Software Engineering for Computational Science: Past, Present, Future

Arne N. Johanson, XING Marketing Solutions GmbH
Wilhelm Hasselbring, Kiel University

Abstract

While the importance of *in silico* experiments for the scientific discovery process increases, state-of-the-art software engineering practices are rarelyadopted in computational science. To understand the underlying causes for this situation and to identify ways for improving the current situation, we conduct a literature survey on software engineering practices in computational science. As a result of our survey, we identified 13 recurring key characteristics of scientific software development that can be divided into three groups: characteristics that results (1) from the *nature of scientific challenges*, (2) from *limitations of computers*, and (3) from the *cultural environment* of scientific software development. Our findings allow us to point out shortcomings of existing approaches for bridging the gap between software engineering and computational science and to provide an outlook on promising research directions that could contribute to improving the current situation.

1 Introduction

With the constantly increasing capabilities of modern computers, *in silico* experiments are becoming more complex and play a more and more important role in the scientific discovery process (Post, 2013). As a consequence, the complexity and lifespan of scientific software is growing as well as the necessity for its output to be reproducible and verifiable. This increases the importance of employing sound software engineering practices in the development of scientific software to guarantee reliable and accurate scientific results. However, surveys show that state-of-the-art software engineering methods are rarely adopted in computational science (Heaton and Carver, 2015; Prabhu et al., 2011). In order to understand the underlying causes for this and to identify ways for improving the current situation, in this paper, we survey literature on software engineering in computational science and identify key characteristics that are unique to scientific software development.

To provide a basis for our survey, we outline the historical development of the relationship between the disciplines of software engineering and computational science (Section 2, describing the **Past**). This relationship is, to a large extent, characterized by an isolation between the two disciplines which resulted in a productivity and credibility crisis of computational science. Based on this viewpoint, we review about 50 publications on case studies and surveys conducted among computational scientists to identify 13 key characteristics of

Preprint of: A. Johanson, W. Hasselbring: "Software Engineering for Computational Science: Past, Present, Future", In: Computing in Science & Engineering, pp. 90-109, March/April 2018.
https://doi.org/10.1109/MCSE.2018.108162940

scientific software development that explain why state-of-the-art software engineering techniques are poorly adopted in computational science (Section 3, portraying the **Present**). The findings of our literature survey allow us to identify shortcomings of existing approaches for bridging the gap between software engineering and computational science and to provide an outlook on promising research directions that could contribute to improving the current situation (Section 4, indicating the **Future**). Concluding remarks are given in Section 5.

2 Software Engineering and Computational Science: Review of the Historical Development

When software engineers started to examine the software development practice in computational science, they noticed a "wide chasm" (Hannay et al., 2009) between how these two disciplines view software development. Faulk et al. (2009) describe this chasm between the two subjects using an allegory which depicts computational science as an isolated island that has been colonized but then was left abandoned for decades:

> "Returning visitors (software engineers) find the inhabitants (scientific programmers) apparently speaking the same language, but communication — and thus collaboration — is nearly impossible; the technologies, culture, and language semantics themselves have evolved and adapted to circumstances unknown to the original colonizers."

The fact that these two cultures are "separated by a common language" created a communication gap that inhibits knowledge transfer between them. As a result, modern software engineering practices are rarely employed in computational science.

2.1 The Origins of the Chasm

The origins of the rift between computational science and software engineering can be traced back as far as the dawn of modern computing in the 1940s. At that time, "scientific computing" was a pleonasm: the electronic digital computer was invented solely to solve complex mathematical problems for the advancement of science and engineering (Ceruzzi, 2003). As the discipline of computer science emerged in the late 1950s and early 60s, it struggled to distinguish itself from electrical engineering and applied mathematics — the disciplines traditionally engaged in the "study of computers" (Newell et al., 1967). To differentiate itself from these applied disciplines, computer science invented the "stigma of all things 'applied'" and aimed for generality in all its methods and techniques (Vessey, 1997). This approach was supposed to ensure that "the core of computer science [...] will remain a field of its own, ahead of, and separate from the application domain specialists" (George E. Forsyth, President of ACM in 1965, as quoted by Vessey (1997)). This first estrangement of computer science from the field that would later on become computational

science was adopted and even reinforced by software engineering.

The term software engineering—and at the same time the corresponding subdiscipline of computer science—was institutionalized by a conference with the aforementioned title organized by NATO in 1968 (Naur and Randell, 1969). That NATO was the sponsor of this conference marks the relative distance of software engineering from computation in the academic context. The perception was that while errors in scientific data processing applications may be a "hassle," they are all in all tolerable. In contrast, failures in mission-critical military systems may cost lives and substantial amounts of money (Ceruzzi, 2003, chap. 3).

Based on this attitude, software engineering—like computer science as a whole— aimed for generality in its methods, techniques, and processes and focused almost exclusively on business and embedded software (Kelly, 2007). Because of this ideal of generality, the question of how specifically computational scientists should develop their software in a well-engineered way, would probably have perplexed a software engineer and the answer might have been: "Well, just like any other application software."

For some time, the coexistence of computational science and software engineering in relative ignorance of one another continued without greater interruptions. One noteworthy exception from this is a paper from Hatton and Roberts (1994) in which they examine the accuracy of over 15 large commercial software libraries of numerical algorithms. Their findings disagree with the typical assumption of computational scientists that their software is accurate to the precision of the machine arithmetic. Hatton and Roberts discovered that in their sample the numerical discrepancy between expected and computed results increases about 1 % per 4000 Lines of Code (LOC). These results, however, did at first not find a larger echo in both communities.

