Software Engineering for Computational Science and Engineering

Jeffrey C. Carver | University of Alabama
Tom Epperly | Lawrence Livermore National Laboratory

Software engineering for computational science and engineering (CSE) is a topic of increasing relevance for many scientific and engineering domains. Historically, this topic has fallen into a void among many related domains. Traditionally, software engineering (SE) research has focused more on the business/IT domain than the scientific or engineering domain. SE researchers have often made the implicit assumption that any solutions they develop should likewise be applicable to the development of CSE software. Conversely, computational developers realize that their constraints often differ from those found in more traditional software development environments, and are wary to use existing SE methods. Moreover, lessons learned by computational developers generally aren't publishable as results, and therefore they often aren't shared.

The ongoing SE-CSE workshop series provides a venue to fill that void. The goal of this workshop series is to bring together SE researchers and computational developers/researchers to discuss problems, share experiences, and work towards solutions. In previous years, some discussion themes that emerged at the workshop included:

1521-9615/14/$31.00 © 2014 IEEE Copublished by the IEEE CS and the AIP

> The adoption of SE practices for CSE is complicated by differing rewards systems.

- the unique characteristics of computational software that affect software development choices;
- the appropriate context dimensions to describe computational software;
- the major software quality goals for computational software;
- crossing the communication chasm between traditional software engineering and computational software;
- effectively involving scientists/engineers in software development and training;
- measuring the impact of SE on scientific/engineering productivity;
- SE tools and methods needed by the computational developers' community; and
- how to effectively test computational software.[1–4]

The SE-CSE workshop described in this special issue occurred during the 2013 International Conference on Software Engineering.[5] The workshop had two phases—it began with presentations by the authors of accepted papers. Extended versions of some of those papers are included in this special issue. The second phase included breakout discussions based upon topics raised during the morning presentations. Here, we provide a brief overview of those discussions. Full notes of the discussion can be found on the workshop website (http://secse13.cs.ua.edu).

Overview of Workshop Discussion

The workshop discussions focused on six key areas: adoption, Agile programming, the culture clash, debugging, education, and verification and validation. These themes reflected the content of the accepted papers' presentations and covered the main challenges and opportunities for SE-CSE. In the following, we briefly detail findings in each area.

Adoption of SE Practices by CSE Developers

The adoption of SE practices for CSE is complicated by differing rewards systems. Scientists are generally rewarded for having numerous influential scientific publications rather than developing high-quality software. In such a case, software is a means to an ends rather than an end in itself. The key to adoption is to focus on a "back to basics" approach to SE rather than promoting cutting-edge SE techniques. Thus, we should focus on the aspects of SE with the highest return on investment, such as source control, software carpentry tools, and lightweight processes. Similarly, we must advocate a cultural shift towards the importance of reproducible results, a big component in other scientific fields that's often neglected in CSE research. A requirement for scientists and engineers to publish their software in a form that other researchers can utilize to reproduce the results would introduce cultural pressure to produce higher-quality code.

Agile Software Development Philosophy

A number of SE and CSE professionals sense that the Agile software development philosophy is a good fit for CSE projects. Adherents to an Agile software development approach use an iterative process to design and develop the software. This process mimics the process of scientific discovery, where the research target may change as knowledge is gained through early results from the software. The Agile approach focuses on producing software with complete functionality and minimal, but adequate, documentation (as opposed to focusing on extensive documentation). As a result, developers produce software in small chunks and "document" requirements as test cases. There are still challenges with developing test cases when there's no oracle for correct answers—particularly for integrated modeling. However, the Agile approach is still seen as a fairly lightweight, flexible process that can adapt to community-driven priorities.

Culture Clash

One approach to promoting the adoption of SE practices for CSE development is to use multidisciplinary teams of software engineers and scientists. Often, this approach encounters difficulties due to the clash of cultures between these groups of people. Scientists generally are indifferent to SE, because they believe that they lack the time and resources to implement such practices. Their priority is on producing scientific publications. For software engineers, software quality is important for providing confidence in the scientific findings produced by the software. They view the use of SE practices as a long-term investment that will ultimately lead to increased scientific productivity. In many cases, the funding is controlled by the scientists, with the software engineers lacking

Debugging large systems is like finding the needle in the haystack.

the standing or resources required to get everyone to adopt a consistent set of SE practices.

Debugging

Debugging CSE software on supercomputers is a particularly difficult challenge. Many bugs only manifest themselves at scales that even surpass the capabilities of today's most advanced debuggers. Nondeterministic bugs that are exceedingly rare at small scales become quite common at larger scales. Due to their random nature, these bugs are notoriously difficult to isolate and debug. For supercomputers involving millions of computing cores, hardware faults will become increasingly common and may be indistinguishable from software faults. Debugging large systems is like finding the needle in the haystack. Statistical analysis can be used to automatically identify normal and abnormal behavior, with developers focusing their attention on the abnormal cases.

Education

Everyone acknowledges there's an education gap in the SE-CSE community. CSE professionals typically don't take any SE courses in their academic programs and may not have taken a single programming course from a computer science department. They often pick up whatever they know from their thesis advisor and their research group. At this point, it's also difficult to know what to teach, because the SE-CSE community hasn't yet completely discovered which SE approaches are most appropriate for the development of CSE software and how those approaches should be tailored. The Software Carpentry project (http://software-carpentry.org), founded by Greg Wilson, seems like a good approach to instill some basic principles for CSE developers.

