How to use Software Heritage for archiving and referencing your source code: guidelines and walkthrough[*]

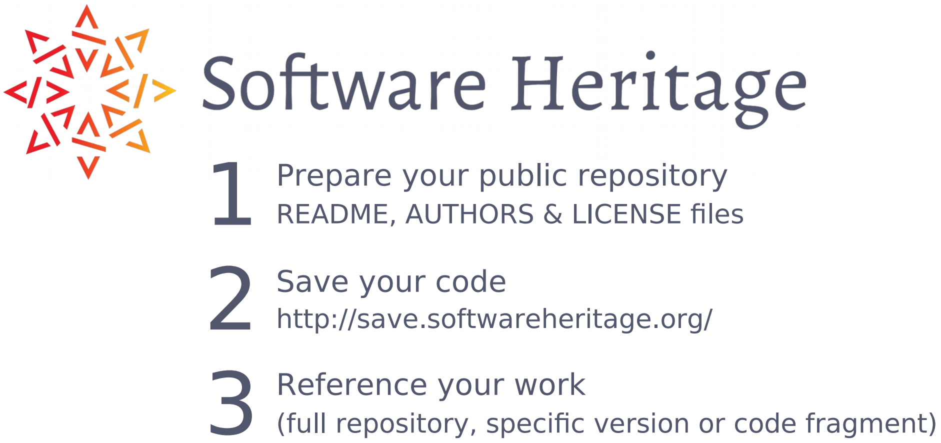
Roberto Di Cosmo
Inria, Software Heritage, University of Paris, France
roberto@dicosmo.org

May 2020

Software source code is *an essential research output*, and there is a growing general awareness of its importance for supporting the research process [4, 19, 16]. Many research communities strongly encourage making the source code of the artefact available by archiving it in publicly-accessible long-term archives. Some have even put in place mechanisms to assess research software, like the *Artefact Evaluation* process introduced in 2011 and now widely adopted by many computer science conferences [5], and the *Artifact Review and Badging* program of the ACM [3].

Software Heritage [14, 1] is a non profit, long term universal archive specifically designed for software source code, and able to store not only a software artifact, but *also its full development history*. It provides the ideal place to *preserve research software artifacts*, and offers powerful mechanisms to *enhance research articles* with precise references to relevant fragments of your source code.

Using Software Heritage for your research software artifacts is straightforward and involves three simple steps, described in the picture below:



Software Heritage

1 Prepare your public repository
README, AUTHORS & LICENSE files

2 Save your code
http://save.softwareheritage.org/

3 Reference your work
(full repository, specific version or code fragment)

In this document we will go through each of these three steps, providing guidelines for making the most out of Software Heritage for your research: Section 1 describes the best practices for preparing your source code for archival; Section 2 shows how to archive your code in Software

[*] This revision: 2ce4cf4d20e8fde040d6ee4ef1bcb303dca9a0bd on branch master

Heritage; Section 3 shows the rich functionalities you can use for referencing in your article source code archived in Software Heritage; finally, in the Appendix you will find a formal description of the different kinds of identifiers available for adressing the content archived in Software Heritage.

1 Prepare your repository

We assume that your source code is hosted on a repository publicly accessible (Github, Bitbucket, a GitLab instance, an institutional software forge, etc.) using one of the version control systems supported by Software Heritage, currently Subversion, Mercurial and Git [1].

It is highly recommended that you provide, in your source code repository, appropriate information on your research artifact: it will make it more appealing and useful to future users (which might actually be *you* in a few months).

Well established best practice is to include, at the toplevel of your source code tree, three *key files*, README, AUTHORS and LICENSE, with the information described below.

README : A description of the software.
This file should contain *at least*

- the name of the software/project
- a brief description of the project.

It is also *highly recommended* to add the following information

- pointers to the project website and documentation,
- pointer to the project development platform,
- license for the project (if not in a separate LICENSE file),
- contact and support information,
- build/installation instructions or a pointer to a file containing them (usually INSTALL)

In could be useful to provide here also some information for the users, like a list of features or informations on how to use the source code

AUTHORS : The list of all authors that need to be credited for the current version.
If you want to specify the role of each contributor in this list, we suggest to use the taxonomy of contributors presented in [2], which distinguishes the following roles: *Design, Architecture, Coding, Testing, Debugging, Documentation, Maintenance, Support, Management.*

LICENSE : The project license terms.
For Open Source Licenses, it is strongly recommended to use the standard names that can be found on the `https://spdx.org/licenses/` website.

Future users that find the artifact useful might want to give you credit by citing it. To this end, you should provide instructions on how you prefer the artifact to be cited.

[1] For up to date information, see `https://archive.softwareheritage.org/browse/origin/save/`

Sophisticated support for bibliographic entries in BibTeX format is available for users of the BibLaTeX package is provided by the `biblatex-software` package [9], available on CTAN [6]. Another option is to use the Citation File Format, CFF (usually in a file named **citation.cff**).