2.2 The Productivity Crisis of Computational Science

In general, computational scientists did not see a reason to be concerned about the quality of their software. This attitude started to change roughly ten years ago, when more and more deficiencies regarding the productivity of scientific software development became apparent. These deficiencies, which led some to speak of a "productivity gridlock" (Faulk et al., 2009) in computational science, were revealed mostly by two parallel developments:

1. The encounter of the clock speed limit for single-core processors and, with it, the introduction of multi-core and heterogeneous, distributed computing systems.

2. The integration of more and more effects into scientific simulations that govern the behavior of the system under study.

Scientific software has a quite long life span (from years to decades) as it often encapsulates the accumulated knowledge and effort of scientists or research groups. Thus, it typically outlives the computing hardware for which it was originally designed. For more than two decades this was not a problem as

it could be taken for granted that each new generation of microprocessors would considerably increase the performance of the scientists' software without greater modifications of the source code. With the encounter of the clock speed limit for single-core processors around the year 2004, this development came to a halt (Fuller and Millett, 2011).

In order to achieve more processing power, chip designers started to scale the number of processor compute cores. At first, this was only realized for the Central Processing Unit (CPU) of the computer (multi-core era). The most recent step was to assist such multi-core CPUs by providing external accelerator devices with even greater numbers (on the order of magnitudes!) of compute cores that are inspired by the design of modern graphics processors (heterogeneous systems era). To harness the power that multi-core and especially heterogeneous systems provide for more and more detailed simulations, computational scientists are now faced with the challenge of adapting their software to exploit parallelism on ever-finer levels of granularity. The rapid push to massive distributed parallel machines in the late 1990's caused similar productivity problems. A problem revealed in the course of this process is a lack of knowledge about software engineering among the scientists, which resulted in a poor maintainability of their "codes." This lack of maintainability impedes the scientists' ability to successfully scale their simulations through adaptation to new hardware architectures (Dongarra et al., 2007; Buttari et al., 2007).

The second development that prevents computational scientists from ignoring modern software engineering techniques if they want to stay—or become again—productive is related to the complexity of their models. For computation to fulfill its role as the "third pillar of scientific inquiry" (Benioff et al., 2005), the scientists have to increase the predictive capabilities of their models by integrating more and more scientific effects into them. This results in the need for coupling or even integrating contributions from a multidisciplinary team of scientists into a single simulation application. As earlier scientific software was developed by small teams of scientists primarily for their own research, modularity, maintainability, and team coordination could often be neglected without a large impact. The shift towards larger, interdisciplinary teams makes these often ignored aspects of software development important for the scientists and, again, exposes a knowledge gap among them that can only be overcome by engaging in a dialog with software engineering (Post, 2013).

2.3 The Credibility Crisis of Computational Science

The challenges regarding the quality of scientific software do not only lead to a decreased development performance but also interfere with the credibility of its results. This aspect becomes especially important as the societal impact of computer simulations has grown in recent times, which can be exemplified by the so-called "Climategate" scandal. The scandal erupted after hackers leaked the e-mail correspondence of scientists from the Climatic Research Unit at the University of East Anglia not long before the 2009 United Nations Climate Change Conference. While the accusations that data was forged for this conference turned out to be unfounded, the e-mails uncovered lacking

programming skills among the researchers and exposed to a large public audience the widely applied practice in climate science to *not* release simulation code and data together with corresponding publications (Merali, 2010). This in itself was, of course, enough to undermine the work of the scientists, as the predictive capabilities of simulations are only as good as their code quality (Hatton and Roberts, 1994) and their code was not even available for peer review—not to mention public review.

Within the scientific community the "Climategate" scandal initiated a debate about the *reproducibility* of computational results that also attracted some attention of the software engineering community (for a discussion of this debate, see Section 4).

2.4 Bridging the Gap

Both the productivity and the credibility crisis, make it abundantly clear that the isolated coexistence of computational science and software engineering cannot continue. Problems with programming and the design of scientific software should not be dismissed as "just a hassle" anymore in order for computational science to advance and to keep its promise of fulfilling the role of a third paradigm for scientific discovery. But even though the events of the past decade have initiated a dialog between software engineering and computational science, progress is still slow. For example, Brown et al. (2015) in a recent publication give a quite sarcastic description of the current status of scientific software and the associated development practices: imagine scientific software as a web browser with no URL entry box—you enter the web address into a configuration file. The browser can use either http or https but not both at the same time, which is controlled by an #ifdef switch requiring you to recompile the browser with the second to last version of a Fortran77 compiler by a specific vendor. In principle, you could change all that as the software is open source but, unfortunately, development is private and you would have to apply to be granted access to the source code, which you will only gain if your intentions are in agreement with those of the main developers.

While such design choices seem absurd to us for modern-day applications, Brown et al. conclude that they "represent the status quo in many scientific software packages" and are often "vehemently defended." However, the mistrust of computational science towards modern software engineering techniques is not totally ungrounded: as software engineering aimed for generality in all its methods and processes, it ignored the unique demands of computational science (Kelly, 2007). Therefore, scientists far too often experienced the methodological offerings of software engineering as being full of "accidental complexities" instead of being helpful (Wilson, 2006). Accordingly, distrust and prejudices are still regularly found on both sides of the "software chasm" that so far has not been closed up again (Storer, 2017).

In order to understand which approaches might be suitable to bridge the gap between the two disciplines, we have to closely examine the characteristics of scientific software development and must take the distinctive requirements of computational science seriously. Only if we—from the perspective of software engineering—abandon the "stigma of all things 'applied'" (Vessey, 1997) and end

the unconditional striving for generality that does not do computational science any justice, can we hope to improve the current situation.

