Manifesto from Dagstuhl Perspectives Workshop 16252

Engineering Academic Software

Alice Allen[1], Cecilia Aragon[2], Christoph Becker[3], Jeffrey Carver[4], Andrei Chiş[5], Benoit Combemale[6], Mike Croucher[7], Kevin Crowston[8], Daniel Garijo[9], Ashish Gehani[10], Carole Goble[11], Robert Haines[12], Robert Hirschfeld[13], James Howison[14], Kathryn Huff[15], Caroline Jay[16], Daniel S. Katz[17], Claude Kirchner[18], Katie Kuksenok[19], Ralf Lämmel[20], Oscar Nierstrasz[21], Matt Turk[22], Rob van Nieuwpoort[23], Matthew Vaughn[24], and Jurgen J. Vinju*[25]

1 University of Maryland, College Park, MD, USA
2 University of Washington, Seattle, WA, USA
3 University of Toronto, Toronto, Canada
4 University of Alabama, Tuscaloosa, AL, USA
5 University of Bern, Bern, Switzerland
6 University of Rennes, Rennes, France; and IRISA, Rennes, France
7 University of Sheffield, Sheffield, UK
8 Syracuse University, Syracuse, NY, USA
9 Technical University of Madrid, Madrid, Spain
10 SRI International, Menlo Park, CA, USA
11 University of Manchester, Manchester, UK
12 University of Manchester, Manchester, UK
13 Hasso-Plattner-Institut, Potsdam, Germany
14 University of Texas at Austin, Austin, TX, USA
15 University of Illinois at Urbana-Champaign, Champaign, IL, USA
16 University of Manchester, Manchester, UK
17 University of Illinois Urbana-Champaign, Champaign, IL, USA
18 INRIA, Le Chesnay, France
19 University of Washington, Seattle, WA, USA
20 Universität Koblenz-Landau, Koblenz, Germany
21 University of Bern, Bern, Switzerland
22 University of Illinois Urbana-Champaign, Champaign, IL, USA
23 VU University Amsterdam, Amsterdam, The Netherlands
24 University of Texas at Austin, Austin, TX, USA
25 CWI, Amsterdam, The Netherlands

Abstract

Software is often a critical component of scientific research. It can be a component of the academic research methods used to produce research results, or it may itself be an academic research result. Software, however, has rarely been considered to be a citable artifact in its own right. With the advent of open-source software, artifact evaluation committees of conferences, and journals that include source code and running systems as part of the published artifacts, we foresee that software will increasingly be recognized as part of the academic process. The quality and sustainability of this software must be accounted for, both *a priori* and *a posteriori*.

The Dagstuhl Perspectives Workshop on "Engineering Academic Software" has examined the strengths, weaknesses, risks, and opportunities of academic software engineering. A key outcome of the workshop is this Dagstuhl Manifesto, serving as a roadmap towards future professional software engineering for software-based research instruments and other software produced and used in an academic context. The manifesto is expressed in terms of a series of actionable "pledges" that users and developers of academic research software can take as concrete steps towards improving the environment in which that software is produced.

Perspectives Workshop June 19–24, 2016 – http://www.dagstuhl.de/16252
1998 ACM Subject Classification D.2.9 [Software Engineering] Management
Keywords and phrases Academic software, Research software, Software citation, Software sustainability
Digital Object Identifier 10.4230/DagMan.6.1.1

* Main contact: Jurgen J. Vinju, `Jurgen.Vinju@cwi.nl`.

Executive Summary

Although the role of software is becoming increasingly important in diverse fields of research, it is commonly not given due recognition. Software is often not cited, or even considered to be citable. Developers of research software are not given due credit. In cases where software embodies a core intellectual contribution of research, its creators may not be invited as co-authors on papers disseminating that research. In cases where software that enabled research can be cited, it may not be. This Dagstuhl Perspectives Workshop explored the *current state* of engineering of academic research software, identified common *problems* with its development, recognition, and sustainability, proposed a set of concrete *actions* to improve the state of academic research software, expressed as a number of personal *pledges*, and put forward numerous *future research directions* to better understand and support academic software.

The personal pledges expressed in this Dagstuhl Manifesto[1] address three general concerns: (i) ensuring that research software is properly *cited*; (ii) promoting the *careers* of research software engineers who develop academic software; and (iii) ensuring the quality and sustainability of software during and following its *development*:

Citation

- I will make explicit how to cite my software.
- I will cite the software I used to produce my research results.
- When reviewing, I will encourage others to cite the software they have used.

Careers

- I will recognize software contributions in hiring and promotion within my institution, and encourage others in my institution to do the same.

Development and Use

- I will develop software as open source from the start, whenever possible.
- I will contribute to sustaining software I use and rely on.
- I will match proposed software engineering practices to the actual needs and resources of the project.
- I will help researchers improve the quality of their software without passing judgment.
- I will publish the intellectual contributions of my research software.
- I will document (including usage instructions, and input and output examples), package, release, and archive versions of my software.

[1] Dagstuhl Perspectives Workshops explore new and emerging topics in Computer Science, and are expected to produce *Dagstuhl Manifestos* that capture trends and developments related to these topics. See: http://www.dagstuhl.de/en/publications/dagstuhl-manifestos/.

Table of Contents

1 Introduction

Academia is software driven. As software is becoming a pervasive technology for automating and innovating every aspect of the human condition, it is also embedded firmly in the academic world. On the one hand, in computer science and in particular software engineering research, we see experimental software and toolkits emerge continuously, either as part of the *output* of research effort, or as part of the *research method*. On the other hand, in general it may be that software is used even more actively in other fields of research such as mathematics, biology, particle physics, astronomy, medicine, and law. Here too, we can distinguish software that is part of the output (*i.e.*, development of innovative production techniques) from software that contributes to the research methods.