We recommend to also provide structured metadata information in a machine readable format. While practices in this area are still evolving, one can use the CodeMeta generator available at `https://codemeta.github.io/codemeta-generator/` to produce metadata conformant to the CodeMeta schema, and put the JSON-LD output in a **codemeta.json** file at the root of the project.

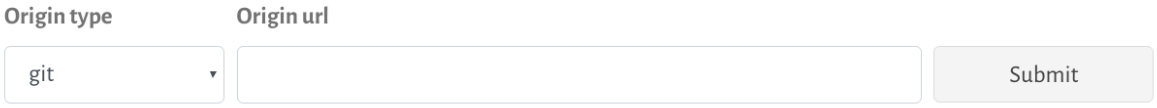
1.1 Learning more

The seminal article *Software Release Practice HOWTO* by E. S. Raymond [18] documents best practices and conventions for releasing software that have been well established for decades, and form the basis of most current recommendations. Interesting more recent resources include the REUSE website [15], which provides detailed guidance and tools to verify compliance with the guidelines, as well as [17], which focuses more on research software.

2 Save your code

Once your code repository has been properly prepared, you only need to:

- go to `https://archive.softwareheritage.org/browse/origin/save/`,
- pick your version control system in the drop-down list, enter the code repository url [2],
- click on the Submit button (see Figure 1).



Origin type: git

Origin url:

Figure 1: The Save Code Now form

That's it, it's all done! No need to create an account or to provide personal information of any kind. If the url you provided is correct, Software Heritage will archive your repository, with its full development history, shortly after. If your repository is hosted on one of the major forges we already know, this process will take just a few hours; if you point to a location we never saw before, it can take longer, as we will need to manually approve it.

For hackers: you can also request archival programmatically, using the Software Heritage API [3]; this can be quite handy to integrate, for example, into a Makefile.

[2] Make sure to use the clone/checkout url as given by the development platform hosting your code. It can easily be found in the web interface of the development platform.

[3] For details, see `https://archive.softwareheritage.org/api/1/origin/save/`

3 Reference your work

Once the source code has been archived, the Software Heritage *intrinsic identifiers*, called SWHID, fully documented online and shown in Figure 2, can be used to reference with great ease any version of it.

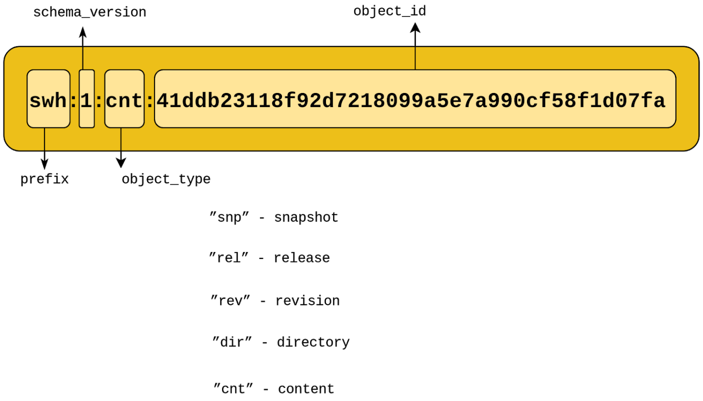


Figure 2: Schema of the core Software Heritage identifiers

SWHIDs are URIs with a very simple schema: the `swh` prefix makes explicit that these identifiers are related to Software Heritage; the colon (`:`) is used as separator between the logical parts of identifiers; the schema version (currently `1`) is the current version of this identifier schema; then follows the type of the objects identified and finally comes a hex-encoded (using lowercase ASCII characters) cryptographic signature of this object, computed in a standard way, as detailed in [12, 13]. These core identifiers may be equipped with the following *qualifiers* that carry contextual *extrinsic* information about the object:

origin : the *software origin* where an object has been found or observed in the wild, as an URI;

visit : persistent identifier of a *snapshot* corresponding to a specific *visit* of a repository containing the designated object;

anchor : a *designated node* in the Merkle DAG relative to which a *path to the object* is specified;

path : the *absolute file path*, from the *root directory* associated to the *anchor node*, to the object;

lines : *line number(s)* of interest, usually within a content object

The combination of the core SWHIDs with these qualifiers provides a very powerful means of referring in a research article to all the software artefacts of interest.

We present here three common use cases: link to the *full repository* archived in Software Heritage; link to a *precise version of the software project*, and link to a *precise version of a source code file*, down to the level of the line of code.