3 Characteristics of Scientific Software Development

In this section, we survey literature from the software engineering community that examines the characteristics of scientific software development. Literature on this topic emerged only after an influential article by Post and Votta (2005), who found that the relatively new discipline of computational science was still "troublingly immature." The topic was investigated mainly by conducting case studies and a few survey studies. Reviewing and integrating the observations of these different studies allows us to reduce the major risk that is commonly associated with case studies: their lack of generalizability. Combining and contrasting the findings of multiple studies in different environments makes it possible to identify a set of characteristics that is likely to be inherent to scientific software development in general. Although the majority of the literature on software development practices in computational science dates back to the years 2006 to 2009, the observations made by them appear to still hold true today, as is indicated by related newer publications (e.g., Brown et al., 2015; Carver and Epperly, 2014; Joppa et al., 2013).

The papers included into our literature survey were identified by querying databases like the ACM Digital Library,[1] the IEEE Computer Society Digital Library,[2] and Google Scholar.[3] Additionally, we searched the articles of journals we expected to be of specific interest, such as *Computing in Science & Engineering* as well as the proceedings of conferences and workshops like the *International Conference on Software Engineering* and the *International Workshop on Software Engineering for Computational Science and Engineering* from 2005 on. Some papers were suggested by peers or identified by references from other articles. A limitation of this strategy is that we necessarily have to rely on a limited number of keywords in our database queries. We tried to mitigate this risk by varying the search phrases and, e.g., using different synonyms (such as scientific computing for computational science etc.).

Since the variety of scientific software and its applications is large (Segal and Morris, 2008), computational scientists do not form a homogeneous group. Scientists develop software ranging from scripts for small-scale data analysis to complex coupled multi-physics simulations executed on high-end hardware. In our literature survey, we focus mostly—though not exclusively—on the latter group which forms the High Performance Computing (HPC) community. The reason for directing our attention to this group is that it is most affected by the productivity and credibility crisis portrayed in the previous section.

As a result of our literature survey, we identified 13 recurring key characteristics of scientific software development that can be divided into three groups:

[1] http://dl.acm.org
[2] http://www.computer.org/csdl
[3] https://scholar.google.com

1. Characteristics resulting from the *nature of scientific challenges*:

   1.a) Requirements are not known up front

   1.b) Verification and validation is difficult and strictly scientific

   1.c) Overly formal software processes restrict research

2. Characteristics resulting from the *limitations of computers*:

   2.a) Development is driven and limited by hardware

   2.b) Use of "old" programming languages and technologies

   2.c) Intermingling of domain logic and implementation details

   2.d) Conflicting software quality requirements (performance, portability, and maintainability)

3. Characteristics resulting from the *cultural environment* of scientific software development:

   3.a) Few scientists are trained in software engineering

   3.b) Different terminology

   3.c) Scientific software in itself has no value but still it is long-lived

   3.d) Creating a shared understanding of a "code" is difficult

   3.e) Little code re-use

   3.f) Disregard of most modern software engineering methods

In the following subsections, we detail our findings with regard to the three groups of key characteristics named above and describe how software engineering approaches for computational science can take these characteristics into account.

3.1 Characteristics Resulting From the Nature of Scientific Challenges

All characteristics of software development in computational science that are listed in this section result from the fact that scientific software is an integral part of a discovery process. When you develop software to explore previously unknown phenomena, it is hard to specify exactly up front what the software is required to do, how its output is supposed to look like, and how to proceed during its development.

a) Requirements Are Not Known Up Front
In science, software is used to make novel discoveries and to further our understanding of the world. Since scientific software is deeply embedded into an exploratory process, you never know where its development might take you. Thus, it is hard to specify the requirements for this kind of software up front as demanded by traditional software processes. Accordingly, most of the

requirements—except for the most obvious high-level ones—are discovered only during the course of development in a highly iterative process (Segal and Morris, 2008). The reason for this is that while the underlying scientific theory is well-established in most scientific software projects, it is unclear in advance how this theory can be applied to the specific problem at hand (Carver et al., 2007). When the sole purpose of the project is to further domain understanding, the exact outcome of the project is—by definition—unknown.

The primary intention of software development in computational science is *not* to produce software but to obtain scientific results. For this reason, it is unsurprising that scientific programmers say about themselves that they are "programming experimentally" (Segal, 2005). The scientific models as well as their implementations are treated as evolving theories to test specific hypotheses (East- erbrook and Johns, 2009). Thus, it is the insights gained from one version of the software that determine what is needed for the next version in relatively short iterations (Hochstein et al., 2005). This iterative nature of the scientific software development process does, therefore, not indicate a lack of programming skills among the scientists but mirrors the growing understanding of the requirements as the software evolves (Segal, 2007).

