Bridging the Chasm: A Survey of Software Engineering Practice in Scientific Programming

TIM STORER, University of Glasgow

The use of software is pervasive in all fields of science. Associated software development efforts may be very large, long lived, and complex, requiring the commitment of significant resources. However, several authors have argued that the "gap" or "chasm" between software engineering and scientific programming is a serious risk to the production of reliable scientific results, as demonstrated in a number of case studies. This article reviews the research that addresses the gap, exploring how both software engineering and research practice may need to evolve to accommodate the use of software in science.

CCS Concepts: • **Software and its engineering** → **Software development methods**; **Software development techniques**; **Software verification and validation**; **Process validation**; *Agile software development*; *Documentation*;

Additional Key Words and Phrases: Software engineering, scientific programming

ACM Reference format:
Tim Storer. 2017. Bridging the Chasm: A Survey of Software Engineering Practice in Scientific Programming. *ACM Comput. Surv.* 50, 4, Article 47 (August 2017), 32 pages.
https://doi.org/10.1145/3084225

1 INTRODUCTION

Software is now an indispensable tool for the conduct of scientific research. Software applications may be used to gather, synthesise, manage, process, analyse, and/or present enormous quantities of data. Several example cases illustrate the diverse use of software in scientific research:

- The Large Hadron Collider facility at CERN is supported by a software development effort consisting of more than five million lines of code, comparable to a small operating system [46, 129]. The collection and analysis of the terabytes of data generated by experiments run on the LHC would be impractical without this infrastructure.
- The 2013 Nobel prize for Chemistry prize was awarded jointly to Karplus, Levitt and Warshel for "for the development of multi-scale models for complex chemical systems" [152]. These models are computer simulations of chemical processes that are either too complex or too costly (or both) to replicate in the physical world for all experiments.
- Software is essential for making long-term predictions about changes to the climate as a result of both natural and anthropogenic factors [49, 171]. As Edwards [50] notes, software is used to integrate a wide variety of sources of historical temperature data to produce a single homogeneous global gridded temperature record. In addition, predictions about

Author's address: T. Storer, School of Computing Science, University of Glasgow, Glasgow, G12 8QQ, United Kingdom; email: timothy.storer@glasgow.ac.uk.

https://doi.org/10.1145/3084225

climate change depend on coupled general circulation models to process this data and make predictions about future temperature variations.

Reviewing these examples, it can be seen that the use of software brings a variety of benefits to scientific research. Large quantities of data gathered from observations can be processed accurately and quickly. Derived data sets can be generated from a range of sources and synthesised in a consistent manner. Very large data sets can be curated using software tools, allowing rapid and consistent dissemination. Analyses employing novel visualisations or statistical techniques can be used to identify new trends and opportunities for future research. Finally, software permits the simulation of physical phenomena for analysis, where collection of primary data is unfeasible.

Despite the acknowledged benefits, growing dependence on software raises questions as to its appropriate role as a tool within the scientific method. Empirical science is characterised by Popper [135] as the proposition of hypotheses concerning some aspect of the world. An experiment is then developed to test this hypothesis. If the results of the experiment contradict the hypothesis, then the hypothesis is rejected and alternative explanations are sought. Hypotheses that are confirmed by repeated experimentation gradually gain acceptance within the scientific community. These hypotheses may eventually be referred to as *theories*. Empirical science therefore progresses through the falsification of invalid hypotheses, rather than the confirmation of valid ones. Popper [135] identified several characteristics of a scientific method that would support this process:

- Scientific theories should be *falsifiable* through experimental contradiction by example. It must be possible to use the hypothesis to make predictions about the result of an experiment that, when contradicted, demonstrate that the hypothesis was incorrect.
- Experiments should be *repeatable*. In practice, repeatability is achieved by thorough documentation of the procedure followed and the tools employed in the experiment, enabling an experimenter to demonstrate the scientific result on demand.
- The results of an experiment should be *reproducible* by an independent experimenter. The individual should be able to follow the experimental procedures employed, using equivalent tools to recreate the same results. If independent experimenters fail to reproduce the results then this should cast doubt on the validity of the original hypothesis.
- The limitations as to the validity of the results of an experiment and any consequent conclusions due to the method employed are made explicit in an experimental report.

The use of software in scientific processes poses challenges to all these requirements. Brooks [20] argued that software is “essentially” complex, intangible, and volatile in nature, leading to many of the failures that beset software projects in all domains [62]. In scientific research, this complexity has manifested itself as difficulties in repetition of experimental methods [161], obstacles to reproducing results [112], and the identification of software defects that invalidate results [67, 68, 116].

Software engineering emerged as a discipline in the 1970s, as a deliberate response to the perceived software “crisis” of the time [120]. The crisis referred to the increasing number of software projects that were perceived as failures, due to over-spent budgets, delayed schedules, incorrect functionality, unacceptable defects, or outright project cancellation [62]. These problems were attributed to the ever-increasing scale, complexity, and consequent volatility of software systems as the capabilities of computers and the number of programmers required to collaborate on a project increased [107].

A variety of software development processes, practices, and tools have been developed since the 1960s to provide for greater control, predictability, and quality in software development efforts. The recognition that software development is intrinsically complex, evolutionary, and concurrent is embodied in the proliferation of agile development disciplines, such as Scrum [154] and Extreme [12]. Recent surveys of software projects have suggested that these developments may finally be providing a remedy [51].

Despite this progress in other domains, the adoption of available software engineering practices and tools in scientific programming, the practice of developing software to support scientific work, remains patchy [179]. Kelly [95] referred to the disconnection between much of software engineering practice and scientific programming as a "chasm." More recently, Faulk et al. [54] raised concerns about productivity in scientific programming that could have been made in the early days of the software crisis. There is a need to improve the transfer of existing practices and tools from other applications of software engineering to scientific programming. In addition, due to the specialised nature of scientific programming, there is a need for research to specifically develop methods and tools that are tailored to the domain.

This article contributes to this research by reviewing the literature that covers the interaction between software engineering and scientific programming. The article summarises the problems encountered when employing software in scientific research; reviews the current state of the art in scientific programming practice as reported in case studies; and ongoing research efforts to better understand and support the needs of researchers working on scientific programming projects. Several sources from the literature were reviewed as starting points for this survey. Specifically, there has been a recent series of workshops examining the relationship between software engineering and science [24–27, 155]. Carver [23] has provided a summary report of the 2009 workshop. In addition, Wilson and Lumsdaine [189] guest edited an edition of *Computing in Science and Engineering* on the role of software engineering in scientific programming. Similarly, Segal and Morris [160] guest edited an edition of IEEE Software, presented the challenges faced by the scientific programming community to software engineers and computing scientists; and Hey et al. [74] edited a collection of articles investigating the growing phenomenon and challenges of data intensive science. Related sources cited by works at these venues that also addressed the challenge of applying software engineering to scientific programming were then also retrieved and reviewed. These articles were supplemented with a search of Google Scholar using combinations of the term "scientific programming," "scientific computing," "computational science," and "software engineering."

The article is structured as follows. Section 2 reviews research over more than three decades that has reported on the problems encountered in scientific programming. The section highlights the conflict between the intrinsically evolutionary and complex nature of software and the demand for stable, documented scientific experiments. Section 3 reviews case studies of scientific programming in practice, identifying where existing practices have been adopted and adapted from software engineering and where ongoing research challenges persist. Sections 4 and 5 reviews the specific challenges in the quality assurance and long-term maintenance of scientific software, as well as advances in addressing these challenges. Section 6 briefly surveys work on data quality as it relates to scientific programming. Finally, Section 7 reviews the material presented and discusses opportunities to revisit both software engineering and scientific practices in light of the growing dependence of science on software.

2 SOFTWARE, REPRODUCIBILITY, AND SCIENTIFIC COMPLEXITY

There are numerous examples of the use of software thwarting attempts at repetition or reproduction of scientific results in a wide variety of disciplines and a sample of more illustrative recent cases are outlined below. The 2010 controversy over software, data, and emails leaked from the Climatic Research Unit at the University of East Anglia in the United Kingdom illustrated many of the challenges [81, 82] in the context of climate science. Software used to derive published results was not routinely released for inspection and verification. When the code was released, several defects were discovered and reported [82]. That code contains defects is perhaps not surprising; however, as Shackley et al. [161] has observed, the sheer complexity of the software used in climate science makes the interaction between software defects and scientific results difficult to interpret.

Similarly, Herndon et al. [71] reported a number of programming defects in a widely cited analysis [144] of public debt to GDP ratios across 20 advanced economies following World War II. Herndon et al. identified errors in the implementation of a spreadsheet that formed the basis for Reinhart and Rogoff's analysis. The error caused the average growth of high debt/GDP ratios to be understated and similarly caused low debt/GDP growth ratios to be overstated. Combined with other problems with the analysis, Herndon et al. [71] refuted Reinhart and Rogoff [144]'s conclusions.

Sanders and Kelly [153] reported that one of her interviewees had found that their application produced significantly different results depending on the hardware platform it was executed on. Similarly, Dubey et al. [47] noted (as an aside) that some compiler optimisation options prevented the production of reliable results during the development of the second release of the FLASH code.

Most starkly, Miller [116] reported that several highly influential articles had to be retracted and more than five years of research work lost as a result of a trivial programming error in a previous researcher's work [116]. The case illustrates the risk of relying on research dependent on complex bespoke software, as the mistake appears to have originated in another lab. Miller also notes the serious implications of such mistakes. One of the retracted articles was reported to be highly cited and the basis for extensive further work. Scientists who had prepared articles contradicting the original research reportedly experienced difficulty getting their own results published.

Several researchers have attempted to systematically assess the feasibility of repetition of software-based experiments [36, 112, 172]. Stodden et al. [172] conducted an empirical study of data and code publication policies adopted by journals. The work found that only a minority of journals maintained a data or code publication policy for peer reviewed research, although (as noted in the study) the work does not account for possible confounding factors, such as the prevalence of computational research in a particular field.

McCullough et al. [112] described an analysis of a code repository maintained by a peer reviewed journal of applied economics. The analysis showed that 73% of the articles reviewed did not comply with the journal's policy requiring code and data submission. Further, many of those submissions that were available contained code that could not be compiled or executed, or was incompatible with the supplied data format. Anderson et al. [5] also reviewed the availability of program source codes and data sets to support reproducible results in economics research. The research concluded that progress towards reproducibility was likely to be minimal without the stricter imposition of mandatory archiving.

Colberg et al. [36] reported a more recent effort to measure the repeatability of results in computing science. The researchers examined research published in recent computing science conferences and attempted to recover, compile and run associated source code. The researchers found that only around a quarter of the research work could be conveniently reproduced in this manner. The researchers also reported a range of obstacles to obtaining source code associated with published research; and found that public funding of the research had no effect on availability.

Several researchers have linked difficulties in achieving repeatability with the management of floating point arithmetic on finite computing hardware. For example, the problems encountered by Hatton [67 ,68] were primarily caused by accumulated losses of precision in floating point calculations due to programming errors that introduced systemic defects. The work reported on defects in software used to determine the placement of oil wells based on computational models. The defects were so significant the authors concluded that the placement method was essentially randomised. Separately, Edwards [50, p. 177] reported that the process of porting global circulation model software to different hardware and software platforms often produced different results due to the accumulation of different round-off operations. Roy and Oberkampf [151] note that when such defects are the result of programming errors rather than necessary approximations due to floating point operations, the use of techniques such as uncertainty quantification can be extremely difficult to apply.

Shaon et al. [162] reported on two case studies of scientific software preservation and maintenance activities. Both studies highlighted the view that long-term software preservation is prohibitively expensive. Surveys of software projects in other domains (Krogstie et al. [102], for example) substantiate this view, suggesting that maintenance accounts for well over half of the lifetime cost of software. These costs arise from curation activities, such as maintaining support for legacy platforms, rather than from the relatively trivial costs of storage. In addition, the peer reviewed publication process actively discourages the active maintenance of software, since the discovery of defects can lead to the costly retraction of results.

Sanders and Kelly [153] also investigated the use of software in scientific domains where the source codes were not always available (such as commercial products). Sanders and Kelly noted that the use of black box software is a significant risk for scientists as they have no effective means of determining if the implementation is correct with respect to a documented theoretical model.

In summary, several challenges can be identified in the literature:

- Experimental software is unavailable or cannot be redeveloped, because sufficient information about specification and design at the time when the experiment was conducted is unavailable.
- The software is available, but the original result cannot be reproduced so that the generated scientific result can be validated. This may be a result of the subsequent identification and removal of defects from the software code between the time of the experiment and the time of publication or due to the execution of the code on different hardware platforms.
- The software is available, but contains defects that, when corrected, may contradict the published results. In this situation, a scientist would in principle be required to retract their article from the peer review process and repeat their analysis.

Fundamentally, these challenges concern the conflict between the documentation and presentation of a stable scientific method and the intangible and volatile nature of software. Software can be represented in a variety of forms, each providing a partial view of the software's structure and/or behaviour. It is unclear how much of this documentation, or what forms should be included as part of a description of an experimental method, given the traditional constraints on article length. In addition, any given software artefact is constantly being altered as new features are added, existing features are enhanced, architecture is improved and defects identified and removed. Any dependencies, including hardware and software platform, libraries and compilers, will also continue to evolve under the same pressures.

3 CASE STUDIES OF SOFTWARE PROCESSES

There is a long history of case studies exploring the relationship between scientific research and software development practices. Reviews of case studies provides an overview of the state of the field and the proliferation of particular methods, tools, and best practices. Each case study reviewed here reported on one or more scientific software development effort of the following types:

- Significant collaborative infrastructure efforts that receive contributions from a mixture of scientific and "professional" software developers, for example, Heroux et al. [72], Matthews et al. [110]. These projects employ a core of professional software developers who act as "gatekeepers" for key infrastructure components. Parts of the software that are considered critical to the scientific domain will also be managed by domain experts. Scientist-developers contribute code developments as part of a managed process.

