"Can I Implement Your Algorithm?": A Model for Reproducible Research Software

Tom Crick
Department of Computing
Cardiff Metropolitan University
Cardiff, UK
Email: tcrick@cardiffmet.ac.uk

Benjamin A. Hall
Microsoft Research
Cambridge, UK
Email: benhall@microsoft.com

Samin Ishtiaq
Microsoft Research
Cambridge, UK
Email: samin.ishtiaq@microsoft.com

***Abstract*—The reproduction and replication of novel scientific results has become a major issue for a number of disciplines. In computer science and related disciplines such as systems biology, the issues closely revolve around the ability to implement novel algorithms and approaches. Taking an approach from the literature and applying it in a new codebase frequently requires local knowledge missing from the published manuscripts and project websites. Alongside this issue, benchmarking, and the development of fair, and widely available benchmark sets present another barrier. In this paper, we outline several suggestions to address these issues, driven by specific examples from a range of scientific domains. Finally, based on these suggestions, we propose a new open platform for scientific software development which effectively isolates specific dependencies from the individual researcher and their workstation and allows faster, more powerful sharing of the results of scientific software engineering.**

I. Introduction

Marc Andreessen famously said in 2011 that "software is eating the world" [1]. It is true: we clearly live in a computational world, with our everyday communications, entertainment, shopping, security, transportation, ... all heavily dependent on (or replaced by) software.

This is particularly true for science and engineering. A 2012 report by the Royal Society stated that computational techniques have "*moved on from assisting scientists in doing science, to transforming both how science is done and what science is done*" [2]. New experiments, simulations, models, benchmarks, even proofs cannot be done without software. And this software does not consist of simple hack-together, use-once, throw-away scripts; scientific software repositories contain thousands, perhaps millions, of lines of code and they need to be actively supported and maintained. More importantly, with reproducibility being a fundamental tenet of science, they need to be re-useable.

However, if we want to be truthful about this, then the scientific literature related to software tools often do not appear to be adhering to the rules themselves [3]. How many of them are reproducible? How many explain their experimental methodologies, in particular the basis for their benchmarking? In particular, can we build the code? [4] We, the authors, are as guilty as anyone in the past, where we have succumbed to publishing papers [5], [6] with benchmarks and promises of code to be released in the near future.

There are many reasons why the wider scientific community is in this state. We are experiencing significant changes in academic dissemination and publication, especially the open access movement, with new models being proposed [7], [8]. There are numerous non-technical impediments to making software maintainable and re-useable, too. The pressure to "make the discovery" and publish quickly disincentivises careful software development. Releasing code prematurely is often seen to give your competitors an advantage, but we should be shining light into these "black boxes" [9].

Things can and should be much better. In this paper, we present a call to action with a set of recommendations which we hope will lead to better, more sustainable, more re-useable software, to move towards an imagined future practice of software development and usage in science and engineering. The basis for many of these recommendations is the basic scientific tenet of openness. There has been previous work in this area [10], [11], as well as a range of manifestos for reproducible research and community initiatives, such as cTuning [1] and the Recomputation Manifesto [2], along with curated recommendations on where to publish research software [3].

II. Can I implement your algorithm?

Reproducibility is a basic tenet of good science. Yet many descriptions of algorithms are too high-level, too obscure, too hand-wavey to allow an easy implementation by a third party. A line in the algorithm might say: "We pick an element from the frontier set" but which element do you pick? Will the first one do? Why will any element suffice? Sometimes the author would like to give more implementation detail but is constrained by the the paper page limit. Sometimes the authors' description in-lines other algorithms or data structures that perhaps only that author is familiar with.

We recommend here that a paper must describe the algorithm in such a way that it is implementable by any reader of that algorithm. This is subjective, of course. So, we also recommend that good scientific conferences have a special track for papers that only re-implement past papers' algorithms, techniques, or tools.

III. Set the code free

There can be no better proof that your algorithm works, than if you provide the source code of an implementation. Software development is hard, but sharing and using others code is relatively easy.