That scientists rarely see design and requirements analysis as distinct steps in software development (Sanders and Kelly, 2008), is in part due to the fact that many scientific applications start out as very small projects and begin to grow only on the basis of their scientific success (Basili et al., 2008). Thus, the requirements for the first version of the software often stem from a single scientist's experience and are usually not explicated by that person. If the software proves to be useful to a broader community, its members tend to make suggestions on features to incorporate into the software and, thereby, they add requirements. These requirements, however, are not explicated in a way that would be detailed enough to form the basis of a contractual document as it is required in established software engineering processes (Segal, 2008). In the case that a sponsor organization demands the documentation of the design and requirements analysis process, the scientists typically do not write these documents before the software is almost complete (Sanders and Kelly, 2008).

b) Verification and Validation Is Difficult and Strictly Scientific

In the context of scientific software, *verification* means to demonstrate that the implementation of algorithms and the equations embodied within them are correct. Thus, verification is purely concerned with theoretical constructs. In contrast, *validation* means to demonstrate that the software and the mathematical model represented by it succeed in capturing all relevant scientific effects correctly. Hence, validation has to ensure that the software output is in sufficient agreement with observations from the real world (Carver et al., 2007). Verification and validation pose serious challenges in all areas of software development but are especially difficult in computational science due to a lack of test oracles, because of complex distributed hardware environments with inadequate tool support, and due to the scientists' undervaluation of software in general (Kanewala and Bieman, 2014).

Validation is particularly challenging as the scientists frequently lack

observational data to compare their model results to—after all, they use simulations precisely because the subject at hand is "too complex, too large, too small, too dangerous, or too expensive to explore in the real world" (Segal and Morris, 2008). But even if observations are available, they can still be incomplete or incorrect and they never extend to the future, with which many simulations are concerned (Sanders and Kelly, 2008). Lastly, if deviations from observations occur, it is hard to trace down their causes which can lie in three distinct dimensions or even a combination of them (Carver et al., 2007):

1. The mathematical model of reality can be insufficient, meaning that scientific aspects are wrong.
2. The algorithm used to discretize the mathematical problem can be inadequate (e.g., have stability problems).
3. The implementation of the algorithm can be wrong due to programming errors.
4. When models for different physical processes are coupled, errors may propagate through the system such that it becomes difficult to trace the error causes.

Therefore, extensive checks of the code and the scientific model have to take place during the development, which highlights the importance of proper verification (Shull et al., 2005).

For the purpose of verification, computational scientists can rely on established testing methods (e.g., unit tests and assertions). In addition to these traditional approaches, they employ checks to test whether theoretically guaranteed results hold true (propositions regarding approximation stability and quality, conservation of certain physical quantities, etc.). However, especially system testing is complicated by the fact that simulation software often runs on distributed hardware that is poorly supported by tools for debugging and profiling (Basili et al., 2008).

Because of the difficulties associated with testing and because of a general disregard for code quality (Section 3.3 c)), formal verification procedures are not common in computational science (Segal, 2007). Prabhu et al. (2011) report that according to their survey, scientists spent more than half of their programming time on finding and fixing errors but only employ "primitive" debugging and testing methods. The testing which is performed is only of cursory nature and consists in manually checking for the answer to questions like "does the software do what I expect it to do with inputs of the type I would expect to use?" (Segal, 2008). In this context, visualization of output data is the most common tool for verification and validation purposes. However, visualization can provide no more than a "sanity check" indicating that the code is behaving "reasonably" (Carver et al., 2006).

A reason for this lack of disciplined testing can be seen in the scientists' regarding their software as imperfect evolving theories that allow them to test hypotheses. From this point of view, they judge model and algorithmic defects to be of far greater significance than coding defects (Basili et al., 2008;

Easterbrook and Johns, 2009). This also explains why almost all testing strategies employed by the scientists are strictly scientific (Faulk et al., 2009). Since they do not perceive the source code to be an entity in its own right and view it as a more or less direct representation of the underlying scientific theory (Section 3.3 c)), they only look at the output of the software and check whether it is in agreement with their current theory. The scientists treat the software like any other (physical) experimentation apparatus that is usually expected to function well. This assumption is only questioned if the data is in conflict with what the scientists would roughly expect (Segal, 2008). For this reason, a software engineering approach for computational science should draw the programmers attention to the important role of the correctness of the source code (Hinsen, 2015). This can be accomplished, for example, by providing easy-to-use methods to test assertions that are meaningful to the scientists on a scientific level.

c) Overly Formal Software Processes Restrict Research

Traditional software development processes that employ a "big design up front" approach—like the waterfall model (Royce, 1970)—are "a poor fit" for computational science (Easterbrook and Johns, 2009). The reason for this is that software development in science is deeply embedded into the scientific method, which makes the up-front specification of requirements impossible (Section 3.1 a)) and introduces challenges with the verification and validation of the implementation (Section 3.1 b)). As scientific software is evolving continuously, no clear-cut requirements analysis, design, or maintenance phases could be discerned (Segal, 2007) and the developers need the flexibility to quickly experiment with different solution approaches (Carver et al., 2007).

Instead of established software engineering processes, scientists apply an informal, non-standard process that is depicted in Figure 1. Their method is highly iterative and starts from a vague idea of which scientific problem the software is supposed to solve and what the application, therefore, could be required to do. Based on this idea, a prototype is developed and is continuously improved guided by the questions "does it do what I want?" and "does it help solve the scientific problem at hand?" (Segal, 2008). When the software reaches a state of maturity which enables it to answer the research question under study, it is subjected to cursory testing as described in Section 3.1 b). If the output of the software does not meet the expectations of the developers, modifications become necessary until "plausible" output is achieved. Note that these modifications almost always involve *both* the code and the underlying scientific theory (Sanders and Kelly, 2008). Therefore, and because the code is often perceived as a mere representation of the theory and not as an entity in its own right (Section 3.3 c)), the development method of the scientists could, in a certain sense, be considered primarily a *theory* development method rather than a software development method. The scientists regard their informal software process as necessarily following from applying the scientific method to scientific reasoning with the help of computing (Kendall et al., 2008).