To make this concrete, in what follows we use as a running example the article *A "minimal disruption" skeleton experiment: seamless map and reduce embedding in OCaml* by Marco Danelutto and Roberto Di Cosmo [7] published in 2012. This article introduced a nifty library for multicore parallel programming that was distributed via the `https://gitorious.org` collaborative development platform, at `https://gitorious.org/parmap`. Since Gitorious has been shut down a few years ago, like Google Code and CodePlex, this example is particularly fit to show why pointing to an *archive* that has your code is better than pointing to the collaborative development platform where you developed it.

3.1 Full repository

In Software Heritage, we keep track of all the *origins* from which source code has been retrieved, and finding a given `origin` is as easy as adding in front of it the prefix `https://archive.softwareheritage.org/browse/origin`

These origins are the exact *URLs of the version control system* that a developer would use to clone a working repository, and are the same urls that you pass to the *Save Code Now* form described in Section 2.

In our running example, for the Parmap code on *gitorious.org*, this origin is `https://gitorious.org/parmap/parmap.git`, so the URL of the *persistently archived full repository* is:

`https://archive.softwareheritage.org/browse/origin/https://gitorious.org/parmap/parmap.git`

Just add this link to your article, and your readers will be able to get hold of the archived copy of your repository even if/when the original development platform goes away (as it has actually happened for `gitorious.org` that has been shut down in 2015).

Your readers can then browse the contents of your repository extensively, delving into its development history, and/or directory structure, down to each single source code file [4].

N.B.: if you are unsure about what is the actual origin URL of your repository, you can look it up using the search box that is available at `https://archive.softwareheritage.org/browse/search/`

3.2 Specific version

Pointing to the full archived repository is nice, but a version controlled repository usually contains all the history of development of the source code, whiche records different states of the project, usually called *revisions*.

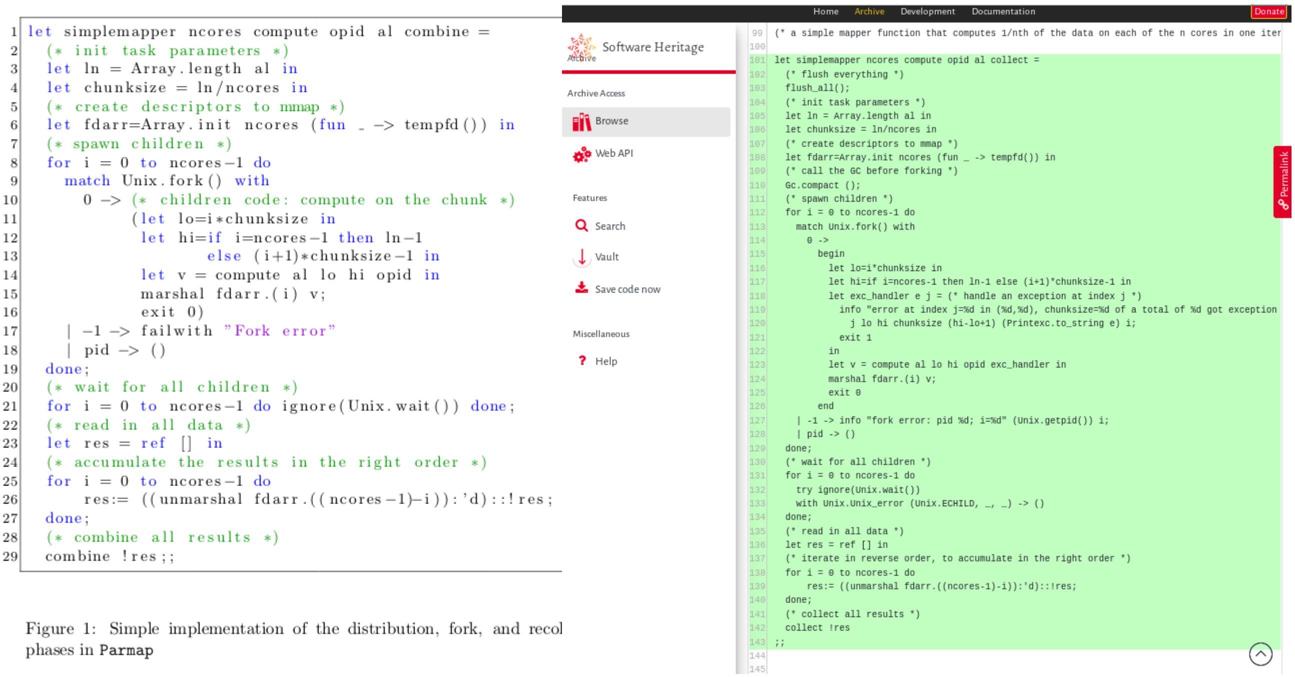
In order to support reproducibility of scientific results, we need to be able to pinpoint precisely the state(s) of the source code used in the article. Software Heritage provides a very easy means of pointing to a precise *revision*, via a standard identifier schema, called SWHID, which is fully documented online and is discussed in the article [12].

