Attributing and Referencing (Research) Software: Best Practices and Outlook from Inria

Pierre Alliez, Université Côte d’Azur, Inria, France, mailto:pierre.alliez@inria.fr
Roberto Di Cosmo, Inria, Software Heritage, University of Paris, France, mailto:roberto@dicosmo.org
Benjamin Guedj, Inria, France and University College London, United Kingdom, mailto:benjamin.guedj@inria.fr
Alain Girault, Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG, 38000 Grenoble, France, mailto:alain.girault@inria.fr
Mohand-Saïd Hacid, Univ. Lyon, University Claude Bernard Lyon 1, LIRIS, Lyon France, mailto:mohand-said.hacid@univ-lyon1.fr
Arnaud Legrand, Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG, 38000 Grenoble, France, mailto:arnaud.legrand@inria.fr
Nicolas Rougier, Univ. Bordeaux, Inria, CNRS, IMN, Labri, Bordeaux, France, mailto:nicolas.rougier@inria.fr

Abstract—Software is a fundamental pillar of modern scientific research, across all fields and disciplines. However, there is a lack of adequate means to cite and reference software due to the complexity of the problem in terms of authorship, roles and credits. This complexity is further increased when it is considered over the lifetime of a software that can span up to several decades. Building upon the internal experience of Inria, the French research institute for digital sciences (http://www.inria.fr), we provide in this paper a contribution to the ongoing efforts in order to develop proper guidelines and recommendations for software citation and reference. Namely, we recommend: (1) a richer taxonomy for software contributions with a qualitative scale; (2) to put humans at the heart of the evaluation; and (3) to distinguish citation from reference.

Keywords — Software citation; software reference; authorship; development process.

I. Introduction

Software is a fundamental pillar of modern scientific research, across all fields and disciplines, and the actual knowledge embedded in software is contained in software source code which is, as written in the GPL license, “the preferred form [of a program] for making modifications to it [as a developer]” and “provides a view into the mind of the designer” [18]. With the rise of Free/Open Source Software, which requires and fosters source code accessibility, access has been provided to an enormous amount of software source code [1]; and (ii) it has experimented a novel process for research software deposit and citation by connecting the French national open access publication portal, HAL (hal.archives-ouvertes.fr), to Software Heritage [1]. Nevertheless, citing and referencing software is a very complex task, for several reasons. First, software authorship is extremely varied, involving many roles: software architect, coder, debugger, tester, team manager, and so on. Second, software itself is a complex object: the lifespan can range from a few months to decades, the size can range to a few dozens of lines of code to several millions, and it can be stand-alone or rely on multiple external packages. And finally, sometimes one may want to reference a particular version of a given software (this is crucial for reproducible
research), while at other times one may want to cite the software as a whole.

In this article, we report on the practices, processes, and vision, both in place and under consideration at Inria, to address the challenges of referencing and accessing software source code, and properly crediting the people involved in their development, maintenance, and dissemination.

The article is structured as follows: Section II briefly surveys previous work. Section III describes the inherent complexity of software, which is the main reason why the topic studied in this paper is challenging. Section IV presents the key internal processes that Inria has established over the last decades to track the hundreds of software projects to which the institute contributes, and the criteria and taxonomies they use. Section V draws the main lessons that have been learned from this long term experience. In particular, we state three recommendations to contribute to a better handling of research software worldwide. Finally, Section VI concludes by providing a set of recommendations for the future.

II. Survey of previous work

The astrophysics community is one of the oldest ones having attempted to systematically describe the software developed and use in their research work. The Astrophysics Source Code Library was started in 1999. Over the years it has established a curation process that enables the production of quality metadata for research software. These metadata can be used for citation purposes, and they are widely used in the astrophysics field [2]. Around 2010, interest in software arose in a variety of domains: a few computer science conferences started an artefact evaluation process (see www.artifact-eval.org) which has spread to many top venues in computer science. This led to the badging system that ACM promotes for articles presenting or using research software [4] and to the cloud-based software hosting solution used and put forward by IEEE and Taylor & Francis for their journals (Code Ocean). The need to take research software into account, making it available, referenceable, and citable, became apparent in many research communities [5], [20], [12], [11], and the limitation of the informal practices currently in use quickly surfaced [13], [7], [14]. An important effort to bring together these many different experiences, and to build a coherent point of view has been made by the FORCE11 Software Citation Working Group in 2016, which led to state a concise set of software citation principles [19]. In a nutshell, this document recognizes the importance of software, credit and attribution, persistence and accessibility, and provides several recommendations based on use-cases that illustrate the different situations where one wants to cite a piece of software.

