IJDC | *General Article*

Persistent Identification and Citation of Software

Catherine Jones
STFC

Brian Matthews
STFC

Ian Gent
University of St Andrews

Tom Griffin
STFC

Jonathan Tedds
University of Leicester

Abstract

Software underpins the academic research process across disciplines. To be able to understand, use/reuse and preserve data, the software code that generated, analysed or presented the data will need to be retained and executed. An important part of this process is being able to persistently identify the software concerned. This paper discusses the reasons for doing so and introduces a model of software entities to enable better identification of what is being identified.

The DataCite metadata schema provides a persistent identification scheme and we consider how this scheme can be applied to software. We then explore examples of persistent identification and reuse. The examples show the differences and similarities of software used in academic research, which has been written and reused at different scales. The key concepts of being able to identify what precisely is being used and provide a mechanism for appropriate credit are important to both of them.

Received 20 October 2015 ~ *Accepted* 24 February 2016 ~ *Revision received* 28 June 2016

Correspondence should be addressed to Catherine Jones, STFC Rutherford Appleton Laboratory, Didcot, OX11 0QX, UK. Email: catherine.jones@stfc.ac.uk

An earlier version of this paper was presented at the 11th International Digital Curation Conference.

Introduction

Software underpins the academic research process across disciplines. To be able to understand, use/reuse and preserve data, the software code that generated, analysed or presented the data will need to be retained and executed. The increased emphasis from funders on the retention and potential reuse of data brings into focus the requirement to understand precisely what software was used in the research process, and for revalidation this knowledge needs to be at a fine grained level.

A starting point is the persistent identification of software to maintain the identity of software as an item over time. This is an emerging area and services such as Zenodo[1] are enabling developers to persistently identify code. In this case the persistence refers to the identifier rather than the object being referred to, since having a persistent identifier does not mean automatically that the object remains persistent. The existence of the identifier means that the information about the object remains persistent and discoverable. It is more discoverable than having the assigned version number.

However, there are additional considerations in the identification of software, as it is a composite artefact and may have different components bundled together. This can be seen in the following definition:

> 'Computer software includes computer programs, libraries and their associated documentation. The word software is also sometimes used in a more narrow sense, meaning application software only.'[2]

Software is written for a purpose, and while programmers might strive for elegance or beauty in the code, the point of software is to be executed. Consequently, if the software is to remain reproducible and reusable, additional consideration needs to be given to maintain its correct execution behaviour.

To be able to cite or manage software for the long term, then the correct collections of components need to be unambiguously identified. To locate software for use or reuse then the metadata describing it needs to be fit for purpose. To actually use software then the original environment needs to be available, or at least enough to reproduce the desired behaviour. The Jisc funded Software Reuse, Repurposing and Reproducibility project has been considering what information is needed to establish these points and uses some examples to demonstrate this. The project builds on the Recomputation project[3] (Gent, 2013; Gent et al., 2014; Arabas, et al., 2014) and earlier work on a framework for software preservation (Matthews et al., 2009; Matthews et al., 2010).

1 Zenodo: https://zenodo.org/
2 Wikipedia – Software: https://en.wikipedia.org/wiki/Software
3 Recomputation project: http://recomputation.org

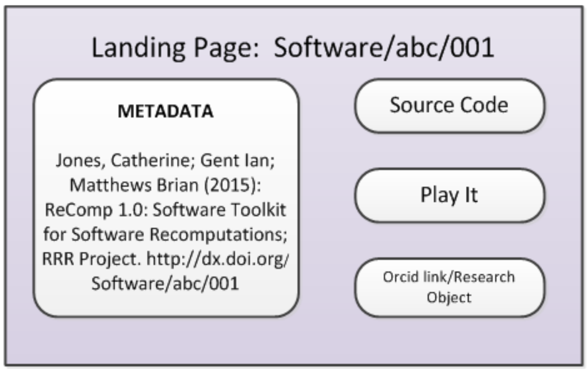

Landing Page: Software/abc/001
METADATA
Jones, Catherine; Gent Ian; Matthews Brian (2015): ReComp 1.0: Software Toolkit for Software Recomputations; RRR Project. http://dx.doi.org/Software/abc/001
Source Code
Play It
Orcid link/Research Object

Figure 1. Software landing page.