- Professional software developers working on scientific software on behalf of domain expert researchers. These professional developers may either be members of the same research group, but with a distinct role in software development, or working to contract, for example, Segal [156].
- Single or small groups of scientists developing analytical software for themselves. These scientist-developers are likely to be self-taught in programming and have little exposure to software engineering principles or practices, for example, Chilana et al. [31] and Hannay et al. [66].

This section provides an overview of these published case studies to date and is organised into themes that emerged from the literature: a review of the general case studies of scientific programming in practice, experiments with agile methods, the impact of project team evolution, and the development of best practice guidance.

3.1 Scientific Programming in Practice

Heroux et al. [72], Matthews et al. [110], and Dubey et al. [47] all reported case studies of long-term scientific software development efforts from the perspective of the project teams. Heroux et al. [72] described the software engineering practices within the Trilinos project. The report emphasised the importance of software quality, maintainability, modularity. A strategy of minimising the effort associated with these goals is pursued through automation of tasks wherever possible. For example, a standard package template is available to minimise the effort associated with sub-project initiation. Matthews et al. [110] described the software development practices at the United Kingdom's Meteorological Office. The study focused on the group's configuration management practices, and the adoption of customised tool support (FCM) for change management and compilation. Dubey et al. [47] reported on the history of experiences of developing code in the FLASH project, a "multi-physics simulation code" that is also a merger of several prior code bases. These projects are representative of many of the challenges faced by large-scale scientific software infrastructure collaborations, including: compromising between feature demands and quality control; code ownership and management during evolution; data organisation and curation; and quality assurance of heterogeneous components. In particular, the studies found that the groups developed customised tool support because the existing tools did not meet their needs.

Other researchers have acted as external observers of scientific software projects. Hannay et al. [66] investigated how scientists develop software, using an online survey. The research concluded that, despite software being crucial to scientific practice, the dissemination of knowledge about the use and development of software occurred internally within disciplines. Generally, scientists do not learn about software engineering techniques from software engineers. The results presented by Hannay et al. could be caused by a number of factors, including poor communication of the effect of poor quality software on science, or the unavailability of appropriate software tools and methods for scientific programming. Nguyen-Hoan et al. [124] conducted a similar survey of participants in scientific programming projects, with many similar findings to Hannay et al. [66].

Chilana et al. [31] compared the software development practices of computing science and molecular biology professionals engaged in the development of software for bioinformatics in both research and production. The research focused in particular on information gathering practices during software development. Similar to Hannay et al. [66], the research found that developers from all backgrounds depended on a decentralised and informal approach to information gathering to solve problems.

Morris [118] reported his experiences of reviewing scientific programming projects. Morris identified a tendency for prototyping practices to be employed even when production scientific software was being written. As a consequence, the software reviewed was of low quality, high complexity and contained a considerable amount of duplication. Morris notes that a number of practices and tools have been developed within software engineering to address the deficiencies identified in the review.

Segal [156] reported a case study in which an external team of professional developers was contracted to provide a library of software components by a research organisation. Segal concludes that linear, plan-based engineering methods are ineffective in the context of scientific software development because scientific domain experts are unlikely to be able to fully state the requirements for the software at the start of the development process. Further, the process of preparing a requirements specification did not establish a common understanding of the requirements. In particular, the requirements specification was treated as a complete document by the software engineers, whereas the scientists assumed that it represented an outline into which additional features could be incorporated later.

Based on the earlier work, Segal [157] later proposed a process model for scientific software development by scientists, based on several case studies. The model illustrated Segal's perception that validation is informal and based on an expert's expectations of the software's output. Segal noted the potential for conflict between this approach to iterative development and validation and traditional plan-based, requirements-driven software development process models.

Later, Segal [158] investigated the early phases of scientific software development, typically involving small project teams. The work illustrated the typical development process in scientific programming using an iterative model, in which successive iterations are driven by an informal assessment of the behaviour of the software artefact, in relation to the expectations of expert users. Segal noted that this approach to software development, while successful for rapid prototyping and proof of concept work, can introduce obstacles to the development of production quality systems that can be disseminated in a peer community.

Based on observations of scientific software development over more than a decade, Kelly [88] argued that activity in the domain is primarily concerned with knowledge acquisition rather than software production. Kelly argued that this model reflects the need for scientists to explore a problem through the software and identifies how this affects the practice of software development. For example, Kelly notes that many of her participants emphasise the preparation of readable, self-documenting code, because readable code is easier to inspect and discuss with colleagues. Similarly, tests and inspections are used to understand how software functions, rather than to explicitly detect or prevent the creation of defects. In later work, Szymczak et al. [175] proposed the use of literate programming tools to capture the knowledge acquisition process described by Kelly.

Carver et al. [29] and Kendall et al. [96] investigated a diverse collection of six case studies (the "Bird" projects) of scientific software development practice using a mixture of questionnaires and follow-on interviews. The purpose of the research was to develop a body of knowledge of domain challenges and good practices in scientific programming. The authors' findings support many of the conclusions of the surveys of practitioners already described. Team sizes were between 3 and 20 participants at any one time, with larger teams generally responsible for longer-term "infrastructure" efforts supporting a larger number of customers and code bases. Two of the case studies were in "maintenance mode," providing volunteer support for bug fixes, but not implementing new features due to a lack of interest from customers and sponsors.

Project practices vary considerably, but most projects had adopted source code management and many of the characteristics of “micro” software development teams [45]. Requirements were generally gathered informally from end users and defect tracking was predominantly handled by internal project communications, rather than formal systems. Working practices varied from team to team, depending on scale and resources. However, a unifying theme for all of the projects was the need for multidisciplinary project teams consisting of both domain specialists and computer scientists/software engineers. This view is validated in the case studies by earlier failures in which mono-disciplinary periods of a project were felt not to deliver systems required by users.

Hochstein and Basili [76] also reported a series of case studies focused on parallelisation efforts in scientific software drawn from the Advanced Simulation and Computing (ASC) Alliance. These projects were generally larger than those reported in the Bird case studies, with typically as many as 75 participants, although core developer teams were somewhat smaller. This project structure is similar to that described for the ATLAS project at CERN [46, 75, 119]. Different parts of the software effort are subject to different verification and validation activities as a consequence of these different forms of contribution. Core infrastructure components often have extensive regression test suites maintained by the core infrastructure team. Conversely, “peripheral” code contributions may be accompanied by unit tests, but this is not mandated by the project. A consequence may be that assessments of software quality in these projects may be more difficult due to the variable coverage and effort [113].

Understanding the practices in scientific programming is an active research area, with several authors detailing plans for future research. For example, Heaton and Carver [69] found a variety of often contradictory claims in the case study literature as to the effectiveness of software engineering practices in scientific programming. Crabtree et al. [37] reported work in progress to understand the application of agile software development methods in case studies of scientific programming projects. Henderson and Perry [70] described plans to conduct similar interview led research at their home institute. Mesh and Hawker [115] and Mesh [114] reported plans to employ grounded theory to develop a process improvement strategy for scientific software development processes.

3.2 Agile Methods

Several studies have specifically explored the use of agile methods and practices in scientific programming teams. These studies have suggested that agile practices for requirements gathering and quality assurance may better fit the dynamic and concurrent nature of scientific software development than plan-based approaches. For example, Ackroyd et al. [1] described the adoption and subsequent adaptation of extreme programming (XP) practices within a software development team supporting scientists working with synchotronic equipment. Ackroyd et al. [1] reported that many agile principles and practices had been incorporated directly into the team’s processes. In particular, they reported that an agile approach to requirements specification and planning enabled the team to elicit and prioritise requirements effectively. Conversely, other practices such as test first development and shared code ownership were found to be less effective, due to time pressures to complete new features, and the specialist natures of particular codes.

Wood and Kleb [191] reported their experiences of introducing agile practices to a proof of concept numerical test bed project at the NASA Langley Research Center. The work illustrated the cultural differences between the nature of the safety critical scientific work undertaken and the ethos and principles of agile methods. For example, the established development processes at the laboratory were highly document- and plan-driven, with an expectation that requirements and design could be delivered upfront. This contrasted with the XP approach, which employs continual re-evaluation of requirements and plans. Overall, the authors found that XP with adaptations could be applied effectively to safety critical scientific programming. Blom [18] drew similar conclusions as to the use of a combination of Scrum and XP based on his own experiences in university-based projects.

Easterbrook and Johns [49] conducted an ethnographic study of the software development practices employed by climate scientists working at the Hadley Centre, part of the United Kingdom's weather forecasting service, the Meteorological Office. In contrast to the work of Hannay et al., Easterbrook and Johns [49]'s study identified examples of agile software engineering practices, including automated version control management, code reviews, automated test harnesses, ticket-oriented defect management and continuous integration. Shull [163] also interviewed climate science researchers and reported similar practices at the Goddard Centre in the United States. Based on interviews with practitioners, Sanders and Kelly [153] also reported a wide variety of agile software engineering practices employed in different scientific programming domains. These practices include iterative development, separation of prototyping and production development lines, and formal user interface design techniques.

Several researchers have explored the application of agile methods in bioinformatics. Kane [84] reported on his experiences of introducing agile methods into a team of contractors working in a bioinformatics research laboratory. The team adopted practices gradually, beginning with the introduction of source code version control and continuous integration. Later, the team adopted more demanding practices that required the cooperation of the project customer. One particular outcome of this was that the team discovered the practice of periodically reviewing and revising the entire backlog to be a useful, if tedious, activity in an ongoing engagement with a customer. This is not a practice advocated in main-stream agile software development guides. However, Kane [84] found this an effective way of reviewing project priorities.

Later, Kane et al. [85] surveyed the practices of six software teams working in bioinformatics research that either employed or had previously employed agile practices. Like Wood and Kleb [191] and Ackroyd et al. [1], Kane et al. also found that many agile practices could be adopted directly, given a development team of appropriate size. In contrast, however, Kane et al. found that testing practices were much more developed in some of their case study teams. In particular, one team's acceptance tests were specified and developed by the project's customer. This suggests that closer collaboration between development team and customer is possible in some scientific programming domains.

Pitt-Francis et al. [134] reported their experiences of applying agile methods to the development of cardiac modelling software. Similar to Wood and Kleb [191], Pitt-Francis et al. found that although many practices can be applied without alteration, some required adaptation for use in scientific programming. For example, rather than frequently releasing software to users, releases were associated with the publication of results, to retain a competitive edge for researchers. Pitt-Francis et al. also found that some aspects of the academic culture in which they worked needed to be adapted to apply agile methods. For example, there was a need to overcome reluctance to share code within a team to achieve collective ownership.

Sletholt et al. [165] reviewed the case studies reported by Easterbrook and Johns [49], Kane [84, Kane et al. [85], Pitt-Francis et al. [134], Wood and Kleb [191]] to understand the extent to which agile practices from Scrum and XP are employed in scientific software development across a range of projects. A key finding was that many agile methods can be used successfully in small-scale scientific programming teams, with some adaptations. Sletholt et al. [166] went on to undertake their own case studies, which concluded in contrast that the application of agile practices was rather more varied. Practices appear to be adopted on an ad hoc basis, rather than as a conscious decision to employ a comprehensive approach to agile methods. It may be noted that Beck and

Andres [12] also advocated a gradual, rather than comprehensive introduction of agile methods into software teams.

3.3 Project Team Evolution and Software Documentation

Hannay et al. [66] and Pitt-Francis et al. [134] separately observe the evolutionary nature of the "development team," as participants (typically graduate students or post-doctoral researchers) join and subsequently leave a research group at regular intervals once their project is complete. This means that the principle contributors to a scientific software code base may change frequently during the lifetime of a project, in a similar (although more regular and predictable) way to open source projects. A consequence of the constantly evolving research team is that no one contributor may have a comprehensive understanding of the system source code [95, 134]. Indeed, the system may begin to adopt the features of a legacy system, from which the original developers have long departed.

The constant change in project contributors may be a reason that comprehensive documentation is valued more highly in scientific programming projects, as reported by Dubey et al. [47], Fangohr et al. [53], and Chilana et al. [31]. Chilana et al. reported the importance of detailed source code documentation in the scientific programming community. This finding contrasts with prevailing philosophy in many domains that have adopted agile practices, in which extensive documentation is deprecated in favour of clearer, "self documenting" code.

By contrast, when Sanders and Kelly [153] investigated programming language choice for scientific development they noted that developers valued domain-specific languages (DSLs) and environments, such as MATLAB because they facilitate a close relationship between documented theory and executable program source code. Domain-specific programming languages have also been proposed for specific scientific fields. For example, the Braincurry language [125] has been designed to support the specification and implementation of experiments in Neuroscience. In related work, Smith et al. [169] and Yu and Smith [194] proposed a method for comparing families of computational models using a standard analysis template. The purpose of the research was to improve the clarity of model documentation and enhance decision making when choosing between related models.

Given the proliferation of domain-specific languages, there is relatively little research in the literature on the software engineering benefits of DSLs for scientific programming, such as readability, maintainability or reproducibility of experiments. However, DSLs are generally restricted to the concepts and concerns of the problem domain at hand [180]. In the context of scientific programming, this could substantially eases the task of comprehension, comparison and analysis of a scientific experiment, since the ways in which an experiment could be implemented are tightly constrained and the key concepts of the science are directly supported in the language. The proliferation of DSLs for a single domain or set of related domains could also pose advantages that the availability of a multitude of general purpose programming languages do not. The restricted nature of DSLs might potentially ease the re-implementation of the same software experiment in several different DSLs to achieve greater confidence as to the reliability of the results, since any variations are less likely to be due to "internal" variations in how an experiment is implemented.