We do acknowledge these valuable efforts, which have contributed to raise the awareness about the importance of research software in the scholarly world.

Nonetheless, we consider that a lot more work is needed before we can consider this problem settled: the actual recommendations that can be found on how to make software citable and referenceable, and how to give credit to its authors, fall quite short of what is needed for an object as complex as software. For example, in most of the guidelines we have seen, making software referenceable for reproducibility (where the precise version of the software needs to be indicated), or citable for credit (to authors or institutions), seems to boil down to simply finding a way to attach a DOI to it, typically by depositing a copy of the source code in repositories like Zenodo or Figshare.

This simple approach, inspired by common practices for research data, is not appropriate for software.

When our goal is giving credit to authors, attaching an identifier to metadata is the easy part, and any system of digital identifiers, be it DOI, Ark or Handles, will do. The difficulty lies in getting quality metadata, and in particular in determining who should get credit, for what kind of contribution, and who has authority to make these decisions. The heated debate spawned by recent experiments that tried to automatically compute the list of authors out of commit logs in version control systems [5] clearly shows how challenging this can be.

As we will see in Section IV-D when looking for reproducibility, it is necessary to precisely identify not only the main software but also its whole environment and to make it available in an open and perennial way. In this context, we need verifiable build methods and intrinsic identifiers that do not depend on resolvers that can be abused or compromised (see Wiley using fake DOIs to trap web crawlers ... and researchers as well), and DOIs are not designed for this use case [9].

To make progress in our effort to make research software better recognized, a first step is to acknowledge its complexity, and to take it fully into account in our recommendations.

III. Complexity of the software landscape

Software is very different from articles and data, with which we have much greater familiarity and experience, as they have been produced and used in the scholarly arena long before the first computer program was ever written. In this section, we highlight some of the main characteristics that render the task of assessing, referencing, attributing and citing software a problem way more complex than what it may appear at first sight.

Software development is a multifaceted and continuously evolving activity, involving a broad spectrum of goals, actors, roles, organizations, practices and time extents. Without pretending to be exhaustive, we detail here in bold the most important aspects that need to be taken into account for assessing, referencing, attributing or citing software.

Structure

A software project can be organized either as a monolithic program (e.g., the Netlib BLAS libraries), or as a composite assembly of modules (e.g., the Clang compiler). It can either be self-contained or have many external dependencies. For example, the Eigen C++ template library for linear algebra (http://eigen.tuxfamily.org) aims for minimal dependencies while listing an ecosystem of unsupported modules.
Lifetime
A software can be produced during a single, short extent of time (referred to as one-shot contribution), or over a long time span, possibly fragmented into several time intervals of activities. Some long running software projects extend over several decades. For example, the CGAL project (https://www.cgal.org/project.html) started in 1996 as a European consortium, became open source in 2004, and has provided more than 30 releases since then.

Community
A software can be the product of a single scholar, a well-identified team or a scattered team of scholars spanning a large scientific community that may be difficult to track precisely. The CGAL open source project lists more than 130 contributors, distinguishing between the former and current developers, and acknowledging the reviewers and initial consortium members (https://www.cgal.org/people.html). In contrast, the Meshlab 3D mesh processing software (https://en.wikipedia.org/wiki/MeshLab) is authored by a single team from the CNR, Pisa.

Authorship
Software developer(s) writing the code are the most visible authors of a software program, but they are not, and by far, the only ones. A variety of activities are involved in the creation of software, ranging from stating the high-level specifications, to testing and bug fixing, through designing the software architecture, making technical choices, running use cases, implementing a demonstrator, drafting the documentation, deploying onto several platforms, and building a community of users. In these contexts the roles of a single contributor can be plural, with contributions spanning variable time extents. Authorship is even more complicated when developers resort to pseudonymity, i.e., disguised identity in order to not disclose their legal identities. For all these reasons, evaluating the real contributions to a significant piece of software is a very difficult problem: in our experience at Inria, automated tools may help in this task, but are by far insufficient, and it is essential to have humans in the loop.