Figure 1 encapsulates the vision of the project: a landing page for a persistently identified software object with effective metadata, links to the downloads, including source code and a runnable version, together with hooks to other entities in the wider context. To realise this vision, we need to provide consistent guidelines for software identification together with local metadata and a virtualized platform for replay and recomputation. In the rest of this paper, we concentrate on issues of persistently identifying software.

Stakeholders and their Motivation

The Software RRR Phase II report (Gent et al., 2015) identified and interacted with three stakeholder groups in the identification, citation and preservation of research software: software developers, computational scientists (researchers) creating software, and digital preservation experts. We add funders and researchers re-using software to this list. In our experience, based on the interactions undertaken which may not be representative of the views of all members of the stakeholder groups, the view of these groups can be generalised as follows.

On the whole, research software engineers who follow good software engineering practice feel that there is no need for the long term maintenance or preservation of runnable versions of software, as these can be generated again from the existing code repository, and aren't convinced of the benefits of putting in place further systems. If they are following good software development practices then each version will have a unique identifier and therefore the view is that there is no need for an external persistent identifier beyond the version number. This may be because they are professional software developers and may not use their software, and so are unlikely to need the provenance trail. It may also be that they are pragmatic people and believe that if the code is important enough, then it will be re-implemented following technology shifts.

There has been a mixed reaction from computational scientists. The idea that persistently identifying code would lead to better acknowledgement was well received by some. The importance of reproducibility of the scientific results from software was raised by a couple of people.

Digital preservation experts are interested in the methods and issues around preserving computer software, both as an artefact in its own right and as a tool to enable the preservation and use of other digital objects. For this group, having a unique identifier is an important step in identifying what is in the collection and should be managed. It was identified as a trend for the coming year at iPres 2015.

Funders are increasing their interest in this area, as for some areas of research the computer code is the key research output; however there is no consistent approach.

It is acknowledged that it can be difficult both to find academic research software which has been written by others and then to re-use it. By assigning persistent identifiers to software used in the research literature and citing this software, discoverability is supported.

So to summarise, the further from the creation of the code, the greater the interest in preserving it. The idea of being able to prove reproducibility of results from software analysis is gaining traction but this is still an area which is novel to many.

Issues in Persistent Identification

A key issue is what exactly needs to be identified; in order to usefully assign persistent identifiers to software, we need a common understanding of what is meant by the term *software*. The Wikipedia definition given in the introduction emphasises its composite nature, while saying nothing explicit about the format of the computer software: is it source code which needs to be compiled/interpreted to be able to use it, or is it some form of running code? Furthermore, having the source code doesn't necessarily mean that one will be able to get it to work without a good understanding of the dependencies and operating system it was designed for.

As software is a complex object and may include one or more physical files – including source code, an executable version and additional items, such as included libraries and documentation – we need to be clear what is being referenced by an identifier. Further, software typically is an evolving artefact, with different expressions being made available through a software release cycle, reflecting changes in functionality and the computing environment.

To analyse this further, we consider some typical scenarios where persistent identifiers for software might be used:

- To attribute credit for the intellectual conception or ownership of a software product, its major design features and functionality; or project leadership of the product's development over a sustained period of time. This may be described in a paper describing the overall nature of the software product, and use the general name by which the software in known: e.g. "The ISIS and SNS Neutron Sources are responsible for Mantid."

- To enable a data reuser to understand which software package is most suitable for processing the data, or to communicate which software was used to process data for a particular result in a paper. In this case, we usually need to know the specific features of the software product being used, so rather than refer to the whole product, which will evolve over its lifetime, we typically wish to know which version of the software is being referenced – that is, which release with a particular functionality. For example, "to process the experimental data we used Mantid version 3.1." We may also need to know details about the operating environment and sometimes hardware the software version is tailored for to understand more exactly the behaviour of the software.

- To enable detailed validation of results we may need to know the details of the specific instance of the software which was actually used. This might include how it was configured, the exact nature of the operating environment and hardware setup, possibly including network performance. For example, “Mantid v 3.5.1 running on a Dell XPS 9350 with Intel i7 2.5GHz CPU and 16GB RAM with Red Hat Enterprise Linux v.7.1.”

