Socio-technical logics of correctness in the scientific software development ecosystem

Position Paper for Workshop on The Changing Dynamics of Scientific Collaboration at CSCW 2010

James Howison
School of Computer Science
Carnegie Mellon University
jhowison@cs.cmu.edu

James D Herbsleb
School of Computer Science
Carnegie Mellon University
jdh@cs.cmu.edu

INTRODUCTION

Science increasingly depends on software. From configuration and control of instruments, to statistical analysis, simulation and visualization, virtually every workflow that generates scientific results involves software.[1] In practice, scientific collaboration in a growing number of disciplines means drawing together different software artifacts produced in different ways, by different people, to build an ensemble artifact that does scientific work and, ultimately, provides reasons to believe scientific conclusions.

In this position paper we present an understanding of the scientific software development ecosystem that is emerging from our interviews of working scientists who develop software in the course of their science. First we describe the types of software and software development being undertaken. We then focus in on three logics of correctness that have emerged from our interviews. We demonstrate that these logics are closely linked to the social circumstances of the software's production and use and the type of software; these are socio-technical logics. We conclude by examining the implications of this understanding for shaping policies designed to maximize the return on the substantial public investments in scientific software production.

The interviews

We have just begun a three year NSF-funded project designed to improve our understanding of the scientific software ecosystem.[2] As a foundation for this study we have interviewed 16 scientists from 5 different scientific collaborations associated with the Open Science Grid. Three collaborations are working in physics (one particle accelerator and two specialized observatories), one works in structural biology and the fifth assists scientists from a variety of domains in accessing Grid computational resources. 14 of the informants describe themselves as scientists, while 2 identify as professional software developers. Their roles include software coordinators for collaborations, infrastructure developers, scientific software consultants and scientists undertaking front-line analysis aimed at publishable results. The interviews are semi-structured with a broad focus on software production and sharing.

[1]This is true both of the software that facilitates collaboration (i.e. communication systems) and of the software that is the subject and outcome of scientific collaboration, which is our focus in this paper.

[2]http://www.nsf.gov/awardsearch/showAward.do?AwardNumber=0943168

Types of scientific software

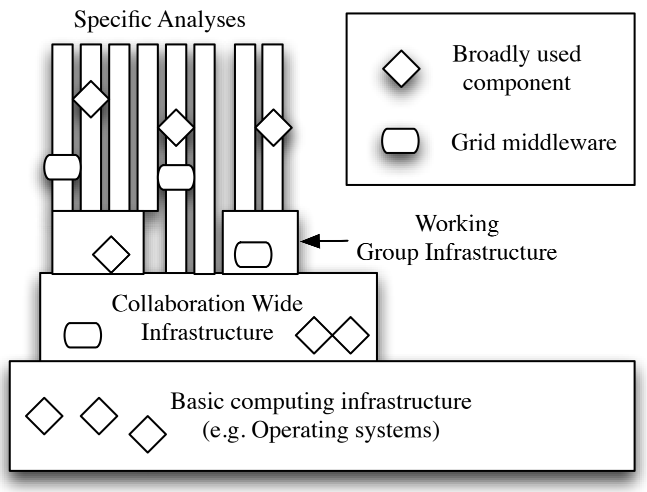
Specific Analyses
Broadly used component
Grid middleware
Working Group Infrastructure
Collaboration Wide Infrastructure
Basic computing infrastructure (e.g. Operating systems)

Fig. 1. A sketch of the scientific software ecosystem.

Figure 1 depicts five different types of software that we find in our interviews. Working from the top to the bottom, we begin at the end, with analyses designed to support particular scientific publications. Beneath these two kinds of infrastructure: collaboration wide and working group infrastructure, differentiated by the number of analyses which rely on them. Underlying all of this we find the broadest kind of shared computing infrastructure: operating systems, such as Linux, Windows and Mac OS X. Literally scattered throughout these different layers we find two additional kinds of software: Grid related middleware and components from outside a specific collaboration, usually open-sourced. Some are more specific to particular fields (such as ROOT in high-energy physics or the

R statistics package), while others are potentially more general (such as high-throughput input/output frameworks). These different types of software are orchestrated into particular ensembles to do scientific work.