Authority
Beyond good practices, most quality or certified software development projects define management processes and authority rules. Authorities are entitled to make decisions, give orders, control processes, enforce rules, and report. They can be institutions, organizations, communities, or sometimes a single person (e.g., Guido van Rossum for Python). Some projects set up an editorial board, similar in spirit to scientific journals, with reviewers, managers and well-defined procedures (See CGAL’s Open Source Project Rules and Procedures at https://www.cgal.org/project_rules.html). Each new contribution must be submitted for review and approval before being integrated. Some decisions can be taken top-down while others are bottom-up. In some cases, a shared governance is implemented. This organization can be somehow compared to the Linux kernel development organization where Linus Torvalds integrates contributions but delegates the responsibility of software quality evaluation to a few trusted colleagues. Another important aspect is the traceability of who did what during the software project. In its simplest form, the number of lines or code or commit logs are used for tracing contributions and changes, but more advanced means such as repository mining-based metrics [17], bug-related metrics, or peer evaluation are common.

Levels of description
Another dimension that adds to the complexity is the variety of levels at which a software project can be described, either for citation or for reference.

Exact status of the source code. For the purpose of exact reproducibility, one must be able to reference any precise point in the development history of a software project, even if it is not labeled as a release; in this case, cryptographic identifiers like those used in distributed version control systems, and now generalized in Software Heritage [19], are necessary. For instance, the sentence “you can find at swsh:1:cnt:cfdf19c4487c43c76f3612557d4dc61f9131790a4; lines=146-187 of swsh:1:snpc:c9c31ee9a3c631472c8817886aa0d37894a3782;origin=https://github.com/rdicosmo/parmap/ the exact core mapping algorithm used in this article” makes two distinct references. The former one points to the lines of a source file while the later one points to the software context in which this file is used.

(Major) release. When a much coarser granularity is sufficient, one can designate a particular (major) release of the project. For instance: “This functionality is available in OCaml version 4” or “from CGAL version 3”.

Project. Sometimes one needs to cite a software project at the highest level; a typical example is a researcher, a team or an institution reporting the software projects it develops or contributes to. In this case, one must list only the project as a whole, and not all its different versions. For instance: “Inria has created OCaml and Scikit-Learn”.

IV. Four processes for four different needs
There are four main reasons why the research software produced at Inria is carefully referenced and evaluated: (i) managing the career of individual researchers and research engineers, (ii) assessing the technology transfer, (iii) visibility and impact of a research team, and (iv) promote reproducible research practices. We detail next these four topics, and the information collected, to cater to these different needs.

A. Career management
Software development is a research output taken into account in the evolution of the career of individual researchers and research engineers. Measuring the impact of a software provides a means to measure the scope and magnitude of contributions of research results, when they are carefully translated into usable software. Evaluating the maturity and breadth of software is also essential to guide further developments and resource allocation.
Inria has an internal evaluation body, the Evaluation Committee (EC), the role of which includes evaluating both individual researchers when they apply for various positions (typically ranging from junior researcher to leading roles such as senior researcher or research director), and organizing the evaluations of whole research teams, which take place every 4 years. In both cases, evaluating a given software revolves around three items: (i) the software itself, which can be downloaded and tested; (ii) precise self-assessment criteria filled-in by the developers themselves; and (iii) a factual and high-level description of the software, including the programming language(s) used along with the number of lines of code, the number of man-months of development effort, and the web site from where the software and any other relevant material (a user manual, demos, research papers, ...) can be downloaded.

Among these three items, the self-assessment criteria play a crucial role because they provide key information on the software, how it was developed, and what role each developer played. Version 1 of these “Criteria for Software Self-Assessment” dates from August 2011 [15]. They are also used by the Institute for Information Sciences and Technologies (INS2I) of The French National Centre for Scientific Research (CNRS). It comprises two lists of criteria using a qualitative scale. The first list characterizes the software itself:

**Audience.** Ranging from A1 (personal prototype) to A5 (usable by a wide public).

**Software Originality.** Ranging from SO1 (none) to SO4 (original software implementing a fair number of original ideas).

