Towards Universal Software Evolution Analysis

Antoine Pietri
Inria
Paris, France
antoine.pietri@inria.fr

Stefano Zacchiroli
University Paris Diderot and Inria
Paris, France
zack@irif.fr

***Abstract*—Software evolution studies have mostly focused on individual software products, generally developed as Free/Open Source Software (FOSS) projects, and more sparingly on software collections like component and package ecosystems. We argue in this paper that the next step in this organic scale expansion is *universal software evolution* analysis, i.e., the study of software evolution at the scale of the whole body of publicly available software.**

We consider the case of Software Heritage, the largest existing archive of publicly available software source code artifacts (more than 5 B unique files archived and 1 B commits, coming from more than 80 M software projects). We propose research requirements that would allow to leverage the Software Heritage archive to study universal software evolution. We discuss the challenges that need to be overcome to address such requirements and outline a research roadmap to do so.

***Index Terms*—software evolution, source code, open source software, free software, digital preservation**

I. Introduction

Nearing the closing of the fourth decade since Lehman's seminal work [1], the literature on software evolution is abundant [2]. In particular, the practice of studying Free/Open Source Software (FOSS) projects from various software evolution angles has flourished providing researchers with treasure troves of software-related data to analyze [3], [4]. And yet, most present-day evolution studies still focus on the evolutive patterns of *individual* software projects, at fine-grained granularity (e.g., commits). There is a clear interest in doing so, for instance to ascertain which factors contribute to maximize software health [5].

Larger-scale studies have been performed [6], [7], reaching up to the scale of the evolution of specific software ecosystems (app stores, package manager repositories, etc.), but paying the price of a more coarse-grained granularity (e.g., releases). The current "ceiling" in software evolution studies hence appears to be, on the one hand, the scale of software ecosystems and, on the other hand, the granularity of commits.

Arguably, one of the next frontier in the field should be studying what we call **universal software evolution**, that is: the study of software evolution at the largest scale possible at the finest possible granularity. At present, the largest scale possible is that of the *software commons*, i.e., the body of all software which is available at little or no cost and which can be reused with few restrictions [8], [9]; the finest observable granularity is that of commits, as captured by state-of-the-art version control systems (VCSs) [10].

In the remainder of this paper we consider the challenge of studying universal software evolution using the Software Heritage archive [11], [12] (described in Sec. II) as a proxy for the software commons. We present the requirements for enabling scientists in analyzing universal software evolution (Sec. III), discuss the challenges to be overcome to address them on top of Software Heritage (Sec. IV), and present a roadmap to do so (Sec. V).

II. Software Heritage

Launched in 2016, the Software Heritage project [11] has the stated mission of collecting, preserving, and sharing all software that is available in source code form, together with its development history. Even though full coverage w.r.t. the software commons is a moving target for any archive, the Software Heritage one is currently the best approximation for enabling studying universal software evolution, both in terms of coverage and granularity.

The Software Heritage data model [12] is a Merkle DAG [13], schematized in Figure 1. Each node is identified by a persistent, cryptographic-strong identifier [14] and deduplication is enforced on all nodes. Each node in the diagram corresponds to an archived source code artifact, produced as part of software development. The following kinds of artifacts are supported:

Contents: (or "blobs") the raw content of files; note that filenames are context-dependent and stored only as part of directory entries.

Directories: lists of named directory entries, where each entry can point to content objects ("file entries"), revisions ("revision entries"), or other directories ("directory entries"). Each entry is associated to a local name (i.e., a relative path without any path separator) and permission metadata and modification timestamps.

Revisions: (or "commits") point-in-time captures of the entire source tree of a development project. Each revision points to the "root" directory of the project source tree. Also, revisions are associated to commit metadata like timestamps, commit message, author, etc.

Releases: (or "tags") revisions that have been marked as noteworthy with a specific, usually mnemonic, name (e.g., a version number). Each release points to a revision and might include additional descriptive metadata.