There is an explosion of available open-data online which is accessed and analysed through the creation of new software – generating more data to analyse. We focus on the *sustainability* of this software, *i.e.*, any software that acquires, cleanses, stores, annotates, transforms, filters, generates (*etc.*) research data. By "sustainability" we mean the *capacity to endure*. We define software as sustainable if it will continue to be available in the future, on new platforms, and meet new needs.

The perspective of this workshop is that of *the research team developing and/or using academic software*. A key role is that of the *research software engineer*[2] (RSE), the person with expertise both in academic research, and in the software engineering tools and technology needed to build the software used in the research, and to make it sustainable. In this context it is critical that the *intellectual contribution of software* be recognized and actively supported.

As software is becoming integral to our processes, the tools we use, and the output we produce, this perspective provides a starting point for a discussion that is both timely and pressing:

- *What is academic software?* How does it differ from other software? What are its most pressing dimensions of quality? What are the major success factors? What are common pitfalls?
- *Is the software that drives our research methods correct?* Are the inputs and outputs sufficiently specified to be able to interpret the difference between incorrect and correct? How can we verify (test or prove) our claims?
- *Is the software we use and produce in an academic context sustainable?* Are we certain we can reproduce previous research methods in the future given arbitrary changes to the technological contexts (machines, operating systems, programming languages, frameworks)? Are we able to incrementally adapt research software to emerging opportunities at the same time, without loss of reproducibility and without incurring prohibitive cost?
- *Is the software process we use fit for the quality we expect?* How can we optimize it in a unique academic context without losing quality? What tools and processes exist to help with this balance? What investments are necessary to discover it?
- *How can we secure academic software quality?* How can we monitor, steer, report on, and review academic software quality? How can we manage and secure trust between academic research teams considering software developed for output and/or research methods?

[2] Depending on the community, other terms may be used for this role, such as "scientific IT support staff", "cyberinfrastructure concierge", or "research infrastructure engineer." In addition, this role may be functionally filled by faculty or students with appropriate expertise.

- *How should we balance domain knowledge and expertise with software engineering knowledge and expertise in an academic research team?* How can we effectively manage heterogeneous research teams where different domains benefit from each other?
- *How can we motivate research teams to invest in academic software*, especially considering the highly competitive and already complex environment they operate in? What is required in terms of long term funding, education and infra-structure to make the goals of academic software feasible?

The goals of the workshop were to plan how to widen and deepen the impact of software engineering knowledge in research labs across the globe and to prioritize pressing open questions for the software engineering community with respect to research software.

1.1 Pledges

A key result of this workshop has been to identify a number of concrete *actions* to be taken by researchers involved in the use, production, and evaluation of academic software. These actions are expressed as a series of personal *pledges* that researchers are encouraged to take and follow.

The pledges are organized into three groups, dealing respectively with: (i) ensuring that research software is properly *cited*; (ii) promoting the *careers* of research software engineers who develop academic software; and (iii) ensuring the quality and sustainability of software during and following its *development*:

Citation

- I will make explicit how to cite my software.
- I will cite the software I used to produce my research results.
- When reviewing, I will encourage others to cite the software they have used.

Careers

- I will recognize software contributions in hiring and promotion within my institution, and encourage others in my institution to do the same.

Development and Use

- I will develop software as open source from the start, whenever possible.
- I will contribute to sustaining software I use and rely on.
- I will match proposed software engineering practices to the actual needs and resources of the project.
- I will help researchers improve the quality of their software without passing judgment.
- I will publish the intellectual contributions of my research software.
- I will document (including usage instructions, and input and output examples), package, release, and archive versions of my software.

1.2 Pledge Discussion

Here we discuss the *background* motivating each pledge or group of pledges, possible *contradictions or concerns* that pose challenges for pledges, *what actions are needed*, *what impact* these actions are expected to bring, and *who else needs to take action.*

1.2.1 Citation Pledges

I will make explicit how to cite my software.
I will cite the software I used to produce my research results.
When reviewing, I will encourage others to cite the software they have used.

Background. When a scholarly work relies on specific software, it should cite that software, to give credit to the creators of that software, to support reproducibility, and to provide a more complete understanding of the scholarly work. This is important in order to raise the bar for transparency in academic work, and to recognize software as a research object worthy of exposure to peer review. This requirement by reviewers will help strengthen the call for increased openness, transparency, and verifiability of software used in research. As researchers are motivated by pressure to publish, reviewers are uniquely able to apply pressure where rigor is needed. Previous work on this topic includes the Open Science Peer Review Oath [1].

Contradictions or Concerns. Research might use a lot of software, so reference sections could potentially be cluttered with unhelpful citations (*e.g.*, citing all of the libraries used by some package). Software evolves over time, so there might be many different citations that dilute its apparent impact. Software can have dynamic authorship, complicating the question of who deserves credit for a particular version.

The software that needs to be cited may not be published, making it hard to determine how to cite it. There may be limits on the number of citations permitted by a journal. There may be a page limit for the text version of the work. There may not be a clearly defined way to add citations for non-text products, such as software, either in the software itself or in the published metadata (*e.g.*, in Zenodo[3] or figshare[4]). There may be resistance from individual authors, even in the open science community, who believe that noting a URL is enough.[5]

Academics whose careers are impacted by their relationships with editors at prestigious journals may worry about annoying those editors. While this is a battle worth fighting for, junior researchers may be anxious about making this stand against more powerful individuals who might misinterpret it. This also increases the effort needed to review papers. Various models for this have been suggested to reduce this, from not having all reviewers required to test the software, to having special software reviewers who are assigned to reviewing software but not the full paper.

For new types of scholarly works, for example software itself or data, there are no commonly accepted standard practices for how to publish that work with citations.

[3] http://zenodo.org
[4] https://figshare.com
[5] For example, see these discussions:
https://github.com/scipy-conference/scipy_proceedings/pull/123#issuecomment-117323363 and
https://github.com/scipy-conference/scipy_proceedings/pull/123#issuecomment-118170251.

What Actions Are Needed? Require a link to the software, or provide the relevant citation if it is available. Screenshots are not enough!