[1]http://ctuning.org/
[2]http://www.recomputation.org/
[3]http://www.software.ac.uk/resources/guides/which-journals-should-i-publish-my-software

A long time ago, Richard Stallman imagined that all code would be free and we would make our money by consulting on the code. Somewhat ironically, this is the case for a significant part of the computing industry now. There are, of course, hard commercial pressures for keeping code closed-source. Even in the scientific domain, scientists and their collaborators may wish to hold onto their code as a competitive advantage, especially if there exists larger competitors who could use the available code to "reverse scoop" the inventors, charging into a promising new research area opened by the inventors.

Closed source is one thing. Licenses that deny the user from viewing, modifying, or sharing the source are another thing. There are, however, even licences on widely adopted tools like GAUSSIAN [12] that prohibit even analysing software performance and behaviour.

There is little doubt that, if science wants to be open and free, then the code that underlies it too needs to be open and free. Code that is available for browsing, modifying, and forking facilitates testing and comparison, and promotes competition. We recommend that code be published under an appropriate open source license [13]; while we defer legal discussiosn of any particular licences, BSD and Apache are good, flexible ones.

A wide variety of licenses exist for molecular dynamics software, with different degrees of openness (GROMACS is LGPL [14], CHARMM/CHARMm and Desmond are Academic/Commercial software licences [15], [16], Amber and NAMD are custom open-like licences). Z3 is an example from the verification area: the code itself is not open source, but the MSR-LA that allows the source code to be read, copied, forked for academic use, provides researchers in the field much more than before [17].

Ultimately: set the code free. Put it on a public space such as GitHub, where it is easy to share and fork. You should embrace the spirit of the CRAPL academic-strength open source license [4] and publish your code – it is good enough [18].

IV. Be a better person

If you have the skills and the experience, you can create better software. We have seen the emergence of initiatives, such as the Software Carpentry [5], Software Sustainability Institute [6] and the UK Community of Research Software Engineers [7] to cultivate world-class research through software, develop software skills and raise the profile of research software engineers.

Some scientists may not have had any formal, or even informal, training in programming. Even basic training in software engineering concepts like revision control, unit tests, etc can help improve the quality of the software written enormously [19] [8]. Interestingly, many of these concepts are taught to computer science undergraduates, but it could be argued that they are taught at the wrong time of the engineers' careers, where the importance of complex, long-running projects is not yet appreciated.

[4]http://matt.might.net/articles/crapl/

[5]http://software-carpentry.org/

[6]http://www.software.ac.uk/

[7]http://www.rse.ac.uk

[8]Also see: http://philipwfowler.wordpress.com/2013/12/19/the-oxford-software-carpentry-boot-camp-one-year-on/

These should be regarded as fundamental literacies for scientists and engineers: we recommend that basic programming and computational skills are taught as a core at undergraduate and postgraduate level.

V. Latin is the language of God

There really is no other scientific or technical field where its participants can just make up a non-principled artefact like a programming language so easily. In a way, it says how much of a "commons" computer science is, that anyone and his dog can create a new programming language, API, framework or compiler. This clearly has advantages and disadvantages.

What is clear is that the use of a principled, high-level programming language to write your software in helps hugely with the maintainability and robustness of the software produced. Such programming languages impose constraints like types: that you can never add a number and a string is the most basic example, but ML's functors provide princpled ways of plugging in components with their implementations completely hidden. Aggressive type checking avoids a subset of bugs which can arise due to incorrectly written functions e.g. well publicised NASA problems with a Mars orbiter [9]. A further example is a pressure coupling bug in GROMACS [14], which arose due to the inappropriate swapping of a pressure term with a stress tensor. A further extension of types, a concept called units of measure that is implemented in languages such as F#, can deal with these kinds of bugs at compile time. Similarly, problems found using in-house software for crystallography led to five papers retractions [20], due to a bug which inverted the phases.

High level languages are often more readable than their competitors. The "density" of a program is often seen to be a good thing, but it is not always the case that a shorter Haskell program is better to maintain than longer Python/C++ one. Nevertheless, what is important is the readability of the code itself. A good example here is from the world of automatic theorem proving: the SSReflect language is much more readable than the original, standard Coq language [21]. SSReflect uses mathematicians' vernacular for script commands, allows reproducibility of automatic proof checking because parameters are named rather than numbered. Even though these proof scripts are really only ever going to be run by a machine, they seek to maintain the basic mathematical idea that a proof should be readable by another mathematician.