In different scenarios we would typically refer to a different collection (or collections) of artefacts of the ‘software’, typically representing the different stages of the software product lifetime. Thus we propose that persistent identifiers should be assigned to logical entities representing the stages of software release cycle, and which can be mapped to a collection of component artefacts.

A Model of Software Entities

Software development generally takes place within a project lifecycle of design, modification and release. As software evolves, changing its scope, functionality and the constraints imposed by the computing environment, new collections of digital artefacts are made available as ‘versions’ with particular features. We define a model describing different software entities and their relationships representing the lifetime of a software product at different levels of detail. It may be appropriate to assign persistent identifiers at any and each of these levels depending on the use they are appropriate for, although in general it might be acceptable to miss out some of these entities. Four entities reflect the release process in a software product’s lifecycle, as shown in Figure 2.

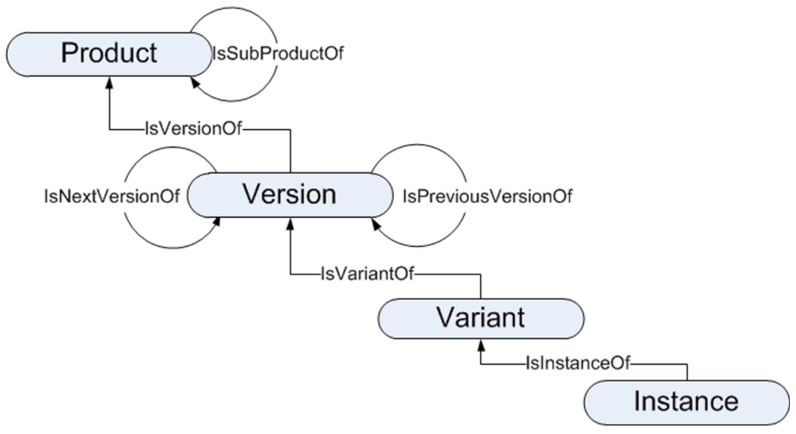


Figure 2. Relationships between software entities.

These entities are defined as follows.

- **Product:** The top-level conceptual entity encompassing the whole development and release lifecycle of the software. It is likely to be how the software is referred to commonly or informally. Using an identifier at this level may be appropriate in order to reference the general concept of a particular software artefact regardless of the specific version, or the continued use or ownership of the software over a long period, If different versions are referenced the product can stand as a unifying entity.

- **Version:** A version of a software product is a single, coherent release of the product with a well-defined functionality and behaviour, defining how it interacts with the computing environment. This level may be needed to identify a specific team of contributors, or where a particular functionality of the version (which may be different in other versions) is being referenced, for example to specify how data and software may be reused.
- **Variant:** Versions may have a number of variations adapted to different operating environments (e.g. version of Windows, MacOS or Linux). A software Variant is usually a manifestation of the product that is adapted for deployment in a specific software operating environment. In this case, the functionality of the version is maintained as much as is practical. However, due to the different behaviours of different platforms, there may be variations in product behaviour, such as in error conditions and user interaction. This may be the appropriate level to use for validation of results or emulation.
- **Instance:** An actual occurrence of a software product which is found on a particular environment or machine[4] is known as an Instance. It may be also referred to as an installation or deployment. This would be an appropriate level for packaging, and for detailed reproduction and validation of results in a specific (virtual) environment.

The model also specifies some relations between the entities as in the diagram we omit a detailed description for brevity as these are fairly self-explanatory. These relations should compose in a natural fashion; thus if a variant entity is omitted, we can relate an instance with the version or product entity with an *IsVariant* relation. Note that there is a *subProduct* relationship; large software developments often have a number of related sub-products within them which have their own attribution and product lifecycles.

Thus, in this model, the persistent identifiers refer to an abstract entity in this hierarchy, which can be described by appropriate metadata. On dereferencing the identifier, we would typically expect the user to be presented with metadata describing the entity (e.g. release notes of a version) and also a collection of (computer) artefacts – an aggregation of other objects which manifest the entity, such as a collection of files. These files may be executables, source code, configuration files, documentation, licences and other (digital and physical) objects.

DataCite Metadata and its use for Software Identification

DataCite[5] issues Digital Object Identifiers for data and other research artefacts. While it is not the only persistent identification system available, its wide adoption means it is an important source for identification of software, and consequently we concentrate on how to adapt Datacite DOIs for the citation of software entities.