Reviewers need to take software seriously, and when a piece of software is mentioned, either by name or by description in the body of the work, in the acknowledgements, or in the citations, check to see if it is well-cited. When software is the published work in question, other software on which it depends should be cited.[6]

Authors need guidance about when to cite software (*i.e.*, which software should be cited) and how to cite software (*i.e.*, citation formats that will be picked up by citation tracking systems).

Why Would Those Small Actions Have Impact? If software citations add to a scholar's h-index, it can be beneficial for the authors. Readers of a paper seeking to build on the published work will have more concrete information about how the work was done. Effort by the reviewers should *trickle up* to journal editors and *trickle down* to research authors. By requiring authors to provide these citations, we will work towards educating the community to produce reusable software.

Who Else Needs to Act? Publishers need to agree to citations being included in publications. Software authors need to provide information about how their software should be cited. Citation tracking systems need to include software citations. Editors and authors too will need to begin to hold their colleagues to this standard in order for this to become a community norm. Conference chairs/journal editors may add this requirement explicitly when publishing a call for papers. Publishers and repository administrators need to create mechanisms for all products to include citations as part of their metadata.

1.2.2 Careers Pledge

I will recognize software contributions in hiring and promotion within my institution.

Background. Funding agencies such as the US National Science Foundation are beginning to recognize research products such as software just as they do for publications. This acknowledges the contributions of software as a primary research product. Including these in CVs and reports as well as requiring them from others is an important step towards normalizing them as having similar stature to publications.

The guidelines used for promotion and evaluation are important because they specify what is valued and thus shape the activities that people undertake. Ultimately promotion also determines who continues to be employed and thus whether people can continue to make software contributions at all. Since promotion guidelines are written by the powerful within institutions, many people feel they can have little or no impact on them; even the powerful feel that they cannot deviate from how these are defined in similar institutions, leading to a stalemate that prevents change. Nevertheless, we are all evaluated and we evaluate others, and we can influence these processes when we participate in these evaluations. Furthermore, we can provide templates and guidelines for recognizing software contributions and encourage respected organizations to adopt them. For example, in 1994 the US National Academy of Sciences released a report titled, "Academic Careers for Experimental Computer Scientists

[6] Although it may not be clear how to do so. See: https://danielskatzblog.wordpress.com/2016/03/25/how-should-we-add-citations-inside-software/.

and Engineers”[7]. The report was intended to provide a reference point for change, and has been quoted in many tenure recommendation letters.

Contradictions or Concerns. Software is often considered to be a scientific *instrument* rather than an output itself.[8] Also, junior researchers may be viewed as wasting effort, accused of padding their resume, or in danger of confusing those who review their CVs. Until it is a community norm, junior researchers must be warned not to rely on these types of research products too heavily, as these notations may be ignored or, even worse, resented.

What Actions Are Needed?

- I will work with my institution’s Human Resources department to develop a formal career pathway for Research Software Engineers. I will share the resulting job descriptions with the community.
- I will work with the wider community of RSEs, *etc.* to share knowledge, experience, and job descriptions that can help others set up and sustain research software groups and careers.
- When participating in processes where I am evaluated, I will list my software contributions. If it is not clear how this may be done, I will explicitly ask where software contributions should be listed in the materials.
- If I sit on a review panel, I will encourage those up for promotion to list their software contributions and, where relevant, bring them up in discussions
- When I am involved in writing guidelines for evaluation, even within my research group or for students, I will include guidance about the type of software contributions I believe add value to research.

Who Else Needs to Act? These actions should help to drive change in the university at higher levels and will involve the actions of higher level administrators.

1.2.3 Development and Use Pledges

I will develop software as open source from the start, whenever possible.
I will contribute to sustaining software I use and rely on.

Background. Some projects are intended to be long-lived and widely usable; others are intended only to create outputs for a single research project or even paper. Potential users should be aware of the author’s intentions for a project before deciding to rely on it.

Initially developing software in closed fora often loses the benefits of open source processes. This includes allowing others to determine whom to credit and follow up regarding technical details. Open access allows developers to adapt code to related contexts. It allows a community of users to share the costs of sustaining the resource. Users can determine how well it fits their needs by inspecting details that are not visible from the external interfaces [7]. In contexts where the quality of the software is critical, open access facilitates verification of correctness. Similarly, security flaws can be identified by a larger group, enabling earlier resolution.

Starting as you intend to continue is important, because projects have their own momentum; it is far more difficult to change once you have established practices and infrastructure. Indeed transitioning from a typical, co-located, PI-directed software development

[7] http://www.nap.edu/read/2236/chapter/1
[8] http://ivory.idyll.org/blog/2015-software-as-a-primary-product-of-science.html

process to an open process involves substantial organizational change across at least three dimensions (i) Governance and Leadership, (ii) Collaboration infrastructure, and (iii) Contribution processes [13].

A commonly used argument for not releasing the code associated with a research paper into the wild is that the code is "bad" and the researcher will be judged negatively based on the poor quality of the code [15]. Dismissive, uncharitable comments about existing code can lead to online shaming, creating and perpetuating a negative attitude towards publishing code developed by researchers. This in turn leads to a body of computational research that is extremely difficult to reproduce. It is also wasteful of resources since it costs a great deal of money to develop that software. It takes longer to build on such research since the foundations need to be reimplemented.

A very large movement behind this openness is already well underway and has resulted in more than one manifesto [10, 3].

Contradictions or Concerns. We want researchers to release their code, or even better develop their code completely in the open, however, we criticize them for releasing code that does not comply to a high standard of quality.

Some institutions prohibit release of software under a FOSS license without IP review. This can often be addressed by planning and stating the code will be open sourced at the time funding is sought, *i.e.*, in the proposal itself.

There is literature on success in open source arguing that projects should start with a (nearly) working software application, not just ideas and plans [22]. One possible solution here is to use infrastructure that can be easily opened, such as private repositories on GitHub, but carefully choose when to flip that switch. That does not, however, change the need for practices such as governance.