**Software Maturity.** Ranging from SM1 ( demos work, rest not guaranteed) to SM5 (high-assurance software, certified by an evaluation agency or formally verified).

**Evolution and Maintenance.** Ranging from EM1 (no future plans) to EM4 (well-defined and implemented plan for future maintenance and evolution, including an organized users group).

**Software Distribution and Licensing** Ranging from SDL1 (none) to SDL5 (external packaging and distribution either as part of e.g., a Linux distribution, or packaged within a commercially distributed product).

As an example, the OCaml compiler is assessed as: Audience A5, Software Originality SO3, Software Maturity SM4, Evolution and Maintenance EM4, Software Distribution and Licensing SDL5.

The second list characterizes the contribution of the developers and comprises the following criteria: **Design and Architecture (DA), Coding and Debugging (CD), Maintenance and Support (MS), and Team/Project Management (TPM).** Each contribution ranges from 1 (not involved) to 4 (main contributor). As an example, the personal contribution of one of OCaml’s main developer might be: Design and Architecture DA3, Coding and Debugging CD4, Maintenance and Support MS3, Team/Project Management TPM4.

Overall, these self-assessment criteria have been in use at Inria for several years now. The feedback from both jury members (for individual researchers) and international evaluators (for research teams) is that they are extremely useful, despite their coarse granularity and being based on self-statement. All praise the relevance of the criteria and the fact that they provide a mean to assess the scope and magnitude of contributions to a given software, much more accurately.

**B. Technology transfer**

Information about authorship and intellectual property is a key asset when technology transfer takes place, either in industrial contracts or for the creation of start-ups. Besides, technology transfer is at the heart of Inria’s strategy to increase its societal and economical impact. However, in the particular case of software, technology transfer raises a number of difficulties. Most of the time, transferring a software to industry starts by sending a copy of the software to a French registration agency named Agence pour la Protection des Programmes (APP: [https://www.app.asso.fr](https://www.app.asso.fr)). When doing so, a dedicated form has to be filled that requires to specify all the contributors of the software, and for each of them the percentage of her/his contribution.

When the software is old (typically more than 10 years old), this involves carrying on some archaeology to retrieve the contribution of the first developers (some of whom may have left Inria, or may have not been Inria employees at all). A dedicated technology transfer team interacts with the researchers in this process, taking into account all the different contributions to software development. In particular, they use a taxonomy of roles that includes the following:

**Coding.**

This seems the most obvious part, but it is actually complex, as one cannot just count the number of lines of codes written, or the number of accepted pull requests. Sometimes a long code fragment may be a straightforward re-implementation of a very well known algorithm or data structure, involving no complexity or creativity at all, while at other times a few lines of code can embody a complex and revolutionary approach (e.g., speeding up massively the execution time). Often, a major contribution to a project is not adding code, but fixing code or removing portions of code by factoring the project and increasing its modularity and genericity.

**Testing and debugging.**

This is an essential role when developing software that is meant to be used more than once. This activity may require setting up a large database of relevant use cases and devising a rigorous testing protocol (e.g., non-regression testing).

**Algorithm design.**

Inventing the underlying algorithm that forms the very basis of the software being transferred to industry is, of course, a key contribution.

**Software architecture design**

This is another important activity that does not necessarily show up in the code itself, but which is essential for maintenance, modularity, efficiency and evolution of the software. As Steve Jobs famously said while promoting Object Oriented Programming and the NeXT computer
Fig. 1. Example of the complexity in direct and indirect dependencies for a specific python package (matplotlib). Boxes represent actual packages (libraries that need to be installed on the system), arrows indicate dependencies to other packages, labels indicates the minimal/maximal version number. In blue the Python dependencies, in red the “true” system dependencies incurred by python (e.g., the libc or libjpeg62), in green some “fake” dependencies incurred by the package management system but which are very likely not used by python (e.g., adduser or dpkg).

more than twenty-five years ago, “The line of code that has no bug and that costs nothing to maintain, is the line of code that you never wrote”.

Documentation.

This activity is essential to ease (re)usability and to support long term maintenance and evolution. It ranges from internal technical documentation to drafting the user manual and tutorials.

The older and bigger the software, the more difficult this authorship identification task is.