The software of the collaborations we have investigated are architecturally organized around a filtering task. That is their analysis chains begin with massive amounts of data and they apply sequential theoretically informed filters, including the results of simulations, to extract evidence of events of scientific interest. Thus there is a temporal ordering to a particular analysis, and steps which prove broadly useful shift into lower layers of the architecture (from analyses to working group infrastructure to collaboration infrastructure). A key component of collaboration infrastructure is therefore the software that directs and manages the step-by-step application of filters and the recording of data and the results of each step. Components such as Grid computing middleware or broadly used components are applied where ever they are needed, ending up throughout the full analysis chain.

Software is mentioned in scientific publications in two ways. The first is citations out to "well-known" scientific analysis packages, falling into our "broadly used component" category. The second, less common, is by reference to a customized analysis workflow corresponding to the specific ensemble that backs the science discussed in the paper. This second kind of reference is often by reference to an internal source code repository label (such as a cvs/svn/git "tag"), although it is clear that while such provenance information is an aim of the scientists it does not always occur.

Each of these components are developed in quite different ways, by different numbers of people in collaboration. Analyses are often the work of a single grad student or post-doc as they work towards specific publication goals. In the words of one informant they are focused on "producing plots" (where plots are the primary format taken as evidence for scientifically interesting events). By contrast collaboration wide infrastructure is typically planned from the start of the collaboration, has many contributors (roughly speaking a core of $\tilde{5}$ whose primary focus is software development and perhaps up to 50 irregular contributors, depending on the size of the collaboration).

Those that identify as professional software developers are relatively rare. Many of the scientists we interviewed indicated that they had difficulty working with non-scientist software developers; indeed the two interviewees that identified as professional software developers had domain specific backgrounds (a Physics PhD in one case and a Physics undergrad from a top institution in the other) and were described by others as physicists.

Grid infrastructure is distinguished because it is built outside scientific collaborations, usually with core contributors funded on infrastructural grant money targeted in large part to software development.

The broadly used components vary substantially in the organization of their production. We found use of commercial components only in one collaboration, structural biology and no use at all in the physics collaborations we studied. Preliminary inspection of the open source projects reveals that they vary hugely in size, from R with 100s of contributors, to input/output libraries with less than 5 contributors, to widely used components irregularly supported by their original, single, authors.

A final distinction between components is how often they are expected to be used: analyses can be useful even if they are only used once, infrastructure is justified by the frequency of its use inside the collaboration, and "broadly used" components are distinguished by their use in multiple, different scientific (or more general) contexts.

THREE LOGICS OF SOFTWARE CORRECTNESS

The theme of software correctness and its entanglement with the organization of software production and assemblage is one theme that has emerged from these interviews. Correctness is a core value of science. As computation and therefore software plays an increasingly central role in science the scientific enterprise comes to rely more heavily on the logic of software correctness. Yet, one scientist we interviewed argued that techniques to demonstrate correctness in computational science are different to those used prior to the computational shift, such as formal proofs in equation systems, but such skills are not yet considered a core part of scientific education.

In our interviews we asked how the informant knew that their software was performing correctly in the course of their science. Three broad classes of answers emerged, here they are paraphrased for brevity:

1) This software is used all the time by everyone; it has been validated and checked by others; there are processes in place to check its correctness. (External/Exposure)
2) I wrote it all myself; and have had it reviewed by close peers; those that understand it best have checked it. (Personal/Local)
3) Mutually supportive results are produced by multiple separate implementations of this software, or alternative theoretically informed methods. (Results Validation)

The first logic shifts the burden of correctness to an external socio-technical arrangement. It draws on two related sources: authority (the author/s are really good at their work) and exposure ("many eyes" have checked this code because it is used all the time). The sum of this is the expectation that the code will perform in a wide variety of cases and usage environments. An example of this is the ROOT analysis software project in high-energy physics, or even the file-system operations of the Linux kernel. This logic lines up cleanly with the classic argument for the correctness of open source software and can be seen clearly in papers where citations are made to existing packages.

The second logic is almost the exact inverse of the first: code is to be trusted because it has been written from scratch for this exact purpose, there is no un-inspected external code in which errors could be hiding. In the high-energy physics collaborations we spoke to it is taken as standard practice to have close colleagues review code used for results in publications, usually by colleagues within the same analysis sub-group. These reviewers are considered close enough to the science to understand the system in detail and thereby judge its adequacy. This logic is rarely, if ever, referenced in a paper, where discussion of custom software built for the analysis is limited to, at most, a source control tag, but discussion of the circumstances of its production is excluded.