3.4 Best Practices

Several authors have proposed sets of best practices for software engineering in scientific programming efforts [48, 73, 91, 173, 187]. These proposals are typically based on the author's own experiences of "what works" or observations of work in case studies. For example, Kelly et al. [91] and separately Wilson et al. [187] summarised best practices for scientific programming based on their previous experiences in this domain. Earlier, Wilson and Lumsdaine based many of their best practices on experiences developing the Software Carpentry course [188]. Heroux and Willenbring [73] focus on practices that are recognisable from agile methods. Indeed, all the proposals included many practices already common in software development in other domains, such as maintaining a separation of concerns, pragmatic decisions about documentation, source code management, code reviews, continuous integration and test case development. Gent et al. [61] made similar arguments (particularly regarding the fundamental importance of source configuration management) more than a decade earlier.

Andersen et al. [4] proposed several existing software development practices that can be employed in a scientific development domain to address challenges identified by Axelrod [8]. For example, close cooperation between software developers and domain experts (as in the Extreme Programming process) is advocated as a means of minimising discrepancies between theoretical models and implementation, whilst enhancing long-term maintainability; the specification of use cases to enhance the accessibility for other scientists to inspect and configure the system; and the partition of software into modules with well defined responsibilities.

Post [136] and later Kendall et al. [97] reported on the applications of lessons learned in previous case studies as best practices in the planning for a large-scale (characterized as $360 million over twelve years) scientific programming project. The aim of the CREATE programme was the development of a suite of computational research tools that could be employed by United States Department of Defense (DoD) equipment acquisition teams for modelling and analysing new weapon systems through access to HPC facilities. Post [136] anticipated that a significant challenge for the CREATE programme would be the tension between the application of agile principles to development team coordination and the demands of the wider organisation's established policies and procedures. This expectation was later confirmed by Kendall et al. [97] who reported that this risk had been mitigated in several ways, including the formation of development teams from multiple DoD institutions; adapting agile planning processes to fit with DoD policies and developed cross-institution communication using video conferencing and other facilities.

4 QUALITY ASSURANCE PRACTICES

The relationship between computational models and their implementation in software is crucial to the reliability of scientific results [178]. There is no direct relationship between software quality assurance and correctness, and quality assurance processes cannot guarantee the absence of defects in software. However, it is generally accepted by software engineers that the absence of quality assurance practices is associated with a higher rate of defects in software. Similarly, the absence of quality assurance processes means that a project suffers from higher risk of the introduction of defects over time.

Several authors have argued for far greater value to be given to the verification and validation of scientific software, particularly implementations of models used for simulation and prediction. Post and Votta [138], for example, refers to the need for a paradigm shift in this area. Indeed, advocates of software validation in scientific programming were making their arguments as early as the 1960s [83, 121]. There are several reasons for a lack of software quality assurance practice in scientific programming reported in the literature:

- Theoretical (often mathematical) models and their associated software implementations and empirical data become conflated [49, 88, 92, 99, 153, 170, 183]. For example: Calder et al. [21], in an extensive discussion of verification and validation of the FLASH multi-physics code refers to verification as "the process of determining that a model implementation accurately represents the developer's conceptual description of the model" but only defines "model" and not "implementation" as a first class artefact; and Kelly and Sanders [92] reported that one of the scientists they interviewed was reluctant to allow software engineers to review or modify program source code because they perceived it as "my model" and noted that they "pursued causes for their outputs not matching *expected results*" (author's emphasis). Consequently, the importance of verifying the correctness of model implementations with respect to a theoretical design (quite apart from validating them) may not be recognised or accepted.

- Software is not perceived as a distinct and valuable contribution to scientific research. Killcoyne and Boyle [99] and Spinellis and Spencer [170], for example, interviewed two researchers working in climate science research. The interviews highlight a common view of the use of software in experimental research, that software development "cannot be allowed to get in the way of the science." Basili et al. [10] makes similar observations from the perspective of their experiences of the high performance computing community. Quite reasonably, the goal of scientists is to produce scientific results, not software. However, an unfortunate consequence may be that the complexities and risks of using software as a scientific instrument are not well understood by end users. Equally, computer scientists can be reluctant for their discipline to be treated as a service to other sciences [190].
- Scientists and engineers over-estimate their ability to produce high quality software. Carver et al. [22] found that awareness of many software engineering practices was relatively low in the respondents to their survey. Despite this, self-assessment of the respondents' ability to produce software of sufficient quality for their work was very high. The authors concluded that the results of the survey supported their contention that scientist-developers "don't know what they don't know" [22].
- There is a poor dissemination of software engineering practices amongst developers in scientific programming projects. As already noted, many developers engaged in scientific programming are largely self-taught, or taught by a scientific domain expert. Umarji et al. [179], after conducting a survey of bioinformatics researchers, found that the dissemination of software engineering practices was variable. Their review of educational material for bioinformatics courses also found few references to software engineering, or the risks associated with low quality software. Consequently, scientific software developers may not be aware of the potential and consequences of discrepancies between design and implementation.
- Software engineering quality assurance practices are inappropriate for scientific programming because they do not fit well with the constraints of the domain [99]. In particular, most software engineering practices assume development efforts of substantial size in order to achieve a cost benefit return on investing in the practice. However, the size of many scientific programming efforts may not justify these upfront costs [55]. Separately, most software development practices assume that requirements can be established and stabilised for a reasonable period of time (even agile methods assume requirements will not change substantially over a single iteration). Conversely, requirements in scientific development may undergo very rapid change, increasing the cost of applying quality assurance practices, such as refactoring and test-driven development.

Despite these challenges, there is evidence in the literature that some scientific software development efforts are adopting (and adapting) existing quality assurance practices from software engineering. In particular, Oberkampf et al. [128] proposed a capability maturity model for physical modelling and simulation software, based on the Carnegie Mellon Capability Maturity Model (CMM) [35]. The model identifies different stages of maturity for different aspects of model and software development assurance, including, for example, the fidelity of the theoretical model to real world phenomenon and software code verification practices. Similar ideas for multiple aspect assessments of simulation software were proposed much earlier by Naylor and Finger [121], with particular emphasis on the predictive capability of the simulation.

The approach adopted by Oberkampf et al. identifies different factors that influence the reliability of modelling and simulation predictions, such as the fidelity of the model to real physical properties of the target system and the extent of efforts to ensure correct numerical implementation of relevant algorithms. Each of these factors is associated with an ordinal maturity level for predictive capability, ranging from 0 (accuracy is based on informal judgement and experience) to 3 (accuracy assessment is formal, detailed and evidenced). A CMM typically provides a framework for directing domain-specific quality assurance efforts, without mandating specific practices.

Other researchers have investigated specific techniques to address quality assurance for scientific programming. Techniques for validating scientific programming artefacts include source code inspection, static analysis, formal refinement techniques and software testing. Experiences of applying these techniques in different contexts are discussed in the sub-sections below.

4.1 Testing

Case studies of scientific software development have suggested that a variable amount of effort can be applied to the development of test harnesses [132]. A key challenge found by Kanewala and Bieman [87] is the availability of a *test oracle*: a means of computing an expected result for comparison with the output from the software under development. Oracles come in a variety of forms, including manual computation, earlier prototypes and third party reference implementations [186]. A challenge in scientific programming is the difficulty of developing a test oracle independent of the software under development [4, 92, 94, 170, 186]. If the purpose of the software is to test a hypothesis, the expected output is unknown and in principle any output from the software could be correct, independently of whether the output supports the hypothesis or not. In practice, partial oracles exist that permit many erroneous outputs to be detected because they are impossible. The more subtle challenge for software testing is to detect outputs that are feasible, but incorrect. In the context of climate science, for example, observations of climate variables such as atmospheric pressure at different altitudes and oceanic temperature in principle provide an oracle for software simulations of climate behaviour as a result of different forcings. However, these records are mostly limited to the twentieth century, and even then require considerable processing and integration to account for different data collection methods and tools, as well as variability in the location and context of weather stations over time [50, 143].

Exacerbating this problem, Chilana et al. [31], Easterbrook and Johns [49], Hook and Kelly [79], Sanders and Kelly [153], Segal [159], Shull [163] all noted the tendency for scientists to depend on their own expertise and expectations as to outputs to validate model implementations. This finding was also reported by Kanewala and Bieman [87] as part of a systematic literature review of testing scientific software. Easterbrook and Johns [49] reported that climate scientists treat the software implementations of models as "evolving theories" and are consequently less concerned with "code correctness" in relation to a theoretical model. In this approach, an experimental change to code is evaluated against a results from a previous version of the model. Sanders and Kelly [153] noted that if an unexpected result occurs it may result in changes to either the underlying theory or the source code. However, it may be difficult to determine whether an unexpected result is a consequence of an invalid theory or an imperfect implementation of the theory in source code [4]. Further, conclusions may be at risk of confirmation bias, because results that appear to confirm a theory may not be investigated further, even though they are an artefact of the implementation (a defect) [139]. Such problems afflict other complex experimental tools, but the tempo of software development and evolution compared with the manufacture of physical artefacts makes this problem particularly acute.

Despite these obstacles, there is evidence in the literature of considerable interest in the testing of scientific codes. For example, Clune and Rood [34] reported a range of quality assurance practices employed in a case study in climate science. Pipitone and Easterbrook [133] compared defect density rates in a selection of global circulation models (GCMs) with those found in the open source projects Apache, VTK and Eclipse. The authors concluded that reported defect rates in the GCMs were generally lower than in the open source projects, suggesting that the software quality of the GCMs was at least as high, if not higher than the open source projects. However, as the authors note in their discussion of threats to validity, reported defect rate is dependent on both the underlying rate of defects in a software system and the extent of efforts to uncover defects through quality assurance practices such as testing. A low defect report rate may be equally indicative of limited efforts to discover defects, particularly since the authors report their own difficulties in identifying defect reports in the GCM project artefacts. Despite these limitations, the work still demonstrates evidence of the practice of bug tracking in scientific programming.

Calder et al. [21], Hochstein and Basili [76] advocate the use of laboratory experimental results as an oracle for validating simulated results. However, the authors also note several obstacle to this approach, during an extensive discussion of the verification and validation activities undertaken on the FLASH multi-physics code. First, the interplay between the different parts of a code that represent physical phenomena can make the separation of concerns during testing difficult. For example, in the case of FLASH, errors in the code may be masked by inappropriate selection of equations of state during testing. Disparity between actual and expected test results may be due to either and the two are difficult to test in isolation. Second, physical experimentation for testing may not be able to adequately replicate the real phenomena of interest. Replicating the physics and chemistry of the internal state of a particular type of star, for example, is not feasible in a conventional laboratory experiment. Third, even when laboratory experiments are available, the diagnostic instrumentation may not have sufficient resolution compared to the results obtainable from simulation (or for the desired results of the science). Finally, natural variation may occur in the setup conditions of the experiment. This real world complexity will complicate comparison with results from an idealised computational simulation.

One option proposed by Trucano et al. [178] to encourage the development of testing infrastructures is to deny funding for experimental validation activities for computational science projects that lack verification processes for simulation codes. As Trucano et al. argues, experimental activities are often very expensive (hence the need for computational simulations) so it may not be unreasonable to focus resources on projects that can demonstrate high confidence in the correct code implementation of (potentially imperfect) simulation models. This approach would imply a staged QA process, in which codes progress through formal or semi-formal verification steps before securing funding for model validation once sufficient confidence in code correctness justifies the resource expenditure. A potential disadvantage of this approach is the disruption caused to a experimental/exploratory approach to scientific research.

Other research has concerned the development of new approaches to testing which account for the oracle problem described above. Hoffman [77, 78] proposed the use of a variety of "real world" test oracles as a means of addressing the paucity of other options in scientific programming. For example, heuristic oracles provide correct expected results for a subset of selected inputs. Inputs between these values are checked using a heuristic that relates them to the selected input output combinations. Similarly, Weyuker [186] observed that testing without an oracle could still be useful if the properties of an incorrect output are known (an output value outside the range [−1…1] for an implementation of the sine function, for example).

Betz and Walker [15] argued that results from previous executions of a software application can be useful as a test oracle, since this allows developers to detect when a change to code has caused an unexpected change in behaviour. Betz and Walker demonstrated this approach through the adoption of continuous integration in the AMBER project. The technique is similar to the development of test cases at the start of a software refactoring process [56]. Similarly, Wang et al. [184] described a proposal for developing regression tests for individual modules by deriving input and expected output combinations from full experimental runs of a larger integrated system. The approach assumes that in well designed systems functions should behave in an identical way, whether integrated in a simulation or exercised from a test harness (implying that functions do not depend on system state or have side effects).

Shull [163] reported that his climate science interviewees would use simulations of simple geographies (perfectly flat planets, for example), for which the correct climate behaviour, according to the model, can be predicted analytically. It is unclear how these simpler test cases can be used to assess the overall correctness of the full implementation. For example, Shull [163] does not indicate what proportion of the source code base was exercised as a consequence of these automated tests. Andersen et al. [4] sketched a similar approach to validating simulations of theoretical models, using older, presumed reliable simulations or simpler configurations as test oracles. In this approach, the results from the new simulation are compared with those from the older tools where their domains of input overlap.

Both Roache [148] and Hook and Kelly [79] argued that the dependence on domain expertise for validation testing is unavoidable and therefore there should be a clear separation between verification and validation activities. The test regime described by Calder et al. [21] for validating FLASH is an example of this dependence: the extensive test suite is built from standard problems in the domain and the selection of the appropriate test case requires the expertise of a domain expert. Hook and Kelly concluded that the scientific validity of software should be considered in terms of trustworthiness (is the result produced by the software believable) rather than correctness. The implication of this approach is that it is important to assess the thoroughness with which a scientific developer has evaluated their code for trustworthiness.

Hook and Kelly [79] and Hook [80], therefore, took an alternative approach by proposing the use of mutation testing [44] to evaluate the effectiveness of a scientific software application's test suite. Mutation testing works by applying the existing test suite to a randomly "mutated" version of the target application. Mutations include substitution of operators, alteration of constants and alteration of conditional structures. The number of tests that fail as a result of a mutation gives an indication of the effectiveness of a test suite in detecting the inadvertent introduction of defects. Later, Kelly et al. [89] used Hook's [2009] mutation testing techniques to investigate the effect of reducing oracle tolerances on test suite performance. The work suggests that the availability of a high precision oracle is more effective in uncovering defects than generating more test cases for a lower precision oracle.