C. Visibility and impact of a research team

Software is a part of the scientific production that any research team exposes. Software that are diffused to a large scholar audience or commercialized to industrial users may become an important source of inspiration for novel research challenges. Feedback from practitioners or academic users is a precious source of knowledge for determining the research problems with high potential practical impact. Software can also be a key instrument for research, central to the daily research activity of a team, and a main support for teaching and education. It may also become a communication medium between young researchers, e.g., Ph.D. students sharing their research topics and experiments via a common set of software components.

Inria considers (research) software to be a valuable output of research, and has always encouraged its research teams to advertise the software project they contribute to: this can be on the public web page of the team, or in its annual activity report. To simplify the collection of the information concerning the software projects, an internal database, called BIL (“Base d’Information des Logiciels”, i.e. “database of information on software”), has been in use for several years. It allows research teams to deposit very detailed meta-data describing the software projects they are involved in. The BIL can then be used to generate automatically the list of software descriptions for the team web page, for the activity report, and also to prefill part of the forms used in the two processes described above for individual career evaluation and for technology transfer, avoiding the burden of typing in the same information over and over again.

D. Reproducible Research

Another important concern of Inria is supporting reproducibility of research results and the reproducibility crisis takes a whole new dimension when software is involved. Scholars are struggling to find ways to aggregate in a coherent compendium the data, the software, and the explanations of their experiments. The focus is no longer on giving credit, but on finding, rebuilding, and running the exact software referenced in a research article. We identified at least three major issues:

First, the frequent lack of availability of the software source code, and/or of precise references to the right version of it, is a major issue [7]. Solving this issue requires stable and perennial source code archives and specialized identifiers [9].

Second, characterizing and reproducing the full software environment that is used in an experiment requires tracking a potentially huge graph of dependencies (a small example is shown in Figure 1). Specific tools to identify and express such dependencies are needed. Finally, although the notion of research compendium is seducing, it should aggregate objects of very different nature (article, data, software) for which
specific archives and solutions may already exist. To ease the
deposit of such objects, we believe the compendium should
thus rather build on stable references to objects than try to
address all problems at once.

In recent years, various building blocks have emerged
to address these challenges and may lead to such a global
approach and stable references to the software artifact themselves. Inria has fostered and supported a few of them, that we briefly present here.

Software Heritage: a universal archive of source code.

Software Heritage (SWH) was started in 2015 to collect,
preserve and share the source code of all software ever
written, together with its full development history [1].
As of today, it has already collected almost 6 billions
unique source code files coming from over 85 million
software origins that are regularly harvested. The
recently added “save code now” feature enables users to
request proactively the addition of new software origins
or to update them. Source code and its development
history are stored in a universal data model based
on Merkle DAGs [9], providing persistent, intrinsic,
unforgeable, and verifiable identifiers for the more
than 10 billion objects it contains [9]. Each intrinsic
identifier is computed on the content and meta-data of
the software itself, through cryptographic hashes, and is
embedded into the software’s persistent identifier. This
universal archive of all software source code addresses
the issue of preserving and referencing source code for
reproducibility.

Reproducible builds.

In the early 2000’s, the ground-breaking notion of
functional package manager was introduced by the Nix
system [10], using cryptographic hashes to ensure that
binaries are rebuilt and executed in the exact same
software environment. Similar notions provide the founda-
dion of the Guix toolchain, which has been developed
over the last decade under the umbrella of the GNU
project, with key contributions from Inria [8]. The essen-
tial property of these tools is that, given the same source
files and the associated functional build recipes, one can
obtain as a result of the build process the very same
binary files in the same environment. Very recently,
Guix has been connected with SWH to ensure long term
reproducibility: when the source code (currently down-
loaded from the upstream distribution sites) disappears
from the designated location, Guix uses transparently
the SWH intrinsic identifiers to fetch the archived copy
from its archive. Functional build recipes are themselves
a form of source code, and they too can be archived and
given intrinsic identifiers, which will provide proper
references also for software environments.

Curation of software deposit in HAL for SWH.