In our running example, the Parmap article, the exact revision of the source code of the library used therein has the following SWHID:

```
swh:1:rev:0064fbd0ad69de205ea6ec6999f3d3895e9442c2;
origin=https://gitorious.org/parmap/parmap.git;
visit=swh:1:snp:78209702559384ee1b5586df13eca84a5123aa82
```

And you can turn this identifier into a clickable URL by prepending to it the prefix `https://archive.softwareheritage.org/` (you can try it live right now by clicking on this link).

[4] For a guided tour see `https://www.softwareheritage.org/2018/09/22/browsing-the-software-heritage-archive-a-guided-tour/`



```
let simplemapper ncores compute opid al combine =
  (* init task parameters *)
  let ln = Array.length al in
  let chunksize = ln/ncores in
  (* create descriptors to mmap *)
  let fdarr=Array.init ncores (fun _ -> tempfd()) in
  (* spawn children *)
  for i = 0 to ncores-1 do
    match Unix.fork() with
      0 -> (* children code: compute on the chunk *)
           (let lo=i*chunksize in
            let hi=if i=ncores-1 then ln-1
                      else (i+1)*chunksize-1 in
            let v = compute al lo hi opid in
            marshal fdarr.(i) v;
            exit 0)
    | -1 -> failwith "Fork error"
    | pid -> ()
  done;
  (* wait for all children *)
  for i = 0 to ncores-1 do ignore(Unix.wait()) done;
  (* read in all data *)
  let res = ref [] in
  (* accumulate the results in the right order *)
  for i = 0 to ncores-1 do
      res:= ((unmarshal fdarr.((ncores-1)-i)):'d)::!res;
  done;
  (* combine all results *)
  combine !res;;
```

Figure 1: Simple implementation of the distribution, fork, and recol
phases in Parmap

```
99  (* a simple mapper function that computes 1/nth of the data on each of the n cores in one iter
100
101 let simplemapper ncores compute opid al collect =
102   (* flush everything *)
103   flush_all();
104   (* init task parameters *)
105   let ln = Array.length al in
106   let chunksize = ln/ncores in
107   (* create descriptors to mmap *)
108   let fdarr=Array.init ncores (fun _ -> tempfd()) in
109   (* call the GC before forking *)
110   Gc.compact ();
111   (* spawn children *)
112   for i = 0 to ncores-1 do
113     match Unix.fork() with
114       0 ->
115         begin
116           let lo=i*chunksize in
117           let hi=if i=ncores-1 then ln-1 else (i+1)*chunksize-1 in
118           let exc_handler e j = (* handle an exception at index j *)
119             info "error at index j=%d in (%d,%d), chunksize=%d of a total of %d got exception
120               j lo hi chunksize (hi-lo+1) (Printexc.to_string e) i;
121             exit 1
122           in
123           let v = compute al lo hi opid exc_handler in
124           marshal fdarr.(i) v;
125           exit 0
126         end
127     | -1 -> info "fork error: pid %d; i=%d" (Unix.getpid()) i;
128     | pid -> ()
129   done;
130   (* wait for all children *)
131   for i = 0 to ncores-1 do
132     try ignore(Unix.wait())
133     with Unix.Unix_error (Unix.ECHILD, _, _) -> ()
134   done;
135   (* read in all data *)
136   let res = ref [] in
137   (* iterate in reverse order, to accumulate in the right order *)
138   for i = 0 to ncores-1 do
139       res:= ((unmarshal fdarr.((ncores-1)-i)):'d)::!res;
140   done;
141   (* collect all results *)
142   collect !res
143 ;;
144
145
```

(a) as presented in the article [7]

(b) as archived in Software Heritage

Figure 3: Code fragment from the published article compared to the content in the Software Heritage archive

3.3 Code fragment

A particularly nifty feature of the SWHIDs supported by Software Heritage is the ability to pinpoint a fragment of code inside a specific version of a file, by using the `lines=` qualifier available for identifiers that point to files.

Let's see this feature at work in our running example, which shows clearly how an article can be greatly enhanced by providing pointers to code fragments.

In Figure 1 of [7], which is shown here as Figure 3a, the authors want to present the core part of the code implementing the parallel functionality that constitutes the main contribution of their article. The usual approach is to typeset in the article itself *an excerpt of the source code*, and let the reader try to find it by delving into the code repository, which may have evolved in the mean time. Finding the exact matching code can be quite difficult, as the code excerpt is *often edited* a bit with respect to the original, sometimes to drop details that are not relevant for the discussion, and sometimes due to space limitations.

In our case, the article presented 29 lines of code, slightly edited from the 43 actual lines of code in the Parmap library: looking at 3a, one can easily see that some lines have been dropped (102-103, 118-121), one line has been split (117) and several lines simplified (127, 132-133, 137-142).