Kanewala and Bieman [86] also investigated the problem of testing without an oracle and proposed the use of metamorphic testing. This technique identifies relations over the properties of the inputs and outputs of a software process that should hold if the input changes. A function that sorts a list of *n* elements should always output a list of *n* elements, for example. Consequently, metamorphic testing is useful in identifying incorrect outputs as described by Weyuker [186] (outputs that cannot be right), rather than for acceptance testing. Later, Lundgren and Kanewala [106] evaluated metamorphic testing for a gene sequencing toolkit. The results suggested that metamorphic testing was more effective at detecting faults than through comparison with an alternative sequencing tool.

Remmel et al. [145] proposed another methodology for developing test suites by treating a family of scientific software applications based on a common software frameworks as a single software product line. Remmel et al. argue that this approach allows the development of test cases for the framework based on an analysis of typical use cases amongst the family of applications. In addition, the method allows the development of reusable test cases that can be employed in future applications. In effect, this approach allows for the shared development of a test oracle between a family of related applications.

A particular challenge reported in the ASC projects was the difficulty of verifying (and validating) parallel codes [76]. It is not clear whether defects were predominantly resident in the design or implementation of the algorithms, since (as noted above) these are often not distinguished by developers in scientific computing projects. For example, the authors report situations in which the number of processors assigned to a task is a factor in the manifestation of defects, which could indicate either design or implementation defects (or both). The MPI framework was identified as a particular cause of these problems, due to a lack of abstraction of parallelisation mechanisms.

4.2 Inspections

Several authors have proposed the use of inspections as an alternative quality assurance practice to software testing. Inspections depend on domain expertise for the discovery of defects, rather than the availability of a test oracle, so may be more effective for scientific programming. Hatton [67, 68], for example, investigated software quality in a large (several million lines of code) seismic data processing application. Hatton employed both static and dynamic analysis of the software. The dynamic analysis in particular demonstrated that the presence of loss of precision defects was so severe as to make the results of the software application essentially equivalent to a random function [67].

Kreyman et al. [101] reviewed sources of defects in software-based scientific models and proposed an inspection technique based on this taxonomy. The taxonomy covered defects in requirements (inappropriate selection of models), design (discrepancies between model and program), implementation (such as logical and numerical defects) and during component integration. Although the inspection procedure is accompanied by an illustrative example, it is unclear how much effort and expertise is required to apply the technique effectively. The authors note that the approach is highly dependent on the adoption of an interdisciplinary approach, and thus engagement and interaction of software engineers and scientists.

Kelly and Shepard [93] proposed a technique based on software inspections for detecting discrepancies between models and source code implementations of software-based experiments. The technique combined a mandated software documentation exercise with inspections intended to identify discrepancies. A key aspect of the technique was to allow inspections to run over the long term, collating reports as individual inspection tasks were completed. This meant that new versions of the software system were released during the inspections, requiring ongoing integration.

Later, Kelly and Sanders [92] investigated the methods employed for assessing the quality of scientific codes in practice, based on interviews with scientists in a variety of domains. Kelly and Sanders observed that none of the interviewees reported the use of software inspections, noting that this techniques depends on recruiting inspectors with expertise in both the problem domain and software development. Kelly and Hook [90] went on to explore the use of inspections as a means of driving test case development. In the case study, a domain specialist used a debugger to follow the steps of a software system to improve their understanding of its behaviour. White box test cases were then developed, leading to the successful discovery of defects that Kelly and Hook claimed would not have been discovered without the inspection step by a domain specialist.

4.3 Continuous Integration

Continuous integration is a relatively recent software engineering practice intended to minimise the disruption caused by continuous and concurrent changes to software. Betz and Walker [15] and Bartlett [9] report the use of continuous integration in scientific programming efforts. Betz and Walker [15] reported on the experience of adopting continuous integration in the AMBER (a molecular simulation application) project. Betz and Walker described the AMBER project as a collaborative, distributed development effort, used for a diverse range of purposes and on a diverse range of hardware platforms (including commodity processors, graphics processing units and super-computers). Consequently, ensuring consistency of results within these parameters is complex and potentially costly in terms of researcher time.

Bartlett [9] investigated a more complex scenario for software development in which the development of a scientific software project is coupled to the concurrent but autonomous development of a a third party library. Bartlett argues that in this context, there is a need to ensure application development is undertaken against the most recent version of the library to minimise disruption caused by significant code changes; whilst also allowing collaborative development of both application and library to be undertaken simultaneously. Bartlett proposes a model of "almost continuous integration," in which most application development occurs against the latest daily release of the library, whilst significant collaborative changes are implemented in branch developments that are only merged with the main product line for a major release.

4.4 Formal Methods

Formal methods, typically involving the mathematical specification and verification of computer programs, appears to have received little attention in the scientific programming community. This may be due to the additional challenge of verifying programs that manage floating point data [64]. However, some experience reports have been produced using formal techniques [41, 65]. Gunnels and van de Geijn [65] explored the application of formal methods to the development of a linear algebra environment (FLAME). Gunnels and van de Geijn argued that there are several benefits to adopting formal methods in this context: the potential to prevent the introduction of defects and establish the correctness of code methodically; a stronger relationship between mathematical descriptions of algorithms and their development in code; and (semi)-automated translation into executable program code. However, it is notable that the authors do not attempt to demonstrate that the C implementation in their case study is correct with respect to the abstract formal description. de Oliveira et al. [41] described their ongoing efforts to integrate formal software assurance methods for parallel computation into the Unitah framework, a multi-physics problem solving environment. The reported aims of the work are to develop techniques for checking for scheduling defects in massively parallel software applications through perturbations of schedules.

5 DESIGN, EVOLUTION, AND MAINTENANCE

The dominance of simulation of physical phenomena in scientific computing has meant considerable attention has been given to improvements in the fidelity and performance of software-based simulations [3]. However, the growing complexity of scientific software applications, coupled with continued improvements in computing power has meant that the costs of software production and (more importantly) maintenance are becoming increasingly significant. Scientific software may be extremely long lived and as a consequence subject to regular maintenance activities as requirements change. For example, Post and Kendall [137] state that codes developed for simulating nuclear explosion yields may have a lifetime of 40 years or more. Sanders and Kelly [153] reported that participants in their case studies of scientific programming were already encountering common problems associated with software maintenance of large-scale, long-term projects. These problems included difficulty in adding new features or repairing defects. The interplay between design activities and software evolution in scientific programming has been addressed by several different research efforts. This section outlines the major themes in the literature concerning the management and maintenance of design in long-term scientific software projects.

5.1 Component Architectures

Boisvert and Tang [19] edited a collection of articles investigating the architecture of scientific software. The collection is divided into two main themes: the integration of heterogeneous components; and the structuring of single components for scientific applications. Much of the work on integration of components mirrored similar activities in other domains during the period. René et al. [146] describes the development of a component framework for parallel computations using CORBA, for example. Similarly, articles grouped in the second category reflected the parallel interests in object-oriented design, such as Ahlander et al. [142] and Thuné et al. [177]'s work, for example.

Allan et al. [3] presents an overview of the Common Component Architecture, a component middleware standard designed specifically for the needs of scientific computing. Allan et al. argued that component middlewares for scientific computing must fulfil several specialised criteria including support for scientific programming languages, such as Fortran, native support for complex numbers, the minimisation of middleware overheads and flexible implementation of local and distributed components following a variety of communication models. The architecture, as described, is intended to fulfill these objectives. At the time of publication (2006), the approach had been demonstrated within a variety of scientific domains, including combustion modelling, climate science, and quantum chemistry [3].

5.2 Design Patterns

Several authors have explored the application of software patterns (as popularised by Gamma et al. [57]) as a means of managing the complexity of scientific software. Decyk et al. [43] investigated the migration of legacy Fortran programs into C++ through the intermediate step of implementing object-oriented concepts in Fortran 90. The work showed that many object-oriented concepts such as encapsulation, inheritance, and (to a certain extent) polymorphism can be expressed in Fortran 90 as software patterns enforced by convention rather than language constructs. It is unclear from the report, however, whether applying these patterns would enhance the quality of code written in Fortran 90, since there is a significant amount of "boilerplate" code required for each construct.

Decyk and Gardner [42], Norton et al. [126] later extended this work by demonstrating the implementation of a selection of object-oriented design patterns [57] in Fortran 90/95. For example, Norton et al. demonstrated the application of the Strategy pattern to the management of variations in algorithm implementation, whilst Decyk and Gardner applied the Factory pattern to the creation of particles of different types in a simulation of plasma.

In parallel work, Markus [108] also illustrated the implementation of a selection of design patterns in Fortran 90/95 and later revisited a selection of these patterns in Fortran 2003 [109]. In addition, Rouson et al. [150] explored the application of creational design patterns in Fortran 2003, with a particular focus on avoiding memory leaks due to unused but none-deallocated objects. Similarly, Gardner and Manduchi [58] presented an extended tutorial on the application of conventional design patterns to a scientific programming project (a waveform browser with a variety of applications). The aim of the book was to demonstrate the applicability of design patterns and refactoring techniques to scientific programming efforts.

Several authors have also considered the development of patterns tailored to specific scientific programming requirements. Cickovski et al. [32] proposed a series of patterns derived specifically from the recurring design problems in scientific software. The work showed that the recurrence of similar software design problems (albeit specific to scientific programming) can be addressed using established software engineering techniques. This suggests that the principles for high-quality software design established in other domains are equally applicable to scientific programming. In addition, the article identifies a domain-specific collection of patterns for modelling molecular dynamics.

Billie [16] proposed a selection of patterns for simulations, including simulations of discrete and continuous phenomena. The work identified the key collaborating classes in each pattern that would form the basis of a re-usable solution. Rouson et al. [149] also explored scientific software patterns, proposing several patterns for managing the interactions between software implementations of semi-discrete simulations. For example, the semi-discrete pattern contains standard methods for advancing the time-step of a simulation, but management and representation of simulation data is handled by the realising class. Similarly, the Puppeteer pattern (a variant of Mediator) reduces the complexity of interactions between different coupled simulations.

5.3 Refactoring and Reengineering Techniques

Design patterns, as discussed in the previous section, are often applied during the refactoring [56, 98] of legacy software code. Researchers have investigated the use of refactoring techniques in the maintenance of legacy scientific development efforts. Arora et al. [7] proposed the reengineering of legacy scientific codes using generative or aspect-oriented technologies, such that existing applications can be augmented with new features without altering core functional behaviour. They demonstrated a proof concept by augmenting an existing application with check points implemented as higher-level abstraction aspects. Woollard et al. [192] addresses a similar problem by demonstrating a technique for encapsulating legacy scientific codes within a component-oriented framework. Woollard et al. [192] argues that this approach, involving only a limited amount of alteration to the legacy codes themselves, eases the process of architectural maintenance as system requirements evolve.

Overbey et al. [131] investigated an alternative technique for refactoring legacy Fortran programs to remove deprecated program constructs with alternatives (replacing `goto` statements with `if` constructs, for example). The work was preliminary, but presented a proof of concept for improving the maintainability of legacy scientific codes without (in principle) affecting functional behaviour. Norton et al. [126] also informally demonstrated the identification of refactoring opportunities during re-engineering of Fortran 77 to Fortran 90/95 program codes. Norton et al. showed that long parameter lists in Fortran 77 could be reduced through the use of Fortran 90/95 module constructs.

Kelly et al. [94] reported a suggestion for developing reverse engineering tools tailored to the scientific domain for extracting formulas from imperative source code. Li [105] outlined a case study in re-engineering a legacy scientific software development through the reverse engineering of a domain model and subsequently system requirements. Like Kelly et al., Li concluded that tools tailored to the needs of the scientific programming community are required to support software maintenance in this domain.

5.4 Workflow Management and Executable Research Papers

Rice and Boisvert [147] reviewed the state of the art in scientific software libraries in the mid 1990s, predicting the trend towards problem solving and scientific *workflow* environments. The intention was that such tools would support the comprehension, management and distribution of computational experiments. There are currently several scientific workflow management tools available for a variety of domains, such as Taverna [130] and the associated MyExperiment platform [63]. Typically, these tools are web-based applications that provide facilities for implementing new work flow components that can then be integrated into a whole workflow using graphical modelling tools.

In the academic literature several other approaches are described. Smith et al. [168] described proposals for easing the management of scientific software projects by treating related codes as a family- or product-line. Smith et al. proposes the adoption of standardised templates for documenting the specifications of scientific software components to ease reuse. Vigder et al. [182] described the development of a software framework for automating the integration of software-based tools into scientific workflows. The work was motivated by an identified lack of automated support for repetitive and time consuming tasks in a scientific development case study. One consequence of the lack of automation was inconsistency in data and software management practices (inhibiting reproducibility, quite apart from productivity).

Neves et al. [123] and Davison [40] have separately advocated automating the traceability of computations to enhance repeatability. Neves et al. [123] proposed a framework to augment workflow management systems with mechanisms to track the evolution of data items during workflow execution. The stated goal of the work is to enhance the provenance of scientific results generated through software workflows, by tracking changes to intermediate data items as the workflow executes. Artefact evolution is tracked using software version control systems and data comparison tools such as `diff`. Davison [40] took a different approach, proposing that computational experiments should be run within a framework that automatically captures contextual information, such as hardware configuration and the releases of dependencies. However, the significant context information may vary considerably between experiments, so it isn't clear what characteristics a general purpose framework should support.

Executable research papers (ERP) offer another mechanism for more closely relating experimental artefacts and documentation [33]. Advocates of this approach argue that it reduces the risk of discrepancy between experiment and report, because documentation is updated as the experiment itself changes, rather than as a (potentially omitted) after-thought. In addition, the experiment itself is portable and can be transferred to other researchers for review, analysis, re-implementation and modification.