Over the past two years, Inria has fostered a collabora-
tion between SWH and HAL, the French national open
access archive, with the goal of providing an efficient
process of research software deposit. Figure 2 provides
a high level overview of this process: researchers submit
software source code and meta-data to the HAL portal;
these submissions are placed in a moderation loop
where humans interact with the researchers to improve
the quality of the meta-data and to avoid duplicates;
once a submission is approved, it is sent to SWH via
a generic deposit mechanism, based on the SWORD
standard archive exchange protocol; it is then ingested in
the SWH archive; finally, the unique intrinsic identifier
needed for reproducibility is returned to the HAL portal,
which displays it alongside the identifier for the meta-
data. Detailed guidelines have been developed to help
researchers and moderators get to a high quality deposit
of their source code.

V. Lessons learned on crediting software

The processes described above have been established inside
Inria and refined over decades to answer the internal needs of
the institution. While their goal has not been to guide exter-
nal processes such as software citation, we strongly believe
they provide a solid basis to build a universal framework for
software citation and reference.

Here are a few important lessons we learned from all the
above: (research) software projects present a great degree of
variability along many axes; contributions to software can
take many forms and shapes; and there are key contributions
that must be recognised but do not show up in the code nor in
the logs of the version control systems. This has several main
consequences:

- the need of a rich metadata schema to describe software
  projects;
- the need of a rich taxonomy for software contributions,
  that must not be flattened out on the simple role of
  software developer;
- last but not least, while tools may help, a careful human
  process involving the research teams is crucial to produce
  the qualified information and metadata that is needed
  for proper credit and attribution in the scholarly world.

We focus here mostly on the two latter issues, as the
question of metadata for software has already attracted
significant attention, with the Codemeta initiative providing
a good vehicle for standardisation, and for incorporating the new entities that may be needed [16].

A. Taxonomy of contributor roles: a proposal

The need to recognise different levels and forms of contributions is not new in academia: in Computer Science and Mathematics we are quite used to separate, for example, the persons that are named as authors, and those that are only mentioned in the acknowledgements.

In the specific case of software projects, the Software Credit Ontology [https://dagstuhleas.github.io/SoftwareCreditRoles/doc/index-en.html] proposes a total of 23 roles, among which 13 are directly concerned with an actual contribution to a software project, under the contributor category: code reviewer; community contributor; designer; developer; documenter; idea contributor; infrastructure supporter; issue reporter; marketing and sales; model driven software engineering expert; packager; requirement elicitor; systems and network engineer. As we can see, this ontology is more focused on the business aspect of software projects (see for instance the marketing and sales role) than on technology aspect (for instance, the developer role is further refined into bug fixer, core developer, and maintainer). The taxonomy we propose in the recommendation below is a refinement and combination of the taxonomies presented in Section IV-A and IV-B.

Recommendation #1: A richer taxonomy for software contributions, with a qualitative scale

Giving credit to contributors of a software project is very similar to giving credit to contributors to research articles. We thus need a rich taxonomy. In the previous sections we discussed two taxonomies, developed and used in two different contexts inside Inria: despite minor differences (for example, maintenance and user support are not taken into account for technology transfer), one can extract rather easily the following taxonomy of contributor roles that covers all the use case seen, and that may be extended in the future:

- Design
- Debugging
- Maintenance
- Coding
- Architecture
- Documentation
- Testing
- Support
- Management

But this is only part of the story: in both of the internal Inria processes we described, contributions are not just classified in different roles, they are also quantified, either at a coarse grain (from 1 to 5 for career evaluation), or at a very fine grain (percentages are used for technology transfer, where a financial return needs to be precisely redistributed). We thus recommend using a coarse grain qualitative scale as it is easy to implement and proves to be very helpful whenever technology transfer occurs.

Recommendation #2: Putting human at the heart of the evaluation

As a bottomline, we strongly suggest to refrain, for research software, from trying to generate software citation and credit metadata, and in particular the list of (main) authors, using automated tools: we need instead quality information in the scholarly world, and currently this can only be achieved with qualified human intervention. We strongly encourage the authors of research software to provide such qualitative information, for example in an AUTHORS file, and to use the aforementioned taxonomy and scale.

As an illustration of this recommendation, the rich metadata collected by HAL in the deposit process are sent to SWH using the now standard CodeMeta schema [16], and will be soon extended with the taxonomy of Section V-A.

CRediT system [3] detailing 14 different possible roles, one of which is software: the key idea is that each person listed as an author needs to specify one or more of the 14 roles.

