Lightning Talk: Bringing Techniques from Software Engineering into Scientific Software

Eric L. Seidel
University of California San Diego

Gabrielle Allen
University of Illinois at Urbana-Champaign

Abstract—**Research software, for example the large body of simulation frameworks and libraries in Computational Science & Engineering (CSE), is often designed, developed, and then supported by students and postdoctoral researchers trained in scientific domains and not in the discipline of software engineering. While there are a number of initiatives to improve the training and support recognition of these research software developers, we believe there has been little attention in bringing the wealth of research and experience in the computer science field of Software Engineering to bear on the development and support of research software.**

This paper is written from an experience base of working as an undergraduate REU scholar as part of a team developing the Cactus Framework — a component-based simulation framework for high performance computing, before training as a PhD researcher in Software Engineering. This lightning talk will describe several areas of Software Engineering that could be investigated for their benefit for improving the quality and productivity of research software, and suggest opportunities and challenges for bringing the two communities together.

I. Software Correctness

A fundamental concern for research software is correctness. Incorrect software can lead to crashes or deadlocks during a long-running computation and waste precious time and money, or worse, it could complete successfully but produce an incorrect result, e.g. due to rounding error in floating-point arithmetic. Researchers in the Programming Languages and Software Engineering communities have studied techniques for ensuring correctness extensively. Though our motivation is seldom connected to scientific software, many of the techniques and tools should be transferrable.

Ensuring correctness usually starts with a high-level specification of a property that we wish our software to exhibit. For example, when writing low-level C code, we would likely want to ensure memory safety. If we are writing concurrent or parallel code, we may want to ensure deadlock freedom. Software that deals with floating-point arithmetic will likely want to ensure numerical stability, i.e. that the rounding error is bounded at some reasonable level. In any program we will want to ensure some form of functional correctness, that the program computes what we expect it to. The specification can be given at varying levels of precision, from a plain english description to a precise mathematical specification that can be mechanically checked.

Armed with a specification, we have three choices: (1) we can verify (i.e. prove) that our implementation satisfies the specification, (2) validate (i.e. test) the implementation against a range of inputs, or (3) synthesize an implementation directly from the specification. Verification is the most difficult option as it inevitably requires the programmer or machine to provide internal invariants that are stronger (and often not obvious) than the top-level property we wish to prove, yet researchers have had much success in designing (semi) automatic systems for proving all of the aforementioned properties [1], [2], [3]. We can trade developer time for a less complete guarantee of correctness by validating our program against a range of inputs, driven by the specification. This can also be done automatically by treating the specification as an oracle, either in a black-box manner where we sample inputs exhaustively [4], [5] or randomly [6], [7], [8], run the program, and check that the corresponding outputs satisfy the specification, or in a white-box manner, where we observe the path taken by a given set of inputs and then choose a subsequent set in order to trigger a different path [9], [10], [11]. Finally, if the problem domain is restricted enough, we can avoid writing an implementation entirely and synthesize one directly from the specification [12] (or from a naive implementation [13]).

II. Reproducibility

Another concern for scientific software is reproducibility. A key part of the scientific method is reproducing prior experiments to increase our confidence in the results. However, the rapidly changing nature of software (and hardware) can make it very difficult to reproduce the exact experiment that led to the published results. We can improve reproducibility by treating each software artifact as a function of its inputs: the source code, external library dependencies, operating system, perhaps even the machine architecture. Viewed through this lens, it becomes natural to describe the process of constructing the software artifact in a (quasi) declarative language [14], [15], [16], [17]. The more precise the specification of the build process — for example we might specify an exact version requirement for inputs instead of a range, or even provide a cryptographic hash of the inputs to avoid ghost updates — the more confident we can be that future researchers will be able to reproduce our experiments.

III. Opportunities and Challenges

In this section we introduce technical and social challenges to adopting Software Engineering research in the Scientific Software community.

A. Technical

It is possible that research software developers are aware of the advances in software verification, but choose not to adopt the tools and techniques because they are simply too difficult to use. The most advanced and expressive verification engines require semi-manual proofs in a logical language, which can often result in proofs that are significantly longer than the program. On the other hand, many automatic verifiers produce errors that are difficult to decipher, containing either too little or too much contextual information to properly guide the user. Even testing can be difficult when it is unclear what the correct answer to a problem is, as is often the case in scientific simulations [18]. A more *human-centered* approach to verification and validation may be beneficial.

B. Social

An equally important challenge to address is the lack of collaboration between researchers in the Software Engineering and Scientific Software communities. Without open avenues of communication between the two disciplines, it is hard for the Software Engineers to keep abreast of the state-of-the-art in Scientific Software development and the challenges we might help address. Similarly, research software developers must go out of their way to stay up-to-date with the advances in Software Engineering and Programming Languages.