VI. Test it to see

Some models may be chaotic and influenced by floating point errors (e.g. molecular dynamics), further frustrating testing. For example: Sidekick is an automated tool for building molecular models and performing simulations [22]. Each system is simulated from an different initial random seed, and under most circumstances this is the only difference expected between replicas. However, on a mixed cluster with both AMD and Intel microprocessors on the nodes, the difference in architecture was found to alter the number of water molecules added to each system by one. This meant that the same simulation performed on different architectures would diverge. Similarly, in a different simulation engine, different neighbour searching strategies gave divergent simulations due to the differing order in which forces were summed.

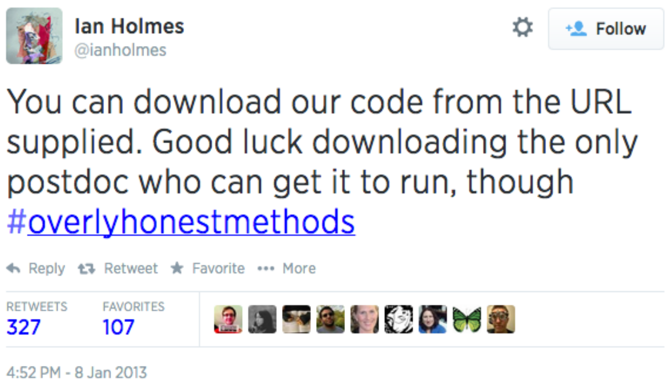
[9]See: http://www.cnn.com/TECH/space/9909/30/mars.metric.02/

Despite these challenges to testing, unshared code is ultimately untestable. Testing new complex scientific software is difficult, as until the software is complete unit tests may not be available. You should thus link from shared code – shared code is more test-able.

VII. Lineage

Research software is not just software – it is the instantiation of novel algorithms and data structures (or at least novel applications of data structures). Thus, lineage is important: the code should include links to papers publishing key algorithms and the code should include explicit relationships to other projects on the repository (i.e. *Project B* was branched from *Project A*). This ensure that both the researchers and software developers working upstream of the current project are properly credited, encouraging future sharing and development. Remember, the people who did the research are not necessarily the same people as the developers and maintainers of the software, so it is important to reward both appropriately with citations (a good way of doing this is the use of CITATION files [10]).

VIII. YMMV


Ian Holmes
@ianholmes
Follow

You can download our code from the URL supplied. Good luck downloading the only postdoc who can get it to run, though #overlyhonestmethods

Reply Retweet Favorite More

RETWEETS 327 FAVORITES 107

4:52 PM - 8 Jan 2013

Figure 1. An #overlyhonestmethod
[source: https://twitter.com/ianholmes/status/288689712636493824]

The tweet is Figure 1 is funny because it is worryingly true. Often, the tool that the paper describes does not exist for download. Or runs only on one particular bespoke platform. Or might run for the author, for a while, but will bit-rot so quickly that even the author cannot compile it in a couple of month's time.

Providing the source code of the tool helps with this, of course. But you must also provide details of precisely *how* you built and wrote the software: you should provide the compiler and build toolchain; you should provide builds tools/makefiles/ant/etc and build instructions; you should list or link to all non-standard packages and libraries that you use; you should note the hardware and OS used. This sounds like a lot of work, but GitHub APIs, Continuous Integration servers, VMs and cloud environments can make it easier; see Section XII at the end for more on this.

IX. Forms of Representation

We often do not, and should not care how things are stored on disk, what their precise representations are. But a common, constrained, standard representation is good for passing tests or models around between different tools. A properly described representation, like the SMT-LIB format [11] for SAT Modulo Theory solvers, where both the syntax and semantics are well understood, aids hugely in developing tools, techniques, benchmarks.

Another example, from biology, is that of the standard representation of qualitative networks and Boolean networks [23], [24]. These networks can be expressed in SMV format, but this would mean that standard QN and BN behaviours have to be hard-coded for each variable, introducing the possibility for errors. In the BioModelAnalyzer tool [25], the XML contains *only* the modifiable parameters limiting the possibility for error.