DataCite provides set of metadata elements to characterise digital objects for search and discovery (DataCite Metadata Working Group, 2015). The DataCite elements have been analysed to propose an appropriate profile for describing software. We do not

4 Including virtual machines or cloud instances.

5 Datacite: www.datacite.org

prescribe the content of any specific element but rather give a suitable interpretation for software and enable the user to establish the appropriate usage for their own situation. We give some examples of these metadata usage guidelines for software in Table 1. A complete list is given in (Gent, Jones and Matthews, 2015).

Table 1. DataCite metadata elements.

Element	Description
Creator	This element identifies the people responsible for the software. However, this may not be straightforward to ascertain as software has a long life-span and may be worked on by many people. The point during the development cycle that the first DOI is given may also affect those identified as creators.
Title	In a software context, there are some specific issues. If it a piece of software written by a single person for a specific project does it actually have a name? Is the official name different from the common name? What effect is versioning or branching of code going to have on the name? Will the name used be unique enough for it to be found and distinguished from other search results?
ResourceTypeGeneral	There is a controlled value of software, but this is a rather wide category and at present there are few suggestions for how this might be broken down further. This is an area with potential for further work.
Description	This field is designed to enable the addition of further information to assist in the understanding of the object being identified. Currently the two subtypes being used for software are Abstract and Other. These do not encourage the use of this field for technical information that may be needed to understand the object and a new subtype with a more descriptive label may be of assistance. This suggestion has been adopted and will be in the imminent V4.0 release5.

The element which, in our opinion, needs the most adjustment to ensure that it is fit for purpose for describing software is the **relationship type**. This describes the relationship between the digital object being identified and other digital objects. The current set of relationships has been developed from the publication world, which doesn't map to the relationships between different software objects. A particular terminology problem is *IsCompiledBy* which has entirely different meanings for the publication and software communities. Work in this area is being started for version 4.1 of the metadata standards.

Of particular interest are the relationships between the objects described in the model above, alongside cases such as the development forking into separate products.

Example of Persistently Identifying Software

We illustrate the use of the model to assign software DOIs with the following example from practice.

Mantid is an open source development for scientific data analysis used within the Neutron Scattering community[6]. The project assigns DataCite DOIs for attribution and reuse, and DOIs have been assigned at different entity levels from our model, with DataCite metadata used to relate versions of the product, as in Figure 3.

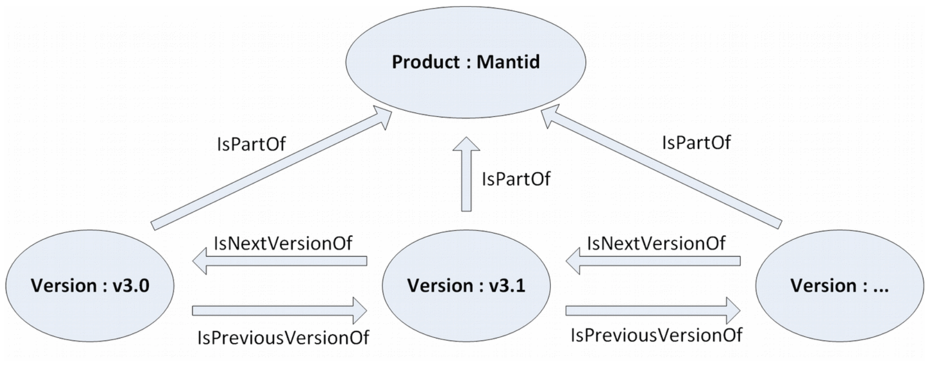


Figure 3. Relationships between MANTID software entities.

Thus a **product** level DOI (doi:10.5286/SOFTWARE/MANTID) is assigned for the whole software package. Each new version has new functionality and a different development team and so there is also a **version** with its own specific DOI. For example, version 3.5.1 is assigned the DOI doi:10.5286/SOFTWARE/MANTID3.5.1.

The relationships between these versions are identified in the DataCite Metadata. However, the DataCite metadata does not have the appropriate fields as specified in our model, and thus we have adopted our profile of the DataCite metadata to represent the entity relations. Thus we have use the DataCite field *IsPartOf* to represent the *IsVersionOf* relation in the software entity model, while the *IsNextVersionOf* and *IsPreviousVersionOf* have analogous relationships in both models.