Quirk [141] proposed the use of the extensible features of the portable document format to embed executable aspects of a computational experiment in a research article. The demonstrated approach allows executable fragments of an experiment to be embedded by an author for later execution, modification, and further experimentation by a reader. Quirk combined this approach with program folds to provide a hierarchical mechanism for viewing program code at different levels of granularity within an article.

Several authors have reported their experiences or provided tutorials of employing literate programming techniques [100] in the practice of scientific programming. Quiney and Wilson [140] advocates the use of literate programming in quantum chemistry research as a means of avoiding the separation of code and documentation. Similarly, Nedialkov [122] provides a tutorial on implementing the VNODE-LP solver using literate programming. The work was motivated by the desire to gain assurance that the solver can be trusted for use in computing proofs as to the bounds on numerical solutions. Nedialkov reflects on the experience of using literate programming, arguing that it is best used after an exploratory, prototyping phase, once the overall design of a program is understood and requires thorough documentation. Singer [164] demonstrated a prototype for software systems research, recording experimental parameters and structure in a program script along with documentation that later forms the basis for an automatically generated research article. Consequently, the experimental implementation and results co-evolve with the documented experimental design and conclusions. Millman and Pérez [117] advocated *literate computing*, a more interactive approach to literate programming, based on the growing availability of electronic scientific notebook applications, such as IPython. In this approach, the notebook becomes the single, interactive focal point for both research development and peer review.

Several other approaches were reported in association with the Elsevier Executable Paper Grand Challenge [59, 127, 181]. van Gorp and Mazanek [181] proposed a web-based portal for creating and sharing ERPs. The aim of the work is to provide a facility for deploying shareable virtual machines containing all the necessary software and dependencies required to execute an experiment. The virtual machines are largely configured at the discretion of journal editors and authors giving considerable flexibility as to the implementation of a software experiment. At the time of writing, the authors had just begun to collect ERPs in the repository, so further evaluation of the effectiveness of the approach will be required.

Gavish and Donoho [59] took a slightly different approach, advocating the adoption of a *discipline of verifiable computational results* as a means of minimising the disruption to existing scientific workflows that could be caused by demands for reproducibility. In this approach, computing platforms (such as a virtual machine runtime) used to generate scientific results (synthesised data sets, plots, etc.) are augmented with a plugin that archives the setup conditions and results of each experimental run. Rather than including a result artefact directly in a publication, an author will reference the result via a universally unique identifier that is generated as the archive is created. As a consequence, a close relationship between scientific results and the conditions under which they were produced is maintained.

6 DATA QUALITY

Many of the issues raised in the literature regarding software quality have also been identified in the wider context of data quality. The growth in the size of research data sets and software processing capabilities have led several researchers to consider quality from a data rather than software process perspective. Wang and Strong [185] proposed a range of parameters for characterising data quality. Later, Bergdahl et al. [13] developed a handbook of data quality assessment methods. This covers a range of qualitative and quantitative techniques for evaluating scientific data sets.

However, the characteristics of modern data sets demands methods that automate data quality assessment and may make many of the assessment techniques proposed by Bergdahl et al. unfeasible [174]. Modern data sets can be: petabytes of data in size; complex to curate, either because individual data items have many or varying attributes, or due to intra-dependencies between items; evolutionary rather than static, as new data items are added and further data cleaning activities are undertaken; and heterogeneous because the individual data items in one data set may originate from several different sources. Stonebraker et al. [174] presented a data curation system, Data Tamer that employs machine learning techniques to partially automate this ongoing assimilation of data.

Climate science presents an example of this challenge. Edwards [50] observes that the production of a relatively simple data set (global gridded temperature records, for example) requires the ongoing acquisition, cleaning and assimilation of data sets from a range of providers, each of whom may employ a variety of data gathering techniques. This data set may then need to be integrated again with other data sets of other observations, such as atmospheric pressure, by other researchers. All these activities imply considerable complexity for users of the data sets and require a considerable amount of supporting information infrastructure.

Bernholdt et al. [14], Mattmann et al. [111], and Crichton et al. [38] describe the challenges encountered in building a large-scale science data management, distribution, and exchange system for NASA's climate science programmes. These include in particular the heterogeneous nature of the different management data sets and accompanying meta-data and the need to maintain autonomy for local data producers/owners. Large-scale software applications, such as the Earth System Grid [14] and the Climate Data Exchange have been developed Crichton et al. [38] to ease the management and manipulation of this data. One consequence is that new, often larger *virtual* data sets are continuously generated, and may also require careful management to support reproducibility.

This view of data curation mirrors modern approaches to software development, in which a software artefact is "cultivated" rather than "produced." The notion of cultivation implies an ongoing evolution of a software artefact (or data set) that incorporates gradual improvements over time as opposed to the periodic releases of end products. An implication of this changing view is that the conception of the scientific method may need to be reconsidered to manage this new complexity.

7 CONCLUSIONS

The challenges for scientific research that is now dependent on software are undoubtedly not new. Rouson et al. [149] cites a Presidential (United States) advisory committee report from 1999, warning that software engineering practices were not being applied effectively to scientific computing. Quirk [141] recalled discussions at a workshop in 1994 that covered many of the same challenges concerning errors in scientific codes. Naylor and Finger [121] discussed the challenges of verifying computer simulation models in the late 1960s. This longevity suggests that there are no straight forward solutions to the challenges posed. In addition, the themes identified in the literature suggest that the practice of both software engineering and scientific research need to be adapted to address the challenge of repeatable, reproducible and falsifiable software-based science.

Many of the practices and tools that have been developed to alleviate the software crisis in other domains of software engineering have been successfully employed (sometimes with adaptation) in scientific programming projects. Further work is required to tailor these tools and practices to support the specific needs of scientific programming, as well as support the transfer of best practices into the domain. In addition, there is a need for software engineers to identify gaps in software engineering practice that leave the requirements of scientific programming practitioners unaddressed. Deshpande's work on software-development models designed for *micro*-teams are an example of advances in this direction [45].

However, many other challenges remain. For example, subtle variations in implementation, software frameworks, compiler configuration, and hardware platform (among others) can all cause small variations to outputs. In most cases, these variations in reproduced results are trivial and are informally accepted as confirmation of a hypothesis. In some cases, these variations, when due to predictable causes, such as floating point rounding, can also be assessed through uncertainty quantification. However, there is a risk in this approach that the level of precision required is set to match that perceived to be achievable, without due consideration for the relevant science. Better procedures, standards, and associated tools are required to document and validate the rounding tolerances for a computational result. A researcher should be able to assert the precision within which they believe an independently reproduced result would support their findings, so this expected precision is both explicit and contestable.

Another unaddressed challenge is that the use of software engineering tools and methods (higher-level programming languages, for example) intended to enhance software quality and ease maintenance may increase the difficulty of verifying the correctness of scientific codes. Higher-level programming languages require a greater number of transformations before they can be executed on the "bare metal" of a physical computer, increasing the opportunity for inconsistencies between what the developer intended and what actually happens at runtime. Thompson [176] famously showed the ease with which a compiler can introduce additional functionality that does not appear in a program's source code. Similarly, Daniel et al. [39] demonstrated that the application of automated refactoring tools can introduce bugs into software, if the refactoring tools (software systems themselves) also contain bugs. The trend towards interpreted and/or domain-specific languages in scientific programming, such as Python and MATLAB accompanied by more sophisticated development environments, for all that it enhances software quality and readability, may exacerbate this problem.

Similarly, the growing interest in the use of virtual machines for packaging computational experiments [60, 181] may inadvertently introduce challenges to experimental comprehension and generalisation. Virtual machine specifications necessarily incorporate a large number of dependencies that either could or should be incidental to the experimental design. Consider, for example, an experiment implemented purely in the Python programming language and packaged in a virtual machine specification. The specification must list all dependencies needed to generate a virtual machine, including, for example, an operating system type and release, Python runtime interpreter release and associated libraries. Many of these choices should be incidental to the results of the experiment, whilst a subset will be justifiably essential dependent variables in the experimental design. Consequently, packaging an experiment within a virtual machine specification can ease repeatability, but may also make reproduction of the experiment more difficult, since dependencies that are critical to the result and those that are incidental are not distinguished [103].

The challenges posed by the use of software in scientific research may mean there needs to be a re-consideration of how scientific research is practised in this context. As has been described, software is fundamentally complex and volatile in nature, and this conflicts with the demands of science for repeatable experimental designs that are sufficiently stable to be submitted to a review and reporting process that may last many months or years. By the time a research article has been published the associated software may have undergone numerous alterations as further enhancements are made and, more critically, defects uncovered and remedied.

Several authors have argued for research articles to be augmented with experimental artefacts, as reviewed in Section 5.4. Several conferences and workshops have experimented with the formation of Artefact Evaluation Committees to undertake peer reviews of codes and data sets submitted alongside manuscripts [17]; and Castelli et al. [30] have reported on the growing development of scientific communication infrastructures for linking articles and datasets. However, the challenges identified in this article suggest these approaches do not go far enough because they do not address the fundamental risk of disconnection between published research results and the experimental artefacts that generated them. Executable artefacts in a peer reviewed research article may be just as outdated as the textual content itself, relative to the current state of an experimental package, as new features are added and defects discovered and repaired. There is a need to develop dissemination mechanisms that more definitively link published scientific reports with continually evolving experimental artefacts.

Fortunately, the gap between software engineering and scientific programming may not be so vast as implied by Kelly [95]. As Killcoyne and Boyle [99] notes, scientific research is often complex and chaotic, with the "process of testing and refining (or discarding) hypotheses lead[ing] to a multitude of elaborate experiments each of which differs, using a unique mix of techniques, technologies, and analyses." Software engineering in other domains can experience a similarly chaotic combination of complex technologies that are continually adapted and re-configured to meet changing needs. A range of techniques have been developed to manage and mitigate the risks that arise in this context. Agile methods such as Scrum and XP have seen widespread adoption in many domains, and several researchers (as described in this article) have adapted and applied them successfully to scientific programming.

Ahalt et al. [2] have outlined a re-conceptualisation of the scientific process based on agile software engineering principles that may address many of these challenges. The authors argue that the increasing complexity of experimental design, coupled with the dependence on software makes the pre-experimental construction of precise hypotheses impractical. Rather, they propose that experimentation should be conducted over short periods of time (cf sprints), based on initial, approximate hypotheses. In addition, the focus of research output from the sprints is shifted from experimental results and conclusions in the form of published articles to the experimental package itself, which should be publicly accessible as soon as practical. As a consequence, the experiment becomes "share-able" within the community of researchers for inspection, improvement and adaptation. Bechhofer et al. [11] also supports this approach, but argues that these packages of data and code, or *research objects*, require much more careful management and documentation to make the extent of their trustworthiness explicit. Falessi and Shull [52] described related ideas for enhancing the automation of software-based science to support reproducibility.

The feasibility of this approach is only just beginning to be explored, with many issues still unaddressed. For example, in 2014, the participants in the Experimental Methodology in Computational Science Research Summer School published a report of their efforts to reproduce a collection of computational experiments, all provided by the participants themselves [6]. The report itself was intended to be open and reproducible, with all experimental code and the report text published on GitHub for inspection and future improvement. Unfortunately, at the time of writing, the article repository has not been modified since the Summer School and the report itself remains largely incomplete, with many of the reproduction attempts abandoned. This experience illustrates that although dissemination of computational experiments is an important and useful development, further work is needed to reduce the friction encountered when reproducing and evaluating computational experiments and enhance the peer-review process.

The 2009 Roundtable on Data and Code Sharing made a number of recommendations in this regard, including the incorporation of code into the peer review process and the development of integrated communities around repositories of code used in scientific programming [193]. Participants in a workshop on Reproducible Research also noted that new review mechanisms may need to be developed for scientific code [104]. In particular, there is a need to distinguish between small, experiment-specific codes and the wider reusable software infrastructure they may depend on.

Software is, of course, not a uniquely fault-prone instrument for scientific research. Normal scientific practice is concerned with gradually establishing confidence in newly published results through repetition and reproduction of experiments. This allows for the discovery and correction of defects in instruments and methods that can just as often strengthen the original result. The key challenge identified in this review is to adjust the tempo of this confidence-building process to match the rate of evolution of scientific software instruments and provide explicit mechanisms to monitor the quality of software and associated scientific data as it evolves over time.

REFERENCES

[1] Karen Ackroyd, Steve H. Kinder, Geoff R. Mant, Mike C. Miller, Christine A. Ramsdale, and Paul C. Stephenson. 2008. Scientific software development at a research facility. *IEEE Softw.* 25, 4 (July/August 2008), 44–51.

[2] Stan Ahalt, Larry Band, Barbara Minsker, Margaret Palmer, Michael Tiemann, Ray Idaszak, Chris Lenhardt, and Mary Whitton. 2013. Water science software institute an open source engagement process, see Reference Carver [27], 40–47.

[3] Benjamin A. Allan, Robert Armstrong, David E. Bernholdt, Felipe Bertrand, Kenneth Chiu, Tamara L. Dahlgren, Kostadin Damevski, Wael R. Elwasif, Thomas G. W. Epperly, Madhusudhan Govindaraju, Daniel S. Katz, James A. Kohl, Manoj Krishnan, Gary Kumfert, J. Walter Larson, Sophia Lefantzi, Michael J. Lewis, Allen D. Malony, Lois C. McInnes, Jarek Nieplocha, Boyana Norris, Steven G. Parker, Jaideep Ray, Sameer Shende, Theresa L. Windus, and Shujia Zhou. 2006. A component architecture for high performance scientific computing. *Int. J. High Perform. Comput. Appl.* 20, 2 (Summer 2006), 163–202.

[4] Peter Bøgh Andersen, Florian Prange, and Søren Serritzlew. 2008. Software engineering as a part of scientific practice. Available at http://imv.au.dk/~pba/Homepagematerial/publicationfolder/softwareengineering.pdf.