Using Software Heritage, the authors can do a much better job, because the original code fragment can now be precisely identified by the following Software Heritage identifier, that can be easily obtained using the permalink box shown in Section 3.3.1 above, and that will **always** point to the code fragment shown in Figure 3b.

```
swh:1:cnt:d5214ff9562a1fe78db51944506ba48c20de3379;
origin=https://gitorious.org/parmap/parmap.git;
visit=swh:1:snp:78209702559384ee1b5586df13eca84a5123aa82;
anchor=swh:1:rev:0064fbd0ad69de205ea6ec6999f3d3895e9442c2;
path=/parmap.ml;
lines=101-143
```

The caption of the original article shown in Figure 3a can then be significantly enhanced by incorporating all the clickable links needed to point to the exact source code fragment that has been edited for inclusion in the article, as shown in Figure 4 (notice that the percent signs at the end of the line are necessary to ensure LaTeX does not break the SWHIDs).

Simple implementation of the distribution, fork, and recollection phases in `Parmap` (slightly simplified from the actual code in the version of Parmap used for this article)

Figure 4: A caption text with links to code fragment and revision

When clicking on the hyperlinked text in the caption shown above, the reader is brought seamlessly to the Software Heritage archive on a page showing the corresponding source code archived in Software Heritage, with the relevant lines highlighted (see Figure 3b).

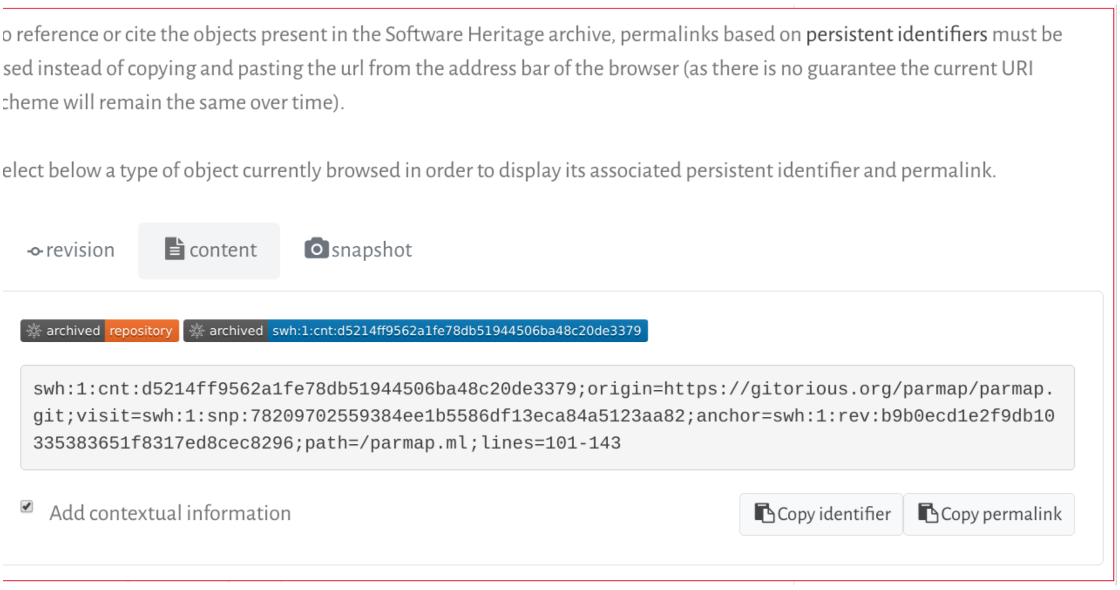
```
\newcommand{\swhurl}[1]{https://archive.softwareheritage.org/#1}
\newcommand{\swhref}[2]{\href{\swhurl{#1}}{#2}}

...

\caption{Simple implementation of the distribution,
fork, and recollection phases in \texttt{Parmap}
(slightly simplified from
\swhref{swh:1:cnt:d5214ff9562a1fe78db51944506ba48c20de3379;%
          origin=https://gitorious.org/parmap/parmap.git;%
          visit=swh:1:snp:78209702559384ee1b5586df13eca84a5123aa82;%
          anchor=swh:1:rev:0064fbd0ad69de205ea6ec6999f3d3895e9442c2;%
          path=/parmap.ml;%
          lines=101-143
 }{the actual code in the version of Parmap used for this article})
}
```

Figure 5: Adding clickable hyperlinks to Software Heritage in LaTeX

LaTeX users can produce the caption of 4 using a few convenient auxiliary macros, as shown in Figure 5. They can also create corresponding bibliography entries using the `biblatex-software` package, like [10] and [11] found in the bibliography of this guide.