Snapshots: point-in-time captures of the full state of a project development repository. Differently from revisions,

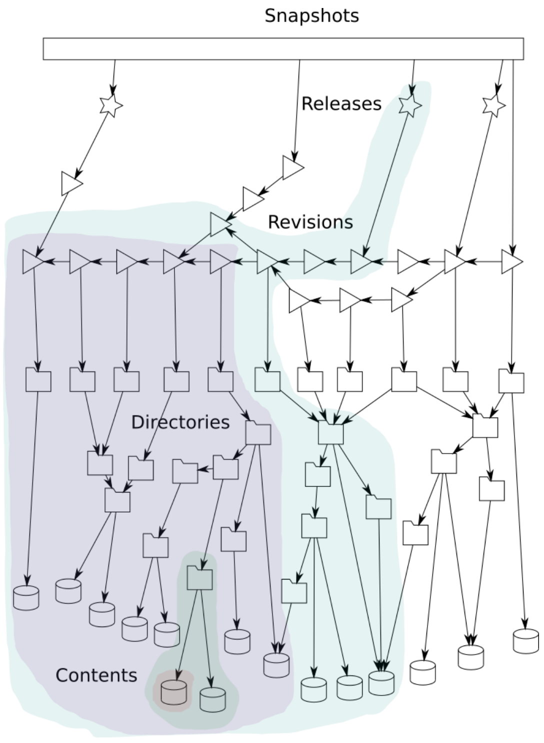


Fig. 1. Software Heritage data model: a uniform Merkle DAG containing source code artifacts and their development history

which capture the state of a single development line (or “branch”), snapshots capture the state of all branches in a repository and allow to deduplicate identical forks of it across the archive.

This data model is uniform in the sense that it captures in a single graph both source code trees and their development histories. It is also canonical as it allows to store (and deduplicate) source code artifacts coming from different version control systems, source packages, etc.

Software Heritage also stores provenance information (not shown in the picture). Each time a software origin is visited, its full state is captured by a snapshot object (novel or reused) and a 3-way mapping between the origin (usually as its URL), the time of the visit, and the snapshot object is added to an append-only journal of crawling activities.

The actual Software Heritage archive[1] covers major active development forges such as GitHub and GitLab.com, gone ones like Gitorious and Google Code, as well as GNU/Linux distributions such as Debian and language specific component ecosystems such as PyPI. All in all the archive contains more than 5 billion unique source code files and more than 1 billion unique commits coming from more than 85 million projects. Active software origins (GitHub, GitLab, Debian, etc.) are periodically re-crawled to keep the archive content fresh. There is some update lag (potentially up to a couple of months for large forges), but that does not appear to be problematic for universal software evolution studies considering typical publication delays.

[1]https://archive.softwareheritage.org, accessed November 1st, 2018

III. Research requirements

There is clear interest on the part of empirical software engineering researchers in accessing the Software Heritage dataset for various purposes. After preliminary conversations with researchers working in the field, we have categorized their needs based on which type of data they are interested in:

Content access. One of the most common requests is to obtain a set of file contents stored in the archive based on some specific criterion. This specifically pertains to the binary data contained in the archived objects. Those requests are usually made in the purpose of analysing the code itself: code patterns recognition, language detection, static analysis, malware detection, and so on.

Occasionally, those requests also require some data preprocessing to be applied to the file contents before the analysis (comments removal, data or binary strings removal, etc.).

Filtering on metadata. It is generally useful to filter the query results depending on some criterion on the metadata of the files. This metadata can be either already present in the archived repositories (file extensions, file names, file sizes, directory depth. . .), or *derived* from the data (MIME types, language detected, license. . .). This metadata has to be precomputed and indexed along with the files.

Content search. The ability to perform full-text search for specific code fragments or patterns is very useful to focus computations on the relevant parts of the code, and it requires an up to date full-text search index.

History graph: When analyzing not only the software itself but its evolution through time, accessing the revision history graph along with the associated metadata (authors, commit messages, etc.) and being able to examine the relationships between the different objects in the revision trees is paramount. In the context of Software Heritage, the relationships are also expressed between different repositories: forks point to the same ancestors, directories that were moved from one repository to another point to the same object, etc.

Provenance indexing: While the software DAG works top-down (the nodes only point to their children i.e., their content, but never their parents), it’s also sometimes necessary to be able to list the different objects that point to a specific object. There are multiple applications for that: find the possible extensions of a file, the different repositories that contain a code fragment, a directory or a revision, etc.

In addition to providing these different data filtering and lookup methods, it should be done in an expressive and generic enough manner, to allow for different categories of data to be accessed in all the steps of the analyses.