X. 9.63sec

The benchmarks the tool describes are fashioned only for this instance of this time. They might claim to be from the Windows device driver set, but the reality is that they are are stripped down versions of the originals. Stripped down so much as to be useless to anyone but the author vs. the referee. It is worse than that really: enough benchmarks are included to beat other tools. The comparisons are never fair (neither are other peoples' comparisons against your tool). If every paper has to be novel, then every benchmark, too, will be novel; there is no monotonic, historical truth in new, synthetically-crafted benchmarks. It is as if, in order to beat Usain Bolt's world record time, you put him in a muddy icy track, and weighed him down with 50kg of excess weight. Given this set up, you could hope to beat his 9.63s on a shorter length track.

Benchmarks should be public. They should allow anyone to contribute, implying that the tests are in a standard format. Further, these benchmarks must be heavily curated. Every test/assertion should be justified. Papers should be penalised if they do not use these public benchmarks.

A good example of some of these points is the protein data bank (http://www.pdb.org) and Systems Biology Markup Language [26], [27]. The software ones we know of, the SMT Competition and the SV-COMP ones [28], [29], are on that journey. Such repositories would allow the tests to be taken and easily analysed by any competitor tool.

XI. Welcome to Web 2.0

The web and the cloud really do open up a whole new way of working. Even small, seemingly trivial features like putting up a web interface to your tool and its tests will allow users who are not able to install necessary dependencies to explore the running of the tool [30]. Ultimately, this can lead to making an "executable paper" appear on the internet. The interactive *Try F#* [12] and Z3 tutorials [13] are a great start that begin to expose what can be done in this area.

Virtual machines on the cloud also make the testing of scaling properties more simple. If you have a tool that you claim is more efficient, you could put together a cluster of slow nodes in the cloud to demonstrate how well the software scales for parallel calculations. Cloud compute is cheap, and getting cheaper. Algorithms that used to require massive HPC resources can now be run cheaply by bidding on the VM spot market. The web is a great leveller.

[10] http://blog.rtwilson.com/encouraging-citation-of-software-introducing-citation-files/

[11] http://smt-lib.org

[12] http://www.tryfsharp.org/Learn

[13] http://rise4fun.com/Z3/tutorial/guide

XII. Conclusion

This is how we imagine the future for research software: suppose you have come up with a better algorithm to deal with some standard problem. You write up the paper on the algorithm, and you also push an C++ implementation of your algorithm to the WSSSPE Cloud's section on this standard problem.

The effect of pushing your implementation is to register your program as a possible competitor in this standard problem competition. There are several dozen widely-agreed tests on this problem already on the WSSSPE Cloud's database. Maybe, after some negotiation due to your novel approach to this standard problem, you add some of your own tests to the database too.

Pushing your code activates the Cloud's Continuous Integration system. The cloud pulls in all the dependencies your code needs, on the platforms you specify, and runs all the benchmarks. This happens every time you push. It also happens every time one of your dependencies (a library, a firmware upgrade for your platform, a new API) changes too.

The proposal above brings together almost all of the points we have discussed in this paper. There are already several web services that nearly do this. Something more complete, and stamped with the authority of the major conferences or journals, would mean that your code would never bit-rot, and no one would have problems reproducing the implementation of your published algorithm.

References

[1] M. Andreessen, "Why Software Is Eating The World," *The Wall Street Journal*, August 2011, available online: http://online.wsj.com/news/articles/SB10001424053111903480904576512250915629460.

[2] Royal Society, "Science as an open enterprise," 2012, available from: https://royalsociety.org/policy/projects/science-public-enterprise/report/.

[3] Editorial, "Devil in the details," *Nature*, vol. 470, no. 7334, pp. 305–306, 2011.

[4] C. Collbery, T. Proebsting, G. Moraila *et al.*, "Measuring Reproducibility in Computer Systems Research," Department of Computer Science, University of Arizona, Tech. Rep., 2014.

[5] T. Crick, M. De Vos, M. Brain *et al.*, "Generating Optimal Code using Answer Set Programming," in *Proceedings of 10th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'09)*, ser. Lecture Notes in Computer Science, vol. 5753. Springer, 2009, pp. 554–559.

[6] J. Berdine, B. Cook, and S. Ishtiaq, "Slayer: Memory safety for systems-level code," in *CAV*, 2011, pp. 178–183.

[7] V. Stodden, P. Guo, and Z. Ma, "Toward Reproducible Computational Research: An Empirical Analysis of Data and Code Policy Adoption by Journals," *PLoS ONE*, vol. 8, no. 6, 2013.