Several researchers point out that the development approach prevalent in

computational science bears some similarity to "agile" software engineering methods,[4] such as Extreme Programming (Beck, 2000). Many computational scientists have been operating with an "agile philosophy" long before the term was even introduced in software engineering (Carver et al., 2007). However, all established development processes—even agile ones—are generally rejected by the community as too formal because the scientists feel that these processes constrain them in experimenting with their software (Segal, 2008). Therefore, any development approach to be adopted by the computational science community must be very lightweight and integrate well with the software/theory method depicted in Figure 1.

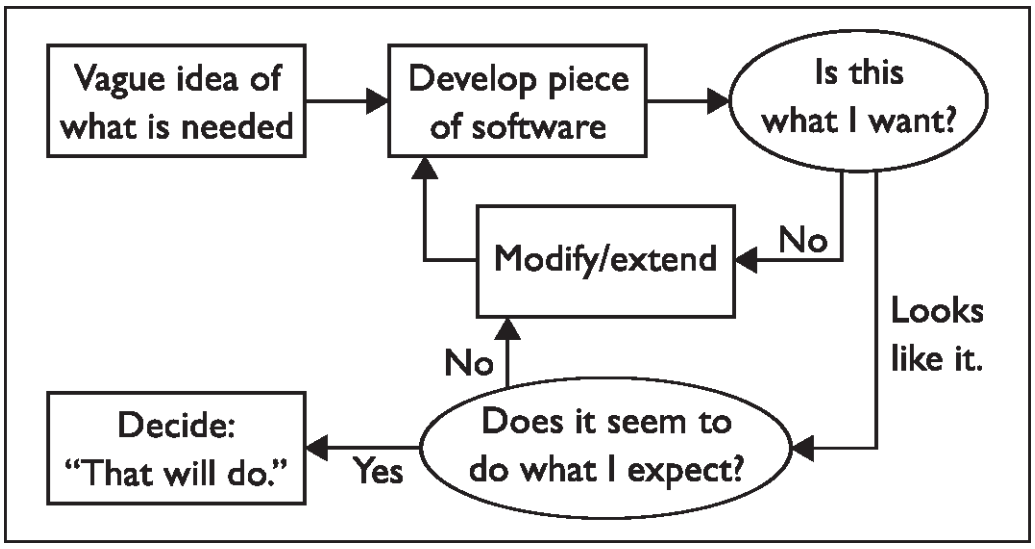

Figure 1: A model of scientific software development (adapted from Segal and Morris (2008)).

3.2 Characteristics Resulting From the Limitations of Computer Hardware

In this section, we discuss characteristics of software development in computational science that are due to limitations regarding available computing resources and their efficient programming.

a) Development Is Driven and Limited by Hardware
Complex simulation software is never perceived as "finished" by the computational scientists. Since it always can only be an imperfect representation of the highly complex reality, one could constantly hope to improve the software and its output by modeling more of the relevant scientific processes or increasing the resolution of discretizations. Therefore, scientific software is typically not limited by theory but by the available computing resources and their efficient utilization (Easterbrook and Johns, 2009).

The development of scientific software is not only limited but also driven by

[4] http://agilemanifesto.org

the available compute hardware in two ways. First, every time new hardware that increases computational power by an order of magnitude becomes available, completely new types of coupled multi-physics simulations suddenly become possible. This necessitates the implementation of new simulation software or, at least, the coupling of simulations in a more complex way. Second, new hardware platforms regularly introduce changes in the underlying hardware architecture. Harnessing the power of these new architectures typically requires to adapt existing simulation software for performance optimization (Faulk et al., 2009).

b) Use of "Old" Programming Languages and Technologies

Especially legacy HPC applications tend to be written to older standards for programming languages such as Fortran and to low-level languages such as C and use long-established technologies like Message Passing Interface (MPI).[5] This is due to several reasons, one being the long lifetime of HPC software (see Section 3.3 c)). In this context, Fortran and C seem to be "safe choices" because it is likely that for many years to come every hardware platform is going to support these languages (Faulk et al., 2009). Scientific programmers are skeptical about new technologies because the history of HPC is full of tools and programming languages that promised productivity increases but were discontinued after a while. Additionally, the low abstraction level of C and older versions of Fortran implies that developers are operating closer to the underlying hardware platform. Therefore, these languages provide predictable performance and allow for more hand-crafted performance optimizations (Basili et al., 2008). It can also be observed that some large scientific communities move towards C++ frameworks such as Trilinos (Heroux et al., 2005), or even Python frameworks such as Jupyter (Ragan-Kelley et al., 2014).

The scientists do not see any reason to adopt newer programming languages as the established ones are easy to learn (which is important for self-teaching; see Section 3.3 a)) and there is a huge amount of legacy code written in those languages (Carver et al., 2007). Their decision is also highly influenced by cultural traditions and believes: interviewees of Sanders and Kelly (2008) reported that object orientation did not "buy [them] anything" and that "a couple lines of C would take a large amount of C++ code." To be accepted by the computational science community, a new programming language would have to be easy to learn, offer reasonably high performance, exhibit stability, and transform language constructs into machine instructions in a predictable way (Carver et al., 2007).

The HPC community uses higher-level languages such as Matlab almost exclusively for prototyping algorithms, which are later re-implemented for higher performance using lower-level languages (Kendall et al., 2008). In disciplines that are less technology-affine—such as biology or psychology—newer languages such a Matlab and Python are more widely adopted for small-scale projects (Prabhu et al., 2011). For larger projects, new technologies have better chances of being accepted if they can coexist with older ones and do not immediately require a full buy-in. This explains why frameworks that dictate the user how to structure their program are seldom used. The scientists prefer re-

[5] http://www.mpi-forum.org/

implementing a lot of existing functionality to giving up control over the code that they want to experiment with (Basili et al., 2008).