[5] Richard G. Anderson, William H. Greene, B. D. McCullough, and H. D. Vinod. 2005. *The Role of Data and Program Code Archives in the Future of Economic Research.* Working Paper 2005-014C. Federal Reserve Bank of St. Louis, MO.

[6] Sylwester Arabas, Michael R. Bareford, Lakshitha R. de Silva, Ian P. Gent, Benjamin M. Gorman, Masih Hajiarabderkani, Tristan Henderson, Luke Hutton, Alexander Konovalov, Lars Kotthoff, Ciaran McCreesh, Miguel A. Nacenta, Ruma R. Paul, Karen E. J. Petrie, Abdul Razaq, Daniël Reijsbergen, and Kenji Takeda. 2014. An Open and Reproducible Paper on Openness and Reproducibility of Papers in Computational Science (September 2014). Retrieved from https://github.com/larskotthoff/recomputation-ss-paper/.

[7] Ritu Arora, Purushotham Bangalore, and Marjan Mernik. 2009. Developing scientific applications using generative programming, see Reference [24], 51–58.

[8] Robert Axelrod. 1997. Advancing the art of simulation in the social sciences. In *Simulating Social Phenomena*, Rosaria Conte, Rainer Hegselmann, and Pietro Rainer (Eds.). Lecture Notes in Economics and Mathematical Systems, Vol. 456. Springer Verlag, 21–40.

[9] Roscoe A. Bartlett. 2009. Integration strategies for computational science & engineering, see Reference Carver [24], 35–42.

[10] Victor R. Basili, Jeffrey Carver, Daniela Cruzes, Lorin Hochstein, Jeffrey K. Hollingsworth, Forrest Shuill, and Marvin V. Zelkowitz. 2008. Understanding the high performance computing community: A software engineer's perspective. *IEEE Softw.* 25, 4 (July/August 2008), 29–36.

[11] Sean Bechhofer, Iain E. Buchan, David De Roure, Paolo Missier, John D. Ainsworth, Jiten Bhagat, Philip A. Couch, Don Cruickshank, Mark Delderfield, Ian Dunlop, Matthew Gamble, Danius T. Michaelides, Stuart Owen, David R. Newman, Shoaib Sufi, and Carole A. Goble. 2013. Why linked data is not enough for scientists. *Future Gen. Comput. Syst.* 29, 2 (2013), 599–611.

[12] Kent Beck and Cynthia Andres. 2005. *Extreme Programming Explained* (2nd ed.). Addison Wesley/Pearson Education.

[13] Mats Bergdahl, Manfred Ehling, Eva Elvers, Erika Földesi, Thomas Körner, Andrea Kron, Kornelia Mag, Peter Lohauß, Vera Morais, Anja Nimmergut, Hans Viggo Sæbø, Ulrike Timm, and Maria Jo ao Zilhão. 2007. *Handbook on Data Quality Assessment Methods and Tools.* European Commission, Wiesbaden.

[14] David Bernholdt, Shishir Bharathi, David Brown, Kasidit Chanchio, Meili Chen, Ann Chervenak, Luca Cinquini, Bob Drach, Ian Foster, Peter Fox, Jose Garcia, Carl Kesselman, Rob Markel, Don Middleton, Veronika Nefedova, Line Pouchard, Arie Shoshani, Alex Sim, Gary Strand, and Dean Williams. 2005. The earth system grid: Supporting the next generation of climate modeling research. *Proc. IEEE* 93, 3 (March 2005), 485–494.

[15] Robin M. Betz and Ross C. Walker. 2013. Implementing continuous integration software in an established computational chemistry software package, see Reference Carver [27], 68–74.

[16] Charles Billie. 2002. Patterns in scientific software: An introduction. *Comput. Sci. Eng.* 4, 3 (May/June 2002), 48–53.

[17] Stephen M. Blackburn, Amer Diwan, Matthias Hauswirth, Peter F. Sweeney, Jos Nelson Amaral, Vlastimil Babka, Walter Binder, Tim Brecht, Lubomr Bulej, Lieven Eeckhout, Sebastian Fischmeister, Daniel Frampton, Robin Garner, Laurie J. Hendren, Andy Georges, Michael Hind, Antony L. Hosking, Richard E. Jones, Tomas Kalibera, Philippe Moret, Victor Pankratius, Nathaniel Nystrom, and Petr Tuma. 2012. *Can You Trust Your Experimental Results?* Technical Report 1. Evaluate Collaboratory.

[18] Martin Blom. 2012. Is scrum and XP suitable for CSE development? See Reference Sloot et al. [167], 1511–1517.

[19] Ronald F. Boisvertand Ping Tak Peter Tang (Eds.). 2001. *The Architecture of Scientific Software. Proceedings of the IFIP TC2/WG2.5 Working Conference on the Architecture of Scientific Software.* IFIP Advances in Information and Communication Technology, Vol. 60. Springer.

[20] Frederick P. Brooks, Jr. 1995. *The Mythical Man-Month* (9 ed.). Addison Wesley.

[21] Alan C. Calder, Bruce Fryxell, T. Plewa, Robert Rosner, L. J. Dursi, V. G. Weirs, T. Dupont, H. F. Robey, J. O. Kane, B. A. Remington, R. P. Drake, G. Dimonte, M. Zingale, Francis X. Timmes, K. Olson, Paul Ricker, P. MacNeice, and H. M. Tufo. 2002. On validating an astrophysical simulation code. *Astrophys. J. Suppl. Ser.* 143 (November 2002), 201–229.

[22] Jeffrey Carver, Dustin Heaton, Lorin Hochstein, and Roscoe Bartlett. 2013. Self perceptions about software engineering: A survey of scientists and engineers. *Comput. Sci. Eng.* 15, 1 (January/February 2013), 7–11.

[23] Jeffrey C. Carver. 2009. Report: The second international workshop on software engineering for CSE. *Comput. Sci. Eng.* 11, 6 (November/December 2009), 14–19.

[24] Jeffrey C. Carver (Ed.). 2009. *Proceedings of the 2nd International Workshop on Software Engineering for Computational Science, in conjunction with the 2009 IEEE 31st International Conference on Software Engineering (ICSE'09).* IEEE Computer Society.

[25] Jeffrey C. Carver (Ed.). 2010. *Proceedings of the 3rd International Workshop on Software Engineering for Computational Science and Engineering.* Elsevier, Amsterdam, The Netherlands.

[26] Jeffrey C. Carver (Ed.). 2011. *Proceedings of the 4th International Workshop on Software Engineering for Computational Science, in Conjunction with the 2011 IEEE 33rd International Conference on Software Engineering (ICSE'11).* ACM.

[27] Jeffrey C. Carver (Ed.). 2013. *Proceedings of the 5th International Workshop on Software Engineering for Computational Science, in Conjunction with the 2013 IEEE 35th International Conference on Software Engineering (ICSE'13).* IEEE Computer Society.

[28] Jeffrey C. Carver and Neil P. Chue Hong (Eds.). 2016. *Proceedings of the International Workshop on Software Engineering for Science (SE4Science@ICSE'16).* ACM.

[29] Jeffrey C. Carver, Richard P. Kendall, Susan E. Squires, and Douglass E. Post. 2007. Software development environments for scientific and engineering software: A series of case studies. In *Proceedings of the 29th International Conference on Software Engineering*. IEEE Computer Society, Minneapolis, MN, 550–559.

[30] Donatella Castelli, Paolo Manghi, and Costantino Thanos. 2013. A vision towards scientific communication infrastructures—on bridging the realms of research digital libraries and scientific data centers. *Int. J. Dig. Libr.* 13, 3-4 (2013), 155–169.

[31] Parmit K. Chilana, Carole L. Palmer, and Andrew J. Ko. 2009. Comparing bioinformatics software development by computer scientists and biologists: An exploratory study, see Reference Carver [24], 72–79.

[32] Trevor Cickovski, Thierry Matthey, and Jesús A. Izaguirre. 2008. Design patterns for generic object-oriented scientific software, see Reference SE-CSE [155].

[33] Jon Claerbout and Martin Karrenbach. 1992. Electronic documents give reproducible research a new meaning. In *Proceedings of the 1992 Meeting of the Society of Exploration Geophysics.*

[34] Thomas L. Clune and Richard B. Rood. 2011. Software testing and verification in climate model development. *IEEE Softw.* 28, 5 (September/October 2011), 49–55.

[35] CMMI Product Team. 2010. *CMMI for Development.* Technical Report. Software Engineering Institute, Carnegie Mellon.

[36] Christian Colberg, Todd Proebsting, Gina Moraila, Akash Shankaran, Zuoming Shi, and Alex M. Warren. 2013. Measuring Reproducibility in Computer Systems Research. Retrieved from http://reproducibility.cs.arizona.edu/tr.pdf.

[37] Carlton A. Crabtree, A. Güneş Koru, and Carolyn Seaman. 2009. An empirical characterization of scientific software development projects according to the Boehm and Turner model: A progress report, see Reference Carver [24], 22–27.

[38] Daniel J. Crichton, Chris A. Mattmann, Luca Cinquini, Amy Braverman, Duane Waliser, Michael Gunson, Andrew F. Hart, Cameron E. Goodale, Peter Lean, and Jinwon Kim. 2012. Shared satellite observations within the climate-modelling community. *IEEE Softw.* 29, 5 (September/October 2012), 73–81.

[39] Brett Daniel, Danny Dig, Kely Garcia, and Darko Marinov. 2007. Automated testing of refactoring engines. In *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Ivica Crnkovicand Antonia Bertolino (Eds.). ACM Press, Cavtat near Dubrovnik, Croatia, 185–194.

[40] Andrew Davison. 2012. Automated capture of experiment context for easier reproducibility in computational research. *Comput. Sci. Eng.* 14, 4 (2012), 48–56.

[41] Diego Caminha B. de Oliveira, Alan Humphrey, Zvonimir Rakamari, Qingyu Meng, Martin Berzins, and Ganesh Gopalakrishnan. 2013. Practical formal correctness checking of million-core problem solving environments for HPC, see Reference Carver [27], 75–83.

[42] Viktor K. Decyk and Henry J. Gardner. 2008. Object-oriented design patterns in fortran 90/95: Mazev1, mazev2 and mazev3. *Comput. Phys. Commun.* 178, 8 (2008), 611–620.

[43] Viktor K. Decyk, Charles D. Norton, and Bolesaw K. Szymanski. 1997. How to express c++ concepts in fortan90. *Sci. Program.* 6, 4 (1997), 363–390.

[44] Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. 1978. Hints on test data selection: Help for the practicing programmer. *IEEE Comput.* 11, 4 (1978), 34–41.

[45] Shweta Deshpande. 2011. *A Study of Software Engineering Practices for Micro Teams.* Master's thesis. Ohio State University.

[46] Tom Doherty. 2007. Integration of the ATLAS VOMS system with the ATLAS Metadata Interface. (2007). Retrieved from ppewww.physics.gla.ac.uk/preprints/2007/01/.

[47] Anshu Dubey, Katie Antypas, Alan Calder, Bruce Fryxell, Don Lamb, Paul Ricker, Lynn Reid, Katherine Riley, Robert Rosner, Andrew Siegel, Francis Timmes, Natalia Vladimirova, and Klaus Weide. 2013. The software development process of flash, a multiphysics simulation code, see Reference Carver [27], 1–8.

[48] Paul F. Dubois. 2005. Maintaining correctness in scientific programs. *Comput. Sci. Eng.* 7, 3 (May/June 2005), 80–85.

[49] Steve M. Easterbrook and Timothy C. Johns. 2009. Engineering the software for understanding climate change. *Comput. Sci. Eng.* 11, 6 (November/December 2009), 64–75.

[50] Paul N. Edwards. 2010. *A Vast Machine.* MIT Press.

[51] Khaled El Emam and A. Güneş Koru. 2008. A replicated survey of it software project failures. *IEEE Softw.* 25, 5 (September/October 2008), 84–90.

[52] David Falessi and Forrest Shull. 2013. Towards flexible automated support to improve the quality of computational science and engineering software, see Reference Carver [27], 88–91.

[53] Hans Fangohr, Maximilian Albert, and Matteo Franchin. 2016. Nmag micromagnetic simulation tool: Software engineering lessons learned, see Reference Carver and Hong [28], 1–7.

[54] Stuart Faulk, Eugene Loh, Michael L. Van De Vanter, Susan Squires, and Lawrence G. Votta. 2009. Scientific computing's productivity gridlock: How software engineering can help. *Comput. Sci. Eng.* 11, 6 (November/December 2009), 30–39.

[55] Mohamed E. Fayad, Mauri Laitinen, and Robert P. Ward. 2000. Software engineering in the small. *Commun. ACM* 43, 4 (March 2000), 115–118.

[56] Martin Fowler. 2000. *Refactoring: Improving the Design of Existing Code.* Addison Wesley, Pearson Education Inc, One Lake Street, Upper Saddle River, NJ 07458.

[57] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1994. *Design Patterns: Elements of Reusable Object-oriented Software* (1st ed.). Addison Wesley.

[58] Henry Gardner and Gabriele Manduchi. 2007. *Design Patterns for e-Science.* Springer Verlag.

[59] Matan Gavish and David Donoho. 2011. A universal identifier for computational results. *Proc. Comput. Sci.* 4 (2011), 637–647.

[60] Ian P. Gent. 2013. The Recomputation Manifesto. Retrieved from http://recomputation.org/sites/default/files/Manifesto1_9479.pdf (April 2013).

[61] Ian P. Gent, Stuart A. Grant, Ewan MacIntyre, Patrick Prosser, Paul Shaw, Barbara M. Smith, and Toby Walsh. 1997. *How Not To Do It.* Research Report 97.27. School of Computer Studies, University of Leeds.

[62] Robert L. Glass. 1997. *Software Runaways: Lessons Learned from Massive Software Project Failures* (1st ed.). Prentice Hall.

[63] Carole Goble and David De Roure. 2007. myExperiment: Social networking for workflow-using e-scientists. In *Proceedings of the 2nd Workshop on Workflows in Support of Large-scale Science (WORKS'07), June 25, 2007, Monterey, CA*, Ewa Deelmanand Ian Taylor (Eds.). ACM Press, 1–2.