What Actions Are Needed? Promote a culture acknowledging that any code contribution attached to a research paper that makes it possible to reproduce research results is valuable, regardless of the quality of the code. Once the code is freely available its quality can be improved. Provide, at the university level, services for code review that researchers wanting to release their code can use. This can increase both their confidence and their technical skills.

Establish a specific plan for releasing scientific source code from day one of development. This should include:

- an initial specification of the software license, compliant with the sponsoring institution's IP policies;
- a plan for relicensing in the future should the business environment surrounding a project's development change;
- a plan for identifying and mitigating license conflicts with software dependencies that may be introduced into the current product;
- a policy for applying this software licensing plan uniformly to all (or as many as allowable) projects under a specific manager's supervision.

Give regular talks, seminars and courses on releasing software to the public. Sit alongside individual researchers and take them through how to use GitHub (say) to actually make their code public for the first time.

Whenever a paper is published within your institution that explicitly mentions the development of code, speak to the author about releasing the software behind it. Write and promote articles giving explicit examples of good practice.

Who Else Needs to Act? Reviewers of papers need to insist that code behind papers is released. Funders need to make software release part of the requirement for funding.

I will match proposed software engineering practices to the actual needs and resources of the project.
I will help researchers improve the quality of their software without passing judgment.

Background. By necessity, software is often developed with a somewhat abstracted sense of the ultimate use case. Researchers who adopt our software solutions must be provided assistance in mapping their intellectual context to that of the software. This includes help in adapting their existing analytical processes, formatting data for use by the software, and guidance on how its outputs should be interpreted. Failing to provide this kind of help can lead to mutual frustration, wasted research time or inappropriate outputs, and ultimately undermines the value and sustainability of the solution. Furthermore, being in the trusted position of providing guidance and feedback has to be continually legitimized by responding to real concerns in ways that empower, rather than shame. Adopting a tool or process demands time and energy, but is also associated with great uncertainty. It is often not clear (i) whether the solution is appropriate, and (ii) when one should stop pushing a particular solution.

Contradictions or Concerns. Evaluating whether a solution is appropriate is time-intensive, and end users may not be equipped to do so. Additional expertise is needed, but the person offering that expertise bears the burden of communicating effectively to the end users. This takes time, energy, and its own skill set.

It is hard enough to get funding to support primary software development. Researchers under any sort of financial pressure may have difficulty prioritizing support for adapting existing software over new development and innovation.

What Actions Are Needed? If I am in the position of assessing the context or suggesting a solution, I will not only focus on the things I am most excited about (the shiny new tool) but make an effort to hear what the researchers/users are excited about, and ways the best parts of existing process and enthusiasm can be channelled into sustained productivity.

When a software solution is proffered to a research community, there should be an explicit plan to provide sustained human support for it. This can include a statement that no such support can be provided, but the expectation should be explicitly set for the end users rather than leaving them to struggle. This allows them to incorporate support availability into their evaluation of the appropriateness of the solution.

Time constraints (*e.g.*, introduced by sprints, workshops, hackathons) can help mitigate the uncertainty associated with adoption and create momentum for an in-depth evaluation of the appropriateness of a demanding technological solution.

I will publish the intellectual contributions of my research software.
I will document (including usage instructions, and input and output examples), package, release, and archive versions of my software.

Background. No software project is an island. We all rely on other software to build our software. If we use free software, such as tools, libraries and compilers, then it is fair and reasonable that we try and support these projects if we can. This might include citing them, providing letters of support, reporting issues, fixing bugs, contributing new features or offering monetary contributions.

If you want to get credit for intellectual contributions in software, you have to explicitly make clear what they are. Writing this down is valuable for other RSEs and researchers alike. Moreover, when writing down your intellectual contributions, you will put them into a larger context, forcing yourself to take a broader viewpoint. Publishing these contributions in international peer-reviewed venues strengthens and legitimizes the case that there are significant intellectual contributions in software.

Learning by example is an established pedagogical framework. Users often find it easier to understand how to use a complex system through examples. Each instance typically corresponds to a common use case. This allows users to start working with the software without investing a great deal of time. Lowering the bar to use can incentivize wider investigational use of the system. If a user finds initial utility, they are likely to then invest the effort into understanding more complex functionality that may require study of detailed documentation (or the code itself).

The process of providing examples can help developers improve both their code and documentation. It encourages consideration of how the software will be utilized in practice. A significant benefit of documented software requirements is the effect they have on the development process. In particular, this can bring to the surface contradictions at design time, even before implementation commences. Of similar importance is the identification of corner cases that may not otherwise have been considered. This can resolve potential problems earlier in the development process.

Contradictions or Concerns. Some projects may be concerned about contributing new features that may compromise their research in some way. In this case new features could be contributed upstream after relevant papers have been published.

Of course, writing a paper will consume valuable time that cannot be spent on the software itself. This should be made explicit before a software project starts. Expectation management is needed. PIs have to be on-board for this to work. Authorship rules for these papers should be discussed in advance (*e.g.*, RSEs only, RSE lead and PIs also co-authors, *etc.*).

What Actions Are Needed? Write papers on intellectual contributions in software. Of course, this also implies that you will perform peer review of similar papers written by others. Go to relevant venues to disseminate the work you did. Document your software by:

- Cataloging the components that can be invoked by a user.
- Providing clear descriptions of the requirements for each component.
- If sensible defaults are absent, offering guidance on configuring the system to use each component.
- Considering common use cases for each such piece of code.
- Constructing a sample invocation of the software, along with a specific input and the expected output.
- If the software allows components to be composed, providing examples of how to do so.

Why Would Those Small Actions Have Impact? The impact of this (small) step is more and wider recognition that software can contain intellectual contributions, which in turn supports the long-term career paths of academic software engineers to publish about this. This will also establish the field of eScience/eResearch as a more mature discipline.