3.3.1 Getting your SWHID

A very simple way of getting the right SWHID is to browse your archived code in Software Heritage, and to navigate to the revision you are interested in. Click then on the *permalinks vertical red tab* that is present on all pages of the archive, and in the tab that opens up you select the *revision* identifier: an example is shown in Figure 6.



o reference or cite the objects present in the Software Heritage archive, permalinks based on persistent identifiers must be sed instead of copying and pasting the url from the address bar of the browser (as there is no guarantee the current URI :heme will remain the same over time).

elect below a type of object currently browsed in order to display its associated persistent identifier and permalink.

```
swh:1:cnt:d5214ff9562a1fe78db51944506ba48c20de3379;origin=https://gitorious.org/parmap/parmap.
git;visit=swh:1:snp:78209702559384ee1b5586df13eca84a5123aa82;anchor=swh:1:rev:b9b0ecd1e2f9db10
335383651f8317ed8cec8296;path=/parmap.ml;lines=101-143
```

☑ Add contextual information

Figure 6: Obtaining a Software Heritage identifier using the permalink box on the archive Web user interface

The two convenient buttons on the botton right allow you to copy the identifiers or the full permalink in the clipboard, to insert in your article as you see fit. Make sure to check the box at the bottom left to generate a SWHID with all relevant qualifiers corresponding to your browsing context.

3.3.2 Generating and verifying SWHIDs (for the geeks)

Version 1 of the SWHIDs uses git-compatible hashes, so if you are using git as a version control system, you can create the right SWHID by just prepending `swh:1:rev:` to your commit hash. This might come pretty handy if you plan to automate the generation of the identifiers to be included in your article: you will always have your code and your article in sync!

Software Heritage identifiers can also be generated and verified independently by anyone using `swh-identify`, an open source tool developed by Software Heritage, and distributed via PyPI as `swh.model` (stable version at swh:1:rev:6cab1cc81118877e2105c32b08653509475f3eaa; origin=https://pypi.org/project/swh.model/).

4 Acknowledgements

These guidelines result from extensive discussions that took place over several years. Special thanks to Alain Girault, Morane Gruenpeter, Julia Lawall, Arnaud Legrand and Nicolas Rougier for their precious feedback on earlier versions of this document.

References

[1] Jean-François Abramatic, Roberto Di Cosmo, and Stefano Zacchiroli. “Building the Universal Archive of Source Code”. In: *Communications of the ACM* 61.10 (Sept. 2018), pages 29–31.

[2] Pierre Alliez et al. “Attributing and Referencing (Research) Software: Best Practices and Outlook From Inria”. In: *Computing in Science and Engineering* 22.1 (Jan. 2020). Available from `https://hal.archives-ouvertes.fr/hal-02135891`, pages 39–52.

[3] Association for Computing Machinery. *Artifact Review and Badging*. `https://www.acm.org/publications/policies/artifact-review-badging`. Retrieved April 27th 2019. Apr. 2018.

[4] Christine L. Borgman, Jillian C. Wallis, and Matthew S. Mayernik. “Who’s Got the Data? Interdependencies in Science and Technology Collaborations”. In: *Computer Supported Cooperative Work* 21.6 (2012), pages 485–523.

[5] Bruce R. Childers et al. “Artifact Evaluation for Publications (Dagstuhl Perspectives Workshop 15452)”. In: *Dagstuhl Reports* 5.11 (2016). Edited by Bruce R. Childers et al., pages 29–35.

[6] *CTAN: the Comprehensive TeX Archive Network*. url: `http://www.ctan.org/` (visited on 04/29/2020).

[7] Marco Danelutto and Roberto Di Cosmo. “A “Minimal Disruption” Skeleton Experiment: Seamless Map & Reduce Embedding in OCaml”. In: *Procedia CS* 9 (2012), pages 1837–1846.

[8] Quynh Dang. “Changes in Federal Information Processing Standard (FIPS) 180-4, Secure Hash Standard.” In: *Cryptologia* 37.1 (2013), pages 69–73.

[9] [Software] Roberto Di Cosmo, *BibLaTeX stylefiles for software products*, 2020. url: `https://ctan.org/tex-archive/macros/latex/contrib/biblatex-contrib/biblatex-software`.

[10] [Software Release] Roberto Di Cosmo and Marco Danelutto, *The Parmap library* version 0.9.8, swhid: ⟨`swh:1:rev:0064fbd0ad69de205ea6ec6999f3d3895e9442c2;origin=https://gitorious.org/parmap/parmap.git;visit=swh:1:snp:78209702559384ee1b5586df13eca84a5123aa82`⟩.

[11] [Software excerpt] Roberto Di Cosmo and Marco Danelutto, “Core mapping routine”, from *The Parmap library* version 0.9.8. swhid: ⟨`swh:1:cnt:d5214ff9562a1fe78db51944506ba48c20de3379;origin=https://gitorious.org/parmap/parmap.git;visit=swh:1:snp:78209702559384ee1b5586df13eca84a5123aa82;anchor=swh:1:rev:0064fbd0ad69de205ea6ec6999f3d3895e9442c2;path=/parmap.ml;lines=101-143`⟩.