[64] David Goldberg. 1991. What every computer scientist should know about floating point arithmetic. *Comput. Surv.* 23, 1 (1991), 5–48.

[65] John A. Gunnels and Robert A. van de Geijn. 2001. Formal methods for high-performance linear algebra libraries, see Reference Boisvert and Tang [19], 193–210.

[66] Jo Erskine Hannay, Carolyn MacLeod, Janice Singer, Hans Petter Langtangen, Dietmar Pfahl, and Greg Wilson. 2009. How do scientists develop and use scientific software? See Reference Carver [24], 1–8.

[67] Les Hatton. 1997. The t experiments: Errors in scientific software. *IEEE Comput. Sci. Eng.* 4, 2 (1997), 27–38.

[68] Les Hatton. 2007. The chimera of software quality. *IEEE Comput.* 40, 8 (2007), 102–103.

[69] Dustin Heaton and Jeffrey C. Carver. 2015. Claims about the use of software engineering practices in science: A systematic literature review. *Info. Softw. Technol.* 67 (2015), 207–219.

[70] Jette Henderson and Dewayne E. Perry. 2013. Exploring issues in software systems used and developed by domain experts, see Reference Carver [27], 96–99.

[71] Thomas Herndon, Michael Ash, and Robert Pollin. 2013. *Does High Public Debt Consistently Stiffle Economic Growth? A Critique of Reinhart and Rogoff.* Working paper 322. Political Economy Research Institute, University of Massachusetts Amherst, Gordon Hall, 418 North Pleasant Street, Amherst, MA 01002.

[72] Michael A. Heroux, Roscoe A. Bartlett, Vicki E. Howle, Robert J. Hoekstra, Jonathan J. Hu, Tamara G. Kolda, Richard B. Lehoucq, Kevin R. Long, Roger P. Pawlowski, Eric T. Phipps, Andrew G. Salinger, Heidi K. Thornquist, Ray S. Tuminaro, James M. Willenbring, and Alan Williams. 2005. An overview of the Trilinos project. *ACM Trans. Math. Software* 31, 3 (September 2005), 397–423.

[73] Michael A. Heroux and James M. Willenbring. 2009. Barely sufficient software engineering: 10 practices to improve your cse software, see Reference Carver [24], 15–21.

[74] Tony Hey, Stewart Tansley, and Kristin Tolle (Eds.). 2009. *The Fourth Paradigm. Data Intensive Scientific Discovery.* Microsoft Research, Redmond, Washington.

[75] Christopher I. Higgins, Michael Koutroumpas, Richard O. Sinnott, John Watt, Thomas Doherty, Ally C. Hume, Andrew G. D. Turner, and David Rawnsley. 2009. Spatial data e-infrastructure. In *Proceedings of the 5th International Conference on e-Social Science.* Cologne, Germany.

[76] Lorin Hochstein and Victor R. Basili. 2008. The asc-alliance projects: A case study of large-scale parallel scientific code development. *IEEE Comput.* 41, 3 (March 2008), 50–58.

[77] Douglas Hoffman. 1998. Analysis of a taxonomy of test oracles. In *Quality Week, 1998.*

[78] Douglas Hoffman. 1999. Heuristic test oracles. *Softw. Test. Qual. Eng.* 1 (March/April 1999), 29–32.

[79] Daniel Hook and Diane Kelly. 2009. Testing for trustworthiness in scientific software, see Reference Carver [24], 59–54.

[80] Daniel Alan Hook. 2009. *Using Code Mutation to Study Code Faults in Scientific Software.* Master's thesis. Queen's University, Kingston, Ontario, Canada.

[81] House of Commons Science and Technology Committee 2010. The disclosure of climate data from the Climatic Research Unit at the University of East Anglia. Eighth Report of Session 2009-10. *The Stationery Office Limited*, London. Volume I Report, together with formal minutes.

[82] House of Commons Science and Technology Committee 2010. The disclosure of climate data from the Climatic Research Unit at the University of East Anglia. Eighth Report of Session 2009-10. *The Stationery Office Limited*, London. Volume II Oral and written evidence.

[83] William E. Howden. 1982. Validation of scientific programs. *Comput. Surv.* 14, 2 (June 1982), 193–227.

[84] David W. Kane. 2003. Introducing agile development into bioinformatics: An experience report. In *Proceedings of the Agile Development Conference.* IEEE Computer Society, 132–139.

[85] David W. Kane, Moses M. Hohman, Ethan G. Cerami, Michael W. McCormick, Karl F. Kuhlmman, and Jeff A. Byrd. 2006. Agile methods in biomedical software development: A multi-site experience report. *BMC Bioinfo.* 7 (2006), 723.

[86] Upulee Kanewala and James M. Bieman. 2013. Techniques for testing scientific programs without an oracle, see Reference Carver [27].

[87] Upulee Kanewala and James M. Bieman. 2014. Testing scientific software: A systematic literature review. *Info. Softw. Technol.* 56 (2014), 1219–1232.

[88] Diane Kelly. 2015. Scientific software development viewed as knowledge acquisition: Towards understanding the development of risk-averse scientific software. *J. Syst. Softw.* 109 (2015), 50–61.

[89] Diane Kelly, Robert Gray, and Yizhen Shao. 2011. Examining random and designed tests to detect code mistakes in scientific software. *J. Comput. Sci.* 2 (2011), 47–56.

[90] Diane Kelly and Daniel Hook. 2011. Scientific software testing: Analysis with four dimensions. *IEEE Softw.* 28, 3 (May/June 2011), 84–90.

[91] Diane Kelly, Daniel Hook, and Rebecca Sanders. 2009. Five recommended practices for computational scientists who write software. *Comput. Sci. Eng.* 11, 5 (September/October 2009), 48–53.

[92] Diane Kelly and Rebecca Sanders. 2008. Assessing the quality of scientific software, see Reference SE-CSE [155].

[93] Diane Kelly and Terry Shepard. 2004. Task-directed software inspection. *J. Syst. Softw.* 73, 2 (October 2004), 361–368.

[94] Diane Kelly, Spencer Smith, and Nicholas Meng. 2011. Software engineering for scientists. *Comput. Sci. Eng.* 13, 5 (September/October 2011), 7–10.

[95] Diane F. Kelly. 2007. A software chasm: Software engineering and scientific computing. *IEEE Softw.* 24, 5 (November/December 2007), 118–120.

[96] Richard Kendall, Jeffrey C. Carver, Andrew Mark, Douglas Post, Clifford E. Rhoades, and Susan Squires. 2008. Developing a weather forecasting code: A case study. *IEEE Softw.* 25, 4 (July/August 2008), 59–65.

[97] Richard P. Kendall, Douglass E. Post, Chris A. Atwood, Kevin P. Newmeyer, Lawrence G. Votta, Paula A. Gibson, Deborah L. Borovitcky, Loren K. Miller, Robert L. Meakin, Miles M. Hurwitz, Saikat Dey, John N. D'Angelo, Richard L. Vogelsong, Oscar A. Goldfarb, and Sunita B. Allwerdt. 2016. A risk-based, practice-centered approach to project management for HPCMP CREATE. *Comput. Sci. Eng.* 18, 1 (January/February 2016), 40–51.

[98] Joshua Kerievsky. 2004. *Refactoring to Patterns.* Addison-Wesley.

[99] Sarah Killcoyne and John Boyle. 2009. Managing chaos: Lessons learned developing software in the life sciences. *Comput. Sci. Eng.* 11, 6 (November/December 2009), 20–29.

[100] Donald Knuth. 1984. Literate programming. *Comput. J.* 27, 2 (1984), 97–111.

[101] Konstantin Kreyman, David Lorge Parnas, and Sanzheng Qiao. 1999. *Inspection Procedures for Critical Programs that Model Physical Phenomena.* CRL Report 368. Department of Computing and Software, Faculty of Engineering, McMaster University, Hamilton, Ontario, Canada.

[102] John Krogstie, Arthur Jahr, and Dag I. K. Sjøberg. 2006. A longitudinal study of development and maintenance in Norway: Report from the 2003 investigation. *Info. Softw. Technol.* 48, 11 (2006), 993–1005.

[103] Sotiria Lampoudi. 2012. *The Path to Virtual Machine Images as First Class Provenance.* Technical Report 2012-05. Department of Computer Science, University of California Santa Barbara.

[104] Randall J. LeVeque, Ian M. Mitchell, and Victoria Stodden. 2014. Reproducible research for scientific computing: Tools and strategies for changing the culture. *Comput. Sci. Eng.* 14, 4 (2014), 13–17.

[105] Yang Li. 2011. Reengineering a scientific software and lessons learned, see Reference Carver [26], 41–45.

[106] Anders Lundgren and Upulee Kanewala. 2016. Experiences of testing bioinformatics programs for detecting subtle faults, see Reference Carver and Hong [28], 16–22.

[107] Donald MacKenzie. 2001. *Mechanizing Proof: Computing, Risk and Trust.* The MIT Press, Massachusetts Institute of Technology, Cambridge, Massachusetts 02142.

[108] Arjen Markus. 2006. Design patterns and fortran 90/95. *ACM Fortran Forum* 25, 1 (2006), 13–29.

[109] Arjen Markus. 2008. Design patterns and fortran 2003. *ACM Fortran Forum* 27, 3 (2008), 2–15.

[110] David Matthews, Greg Wilson, and Steve Easterbrook. 2008. Configuration management for large-scale scientific computing at the UK met office. *Comput. Sci. Eng.* 10 (2008), 56–64.

[111] Chris A. Mattmann, Daniel J. Crichton, Nenad Medvidovic, and Steve Hughes. 2006. A software architecture based framework for highly distributed and data intensive scientific applications, Leon J. Osterweil, H. Dieter Rombach, and Mary Lou Soffa (Eds.). ACM Press, Shanghai, China, 721–730.

[112] B. D. McCullough, Kerry Anne McGeary, and Teresa D. Harrison. 2006. Lessons from the JMCB archive. *J. Money Credit Bank.* 38, 4 (June 2006), 1093–1107.

[113] Zeeya Merali. 2010. Error. Why scientific computing does not compute. *Nature* 467 (October 2010), 777–779.

[114] Erika S. Mesh. 2015. Supporting scientific SE process improvement. In *Proceedings of the 37th International Conference on Software Engineering—Volume 2.* IEEE Press, Piscataway, NJ, 923–926.

[115] Erika S. Mesh and J. Scott Hawker. 2013. Scientific software process improvement decisions: A proposed research strategy, see Reference Carver [27], 32–39.

[116] Greg Miller. 2006. Scientific publishing: A scientist's nightmare: Software problem leads to five retractions. *Science* 314, 5807 (2006), 1856–1857.

[117] K. Jarrod Millman and Fernando Pérez. 2014. Developing open source scientific practice. In *Implementing Reproducible Research*, Victoria Stodden, Friedrich Leisch, and Roger D. Peng (Eds.). Chapman and Hall/CRC Press, 149–183.

[118] Chris Morris. 2008. Some lessons learned reviewing scientific code, see Reference SE-CSE [155].

[119] Hoda Naguib and Yang Li. 2012. (Position paper) applying software engineering methods and tools to CSE Research Projects, see Reference Sloot et al. [167], 1505–1509.

[120] Peter Naurand Brian Randell (Eds.). 1968. *Report on a Conference Sponsored by the NATO Science Committee.* Garmisch, Germany.

[121] Thomas Naylor and J. M. Finger. 1967. Verification of computer simulation models. *Manage. Sci.* 14, 2 (October 1967).

[122] Nedialko S. Nedialkov. 2011. Implementing a rigorous ODE solver through literate programming. In *Modeling, Design, and Simulation of Systems with Uncertainties*, Andreas Rauhand Ekaterina Auer (Eds.). Springer, 3–19.

[123] Vitor C. Neves, Vanessa Braganholo, and Leonardo Murta. 2013. Implicit provenance gathering through configuration management, see Reference Carver [27], 92–95.

[124] Luke Nguyen-Hoan, Shayne Flint, and Ramesh Sankaranayana. 2010. A survey of scientific software development. In *Proceedings of the International Symposium on Empirical Software Engineering and Measurement (ESEM'10)*, Giancarlo Succi, Maurizio Morisio, and Nachiappan Nagappan (Eds.). ACM Press, Bolzano-Bozen, Italy.

[125] Tom Nielsen, Tom Matheson, and Henrik Nilsson. 2011. Braincurry: A domain–specific language for integrative neuroscience. In *Trends in Functional Programming Volume 10*, Zoltán Horváth, Viktória Zsók, Peter Achten, and Pieter Koopman (Eds.). Intellect, The Mill, Parnall Road, Fishponds, Bristol, BS16 3JG, Chapter 11, 161–176.

[126] Charles D. Norton, Viktor K. Decyk, Boleslaw K. Szymanski, and Henry Gardner. 2007. The transition and adoption of modern programming concepts for scientific computing in Fortran. *Sci. Program.* 15, 1 (Spring 2007), 1–27.

[127] Piotr Nowakowskia, Eryk Ciepielaa, Daniel Harężlaka, Joanna Kocota, and Marek Kasztelnika, Tomasz Bartyńskia, Jan Meiznera, Grzegorz Dyka, and Maciej Malawski. 2011. The collage authoring environment. *Proc. Comput. Sci.* 4 (2011), 608–617.

[128] William L. Oberkampf, Martin Pilch, and Timothy G. Trucano. 2007. *Predictive Capability Maturity Model for Computational Modeling and Simulation.* Technical Report SAND2007-5948. Sandia National Laboratories, Albuquerque, New Mexico 97185 and Livermore, California.