When adapting software engineering methods for computational science, one has to take into consideration the reluctance especially of HPC developers regarding any technology that is not tested by time and that runs the risk of ceasing to be supported. Therefore, it is important to make all software aimed at scientific programmers available under open source licenses and *not* to force them to use newer programming languages. This allows the scientists to, at least in principle, keep maintaining discontinued software by themselves. Also a stepwise buy-in into proposed technologies should be made possible.

c) Intermingling of Domain Logic and Implementation Details

The use of older procedural programming languages in computational science (Section 3.2 b)) and a focus on performance (Section 3.2 d)) often impede the separation of domain logic and implementation details in the solution artifacts. This makes it difficult to evolve scientific theory and implementation-specific aspects (such as optimizations for a particular hardware platform) independently of one another and ultimately leads to software that is hard to maintain. It also results in an expertise problem: if all aspects of the implementation are intermingled, the developer should be—but rarely is—equally proficient in all those aspects ranging from the domain knowledge to numerical methods to the specifics of certain processor designs (Faulk et al., 2009). Software engineering approaches, thus, should focus on separating these concerns without negatively affecting performance levels.

d) Conflicting Software Quality Requirements

The ISO/IEC 25010 standard lists eight categories of product quality characteristics that software can be evaluated for: functional suitability, reliability, performance efficiency, usability, security, compatibility, maintainability and portability (ISO 25010, 2011). In their field studies, Carver et al. (2007) find that scientific software developers rank the following characteristics as the most important ones in descending order:

1. Functional correctness
2. Performance
3. Portability
4. Maintainability

It seems clear that scientists perceive the correctness of the results of their software as the topmost priority. After all, the results are supposed to accurately represent processes in the real world and are used as a starting point for scientific reasoning.

Especially in the HPC context, it is also not surprising that the scientists value performance as large simulations can take days or even months to run. But however valuable performance is to the scientists, it is not an end in itself—the real goal is to do science. Therefore, the most adequate performance metric for scientific software is not given in Floating Point Operations Per Second

(FLOPS) but rather in "scientifically useful results per calendar time" (Basili et al., 2008; Carver et al., 2006). Furthermore, performance is in conflict with portability and maintainability because it is usually achieved by introducing hardware-specific optimizations that reduce the readability of the code. The additional quality attributes, portability and maintainability, are also of great importance to the scientists as scientific software is long-lived (Section 3.3 c)). During its long lifetime, hardware platforms change frequently, which limits the possibility for hardwarespecific performance tuning (Kendall et al., 2008).

The conflict between performance and portability is experienced as problematic by the scientists. However, software engineering can, so far, offer little guidance in this aspect because performance and portability are among the least significant quality characteristics for most software engineering approaches (Faulk et al., 2009). Therefore, adaptations of software engineering techniques for computational science must pay special attention to alleviating the performance / portability issue.

3.3 Characteristics Resulting From the Cultural Environment of Scientific Software Development

The characteristics that are listed in this section result from the cultural environment in which scientific software development takes place. This environment is shaped, for example, by the training of computational scientists and the funding schemes of scientific research projects.

a) Few Scientists Are Trained in Software Engineering

Segal (2007) describes computational scientists as "professional end user developers" who work in very technical and "knowledge-rich" domains and typically develop software solely to advance their own professional goals. What they have in common with conventional end user developers is that most of them lack any kind of formal computer science training and do not perceive themselves as software engineers but as domain experts even though they spend a considerable amount of their research time on developing software.[6] In contrast to most conventional end user developers, however, computational scientists rarely experience any difficulties learning general-purpose programming languages.

The self-perception of scientific software developers as scientists rather than developers is grounded in the cultural values of the community: because the ultimate goal is to further scientific knowledge, domain expertise is seen as "intellectual capital," whereas software development skills are just "techniques"—a means to an end. This also implies that possessing software engineering skills is not valued when it comes to recruitment and promotion decisions. Jobs are awarded to those candidates who are qualified best for what is usually viewed as the highest priority in computational science: scientific theory (Sanders and Kelly, 2008).

[6] Prabhu et al. (2011) in their study of 114 research scientists from diverse fields find that more than a third of their subjects' research time is spent on software development tasks.

In addition to not being appreciated, learning software engineering skills is perceived as an excessive demand by computational scientists as they already have enough to do with performing as a scientists (write papers and grants, give presentations, etc.) and keeping up with their fast-developing fields of study (Killcoyne and Boyle, 2009). This problem is reinforced by the fact that computational science is already becoming more and more interdisciplinary, and thus more complicated, purely from the scientific side. As more and more effects are to be considered by ever more complex simulations, computational scientists already have to be able to collaborate with researchers from other disciplines and "speak their language" (Carver et al., 2007). All of this leaves little room for software engineering education. The knowledge of programming languages that scientists possess—which is obviously not identical with software engineering knowledge—is usually acquired by self-study or from co-workers (Carver et al., 2013; Basili et al., 2008).

Even though software development is largely perceived as a burden, computational scientists do not like delegating it to others. They feel they possess the necessary technical skills and find it easier to do it themselves than to explain their needs to somebody else (Easterbrook and Johns, 2009). Furthermore, the development process critically depends on domain knowledge (Segal, 2009; Segal and Morris, 2008). It is perceived as easier to teach scientists how to program than to make software engineers understand the domain science because many of the applications "require a PhD in physics or a branch of engineering just to understand the problem" (Carver et al., 2007). This view is backed up by a study from Segal (2005) in which software engineers implemented a scientific software library based on requirement and specification documents written by scientists. Even though formal minuted meetings were held during the development process to establish a shared understanding between the scientists and the software engineers, the final product did not meet the requirements of the scientists.