[12] Roberto Di Cosmo, Morane Gruenpeter, and Stefano Zacchiroli. “Identifiers for Digital Objects: the Case of Software Source Code Preservation”. In: *Proceedings of the 15th International Conference on Digital Preservation, iPRES 2018, Boston, USA*. Sept. 2018.

[13] Roberto Di Cosmo, Morane Gruenpeter, and Stefano Zacchiroli. “Referencing Source Code Artifacts: a Separate Concern in Software Citation”. In: *Computing in Science and Engineering* 22.2 (Mar. 2020), pages 33–43.

[14] Roberto Di Cosmo and Stefano Zacchiroli. “Software Heritage: Why and How to Preserve Software Source Code”. In: *Proceedings of the 14th International Conference on Digital Preservation, iPRES 2017*. Sept. 2017.

[15] Free Software Foundation Europe. *REUSE Software*. https://reuse.software. Accessed on 2019-09-24. Sept. 2019.

[16] Konrad Hinsen. "Software Development for Reproducible Research". In: *Computing in Science and Engineering* 15.4 (2013), pages 60–63.

[17] Michael Jackson (ed). *Software Deposit: What to deposit (Version 1.0)*. https://softwaresaved.github.io/software-deposit-guidance/WhatToDeposit.html. doi:10.5281/zenodo.1327325. Aug. 2018.

[18] Eric S Raymond. *Software Release Practice HOWTO*. https://www.tldp.org/HOWTO/html_single/Software-Release-Practice-HOWTO/. Accessed on 2019-06-05. Jan. 2013.

[19] Victoria Stodden, Randall J. LeVeque, and Ian Mitchell. "Reproducible Research for Scientific Computing: Tools and Strategies for Changing the Culture". In: *Computing in Science and Engineering* 14.4 (2012), pages 13–17.

A Appendix: Reference for SWHID identifiers

The SWHID identifier schema is fully documented online and is discussed in the article [12], but we reproduce here for completeness an excerpt of the documentation.

A SWHID consists of two separate parts, a mandatory *core identifier* that can point to any software artifact (or “object”) available in the Software Heritage archive, and an *optional list of qualifiers* that allows to specify the context where the object is meant to be seen and point to a subpart of the object itself.

A.1 Syntax

Syntactically, SWHIDs are generated by the `<identifier>` entry point of the EBNF grammar given in Table 1.

Table 1: EBNF grammar of Software Heritage persistent identifiers

⟨*identifier*⟩ ::= ⟨*identifier_core*⟩ [⟨*qualifiers*⟩]

⟨*identifier_core*⟩ ::= ‘swh’ ‘:’ ⟨*scheme_version*⟩ ‘:’ ⟨*object_type*⟩ ‘:’ ⟨*object_id*⟩

⟨*scheme_version*⟩ ::= ‘1’

⟨*object_type*⟩ ::= ‘snp’ | ‘rel’ | ‘rev’ | ‘dir’ | ‘cnt’

⟨*object_id*⟩ ::= 40 * ⟨*hex_digit*⟩ (* intrinsic object id, as hex-encoded SHA1 *)

⟨*dec_digit*⟩ ::= ‘0’ | ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ | ‘7’ | ‘8’ | ‘9’

⟨*hex_digit*⟩ ::= ⟨*dec_digit*⟩ | ‘a’ | ‘b’ | ‘c’ | ‘d’ | ‘e’ | ‘f’

⟨*qualifiers*⟩ := ‘;’ ⟨*qualifier*⟩ [⟨*qualifiers*⟩]

⟨*qualifier*⟩ ::= ⟨*context_qualifier*⟩ | ⟨*fragment_qualifier*⟩

⟨*context_qualifier*⟩ ::= ⟨*origin_ctxt*⟩ | ⟨*visit_ctxt*⟩ | ⟨*anchor_ctxt*⟩ | ⟨*path_ctxt*⟩

⟨*origin_ctxt*⟩ ::= ‘origin’ ‘=’ ⟨*url_escaped*⟩

⟨*visit_ctxt*⟩ ::= ‘visit’ ‘=’ ⟨*identifier_core*⟩

⟨*anchor_ctxt*⟩ ::= ‘anchor’ ‘=’ ⟨*identifier_core*⟩

⟨*path_ctxt*⟩ ::= ‘path’ ‘=’ ⟨*path_absolute_escaped*⟩

⟨*fragment_qualifier*⟩ ::= ‘lines’ ‘=’ ⟨*line_number*⟩ [‘-’ ⟨*line_number*⟩]

⟨*line_number*⟩ ::= ⟨*dec_digit*⟩ +

⟨*url_escaped*⟩ ::= (* RFC 3987 IRI *)

⟨*path_absolute_escaped*⟩ ::= (* RFC 3987 absolute path *)

Where <path_absolute_escaped> is an <ipath-absolute> from RFC 3987, and <url_escaped> is a RFC 3987 IRI. In either case all occurrences of ; (and %, as required by the RFC) have been percent-encoded (as %3B and %25 respectively). Other characters can be percent-encoded, e.g., to improve readability and/or embeddability of SWHID in other contexts.