Who Else Needs to Act? We need sufficient venues for this type of work (journals, conferences, *etc.*). We need sufficient reviewers. We need management of the RSE to be supportive. We need the collaborating PIs to be supportive. We need to build funding for the time needed to document the contributions and to present them to be either built into the project or to be seen as standard overhead on the work.

2 Future Research Directions

The research areas outlined below are concerned with gaining a deeper understanding of both the process and the output of research software engineering activities. Because both tools and practices change so rapidly, and because such a wide diversity of these exists across different scientific disciplines and contexts, the overarching challenge is to understand how findings about specific projects or groups relate to broader trends over time. The research agenda includes developing ways to measure meaningful aspects of both practice and output of scientific programming and questions regarding design and development of tools to support aspects of relevant activities.

2.1 Quantifying the Availability of Scientific Software

Existing work has included literature surveys of particular publication venues that aim to provide some measure of how much software is available [12, 8, 18]. However, there is limited consensus on what constitutes code that is expected to be released, and the level of scrutiny with which to evaluate a particular publication. These limitations make comparisons across venues difficult, let alone fields or disciplines. Such comparisons constitute a component of understanding the change which we try to bring about with improved tools and advocacy programs.

Specific questions to be addressed:

- Which stakeholders (institutional, funding, *etc.*) are invested in this information, and how to support the systematic gathering of relevant data? Map out existing work that has done these kinds of surveys and collate the various means to measure code availability and create a consistent recommended primer for conducting this survey, in order to enable a broader scope of study and comparability.
- In published scientific literature, how are rates of available, runnable, and hidden software changing over time?
- How can code duplication be identified? What are reasons for duplicated code, and what proportion of these are the result of breakdown in code discovery (rather than a reason arising from other needs)?

2.2 Facilitating Software Discovery Within and Across Disciplines

A major challenge of RSE is *discoverability* of relevant available software and techniques for a given research problem. This challenge is further complicated by the *diversity* of programmer skill, project scale, level of concern and scale of action. The new and ambitious *Software Heritage*[9] initiative could be used to make software available and discoverable, and provides a structured and rich repository of currently more than 3 billion source files.

Understanding the metadata ecosystem is another major research focus, but what other approaches, aside from (or complementary to) standards can support scaling of discoverability? Other questions on this agenda are concerned with measuring the amount and cost of hidden software, but we already recognize it as a pain point that may be addressed with design, tooling, and workshops.

[9] https://www.softwareheritage.org

Claims of cost of hidden software or infrastructure maintenance remain (often) anecdotal or, at best, qualitative self-report. It is difficult to even measure the amount of hidden code (see Q1) let alone quantify its impact. However, this is needed for evaluation of interventions. Some starting points for further research follow:

- HCI/design research: Understand user intention to support discoverability, explore opportunities for machine learning application.
- Instrumentation of existing infrastructure (*e.g.*, GitHub), or use of annotation infrastructure such as hypothes.is, JSON-LD.
- What is the cost of code that is undiscoverable, unavailable, or unrunnable?
- How much does it cost to support / not support software that others use?
- Map out existing models to share the costs, to inform recommendations.

2.3 Sustainability of Software Experimentation

Reproducibility of scientific experiments with major computational components requires prepared environments that will, in principle, allow experiments to be redone in 2, 10, 100, and perhaps even 1000 years. We envision a future in which programs should be rerunnable and feel as persistent as email seems today. A journal like IPOL[10], for example, allows authors to publish a paper together with a runnable program, at least on the current machine available in 2016. Jupyter Notebook[11] allows all necessary components and dependencies to be bundled. Other initial work in this area includes Elsevier's executable paper competition [23], the Collage Authoring Environment [19], SHARE [9], the Scientific Paper of the Future[12] [20], and RunMyCode[13], and many other examples. To extend the persistence of software execution across environments and tools opens new research tracks in the rapidly changing hardware and software environments that we currently live with.

The following specific questions need to be addressed:

- What would we like to have as a research environment for reproducibility, including software and data, 10 years from now?
- How to design and implement a virtual machine for forever-sustainable executable environments?
- How to deal with software legacy in the long term, in particular:
  - De-optimization, porting, refactoring of academic software?
  - Translation of legacy data formats?
- How/when to transition grant funded scientific software projects to alternative models of support including commercialization or building true peer production open source communities in which those using the software contribute enough time to keep the software scientifically useful [14, 11, 6, 16]

2.4 Software Engineering Tools Improving Productivity by Tailoring to Intent and Skill

Many common tools that embody SE best practices are not written for researchers. Tools like Jupyter Notebook provide a more natural environment. In much SE research, the

[10] http://www.ipol.im
[11] http://jupyter.org
[12] http://www.scientificpaperofthefuture.org/gpf/.
[13] http://www.runmycode.org.

assumption is that the programmer is building and maintaining an artefact, but many researchers write code that should work and be correct, but not necessary constitute the same scale or persistence of artefact. There are ongoing SE research projects in this direction. SE is a scholarly community well-positioned to understand and design for these forms of programming.

Here are some possible starting points for further research:

- How can tool design help to encourage SE best practices (*e.g.*, related testing, debugging, reproducibility) in the development of scientific software?
- How do different tools augment programmer cognition?
- Develop typologies of software and corresponding design requirements. For example:
  - Support of "what-if" scenarios
  - Multi-scale interaction – capability to "zoom" in and out of levels of abstraction
  - Avoid independent co-evolution
- Tooling and methodology to enable researchers to think about their models as first class entities
- Enabling co-research of the (visual) language needed to describe new theory, and enabling immediate simulation and verification of the new language.
- How can we arrive at a generalized/moldable Matlab, a moldable maple?

2.5 Re-Tooling the Bibliographic Software Toolchain for Software Citation

For the vision of software citation to be realized, authors need to be able to store a metadata record for software with the needed fields, have the toolchain pass these fields through, and have style files produce output that includes them. To achieve this, interfaces for metadata storage will need to be upgraded to meet these needs and improved style files will need to be pushed into the hands of users.