Although it seems neither desirable nor feasible to delegate the work of computational scientists to external software engineers, it is regarded beneficial to have a few software engineers working in scientific research institutions to provide development support (Killcoyne and Boyle, 2009). However, such positions have typically not been supported by funding agencies in the past (Carver et al., 2007).

b) Different Terminology

Due to the isolated development of computational science and software engineering, both fields have established distinct terminologies even for shared concepts (Faulk et al., 2009). The terms and metaphors of the computational scientists are typically drawn either from the scientific method itself or from rather low-level concepts of computation. For example, scientific programmers do not call their applications "software" but rather speak of "codes." A "serial code" is a piece of software that does not utilize parallelism and "scaling" such a code means adapting it for parallel execution etc.

Because of their distinct terminology, scientific programmers sometimes (have to) re-invent existing software engineering techniques: they just do not find the

existing methods that would fit their needs because they look for them using the "wrong" vocabulary. For them, these techniques are just "natural" aspects of a research method rather than being a general tool for software development. Therefore, scientists in some cases do not recognize that they are already using software engineering methods if they are confronted with them in the vocabulary of software engineering (Easterbrook and Johns, 2009).

It appears that software engineers have to adapt their vocabulary in order to be understood and taken seriously in the domain of computational science. The terminology of software engineering is often regarded by computational scientists as consisting mostly of "glitzy" marketing terms that are nothing but empty promises (Killcoyne and Boyle, 2009).

c) Scientific Software in Itself Has no Value but Still it Is Long-Lived

For computational scientists, the software they produce has no value in itself; its value is solely based on its ability to efficiently solve problems at hand and make new scientific discoveries (Faulk et al., 2009). This focus on novelty and discovery leads to the perception that software skills are just a "necessary craft," just a means to an end, and that acquiring them is not "real work" (Killcoyne and Boyle, 2009). While domain knowledge is considered "intellectual capital," software development knowledge is merely a "technique," which consequently renders all technical decisions comparably unimportant (Segal, 2007).

Additionally, many computational scientists do not regard software as an entity in its own right. In their mind, source code is a more or less direct representation of the underlying scientific theory (Sanders and Kelly, 2008). Thus, the only value even a code that has been developed and maintained for decades has, does not stem from the engineering effort put into it but from the scientific knowledge accumulated in it.

Such a perspective on software leads to a situation in which code quality is not considered important either—even though it is strongly related to the quality of the scientific results (Hatton and Roberts, 1994). Instead of defect rates, the only code metric that is applied to scientific software is that of novel, publishable results per LOC (Easterbrook and Johns, 2009). There are even cases in which non-trivial software is implemented for the mere purpose of getting a single article published. Because the time-to-solution has to be low in such cases, not much thought is spent on quality attributes like maintainability, extensibility, or reusability. If such a rather poorly engineered code happens to keep being extended—which is how many large codes emerge—, it is hard to remedy these deficiencies (Killcoyne and Boyle, 2009).

Even though the scientists see no value in scientific software in itself, many codes have a long lifetime on the order of decades. The software may not be valuable as such but the accumulated knowledge of the researchers that is embodied in it makes it a long-time investment (Faulk et al., 2009).

During such a long life cycle, the software continuously needs to be developed further in order to reflect the advances in scientific theory and computational hardware (Easterbrook and Johns, 2009; Carver et al., 2007). Because of the potentially very long lifetimes, many scientific software

developers try to avoid dependencies on technologies that could become unavailable. For this reason, the number of dependencies, such as software libraries, is kept to a minimum and only such tools and programming languages are used that have already withstood the ravages of time. This is especially true in the HPC community as their codes are those most likely to be long-lived.

Despite the long life span of scientific software, the effort devoted to its maintenance is low because of a focus on the implementation of new features. Carrying out maintenance tasks is discouraged, firstly, by simply not being rewarded as it does not lead to new publishable results and, secondly, by putting the burden on the developers to demonstrate that their changes do not affect the accuracy of the simulation results (Easterbrook and Johns, 2009). Additionally, the grant-based funding schemes in many branches of science make it hard to assume a long-term perspective on "caring" for scientific software, which is why "quick and dirty" solutions are selectively favored (Howison and Herbsleb, 2011; Killcoyne and Boyle, 2009). Consequently, any software engineering approach for computational science should try to ensure that quality properties like maintainability are built into the software right from the beginning "quasi-automatically."

d) Creating a Shared Understanding of a "Code" Is Difficult

While all scientists eagerly document their scientific results in papers and technical reports, they typically do not produce documentation for the software they implement. User guides are created only in the less frequent case that the software is intended to be used by a larger user base outside of the research group of the original developers (Sanders and Kelly, 2008). Instead of relying on documentation, the scientists prefer informal, collegial ways of knowledge transfer to create a shared understanding of a piece of software. As the users and developers of scientific codes usually overlap, they can rely on a shared background knowledge. Therefore, the scientists find it harder to read and understand documentation artifacts than to contact the author of a certain part of the software and discuss their questions with them (Segal, 2007).