Software Reuse in Practice

We next illustrate some of the difficulties encountered in identifying and reusing research software, once identified, in a particular biomedical working environment over a number of years in a second example at the Cancer Research Biobank at the University of Leicester[7]. This group worked with the BRISSKit project[8] to tailor and document an instance of the OpenSpecimen[9] software for their active sample data management requirements during 2013-14. The Biobank plans to continue to use and tailor the software in a range of future research projects. Rather than rely on individual project funded researchers with varying technical experience to curate and update the

6 MANTID is a framework that supports high-performance computing and visualisation of materials science data. See: http://www.mantidproject.org/Main_Page

7 Cancer Research Biobank: http://www2.le.ac.uk/partnership/lcrc/facilities/cancer-biobank

8 BRISSKit: http://www.brisskit.le.ac.uk

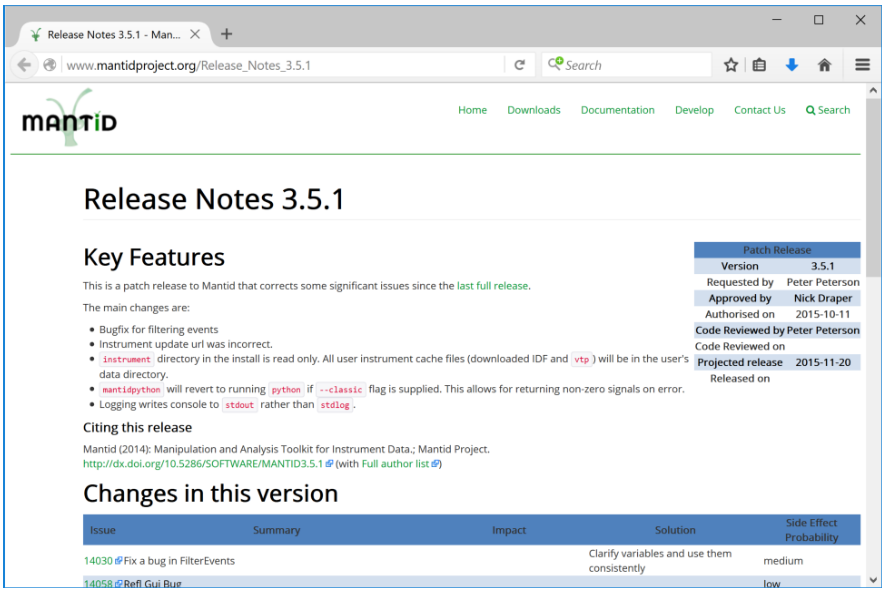
9 OpenSpecimen: http://www.openspecimen.org/

relevant software, the Biobank intended to preserve the tailored instance of the previous open source build in order to reuse and adapt it in future. However, no permanent identifiers had been assigned, just a standard code repository deposit and documentation.

In 2015, on application of OpenSpecimen to a new use case, an urgent need arose to further adapt the previous tailored version. As often is the case, it would need to be undertaken by a different researcher. In the meantime, a new major release of the core OpenSpecimen code itself was now available and the focus of the open source community support. Reusing a locally tailored but significantly earlier version of the code was therefore impractical for all but the most technically advanced researcher.

It may not be enough to simply deposit versioned code in a Github repository since this does not allow redeployment, as will often be desired, without the involvement of a more technically proficient researcher or support resource. This argues for the creation of recomputation platforms allowing a less technical researcher to redeploy previously tailored code and environment where needed.

In comparison, Mantid is designed to be used 'as is'. Further development is being done by a resourced development team rather than the expectation that an individual researcher reusing the software would modify the code. Figure 4 gives the landing page for Mantid version 3.5.1, which provides the release notes for this particular version and a link to downloads to enable the code to be installed locally.


Release Notes 3.5.1 - Man...
www.mantidproject.org/Release_Notes_3.5.1
Search
MANTID
Home
Downloads
Documentation
Develop
Contact Us
Search

Release Notes 3.5.1

Key Features

This is a patch release to Mantid that corrects some significant issues since the last full release.