[129] Emil Obreshkov. 2010. Software release build process and components in Atlas offline. In *Proceedings of the Conference on Computing in High Energy and Nuclear Physics 2010.*

[130] Tom Oinn, Mark Greenwood, Matthew Addis, M. Nedim Alpdemir, Justin Ferris, Kevin Glover, Carole Goble, Antoon Goderis, Duncan Hull, Darren Marvin, Peter Li, Phillip Lord, Matthew R. Pocock, Martin Senger, Robert Stevens, and Anil Wipat and Chris Wroe. 2006. Taverna: Lessons in creating a workflow environment for the life sciences.*Concurr. Comput.: Pract. Exp.* 18, 10 (August 2006), 1067–1100.

[131] Jeffrey L. Overbey, Stas Negara, and Ralph E. Johnson. 2009. Refactoring and the evolution of Fortran, see Reference Carver [24], 28–34.

[132] Michael Pernice. 2013. *Survey of Software Quality Assurance and Code Verification Practices in CASL.* Technical Report CASL-U-2013-0079-000. Idaho National Laboratory, Oak Ridge Laboratory.

[133] Jon Pipitone and Steve Easterbrook. 2012. Assessing climate model software quality: A defect density analysis of three models. *Geosci. Model Dev.* 5 (2012), 1009–1022.

[134] Joe Pitt-Francis, Miguel O. Bernabeu, Jonathan Cooper, Alan Garny, Lee Momtahan, James Osborne, Pras Pathmanathan, Blanca Rodriguez, Jonathan P. Whiteley, and David J. Gavaghan. 2008. Chaste: Using agile programming techniques to develop computational biology software. *Philos. Trans. Ser. A, Math. Phys. Eng. Sci.* 366, 1878 (September 2008), 3111–3136.

[135] Karl Popper. 2005. *The Logic of Scientific Discovery.* Routledge, 11 New Fetter Lane, London, EC4P 4EE.

[136] Douglas E. Post. 2008. A new DoD initiative: The computational research and engineering acquisition tools and environment (CREATE) program. *J. Phys.: Conf. Ser.* 125 (2008), 012090.

[137] D. E. Post and R. P. Kendall. 2004. Software project management and quality assurance practices for complex, coupled, multiphysics, massively parallel computational simulations: Lessons learned from ASCI. *Int. J. High Perform. Comput. Appl.* 18, 4 (2004), 399–416.

[138] Douglass E. Post and Lawrence G. Votta. 2005. Computational science demands a new paradigm. *Phys. Today* 58, 1 (January 2005), 35–40.

[139] Patrick Prosser and Chris Unsworth. 2010. Limited Discrepancy Search: Revisited (2010). Private communication.

[140] H. M. Quiney and S. Wilson. 2005. Literate programming in quantum chemistry: A simple example. *Int. J. Quant. Chem.* 104, 4 (2005), 430–445.

[141] James J. Quirk. 2005. Computational science "Same old silence, same old mistakes" "Something more is needed." In *Adaptive Mesh Refinement—Theory and Applications. Proceedings of the Chicago Workshop on Adaptive Mesh Refinement Methods, Sept. 3–5, 2003* (Lecture Notes in Computational Science and Engineering), Tomasz Plewa, Timur Linde, and V. Gregory Weirs (Eds.), Vol. 41. Springer Verlag, 3–28.

[142] Krister Ahlander, Magne Haveraaen, and Hans Z. Munthe-Kaas. 2001. On the role of mathematical abstractions for scientific computing, see Reference Boisvert and Tang [19], 145–158.

[143] Thomas Reichler and Junsu Kim. 2008. How well do coupled models simulate today's climate. *Bull. Am. Meteorol. Soc.* 89, 3 (March 2008), 303–311.

[144] Carmen M. Reinhart and Kenneth S. Rogoff. 2010. Growth in a time of debt. *Am. Econ. Rev.: Papers Proc.* 100 (May 2010), 573–578.

[145] Hanna Remmel, Barbara Paech, Christian Engwer, and Peter Bastian. 2013. Design and rationale of a quality assurance process for a scientific framework, see Reference Carver [27], 58–67.

[146] Christophe René, Thierry Priol, and Guillaume Alléon. 2001. Code coupling using parallel Corba objects, see Reference Boisvert and Tang [19], 105–118.

[147] John Rice and Ronald F. Boisvert. 1996. From scientific software libraries to problem solving environments. *IEEE Comput. Sci. Eng.* 3 (1996), 44–53.

[148] Patrick Roache. 2004. Building PDE codes to be verifiable and validatable. *Comput. Sci. Eng.* 6, 5 (September/October 2004), 30–38.

[149] Damian W. I. Rouson, Helgi Adalsteinsson, and Jim Xia. 2010. Design patterns for multiphysics modeling in Fortran 2003 and C++. *ACM Trans. Math. Software* 37, 1 (January 2010), Article 3.

[150] Damian W. I. Rouson, Jim Xia, and Xiaofeng Xu. 2012. Object construction and destruction design patterns in Fortran 2003, see Reference Sloot et al. [167], 1495–1504.

[151] Christopher J. Roy and William L. Oberkampf. 2011. A comprehensive framework for verification, validation, and uncertainty quantification in scientific computing. *Comput. Methods Appl. Mech. Eng.* 200, 25 (June 2011), 2131–2144.

[152] Royal Swedish Academy of Sciences. 2013. Scientific Background on the Nobel Prize in Chemistry 2013. Development of Multiscale Models for Complex Chemical Systems. (October 2013). Retrieved from http://www.nobelprize.org/nobel_prizes/chemistry/laureates/2013/advanced-chemistryprize2013.pdf.

[153] Rebecca Sanders and Diane Kelly. 2008. Dealing with risk in scientific software development. *IEEE Comput. Softw.* 25 (2008). Issue 4.

[154] Ken Schwaber and Mike Beedle. 2001. *Agile Software Development with SCRUM.* Prentice Hall.

[155] SE-CSE 2008. *Proceedings of the 1st International Workshop on Software Engineering for Computational Science and Engineering.* Leipzig Germany.

[156] Judith Segal. 2005. When software engineers met research scientists: A case study. In *Empirical Software Engineering.* Springer Verlag, 517–536.

[157] Judith Segal. 2008. Models of scientific software development, see Reference SE-CSE [155].

[158] Judith Segal. 2009. Software development cultures and cooperation problems: A field study of the early stages of development of software for a scientific community. *Comput. Support. Coop. Work (CSCW'09)* 18, 5–6 (2009), 581–606.

[159] Judith Segal. 2009. Some challenges facing software engineers developing software for scientists, see Reference Carver [24], 9–14.

[160] Judith Segal and Chris Morris. 2008. Developing scientific software. *IEEE Comput. Softw.* 25 (July/August 2008), 18–20. Issue 4.

[161] Simon Shackley, Peter Young, Stuart Parkinson, and Brian Wynne. 1998. Uncertainty, complexity and concepts of good science in climate change modelling: Are GCMS the best tool? *Climate Change* 38 (1998), 159–205.

[162] A. Shaon, Jim Woodcock, and E. Conway. 2009. *Tools and Guidelines for Preserving and Accessing Software as a Research Output Report II: Case Studies.* Technical Report. The University of York.

[163] Forrest Shull. 2011. Assuring the future? A look at validating climate model software. *IEEE Softw.* 28, 5 (September/October 2011), 4–8.

[164] Jeremy Singer. 2011. A literate experimentation manifesto. In *Proceedings of the ACM Symposium on New Ideas in Programming and Reflections on Software, Onward! 2011, part of SPLASH'11*, Robert Hirschfeldand Eelco Visser (Eds.). ACM Press, Portland, OR, 91–102.

[165] Magnus Thorstein Sletholt, Jo Hannay, Dietmar Pfahl, Hans Christian Benestad, and Hans Petter Langtangen. 2011. A literature review of agile practices and their effects in scientific software development, see Reference Carver [26], 1–9.

[166] Magnus Thorstein Sletholt, Jo Erskine Hannay, Dietmar Pfahl, and Hans Petter Langtangen. 2012. What do we know about scientific software development's agile practices?*Comput. Sci. Eng.* 14, 2 (March/April 2012), 24–36.

[167] Peter M. A. Sloot, G. Dick van Albada, and Jack Dongarra (Eds.). 2012. *Proceedings of the International Conference on Computational Science (ICCS'10).* Procedia Computer Science, Vol. 1. Elsevier, University of Amsterdam, The Netherlands.

[168] Spencer Smith, Lei Lai, and Ridha Khedri. 2007. Requirements analysis for engineering computation: A systematic approach for improving reliability. *Reliable Comput.* 13, 1 (2007), 83–107.

[169] W. Spencer Smith, Jacques Carette, and John McCutchan. 2008. Commonality analysis of families of physical models for use in scientific computing, see Reference SE-CSE [155].

[170] Diomidis Spinellis and Henry Spencer. 2011. Lessons from space. *IEEE Softw.* 28, 5 (September/October 2011).

[171] Nicholas Stern. 2007. *The Economics of Climate Change. The Stern Review. Cambridge University Press.*

[172] Victoria Stodden, Peixuan Guo, and Zhaokun Ma. 2013. Toward reproducible computational research: An empirical analysis of data and code policy adoption by journals. *PLOS One* 8, 6 (June 2013), e67111.

[173] Victoria Stodden and Sheila Miguez. 2013. Best practices for computational science: Software infrastructure and environments for reproducible and extensible research. *J. Open Res. Softw.* 2, 1 (2013), e21.

[174] Michael Stonebraker, Daniel Bruckner, Ihab F. Ilyas, George Beskales, Mitch Cherniack, and Stan Zdonik. 2013. Data curation at scale: The data tamer system. In *Proceedings of the 6th Biennial Conference on Innovative Data Systems Research (CIDR'13). Online Proceedings.* www.cidrdb.org, Asilomar, CA.

[175] Dan Szymczak, Spencer Smith, and Jacques Carette. 2016. A knowledge-based approach to scientific software development: Position paper, see Reference Carver and Hong [28], 23–26.

[176] Ken Thompson. 1984. Reflections on trusting trust. *Commun. ACM* 27, 8 (August 1984), 761–763.

[177] Michael Thuné, Krister øAhlander, Malin Ljungberg, Markus Nordén, Kurt Otto, and Jarmo Rantakokko. 2001. Object-oriented modeling of parallel PDE solvers, see Reference Boisvert and Tang [19], 159–174.

[178] T. G. Trucano, Douglass E. Post, M. Pilch, and W. L. Oberkampf. 2005. *Software Engineering Intersections with Verification and Validation (V&V) of High Performance Computational Science Software: Some Observations.* Technical Report SAND2005-3662P. Sandia National Laboratories, PO Box 5800, Albuquerque, New Mexico.

[179] Medha Umarji, Carolyn Seaman, A. Gunes Koru, and Hongfang Liu. 2009. Software engineering education for bioinformatics. In *Proceedings of the 2009 22nd Conference on Software Engineering Education and Training.* IEEE Computer Society, 216–223.

[180] Arie van Deursen, Paul Klint, and Joost Visser. 2000. Domain-specific languages: An annotated bibliography. *ACM SIGPLAN Notices* 35, 6 (June 2000), 26–36.

[181] Pieter van Gorp and Steffen Mazanek. 2011. share: A web portal for creating and sharing executable research papers. *Proc. Comput. Sci.* 4 (2011), 589–597.

[182] Mark Vigder, Darlene Stewart, and Janice Singer. 2008. Software automation in scientific research organizations, see Reference SE-CSE [155].

[183] Alexey Voinov and Herman H. Shugart. 2013. "Integronsters," integral and integrated modeling. *Environ. Model. Softw.* 39 (2013), 149–158.

[184] Dali Wang, Tomislav Janjusic, Colleen Iversen, Peter E. Thornton, Misha Karssovski, Wei Wu, and Yang Xu. 2015. A scientific function test framework for modular environmental model development: Application to the community land model. In *Proceedings of the 1st IEEE/ACM International Workshop on Software Engineering for High Performance Computing in Science (SE4HPCS'15), 2015*, Jeffrey Carver, Paolo Ciancarini, and Neil Chue Hong (Eds.). IEEE Computer Society, 16–23.

[185] Richard Y. Wang and Diane M. Strong. 1996. Beyond accuracy, what data quality means to data consumers. *J. Manage. Info. Syst.* 12, 4 (Spring 1996), 5–33.
[186] Elaine J. Weyuker. 1982. On testing non-testable programs. *Comput. J.* 25, 4 (1982), 465–470.
[187] Greg Wilson, D. A. Aruliah, C. Titus Brown, Neil P. Chue Hong, Matt Davis, Richard T. Guy, Steven H. D. Haddock, Kathryn D. Huff, Ian M. Mitchell, Mark D. Plumbley, Ben Waugh, Ethan P. White, and Paul Wilson. 2014. Best practices in scientific computing. *PLOS Biol.* 12, 1 (January 2014), 1–7.
[188] Greg Wilson and Andrew Lumsdaine. 2006. Software carpentry getting scientists to write better code by making them more productive. *Comput. Sci. Eng.* 8, 6 (November/December 2006), 66–69.
[189] Greg Wilson and Andrew Lumsdaine. 2009. Software engineering and computational science. *Comput. Sci. Eng.* 11, 6 (November/December 2009), 12–13.
[190] Marianne Winslett and Vanessa Braganholo. 2012. Erich Neuhold speaks out. *SIGMOD Rec.* 41, 2 (June 2012), 37–46.
[191] William A. Wood and William L. Kleb. 2003. Exploring XP for scientific research. *IEEE Softw.* 20, 3 (May/June 2003), 30–36.
[192] David Woollard, Chris Mattmann, and Nenad Medvidovic. 2009. Injecting software architectural constraints into legacy scientific applications, see Reference Carver [24], 65–71.
[193] Yale Law School Roundtable on Data and Code Sharing. 2010. Reproducible research. *Comput. Sci. Eng.* 12, 5 (September/October 2010), 8–12.
[194] Wen Yu and Spencer Smith. 2009. Reusability of FEA software: A program family approach, see Reference Carver [24], 43–50.

Received August 2015; revised April 2017; accepted April 2017