, mapping mergeinfo into the SWH DAG structure is impossible in general because mergeinfo can track partial merges (a.k.a. "cherry-pick" merges), which the SWH data model cannot do. As in the Git data model which the SWH model was based on, partial merges are not represented in a special way but appear as regular changesets.

3.1.4 Preserving branching structure

The branching structure of SVN-based projects could be preserved by splitting a given SVN project into a set of branch paths of interest, for example by implementing an SVN Lister as proposed in section 3.1.2.

A (path, revision) tuple data structure would suffice to represent a branch tip. The history of each tip would then need to be traced until a copy event brings the path into existence, at which point the lifetime of a branch stops and the lifetime of another potential branch, which the previous branch was based on, begins. The SVN Loader would import changes from each known branch in parallel, sorted according to the SVN revision number which made the change, creating several lines of SWH revisions history which correspond to one SVN branch each.

Because SVN branches are often deleted when they are no longer needed all revisions must be searched for potential branch paths. Deletion of branches cannot be represented as such by SWH as this would require the modification of existing SWH snapshots or would render such branches irretrievable to users of the SWH archive. A special case to account for are branches which occupy the same path in different revision ranges. Encoding both the path and the SVN revision number in which the branch first appeared in the SWH snapshot branch name should resolve any ambiguity.

[8]Line-ending convention a text file should use.

[9]Whether the "executable" bit should be set on a file when checked out into a UNIX filesystem.

[10]Whether a file is a non-regular file. In practice only used to tell whether a file should be a symbolic link when checked out into a UNIX filesystem.

Due to differences in the data models, some branches in the SVN repository cannot be mapped to SWH branches. This includes branch paths which are nested in the path of another branch, as well as branches of single files. Changes that occurred on nested branches would need to be represented as changes which occurred on the outer-most branch (i.e. the parent SVN DAG node with is reachable via the shortest path and has a non-zero SVN copy ID).

An example of a nested branch exists in the Subversion project itself. In `https://svn.apache.org/r1498089` the subdirectory `libsvn_fs_fs` was copied to `libsvn_fs_x` in order to begin development on a new filesystem backend. Both subdirectories were children of the same outer branch `/subversion/branches/fsx`. This outer branch was eventually merged to the main branch `/subversion/trunk` in `https://svn.apache.org/r1509915`. Changes were regularly merged between the two backends throughout this development, relying on the ability of SVN to perform and track merges between nested branches.

3.1.5 Preserving merges between branches

Because SVN mergeinfo can track merge operations that can only be represented as regular changesets in the SWH date model (see section 3.1.3), preserving information about `svn merge` operations in detail would be complicated.

It is possible to recognize SVN merges which can be expressed as SWH merge revisions that have multiple parent revisions, but even this would require a lot of effort to do correctly. The main problem lies in the complicated nature of how SVN merge information is stored.[4] Mergeinfo is stored per-path, has explicit and implicit inheritance rules, may contain redundancies, and may even contain inaccuracies because of SVN user errors or because of bugs in older versions of SVN. Additionally, the rules which say when mergeinfo should be created or updated have changed over time.[2] Attempting to teach the SVN Loader to understand mergeinfo in detail would likely do more harm than good because of such complications.

It should suffice to represent SVN merges as regular changesets and store SVN mergeinfo properties verbatim as extrinsic meta data of the corresponding file and directory nodes in the SWH DAG. This would allow interested users of the SWH archive to understand when a merge was made in the SVN repository origin. It would also allow for possible future enhancements of the SWH software suite if necessary by making use of the additional information about merges.

3.2 Assorted issues

This section discusses assorted issues found during code review of the SVN Loader.

3.2.1 Commit log message encoding

The SVN Loader attempts to re-encode log messages instead of treating them as opaque byte strings. However, the function `convert_commit_message()` cannot know the encoding of its input for certain.

While SVN commit messages should always be encoded in UTF-8, this was only enforced by the standard SVN client. Older SVN servers allow clients to write log messages in arbitrary encodings, and some Java-based SVN clients used encodings such as latin1[11]. In practice, this means SVN repositories may contain arbitrary character encodings in their history and there is no meta-data which specifies which encoding was used.

For preservation purposes, treating log message as binary data would probably be the best approach.

3.2.2 Recovery from truncated SVN dump streams

The way recovery from partial svnrdump results was implemented is problematic.

The function `dump_svn_revisions()` searches for the character sequence `Revision-number:...` in the dump stream, and assumes that this sequence always indicates the beginning of a new dump stream record. However, the same character sequence could occur anywhere in versioned file content data present in the dump stream.

The SVN Loader should use a suitable parser for the dump stream format in order to truncate the stream properly, or simply retry the entire operation if there was an error while obtaining the dump stream.

3.2.3 Support for SVN externals

The SVN Loader currently aborts a loading operation if an SVN externals property is encountered. At the time of writing, there is already work being done to address this.[12]

[11] `https://issues.apache.org/jira/browse/SVN-3313`
[12] `https://forge.softwareheritage.org/T611`

4 Conclusion

The goal of the SVN Loader is to preserve information stored in an SVN repository as accurately as possible. Unfortunately, perfect preservation will never be achievable without a fully compatible destination format.

However, to some degree it is possible to convert the source SVN data model into the target SWH data model in a more accurate way. Making further improvements and refinements to the SVN Loader's current design could also provide useful insights for future implementations of additional SWH Loaders facing similar challenges.

References

[1] Extrinsic metadata specification. `https://docs.softwareheritage.org/devel/swh-storage/extrinsic-metadata-specification.html`.

[2] Subversion 1.7 release notes: Merge-tracking enhancements. `https://subversion.apache.org/docs/release-notes/1.7.html#merge-tracking-enhancements`.

[3] Subversion loader design documentation. `https://archive.softwareheritage.org/browse/content/sha1_git:2f56aaa4728cf21e5069c9982c0e854534cc22a6/?origin_url=https://forge.softwareheritage.org/source/swh-loader-svn.git&path=docs/swh-loader-svn.txt`.

[4] Paul T. Burba. Subversion 1.5 mergeinfo - understanding the internals. `https://www.open.collab.net/community/subversion/articles/merge-info.html`.

[5] Ben Collins-Sussman, Brian W. Fitzpatrick, and C. Michael Pilato. *Version Control with Subversion, First Edition.* O'Reilly, June 2004. Full text available under a Creative Commons licence at `https://svnbook.red-bean.com`.

[6] Roberto Di Cosmo and Stefano Zacchiroli. Software Heritage: Why and How to Preserve Software Source Code. In *iPRES 2017 - 14th International Conference on Digital Preservation*, pages 1–10, Kyoto, Japan, September 2017. `https://hal.archives-ouvertes.fr/hal-01590958/document`.

[7] Karl Fogel. Subversion's delta editor: Interface as ontology. `https://www.red-bean.com/kfogel/beautiful-code/bc-chapter-02.html`.

[8] Sanjeev Khanna, Keshav Kunal, and Benjamin Pierce. A formal investigation of diff3, 2007. `https://www.cis.upenn.edu/~bcpierce/papers/diff3-short.pdf`.

[9] C. Michael Pilato. Subversion filesystem history. `https://svn.apache.org/repos/asf/subversion/trunk/subversion/libsvn_fs_base/notes/fs-history?p=1875971`.