Verification and Validation

The level of verification and validation (V&V) varies a great deal in the CSE community. Convincing CSE team members to perform regular testing can be challenging, because it requires time and effort. Some teams have high standards of testing and code coverage, while others have practically none. For cutting-edge research, it's often difficult to test systems, because there's no prior work or experimental data to compare against. Even when data is available, many CSE applications aren't able to produce bit-for-bit reproducible results, due to inherent nondeterminism in parallel algorithms. Testing might require access to a supercomputer; but supercomputing centers don't like to provide cycles for testing.

In This Issue

Now that we've outlined some of the issues in the field, let's look at the articles for this special issue. Each article has a unique take on the topic of SE-CSE and provides a novel approach or solution. There are six articles, with each being an extension of a paper presented at the workshop. For each paper, the authors were invited to revise to include a significant amount of new content beyond the workshop paper.

In the first article, "Streamlining Development of a Multimillion-Line Chemistry Code" Robin Betz and Ross Walker describe the use of a standard software engineering practice, *continuous integration*, to support the development of the molecular dynamics code, AMBER. The team used a customized version of Cruise Control to support continuous integration. To implement this practice, the team followed five key practices:

1. Maintain a single source repository.
2. Automate the build and tests.
3. Simplify deployment and executable distribution.
4. Every push should result in a build on the integration machine.
5. Development should be communicative and collaborative.

The authors' work describes the implementation details, barriers faced, and methods used to address those barriers.

Next, Stan Ahalt and his colleagues describe the application of open source mechanisms and software engineering to research about water science in their article, "Water Science Software Institute: Agile and Open Source Scientific Software Development." After providing an overview of the computational challenges faced by the water research community and a brief introduction to the Water Sciences Institute, the article describes the Open Community Engagement Model. The piece provides an example of how this model works in practice. The evaluation of this exercise showed positive results.

In the third article, “Leveraging Expertise to Support Scientific Software Process Improvement Decisions,” Erika Mesh and her colleagues describe the development of the Scientific Software Process Improvement Framework (SciSPIF). This framework is designed to provide a flexible approach for scientific developers to make decisions that are relevant to their own projects while still having access to a library of standard software engineering practices. The article details the use of a grounded theory approach to analyze the literature and to conduct a series of case studies to provide information for initially populating the framework.

Fourth, in “Building CLiiME via Test-Driven Development: A Case Study,” Aziz Nanthaamornphong and his colleagues describe a case study of using test-driven development (TDD) to support the development of a computational science software package. Using TDD enabled the development team to create a software package that’s extensible by other developers. The article provides a number of lessons learned, based on the authors’ experience with applying TDD for the first time. The piece also discusses some of the benefits that Agile methods in general—and TDD specifically—can provide to computational science projects.

Alan Humphrey and his colleagues address the problem of latent defects in large, high-performance computing scientific software in the fifth article, “Systematic Debugging Methods for Large-Scale HPC Computational Frameworks.” The authors introduce a new debugging approach based on coalesced stack trace graphs that supports a systematic debugging process. The article then illustrates the new approach with a case study in which the authors identified and fixed a real defect in the Unitah Computational Framework.

Finally, in “A Case Study on a Quality Assurance Process for a Scientific Framework,” Hanna Remmel and her colleagues describe the results of a case study to evaluate the feasibility and developer acceptance of two quality assurance practices. First, the case study showed that developers found *variability modeling*—a technique to assist developers in systematically creating system tests—useful and easy to learn, and they reported on their intent to use variability modeling in their work. Second, the case study showed that the technique of *desk checking* also was useful and easy to learn.

We hope you find these articles beneficial in understanding the complexity of issues for SE-CSE. We also hope this special issue creates an awareness of the work being done in the field, while also bridging the gap between traditional software engineering and computational software.

References

1. J.C. Carver, “First International Workshop on Software Engineering for Computational Science & Engineering,” *Computing in Science & Eng.*, vol. 11, no. 2, 2009, pp. 7–11.
2. J.C. Carver, “Report: The Second International Workshop on Software Engineering for CSE,” *Computing in Science & Eng.*, vol. 11, no. 6, pp. 14-19, November/December, 2009.
3. J.C. Carver, “Third Int’l Workshop Software Engineering for Computational Science and Engineering,” *Procedia Computer Science*, vol. 1, no. 1, 2010, pp. 1485–1486.
4. J.C. Carver, “Software Engineering for Computational Science and Engineering,” *Computing in Science & Eng.*, vol. 14, no. 2, 2012, pp. 8–11.
5. J.C. Carver et al., “5th International Workshop on Software Engineering for Computational Science and Engineering,” *Proc. 2013 Int’l Conf. Software Eng.*, 2013, pp. 1547–1548.

Jeffrey C. Carver is an associate professor in the Department of Computer Science at the University of Alabama. His research interests include empirical software engineering, software quality, software engineering for computational science and engineering, software architecture, human factors in software engineering, and software process improvement. Carver has a PhD in computer science from the University of Maryland. He’s a senior member of the IEEE Computer Society and the ACM. Contact him at carver@cs.ua.edu.

Tom Epperly is the Computer Science Group Leader in the Center for Applied Scientific Computing at Lawrence Livermore National Laboratory. He holds a Ph.D. in Chemical Engineering from the University of Wisconsin-Madison, and his research interests involve integration technology and frameworks for computational science & engineering. He can be reached at epperly2@llnl.gov.