Most of the time the final result of the computation is not very large compared to the volume of data processed, but rather just a product of a reduce (in the MapReduce [15] sense) operation on the computed results, so aggregating computations should have a small overhead and avoid large data transfers.

IV. Challenges

Satisfying all the requirements outlined above for the entire dataset requires addressing a lot of technical challenges.

A. Data volume

Most of the challenges in the way of providing some form of access to the data stem simply from the sheer size of the software archive. Allowing people to locally retrieve a large chunk of the archive to perform a local computation is very impractical at the Software Heritage scale in most cases, both from a network transfer perspective and form a local storage one.

Handling the file contents of the archive requires a lot of resources and expertise. The sheer size of the blobs (currently ~200 TB) demands a lot of storage capacity and the blobs cannot easily be stored on a single machine using consumer-grade storage. The unusual size distribution of the blobs, whose median size is approximately 3 kB, makes it also hard to use industry-grade storage solutions, because they often are not designed to store a very large quantity of very small files. On conventional systems, some limitations on inode management may apply. Other distributed storage solutions like Ceph cannot easily handle a large amount of small files (because of the per-file overhead needed for replication [16]), and require some form of content packing to take place beforehand.

While compressing the blobs works well to reduce the size with a compression ratio of ~2 (estimated on a random sample of the archive of about 1%), compressing similar contents together based on content chunking techniques with rolling hashes [17] [18] are usually not worth the added complexity, with a compression ratio of ~1.5 on the whole archive (estimated on 0.1% of the blobs, sample taken contiguously by date of insertion in the archive to preserve the locality of similar files).

The size of the Merkle DAG itself is more reasonable (the ~10 billion nodes and ~100 billion edges of the graph take ~1.2 TB in the Apache Parquet format), but using it efficiently often requires some indexes on the hashes, which significantly increases its size on-disk. Moreover, some intensive processing on the graph itself could require having it stored directly in main memory, which is difficult to achieve on standard machines that have orders of magnitude less RAM.

Even if the recipient of the dataset already has the storage capacity and expertise to handle such a vast amount of data, transferring it through the network is impractical and expensive. Sending the whole dataset through a connection with a speed matching the common industry standard of 1 Gbps would take more than 20 days, assuming the absence of sequential overhead between fetches.

B. Representation mismatch

Researchers and data scientists usually try their experiments by prototyping on small sample sizes, before reaching out to experiment on larger datasets. Doing so, they generally use tools that are fit for specific data representations. Thus, they often expect the data to be presented in specific formats. One of the challenges of making software analysis accessible to them is to help them transform the data from a format well-suited for archival to a format suited for large-scale analysis/massive computation.

Files and directories contained in a specific revision are usually expected to be represented as on-disk filesystem trees, so the children of the directories can directly be accessed through the primitives of the filesystem. In the Software Heritage archive, the de-duplication requires this to go through an additional index of the hashes of the directories. The interface therefore has to provide a utility to "flatten" the compact representation into a more classical directory structure, although doing so systematically would invalidate the benefits of de-duplication.

For the revision graph itself, there is no current standard of representation, so most of the research experiments so far have worked on tool-specific representations (often depending on the version control system used). While there is value in providing a universal representation for commit-level software evolution from different sources, it is still important to provide a data representation that does not stray too far from what those domain-specific analysis tools usually expect.

C. Provenance mappings

Making provenance mappings accessible to allow looking up the different places where an object can be found is a hard problem, because of the combinatorial explosion of the ancestor relationship mappings. A single file content can be found in thousand if not millions of origins. There is a difficult balance to strike between reducing the size of the mappings using intermediate objects in the relationship as layers to compress the volume of edges, and reducing as much as possible the amount of indirections that require index hits for performance reasons.

Moreover, maintaining this (bottom-up) provenance index is harder than its top-down counterparts, since there is no way to know in advance all the objects of the relationship, and thus represent them with an intrinsic hash for indexing.

D. Metadata misalignment

Usually, the code unit on which experiments are performed are software "projects". While this concept makes sense at the metadata level (a project usually has a homepage URL, a name, and a single origin for its source code), this concept is lost in a fully-deduplicated representation of Software Heritage. This is because: repositories can contain full clones of other repositories, it is hard to distinguish random forks of a project from the official one, etc.