There are three broadly used bibliographic toolchains (although there are many other systems.)[14]

1. BibTeX,
2. citation style language (Zotero | Papers | Mendeley → citeproc → Word | pandoc | Pages), and
3. Endnote.

In the BibTeX world, this means standardizing a type definition (*e.g.*, @software, although there may be existing partial standards) and have BibTeX interfaces (such as bibdesk) handle it properly. The modern implementations of BibTeX pass through unknown types, so that is fine. Then the bst files need to know what to do when receiving an @software record.

In the Citation Style Language (CSL) world, used by Zotero, Mendeley, and Papers, we need a CSL type that is well support by the user interfaces, the citeproc implementations to pass it through to the CSL files[15] and the CSL files to know what to do with it.

[14] https://en.wikipedia.org/wiki/Comparison_of_reference_management_software
[15] *e.g.*, https://github.com/citation-style-language/styles

In the Endnote world, Endnote and its proprietary style system makes some things difficult to change from the outside. Endnote has to have an appropriate type, and the style files have to process it.[16]

We can define a set of example software citations and produce metadata records for them. We can then use a continuous integration tool to run each of those through each style in each bibliographic tool chain. We can then compare the outputs to the software citation principles and say whether the style file produces valid software citations. By automating that comparison we can define a test.

In the CSL world style file distribution and editing is centralized; by changing the style files in the GitHub repository they will eventually be distributed to users via their software client.

In the BibTeX world, however, bst files are not centralized but distributed in a heterogenous manner (partly through latex distributions, partly through publisher websites). Often the closest thing to a canonical distribution point is the publishers' sites, so we would have to convince each publisher to take an edited bst file. One approach to drive this change could be to regularly tweet the failing test to the publisher and provide a link to a file that does not fail the tests.

One additional necessary element would be tool support to make it easy to get appropriate metadata records into bibliographic software. For example, one could write web importers for Zotero that created an appropriate metadata record from the software landing page, with a generic fallback that works on repository home pages. These importers could read data from CITATION files, micro-format embedded metadata (*e.g.*, schema.org SoftwareApplication) or data composed from repository APIs.

2.6 Analysis of Scientific Software Ecosystem Metadata

With better metadata about projects we can pursue systems that provide insight into the scientific software ecosystem. Example efforts currently include Depsy[17], Libraries.io[18], and the Scientific Software Network Map[19] [17, 14]. These take data on software, authors, mentions in publications, and software projects and compose them into maps that provide information on direct and indirect impact, as well as data often hidden from developers (such as which projects are used with others by end users).

Specific questions in the area of ecosystem analysis and metadata are:

- What kind of metadata is needed? What formats are appropriate?
- What motivates projects to generate and maintain metadata?
- How well can software be observed in the literature?
- What are the moments in which participants are well motivated (*e.g.*, people are motivated just after publishing a new paper to add it to their request for citation)?
- How can we motivate software developers to improve metadata?
- How can we quantify and measure code meaningfully?

[16] Some information on this is available here:
http://endnote.com/sites/en/files/m/pdf/en-x7-win-editing-reference-types-styles.pdf
[17] http://depsy.org,http://blog.impactstory.org/introducing-depsy/
[18] https://libraries.io
[19] http://scisoft-net-map.isri.cmu.edu:7777

3 Participants

- Alice Allen
University of Maryland – College Park, US
- Cecilia Aragon
University of Washington – Seattle, US
- Christoph Becker
University of Toronto, CA
- Jeffrey Carver
University of Alabama, US
- Andrei Chiş
University of Bern, CH
- Benoit Combemale
University of Rennes 1, FR
- Mike Croucher
University of Sheffield, UK
- Kevin Crowston
Syracuse University, US
- Daniel Garijo
Technical Univ. of Madrid, ES
- Ashish Gehani
SRI – Menlo Park, US
- Carole Goble
University of Manchester, UK
- Robert Haines
University of Manchester, UK
- Robert Hirschfeld
Hasso-Plattner-Institut – Potsdam, DE
- James Howison
University of Texas – Austin, US
- Kathryn Huff
University of Illinois at Urbana-Champaign, US
- Caroline Jay
University of Manchester, UK
- Daniel S. Katz
University of Illinois at Urbana-Champaign, US
- Claude Kirchner
Inria – Le Chesnay, FR
- Katie Kuksenok
University of Washington – Seattle, US
- Ralf Lämmel
Universität Koblenz-Landau, DE
- Oscar Nierstrasz
University of Bern, CH
- Matt Turk
University of Illinois at Urbana-Champaign, US
- Rob van Nieuwpoort
VU University Amsterdam, NL
- Matthew Vaughn
University of Texas – Austin, US
- Jurgen J. Vinju
CWI – Amsterdam, NL

Acknowledgements

The participants wish to thank Schloss Dagstuhl for their strong support of this workshop. We thank Tobias Rawald (GFZ Potsdam) for his contributions to the phrasing of the pledges, and Cédric Vonesch for reviewing the final draft.

References

1 Jelena Aleksic, Adrian Alexa, Teresa K. Attwood, Neil Chue Hong, Martin Dahlö, Robert Davey, Holger Dinkel, Konrad U. Förstner, Ivo Grigorov, Jean-Karim Hériché, Leo Lahti, Dan MacLean, Michael L. Markie, Jenny Molloy, Maria Victoria Schneider, Camille Scott, Richard Smith-Unna, and Bruno Miguel Vieira. An Open Science Peer Review Oath. *F1000Research*, January 2015. As part of the “AllBio: Open Science & Reproducibility Best Practice Workshop”. doi:10.12688/f1000research.5686.2.