A.2 Semantics

The swh prefix, which is IANA registered, makes explicit that these identifiers are related to Software Heritage, and the colon (:) is used as separator between the logical parts of identifiers. The scheme version (currently 1) is the current version of this identifier scheme.

A persistent identifier points to a single object, whose type is explicitly captured by <object_type>:

snp identifiers points to snapshots,

rel to releases,

rev to revisions,

dir to directories,

cnt to contents.

The actual object pointed to is identified by the intrinsic identifier <object_id>, which is a hex-encoded (using lowercase ASCII characters) SHA1 [8] computed on the content and metadata of the object itself.[5]

A.3 Git compatibility

Intrinsic object identifiers for contents, directories, revisions, and releases are, at present, compatible with the Git way of computing identifiers for its objects. A Software Heritage content identifier will be identical to a Git blob identifier of any file with the same content, a Software Heritage revision identifier will be identical to the corresponding Git commit identifier, etc. This is not the case for snapshot identifiers as Git doesn't have a corresponding object type.

Git compatibility is incidental and is not guaranteed to be maintained in future versions of this scheme (or Git), but is a convenient feature for developers, for the time being.

A.4 Examples

Here are a few interesting examples of what the Software Heritage *core identifiers* look like. It is possible to access the corresponding artefact by prepending to the identifier the prefix of the Software Heritage resolver: https://archive.softwareheritage.org/

```
swh:1:cnt:94a9ed024d3859793618152ea559a168bbcbb5e2
```

points to the content of a file containing the full text of the GPL3 license

[5] See https://docs.softwareheritage.org/devel/swh-model/persistent-identifiers.html for more details.

```
swh:1:dir:d198bc9d7a6bcf6db04f476d29314f157507d505
```

points to a directory containing the source code of the Darktable photography application as it was at some point on 4 May 2017

```
swh:1:rev:309cf2674ee7a0749978cf8265ab91a60aea0f7d
```

points to a commit in the development history of Darktable, dated 16 January 2017, that added undo/redo supports for masks

```
swh:1:rel:22ece559cc7cc2364edc5e5593d63ae8bd229f9f
```

points to Darktable release 2.3.0, dated 24 December 2016

```
swh:1:snp:c7c108084bc0bf3d81436bf980b46e98bd338453
```

points to a snapshot of the entire Darktable Git repository taken on 4 May 2017 from GitHub.

A.5 Qualifiers

The semi-colon (`;`) is used as separator between the core identifier and the optional qualifiers, as well as between qualifiers. Each qualifier is specified as a key/value pair, using = as a separator. The following qualifiers are available:

origin : the software origin where an object has been found or observed in the wild, as an URI;

visit : the core identifier of a snapshot corresponding to a specific visit of a repository containing the designated object;

anchor : a designated node in the Merkle DAG relative to which a path to the object is specified, as the core identifier of a directory, a revision, a release or a snapshot;

path : the absolute file path, from the root directory associated to the anchor node, to the object; when the anchor denotes a directory or a revision, and almost always when it's a release, the root directory is uniquely determined; when the anchor denotes a snapshot, the root directory is the one pointed to by HEAD (possibly indirectly), and undefined if such a reference is missing;

lines : line number(s) of interest, usually within a content object

A.6 Examples with qualifiers

The following SWHID points to the source code root directory of the game Quake III Arena[6] with the origin URL where it was found

[6] See `https://en.wikipedia.org/wiki/Quake`$_{\text{I}}$`II`$_{\text{A}}$`rena`

```
swh:1:dir:c6f07c2173a458d098de45d4c459a8f1916d900f;
origin=https://github.com/id-Software/Quake-III-Arena
```

And the following SWHID points to a comment fragment in an example program for the OCamlP3l parallel programming library.

```
swh:1:cnt:4d99d2d18326621ccdd70f5ea66c2e2ac236ad8b;
origin=https://gitorious.org/ocamlp3l/ocamlp3l_cvs.git;
visit=swh:1:snp:d7f1b9eb7ccb596c2622c4780febaa02549830f9;
anchor=swh:1:rev:2db189928c94d62a3b4757b3eec68f0a4d4113f0;
path=/Examples/SimpleFarm/simplefarm.ml;lines=9-15
```