In order to bridge the notion of "projects" to the Software Heritage archive, there has to be ways of cross-referencing the objects in the archive with the metadata surrounding the projects collected during archival.

E. Repeatability and reproducibility

An important part of scientific experiments is reproducibility, which is something to keep in mind in making a very large and constantly-evolving dataset available for research applications. While intrinsic hashes guarantee full consistency of the data at the snapshot level, it might be useful to provide a way to describe the state of the *whole archive* at some point in time. If we are able to reconstruct a previous state of the whole archive from a timestamp, including this timestamp along with the experiment methodology will allow the experiment to be repeated on the exact same dataset as when it was executed for the first time. That way, an experiment can be *repeated* by executing it on the timestamped state of the archive, and *reproduced* by executing it on a different timestamp.

While the trivial way of timestamping a Merkle DAG is simply to hash the list of roots, it does not work when the graph is incomplete. As the Software Heritage DAG has a lot of holes (missing revisions, files, etc.) that can be added or removed without changing the intrinsic identifier of the nodes, relying solely on those hashes is not sufficient to reliably get a timestamp of the archive as a whole.

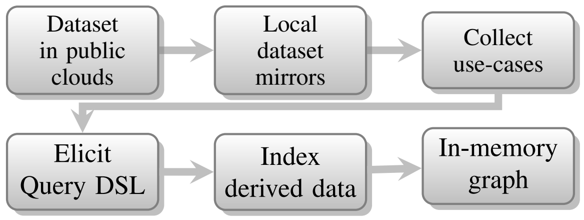
One actual way to achieve this would be to store the evolution of the archive as a journal of insertions that can be rewinded or replayed, and some static data containing the time of insertion of objects.

F. *Expressivity*

Researchers who want to run analyses on the Software Heritage dataset will perform *queries* on the archive to describe the computations and the part of the archive they will be run on. Running these queries on the archive will require a query API that can express these different use cases.

The expressive power of the query language determines how easy it is to use the different data selection features, computation primitives and result aggregations when running queries on the dataset. The semantics of the language have to provide ways of representing and combining those different operations so that the breadth of computations that queries are able to represent is as wide and generic as possible for the use-cases we identified.

V. Roadmap



A first, preliminary step of this work is to make the dataset available in some format suitable for scale-out analysis, so that Software Heritage and other researchers can perform preliminary experiments on it. Some public cloud computing providers like Amazon Web Services or Google Cloud have public datasets programs, on which we can make the Software Heritage dataset publicly available without bearing the cost of the storage.

While allowing people to run queries directly on a public cloud instance is well-suited for one-off experiments, it does not work well for a more intensive use. Researchers having access to hardware resources and software engineering skills might find it more cost-efficient to run their experiments on a local copy of the archive. As we build a mirroring pipeline and infrastructure to keep the Software Heritage public datasets up to date, we need to provide a way for researchers to have their own local copy of the dataset for intensive uses.

Once the dataset is available in some format for people to run queries on it, the paramount step to guide the way of making analysis as accessible as possible will be to collect real-world use cases: what are the types of queries that scientists want to run? What are the data and metadata filters that they need? What is the kind of information that is the most often retrieved from the data model?

After having collected these use-cases, it will then be possible to elicit a domain specific language that would cover the kind of queries required for the use-cases we identified, to offer a better accessibility than simply doing raw queries on the database. This language will have to be expressive enough to address the difficulties we outlined, notably by allowing flexible manipulations of the data representations. It should also be as simple as possible to express constraints on the data used for the computations, and to limit the span of the queries. The language has to possess powerful aggregation capabilities, to avoid transferring large amounts of data and instead sending the result of the reduction of the computation directly.

Real-world use cases might also exhibit patterns of access to data derived from the dataset, like diffs between revisions, branching and merging histories, etc. Isolating this “derived data” to index it in the dataset would also be very useful, as it would significantly improve the performance of computations on these common use cases.

While the cost of disk access for the file contents cannot be avoided, it might be interesting performance-wise to store as much of the history graph as possible directly in memory. The on-disk size of the database is currently around 4 TiB, which indicates an order of magnitude of a size that could be fitted reasonably well in memory. We examined some compression techniques [19] that would allow us to represent the history graph as a sparse adjacency matrix. If these results allow us to drastically compress the graph, it might lower the cost barrier for fast in-memory computations.