The third logic argues that the correctness of the software is demonstrated because the results it produces are supported in multiple independent ways, including the results of other unrelated software implementations (other labs or collaboration sub-groups or even other students) and theoretic expectations elaborated in other ways (for example through systems of mathematical equations or laborious manual inspection or manipulation). The specific piece of software is embedded in a matrix of theory-laden artifacts which together are responsible for scientific correctness. This logic is strong in actual publications.

In practice, of course, these multiple logics are combined: (again to paraphrase) "I deeply understand and checked the parts I wrote, many others checked the parts I didn't write and the entire ensemble produces results mutually supported by previous work and theory". In particular the second and third logics are often combined when considering the correctness of code written for specific analyses.

Each logic has assumptions: Each logic, however, has key assumptions. The first relies heavily on there actually having been multiple uses and users of a piece of software. In scientific work this is hard to take for granted, since requirements can be so specialized. In fact this is a key difference between scientific software and other types of software: much of it is written with the intention that it will only be run a very small number of times, perhaps just once to get the intended scientific result. Certainly the fact that the software is open source is no guarantee by itself at all that the software is widely used. While some open science software is heavily used and designed to perform in a wide set of circumstances, other software labeled open source may only have been used once or twice in very specific ways.

The second logic places significant onus on the skills of the analysis author and the surrounding reviewers. Our informants have been unanimous in their experience that actual code-reading reviews are never done while a paper is being reviewed by a journal, for example. Rather reviews happen very locally, often just one level above the original author within a collaboration, even when the rest of the paper undergoes further rounds of internal and external review. This practice maintains the deep comprehension of the issue, but raises concerns about shared blind-spots, training or groupthink.

The third logic relies on separate implementations not making similar errors. More importantly it relies on previous work or alternative representations being close enough in aim, and results being in a comparable format, so that agreement can be assessed at a fine enough level of granularity.

CONCLUSION

In an effort to maximize the returns on public investment in science a series of policies have been adopted by scientific funding agencies and journals, each driving towards increased openness. Well known examples include the NIH's policy requiring the deposition of publications in open-access repositories, such as PubMed, and data in databases such as the NCBI's Gene Expression Omnibus (GEO). Agencies are considering similar policies with regard to software production, including requirements that all publicly funded code be "open-sourced" with the expectation that this will reduce redundancy, promote scientific openness and thus simultaneously advance science while saving money.

The position in this paper suggests that there is a risk that broad, sweeping policies for software production in science will result in over-emphasizing specific logics, even when its assumptions are not met.

For example, one concern with a blanket policy of open sourcing is that code that is, in fact, not widely used and therefore wouldn't otherwise have become an open source project will be labeled as such. In that way code which had been judged based on a combination of the second and third logics, would then be at risk of being judged based on the first logic. Programmers might be less likely to deeply inspect such open source code and scarce review resources would be less likely to concentrate on such code.

Secondly, it is worth considering that each logic relies on a set of supporting institutional arrangements. For example, a logic of exposure relies on there being enough shared interest to draw together a large enough body of developers and users to run a functional open source project, as well as the collaboration and "gentle persuasion" skills required in such projects. Conversely, the second logic relies on there being adequate programming skills and time available to write important software from scratch. Such skills and resources are not likely to be substitutable.

Our interviews confirm that much analysis software is written by early-stage PhD students in scientific disciplines, who struggle to learn programming skills, often from scratch. While this situation is far from perfect it seems to match more closely with the requirements of the second logic, than to expect these students to become contributors to well-organized open source projects, exposing their early programming efforts to intense scrutiny while learning. Somewhat counter-intuitively, at least to us, we therefore suggest that little used code designed for specific analyses should be archived and re-written when required, rather than 'dumped' as open-source scientific software, retaining an appropriate emphasis on the logics of close understanding and multiple re-implementation.

The scientific software production ecosystem is complex; we are only beginning to build a nuanced view and plan to bolster our understandings by explicit analyses of individual scientific workflows, mapping out all the software used and tracing its origins.