[8] G. Fursin and C. Dubach, "Community-Driven Reviewing and Validation of Publications." ACM Press, 2014, pp. 1–4.

[9] A. Morin, J. Urban, P. D. Adams *et al.*, "Shining Light into Black Boxes," *Science*, vol. 336, no. 6078, pp. 159–160, 2012.

[10] S. Sim, S. Easterbrook, and R. Holt, "Using benchmarking to advance research: a challenge to software engineering," in *Proceedings of the 25th International Conference on Software Engineering (ICSE 2003)*. IEEE Press, 2003, pp. 74–83.

[11] F. Chirigati, M. Troyer, D. Shasha *et al.*, "A Computational Reproducibility Benchmark," *IEEE Data Engineering Bulletin*, vol. 36, no. 4, pp. 54–59, 2013.

[12] J. Giles, "Software company bans competitive users," *Nature*, vol. 429, no. 6989, p. 231, 2004.

[13] "Open Source Licenses," http://opensource.org/licenses.

[14] B. Hess, C. Kutzner, D. van der Spoel *et al.*, "Gromacs 4: Algorithms for highly efficient, load-balanced, and scalable molecular simulation," *Journal of Chemical Theory and Computation*, vol. 4, no. 3, pp. 435–447, 2008.

[15] B. R. Brooks, C. L. Brooks, A. D. Mackerell *et al.*, "Charmm: The biomolecular simulation program," *Journal of Computational Chemistry*, vol. 30, no. 10, pp. 1545–1614, 2009.

[16] K. J. Bowers, E. Chow, H. Xu *et al.*, "Scalable algorithms for molecular dynamics simulations on commodity clusters," in *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*. IEEE Press, 2006, p. 43.

[17] L. de Moura, "Releasing the Z3 source code," 2012, available online: http://leodemoura.github.io/blog/2012/10/02/open-z3.html.

[18] N. Barnes, "Publish your computer code: it is good enough," *Nature*, vol. 467, no. 753, 2010.

[19] G. Wilson, "Software Carpentry: lessons learned," *F1000Research*, vol. 3, no. 62.

[20] G. Miller, "A Scientist's Nightmare: Software Problem Leads to Five Retractions," *Science*, vol. 314, no. 5807, pp. 1856–1857, 2006.

[21] G. Gonthier, B. Ziliani, A. Nanevski *et al.*, "How to make ad hoc proof automation less ad hoc," *J. Funct. Program.*, vol. 23, no. 4, pp. 357–401, 2013.

[22] B. A. Hall, K. B. A. Halim, A. Buyan *et al.*, "Sidekick for membrane simulations: Automated ensemble molecular dynamics simulations of transmembrane helices," *Journal of Chemical Theory and Computation*, vol. 10, no. 5, pp. 2165–2175, 2014.

[23] S. A. Kauffman, "Metabolic stability and epigenesis in randomly constructed genetic nets," *Journal of Theoretical Biology*, vol. 22, no. 3, pp. 437–67, 1969.

[24] M. A. Schaub, T. A. Henzinger, and J. Fisher, "Qualitative networks: a symbolic approach to analyze biological sign aling networks." *BMC Systems Biology*, vol. 1, p. 4, 2007.

[25] D. Benque, S. Bourton, C. Cockerton *et al.*, "BMA: visual tool for modeling and analyzing biological networks," *Computer Aided Verification*, vol. 7358, no. 50, pp. 686–692, 2012.

[26] M. Hucka, A. Finney, H. M. Sauro *et al.*, "The systems biology markup language (SBML): a medium for representation and exchange of biochemical network models," *Bioinformatics*, vol. 19, no. 4, pp. 524–31, 2003.

[27] C. Chaouiya, D. Berenguier, S. M. Keating *et al.*, "SBML qualitative models: a model representation format and infrastructure to foster interactions between qualitative modelling formalisms and tools," *BMC Syst Biol*, vol. 7, p. 135, 2013.

[28] "SMT COMP," 2014, http://smtcomp.sourceforge.net/2014/.

[29] "SV COMP," 2015, http://sv-comp.sosy-lab.org/2015/.

[30] B. A. Hall, E. Jackson, A. Hajnal *et al.*, "Logic programming to predict cell fate patterns and retrodict genotypes in organogenesis," *Journal of The Royal Society Interface*, vol. 11, no. 98, 2014.