One mechanism which could be used to bridge the gap between the academic software engineering community and the teams developing research software would be to provide targeted graduate student internships. Currently, many graduate students in computer science spend short periods working for industry or at national laboratories, but these internships usually remain within the student's own discipline. We propose that software engineering students be encouraged to intern with research software development teams and become embedded in these groups to understand the culture, software stack, and development challenges and take these experiences back with them as they develop new tools and approaches. Such internships could be funded by agencies such as NSF already funding software development, potentially as supplements to existing awards.

One challenge of this approach will be to make sure that the students involved would be able to publish a creditable output from their internship which would be recognized as part of their career development — there is a danger that work carried out in this area would be viewed by peers as simply engineering rather than research.

Another challenge faced by the Software Engineering community is how to reward researchers for producing not just techniques, but real, *usable* software artifacts that employ those techniques. As in many fields, the reward structure for Software Engineering researchers heavily emphasizes publication of novel techniques, which is somewhat at odds with providing (and supporting!) a usable artifact. The research software community may be able to help in that regard by making the case to funding agencies, and hiring and tenure committies, that well-supported software engineering tools would aid their own work.

References

[1] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar, "The software model checker blast," *Int. J. Softw. Tools Technol. Trans.*, vol. 9, no. 5-6, pp. 505–525, 13 Sep. 2007.

[2] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata, "Extended static checking for java," in *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, ser. PLDI '02. New York, NY, USA: ACM, 2002, pp. 234–245.

[3] P. M. Rondon, M. Kawaguci, and R. Jhala, "Liquid types," in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '08. New York, NY, USA: ACM, 2008, pp. 159–169.

[4] D. Marinov and S. Khurshid, "TestEra: A novel framework for automated testing of java programs," in *Automated Software Engineering, 2001.(ASE 2001). Proceedings. 16th Annual International Conference on*. IEEE, 2001, pp. 22–31.

[5] C. Runciman, M. Naylor, and F. Lindblad, "Smallcheck and lazy smallcheck: Automatic exhaustive testing for small values," in *Proceedings of the First ACM SIGPLAN Symposium on Haskell*, ser. Haskell '08. New York, NY, USA: ACM, 2008, pp. 37–48.

[6] K. Claessen and J. Hughes, "QuickCheck: A lightweight tool for random testing of haskell programs," in *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '00. New York, NY, USA: ACM, 2000, pp. 268–279.

[7] C. Csallner and Y. Smaragdakis, "JCrasher: an automatic robustness tester for java," *Softw. Pract. Exp.*, vol. 34, no. 11, pp. 1025–1050, 1 Sep. 2004.

[8] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-Directed random test generation," in *29th International Conference on Software Engineering*, ser. ICSE '07, 2007, pp. 75–84.

[9] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed automated random testing," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05. New York, NY, USA: ACM, 2005, pp. 213–223.

[10] K. Sen, D. Marinov, and G. Agha, "CUTE: A concolic unit testing engine for C," in *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. ESEC/FSE-13. New York, NY, USA: ACM, 2005, pp. 263–272.

[11] N. Tillmann and J. de Halleux, "Pex–White box test generation for .NET," in *Tests and Proofs*, ser. Lecture Notes in Computer Science, B. Beckert and R. Hähnle, Eds. Springer Berlin Heidelberg, 2008, pp. 134–153.

[12] E. Darulova and V. Kuncak, "Sound compilation of reals," in *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL '14*. New York, New York, USA: ACM Press, 2014, pp. 235–248.

[13] P. Panchekha, A. Sanchez-Stern, J. R. Wilcox, and Z. Tatlock, "Automatically improving accuracy for floating point expressions," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2015. New York, NY, USA: ACM, 2015, pp. 1–11.

[14] "Chef — IT automation for speed and awesomeness — chef," https://www.chef.io/chef/, accessed: 2016-7-10.

[15] "Puppet - the shortest path to better software," https://puppet.com/, accessed: 2016-7-10.

[16] E. Dolstra, A. Löh, and N. Pierron, "NixOS: A purely functional linux distribution," *J. Funct. Programming*, vol. 20, no. Special Issue 5-6, pp. 577–615, 2010.

[17] G. Allen, F. Löffler, E. Schnetter, and E. L. Seidel, "Component specification in the cactus framework: The cactus configuration language," in *GRID*. IEEE, 2010, pp. 359–368.

[18] J. C. Carver, R. P. Kendall, S. E. Squires, and D. E. Post, "Software development environments for scientific and engineering software: A series of case studies," in *29th International Conference on Software Engineering (ICSE'07)*. ieeexplore.ieee.org, May 2007, pp. 550–559.