B. The importance of the human in the loop

This quantification is essential, in particular considering that an academic credit system will be inevitably built on top of software citations, which brings us to our next key point: the importance of having humans in the loop, which has already been clearly advocated in a different context by the team behind the Astronomic Source Code Library [2].

As we have already noted, many of the contributor roles identified above are not reflected in the code. In order to assess these roles, in kind and quantity, it is necessary to interact with the team that has created and evolved the software: this is what the technology transfer service at Inria routinely does.

What about the activities that are tightly related to the software source code itself, like coding, testing, and debugging? Here it is very tempting to try to use automated tools to determine the role of a contributor, and the importance of each contribution. There are indeed a wealth of different developer scoring algorithms that target GitHub contributors (see for example [http://git-awards.com], [https://github.com/msparks/git-score] and GitHub’s own scoring using the number of commits, deletions, or additions). Unfortunately these measures are far from robust: refactoring (that may be just renaming or moving file around or even changing tabs in spaces!) can lead to huge score increases, while the actual developer contribution is marginal. And even if one could rule out irrelevant code changes, our experience at Inria is that the importance and quality of a contribution cannot be assessed by counting the number of lines of code that have been added (see our description of the coding role in Section IV-B). This is particularly the case for research software that involves significant innovations.
C. Distinguish citation from reference

We have extensively covered the best practices for assessing and attributing software artefacts: they are essential for giving qualified academic credit to the people that contribute to them, and are key prerequisites for creating citations for software. This complex undertaking requires significant human intervention, and proper processes and tools.

The overall problem of reproducible research is quite different: while there are examples of rather comprehensive solutions in very specialised domains, it seems very difficult to find a unique solution general enough to cover all the use cases. An example of domain specific solution is provided by the IPOL journal (Image Processing On Line, https://www.ipol.im/ an Open Science journal dedicated to image processing): Each article describes an algorithm and contains its source code, with an online demonstration facility and an archive of experiments.

We believe that the three building blocks described in Section IV-D (Software Heritage, NiX/GUIX, curated connections between SWH and HAL) will allow to provide precise references (as illustrated in the end of Section III) to both specific software excerpt, context, and environment and to permanently bind them with research articles.

Recommendation #3: Distinguish citation from reference

It is essential to distinguish citations to projects or results from exact references to software and their environment, and we believe that both should be used in articles. We also strongly encourage the use of tools like GUIX and Software Heritage to build such perennial references.

Although neither a consensus nor a standard exists yet on how to use references in articles, we are currently working on proposing concrete guidelines and adding support in Software Heritage to easily provide the corresponding \LaTeX snippets.

VI. Conclusion

It this article we presented for the first time the internal processes and efforts in place at Inria for assessing, attributing, and referencing research software. They play an essential role for the careers of individual Inria researchers and engineers, the evaluation of whole research teams, the technology transfer activities and incentive policies, and the visibility of research teams.

These processes have to cope with the great complexity and variability of research software, in terms of the nature of its relating activities and practices, roles of its contributing actors, and diversity of lifespans.

Recommendations

Based on our experience over several decades, we have distilled the important lessons learned and are happy to provide a set of recommendations that can be summarized as follows:

Recognise the diversity of contributor roles

The taxonomy of contributors described in Section V-A has been extensively tested internally at Inria. We recommend that it be incorporated in the CodeMeta standard, and all the platforms and tools that support software attribution and citation. In the meanwhile, researchers can adopt it right away in the metadata they incorporate in their own source code.

Keep the human in the loop

To obtain quality metadata, as seen in Section V-B it is essential to have humans in the loop. We strongly advise against the unsupervised use of automated tools to create such metadata. While these automated tools can save a lot of time, we recommend instead the implementation of a metadata curation and moderation mechanism in all tools and platforms that are involved in the creation of metadata for research software, like Zenodo or FigShare. We also recommend that research institutions and academia in general rely on human experts to assess the qualitative contributions of research software, and refrain from adopting as evaluation criteria automated metrics that are easily biased.

Distinguish citation from reference

As explained in Section III citations, used to provide credit to contributors, are conceptually different from references designed to support reproducibility. While the latter can be largely automated using platforms like Software Heritage and tools like GUIX, the former require careful human curation. Research articles will then be able to provide both software citations and software references, and we are currently working on concrete guidelines that we will make publicly available.

References