2 Monya Baker. Why scientists must share their research code. *Nature*, September 2016. doi:10.1038/nature.2016.20504.

3 Lorena A. Barba. Reproducibility PI Manifesto, December 2012. Slides for lightning talk at the ICERM Workshop on “Reproducibility in Computational and Experimental Mathematics”, December 2012; last checked: 2017-05-12. doi:10.6084/m9.figshare.104539.v1.

4 Nick Barnes. Science Code Manifesto, 2013. Last checked: 2017-05-12. URL: https://web.archive.org/web/20160218093215/http://sciencecodemanifesto.org.

5 Christoph Becker, Ruzanna Chitchyan, Leticia Duboc, Steve Easterbrook, Martin Mahaux, Birgit Penzenstadler, Guillermo Rodriguez-Navas, Camille Salinesi, Norbert Seyff, Colin Venters, Coral Calero, Sedef Akinli Kocak, and Stefanie Betz. The Karlskrona manifesto for sustainability design, October 2014. arXiv:1410.6968 [cs]. URL: http://arxiv.org/abs/1410.6968.

6 Matthew J. Bietz, Toni Ferro, and Charlotte P. Lee. Sustaining the development of cyberinfrastructure: an organization adapting to change. In *Proceedings of the ACM 2012 Conference on Computer Supported Cooperative Work (CSCW 2012)*, pages 901–910. ACM, 2012. doi:10.1145/2145204.2145339.

7 Mário Rosado de Souza, Robert Haines, and Caroline Jay. Defining sustainability through developers’ eyes: Recommendations from an interview study. In *Proceedings of the 2nd Workshop on Sustainable Software for Science: Practice and Experiences (WSSSPE 2014)*, 2014. URL: https://www.escholar.manchester.ac.uk/item/?pid=uk-ac-man-scw:296273, doi:10.6084/m9.figshare.1111925.

8 Geraint Duck, Goran Nenadic, Michele Filannino, Andy Brass, David L. Robertson, and Robert Stevens. A Survey of Bioinformatics Database and Software Usage through Mining the Literature. *PLOS ONE*, 11(6):e0157989, 2016. doi:10.1371/journal.pone.0157989.

9 Pieter Van Gorp and Steffen Mazanek. SHARE: a web portal for creating and sharing executable research papers. In *Proceedings of the International Conference on Computational Science (ICCS 2011)*, volume 4 of *Procedia Computer Science*, pages 589–597, 2011. doi:10.1016/j.procs.2011.04.062.

10 Alex Holcombe. Open Access Pledge – Making and tracking open access pledges, 2011. Last checked: 2017-05-12. URL: http://www.openaccesspledge.com/.

11 James Howison. Sustaining scientific infrastructures: transitioning from grants to peer production (work-in-progress). In *Proceedings iConference 2015*, March 2015. URL: https://www.ideals.illinois.edu/handle/2142/73439.

12 James Howison and Julia Bullard. Software in the scientific literature: Problems with seeing, finding, and using software mentioned in the biology literature. *Journal of the Association for Information Science and Technology*, 67(9):2137–2155, 2016. doi:10.1002/asi.23538.

13 James Howison and Kevin Crowston. Collaboration through open superposition: A theory of the open source way. *MIS Quarterly*, 38(1):29–50, 2014. URL: http://aisel.aisnet.org/cgi/viewcontent.cgi?article=3156&context=misq.

14 James Howison, Ewa Deelman, Michael J. McLennan, Rafael Ferreira da Silva, and James D. Herbsleb. Understanding the scientific software ecosystem and its impact: Current and future measures. *Research Evaluation*, 24(4):454–470, 2015. doi:10.1093/reseval/rvv014.

15 Caroline Jay, Rawan Sanyour, and Robert Haines. “Not everyone can use Git”: Research Software Engineers’ recommendations for scientist-centred software support (and what researchers really think of them). To appear in the Journal of Open Research Software.

16 Daniel S. Katz, C. Titus Brown, Arfon M. Smith, and Adam Slagell. A guide to sustainability models for research software projects, 2016. [Last checked: 2017-05-12]. URL: https://github.com/danielskatz/sustaining-research-projects.

17 Amber L. McConahy, Ben Eisenbraun, James Howison, James D. Herbsleb, and P. Sliz. Techniques for monitoring runtime architectures of socio-technical ecosystems, 2012. URL: http://hkl.hms.harvard.edu/uploads/image/pdfs/eisenbraun_2012.pdf.

18 Ivelina Momcheva and Erik Tollerud. Software Use in Astronomy: an Informal Survey, July 2015. arXiv:1507.03989 [astro-ph]. URL: http://arxiv.org/abs/1507.03989.

19 Piotr Nowakowski, Eryk Ciepiela, Daniel Harezlak, Joanna Kocot, Marek Kasztelnik, Tomasz Bartynski, Jan Meizner, Grzegorz Dyk, and Maciej Malawski. The collage authoring environment. In *Proceedings of the International Conference on Computational Science (ICCS 2011)*, volume 4 of *Procedia Computer Science*, pages 608–617, 2011. doi:10.1016/j.procs.2011.04.064.

20 OntoSoft. What is a Geoscience Paper of the Future – The Geoscience Papers of the Future Initiative, 2016. Last checked: 2017-05-12. URL: http://www.scientificpaperofthefuture.org/gpf/what-is-a-gpf.

21 Research Software Engineers Association (UKRSE). What is a Research Software Engineer?, 2013. Last checked: 2017-05-12. URL: https://ukrse.github.io/who.html.

22 Anthony Senyard and Martin Michlmayr. How to have a successful free software project. In *Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC 2004)*, pages 84–91. IEEE, 2004. Pre-print available at http://www.cyrius.com/publications/senyard_michlmayr-successful_project.pdf. doi:10.1109/APSEC.2004.58.

23 Peter Sloot. The executable paper grand challenge, 2011. Part of the International Conference on Computational Science 2011; last checked: 2017-05-12. URL: http://www.iccs-meeting.org/iccs2011/index.html.