VI. Conclusion

This paper presents the challenges of making Software Heritage the universal analysis platform for software evolution. We have presented a roadmap for answering the main needs we outlined. While the end goal is to build a fully-fledged platform with APIs that allow running remote computations on the archive, the raw form of the dataset can be made available sooner on public clouds, and will allow us to collect use-cases to refine our vision of the platform. We expect this availability to ease the work of scholars working on software evolution analysis, both at the micro and macro level.

References

[1] Meir M Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, 1980.

[2] Israel Herraiz, Daniel Rodriguez, Gregorio Robles, and Jesus M Gonzalez-Barahona. The evolution of the laws of software evolution: A discussion based on a systematic literature review. *ACM Computing Surveys (CSUR)*, 46(2):28, 2013.

[3] MM Mahbubul Syeed, Imed Hammouda, and Tarja Systa. Evolution of open source software projects: A systematic literature review. *Journal of Software*, 8(11):2815–2830, 2013.

[4] Hongyu Pei Breivold, Muhammad Aufeef Chauhan, and Muhammad Ali Babar. A systematic review of studies of open source software evolution. In *Software Engineering Conference (APSEC), 2010 17th Asia Pacific*, pages 356–365. IEEE, 2010.

[5] Bram Adams, Eleni Constantinou, Tom Mens, and Gregorio Robles, editors. *Proceedings of the 1st International Workshop on Software Health, SoHeal@ICSE 2018, Gothenburg, Sweden, May 27, 2018*. ACM, 2018.

[6] Jesus M Gonzalez-Barahona, Gregorio Robles, Martin Michlmayr, Juan José Amor, and Daniel M German. Macro-level software evolution: a case study of a large software compilation. *Empirical Software Engineering*, 14(3):262–285, 2009.

[7] Matthieu Caneill, Daniel M. German, and Stefano Zacchiroli. The debsources dataset: Two decades of free and open source software. *Empirical Software Engineering*, 22:1405–1437, June 2017.

[8] Donald Beagle. Conceptualizing an information commons. *The Journal of Academic Librarianship*, 25(2):82–89, 1999.

[9] Nancy Kranich and Jorge Reina Schement. Information commons. *Annual Review of Information Science and Technology*, 42(1):546–591, 2008.

[10] Diomidis Spinellis. Version control systems. *IEEE Software*, 22(5):108–109, 2005.

[11] Jean-François Abramatic, Roberto Di Cosmo, and Stefano Zacchiroli. Building the universal archive of source code. *Communications of the ACM*, 61(10):29–31, October 2018.

[12] 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*, 2017.

[13] Ralph C. Merkle. A digital signature based on a conventional encryption function. In Carl Pomerance, editor, *Advances in Cryptology - CRYPTO '87, A Conference on the Theory and Applications of Cryptographic Techniques, Santa Barbara, California, USA, August 16-20, 1987, Proceedings*, volume 293 of *Lecture Notes in Computer Science*, pages 369–378. Springer, 1987.

[14] Roberto Di Cosmo, Morane Gruenpeter, and Stefano Zacchiroli. Identifiers for digital objects: the case of software source code preservation. In *iPRES 2018: 15th International Conference on Digital Preservation*, 2018.

[15] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[16] [ceph-users] ceph behavior on (lots of) small objects (rgw, rados + erasure coding)? Ceph Users Mailing List, 2018.

[17] Wen Xia, Yukun Zhou, Hong Jiang, Dan Feng, Yu Hua, Yuchong Hu, Qing Liu, and Yucheng Zhang. Fastcdc: a fast and efficient content-defined chunking approach for data deduplication. In *USENIX Annual Technical Conference*, pages 101–114, 2016.

[18] Athicha Muthitacharoen, Benjie Chen, and David Mazieres. A low-bandwidth network file system. In *ACM SIGOPS Operating Systems Review*, volume 35, pages 174–187. ACM, 2001.

[19] Nieves R Brisaboa, Susana Ladra, and Gonzalo Navarro. Compact representation of web graphs with extended functionality. *Information Systems*, 39:152–174, 2014.