The main changes are:

- Bugfix for filtering events
- Instrument update url was incorrect.
- `instrument` directory in the install is read only. All user instrument cache files (downloaded IDF and `vtp`) will be in the user's data directory.
- `mantidpython` will revert to running `python` if `--classic` flag is supplied. This allows for returning non-zero signals on error.
- Logging writes console to `stdout` rather than `stdlog`.

Patch Release	
Version	3.5.1
Requested by	Peter Peterson
Approved by	Nick Draper
Authorised on	2015-10-11
Code Reviewed by	Peter Peterson
Code Reviewed on	
Projected release	2015-11-20
Released on	

Citing this release

Mantid (2014): Manipulation and Analysis Toolkit for Instrument Data.; Mantid Project. http://dx.doi.org/10.5286/SOFTWARE/MANTID3.5.1 (with Full author list)

Changes in this version

Issue	Summary	Impact	Solution	Side Effect Probability
14030	Fix a bug in FilterEvents		Clarify variables and use them consistently	medium
14058	Refl Gui Bug			low

Figure 4. Landing page for Mantid 3.5.1.

This enables the reuse of a clearly identified, specific version of the software. Note that the page includes a recommended citation for inclusion in citing papers, including the use of the DOI.

Conclusions and Further Work

The Jisc funded project has now been completed, but work around the citation of software is being carried on by the Force11 Software Citation working group[10], which the authors have joined, along with a UK-based Research Software Engineers network[11] which can also help facilitate further work in this area of rapidly growing importance.

For persistent identification, citation and reproducibility to become common place there needs to be changes to researcher culture and credit. However, it is important to enable these changes by providing consistent guidance and usage.

The examples show the differences and similarities of software used in academic research, which has been written and reused at different scales. The key concepts of being able to identify what precisely is being used and provide a mechanism for appropriate credit are important to both of them.

Finally, we highlighted the key importance for the many researchers who need to reuse existing research software that they themselves may not have written or adjusted. Instances of such software are not normally preserved within a group or at institutional level between projects without a unique business case to do so at institutional level. Hence making a recomputable instance available via a neutral cloud type platform takes on critical importance if researchers are to be able to sustain, reuse and gain continued credit for their own and modified open source software.

Acknowledgments

This work has been funded by Jisc in the Research@Risk scheme. We are grateful to our colleagues Simon Dobson, John McDermott, Rob Hastings and Shajid Issa for their comments and advice.

References

Arabas, S. et al. (2014). Case studies and challenges in reproducibility in the computational sciences. 1st Summer School on Experimental Methodology in Computational Science Research, St Andrews. arXiv:1408.2123

Datacite Metadata Working Group. (2015). DataCite metadata schema for the publication and citation of research data, version 3.1.

Gent, I. (2013). *The Computation Manifesto*. arXiv:1304.3674

Gent, I., Jones. C.M., & Matthews, B.M. (2015). Guidelines for persistently identifying software using DataCite. Jisc Research@Risk project report. Retrieved from http://purl.org/net/epubs/work/24058274

10 Force11 Software Citation working group: https://www.force11.org/group/software-citation-working-group

11 Research Software Engineers network: http://www.rse.ac.uk/

Gent, I.P., Kotthoff, L. (2014). Recomputation.org: Experiences of its first year and lessons learned. 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing (UCC), 968 – 973. doi:10.1109/UCC.2014.158

Gent, I., McDermott, J., Wu C., Jones, C., Matthews, B., Lach, P., Lamerton, S., & Tedds, J. (2015). Software reuse, repurposing and reproducibility phase II report Retrieved from http://rrr.cs.st-andrews.ac.uk/wp-content/uploads/2015/12/SoftwareRRR-Report-Dec151.pdf

Matthews, B., Shaon, A., Bicarregui, J., Jones, C., Woodcock, J., & Conway, E. (2009). Towards a methodology for software preservation. In 6th International Conference on Preservation of Digital Objects (iPres 2009), San Francisco, USA. Retrieved from http://escholarship.org/uc/item/8089m1v1

Matthews, B., Shaon, A., Bicarregui, J., & Jones, C. (2010). A framework for software preservation. *International Journal of Digital Curation 5*(1). doi:10.2218/ijdc.v5i1.145