24 Arfon M. Smith, Daniel S. Katz, Kyle E. Niemeyer, and FORCE11 Software Citation Working Group. Software Citation Principles. *PeerJ Computer Science*, 2:e86, 2016. doi:10.7717/peerj-cs.86.

25 Benjamin J. Weiner, Michael R. Blanton, Alison L. Coil, Michael C. Cooper, Romeel Davé, David W. Hogg, Bradford P. Holden, Patrik Jonsson, Susan A. Kassin, Jennifer M. Lotz, John Moustakas, Jeffrey A. Newman, J. X. Prochaska, Peter J. Teuben, Christy A. Tremonti, and Christopher N. A. Willmer. Astronomical Software Wants To Be Free: A Manifesto, March 2009. arXiv:0903.3971 [astro-ph.IM]. URL: https://arxiv.org/abs/0903.3971.

26 Mark D. Wilkinson, Michel Dumontier, IJsbrand Jan Aalbersberg, Gabrielle Appleton, Myles Axton, Arie Baak, Niklas Blomberg, Jan-Willem Boiten, Luiz Bonino da Silva Santos, Philip E. Bourne, Jildau Bouwman, Anthony J. Brookes, Tim Clark, Mercè Crosas, Ingrid Dillo, Olivier Dumon, Scott Edmunds, Chris T. Evelo, Richard Finkers, Alejandra Gonzalez-Beltran, Alasdair J.G. Gray, Paul Groth, Carole Goble, Jeffrey S. Grethe, Jaap Heringa, Peter A.C ’t Hoen, Rob Hooft, Tobias Kuhn, Ruben Kok, Joost Kok, Scott J. Lusher, Maryann E. Martone, Albert Mons, Abel L. Packer, Bengt Persson, Philippe Rocca-Serra, Marco Roos, Rene van Schaik, Susanna-Assunta Sansone, Erik Schultes, Thierry Sengstag, Ted Slater, George Strawn, Morris A. Swertz, Mark Thompson, Johan van der Lei, Erik van Mulligen, Jan Velterop, Andra Waagmeester, Peter Wittenburg, Katherine Wolstencroft, Jun Zhao, and Barend Mons. The FAIR Guiding Principles for scientific data management and stewardship. *Scientific Data*, 3, March 2016. Article number 160018 (2016). doi:10.1038/sdata.2016.18.

A Related Material

Pointers to, and commentary on, existing manifestos

The open science and research software communities have been very active in creating manifestos in the style of calls to action. In this section, we discuss some of them, based on our discussions at the Dagstuhl EAS meeting.

Some such manifestos have calls for improved software and bibliography metadata for persistent citation of software. These include both the FAIR guiding principles and the FORCE11 Software Citation Principles and the Science Code Manifesto [26, 24, 4].

Other topics addressed by such manifestos include emphasis on access to source code, which was also a topic of interest in this workshop. Manifestos that address this topic include the Science Code Manifesto, and the Karlskrona Manifesto for Sustainability Design.

Emphasis on grass-roots support and institutional change toward visibility of the research contributions of software and software engineers are also addressed by many of these manifestos. A domain-specific manifesto, "Astronomical Software Wants To Be Free: A Manifesto," is a key example of a call to arms for that community to value the software underlying their science [25]. Broader calls for this community action are included in the UK RSE objectives and the Science Code Manifesto.

Many are more focused on the practice of reproducible software development for research. These include the Open Science Peer Review Oath, the Reproducibility PI Manifesto, the GeoScience Paper of the Future Initiative, and the FAIR principles. The Journal of the American Statistical Association (JASA) will now insist on the availability of code and data during the review of manuscripts [2].

Science Code Manifesto [4]. [20] Focuses on "source code written to specifically process data for a published paper." Therefore, may only cover a portion of the types of software of interest (e.g. from the wording it appears that software that produces scientific data may not be included here). It does cover Code, Copyright, Citation, Credit, and Curation.

FORCE11 Software Citation principles [24][21]. Emphasize persistence and clarity.

Open Access Pledge [10][22]. Focuses on promising to publish software and papers in open access venues. Also notes an emphasis on contributing more.

Open Science Peer Review Oath[23]. Focuses on leveraging one's power as a reviewer to demand open software access, reproducible practices, and transparent review ideals.

Karlskrona Manifesto for Sustainability Design [5][24]. Includes a broad definition of sustainability. In the technical sense, this is along the lines of WSSSPE, but it goes well beyond, including the environmental and social interpretations of sustainability.

Astronomical Software Wants To Be Free: A Manifesto [25][25]. Emphasizes various principles that would improve the visibility and esteem of software within astronomy. It emphasizes structural incentives and adoption of values which recognize this.

[20] https://web.archive.org/web/20160218093215/http://sciencecodemanifesto.org
[21] https://www.force11.org/software-citation-principles
[22] http://www.openaccesspledge.com/
[23] http://f1000research.com/articles/3-271/v2
[24] http://sustainabilitydesign.org/
[25] http://arxiv.org/abs/0903.3971

UK RSE [21][26]. Emphasizes the raising of awareness of the role of Research Software Engineers through communication and support of institutional incentives. Objectives of UKRSE: http://www.rse.ac.uk/objectives.html.

Reproducibility manifesto [3][27]. Includes terms to make software reusable by others. It is focused on reproducibility, leaving sustainability of software out of the question.

The GeoScience paper of the future initiative [20][28]. Has a set of requirements for software to be included in a paper: "with documentation, a license for reuse, and a unique and citable persistent identifier". Differences with the scope of our workshop: it involves also data and its provenance, focusing more on the paper itself rather than the software.

FAIR principles [26][29]. Includes focus on research data. The goal is to make them findable, accessible, interoperable and reusable. These principles can generally be applied to software as well.

[26] http://www.rse.ac.uk/who.html
[27] http://lorenabarba.com/gallery/reproducibility-pi-manifesto/
[28] http://www.ontosoft.org/gpf/what-is-a-gpf
[29] http://www.nature.com/articles/sdata201618