The high personnel turnover rates in scientific software development, however, render such an informal knowledge transfer problematic. Most developers in this area are novices (PhD students and early post docs) because scientists typically do not develop software for their whole careers. As they ascend the career ladder—and often move to other institutes—, their knowledge of the software becomes harder to access (Shull et al., 2005). This means that over and over again novices, without the help of any documentation material, have to familiarize themselves with codes that have not been written with program comprehension in mind (Carver et al., 2006; Segal, 2007). Therefore, software engineering methods for computational science should raise the abstraction level of the implementation artifacts produced by scientific developers to make these artifacts, at least to some extent, self-documenting.

However, the situation is different at government labs in which long-living software is primarily developed by scientists that spend much of their career within that one institution. Such institutions often have resources for a team

of experts that are more focused on the software itself, with responsibilities for porting, optimization, and maintenance. For such long-living software, it is important to keep the documentation and knowlegde about the software up to date (Goltz et al., 2015).

e) Little Code Re-Use

Scientific software developers tend to rarely re-use code developed by others. Frameworks, for example for abstracting from the often tedious details of using MPI, are not adopted because they make certain assumptions as to how their users should structure their code. The scientists fear that later on in a project, these structural assumptions could turn out to be too restrictive but cannot be circumvented. Instead, the researchers tend to re-develop such frameworks by themselves for every application to make them exactly match their needs (Basili et al., 2008; Carver et al., 2006). The same is true even for the use of software libraries. For example, many scientists implement their own linear algebra libraries while there are numerous well-tested, cache-optimized, parallel implementations available under open source licenses. Thereby, these scientists waste much effort on re-inventing existing technologies and, very likely, re-create them with inferior quality (Prabhu et al., 2011).

Limited re-use of existing code cannot only be observed for software developed by others but is even prevalent when it comes to the scientists' own. Because the majority of scientific codes is not programmed with comprehensibility in mind, scientists prefer re-writing code for new projects instead of spending a large amount of time on understanding the old one—even if they are the author of the old code (Segal, 2007). Raising the level of abstraction in implementation artifacts could help to promote code re-use among scientists because it simplifies the comprehension process.

f) Disregard of Most Modern Software Engineering Methods

Surveys among scientific software developers show that they believe to have adequate software engineering knowledge to achieve their development goals. However, when asked about their knowledge and adoption of specific modern software engineering best practices and techniques (such as testing, profiling, and refactoring), both knowledge and adoption are relatively low. Therefore, it appears as if the scientists simply "don't know what they don't know" (Carver et al., 2013; Hannay et al., 2009). And even if the scientists are familiar with tools such as profilers, they rarely actually use most of them—either because of prejudice against the tools ("will not help") or because they think they do not really need them ("I know where time is spent in my code") (Prabhu et al., 2011).

But it is not just ignorance that leads to the non-adoption of software engineering methods. Many methods and tools are just not a good match for the scientists because their functioning is based on (often implicit) assumptions that are violated in the computational science context (Heaton and Carver, 2015). Or they do not fit because they ignore the specific requirements that the scientists have (especially when it comes to tools that could support them). An example of the first type of mismatch due to wrong assumptions are software engineering processes that do not adequately

consider the long life cycles of scientific software or the lack of up front requirements (Carver et al., 2007). Integrated Development Environments (IDEs) for the HPC community are an example of the second mismatch due to neglecting the specific requirements for tools. The use of IDEs is limited in this community because the development environments usually do not feature convenient support for building, profiling, and deploying HPC applications on large-scale distributed systems. Therefore, the scientists only feel constrained by IDEs and, hence, do not adopt them (Prabhu et al., 2011; Carver et al., 2006).

The failure of software engineerings to adequately address the needs of computational science leads to a situation in which the scientists are suspicious about software engineers' claims and overwhelmingly favor handcrafted solutions (Faulk et al., 2009). However, if the scientists are exposed to a certain software engineering technique that they find well-matched for their specific working environment, it is readily adopted. Examples of this are version control systems, regression testing frameworks that can be adapted to the scientists' needs for testing, and reuse in the small via libraries for equation solvers, mesh handling, etc. (Basili et al., 2008). In order to be accepted by the scientists, these tools must introduce a minimum of technicalities as the scientists are busy enough following the fast developments in their own field (Killcoyne and Boyle, 2009).

All in all, we can conclude that software engineering approaches will only be adopted by scientists if these approaches honor the distinct characteristics and constraints of scientific software development which we described above.

4 What Software Engineering has to Offer to Computational Science

Our detailed analysis of the specific characteristics of scientific software development enables us to identify some shortcomings of existing elaborated proposals that are concerned with bridging the "chasm" between software engineering and computational science in Section 4.1. Furthermore, we provide an overview on more recent attempts at closing the gap between the two disciplines in Section 4.2 and we give an outlook on possible research directions that could contribute to improving the current situation in Section 4.3 and 4.4.

4.1 Bridging the Software Chasm

Previous attempts at addressing the credibility and productivity crisis of computational science can be categorized into three groups:

1. Publish and review source code along with scientific articles to ensure reproducibility or at least repeatability of *in silico* experiments.

2. Let software engineers build or re-engineer (parts of) the scientific software.

3. Train scientists to enable them to use state-of-the-art software engineering

methods.

In the context of the credibility crisis of computational science, a discussion about the *reproducibility* of scientific results that rely on computation emerged both within science itself (Peng, 2011) and in the software engineering community (LeVeque et al., 2012). Being able to—at least in principle—validate the findings of other scientists by reproducing their experiments is at the heart of the scientific method. However, it is common practice in many areas of computational science *not* to release the source code on which the findings of a publication are based. This practice impedes the reproduction of published results or even renders it outright impossible. Therefore, several authors suggest to make public disclosure of the source code mandatory for peer-reviewed publications and some even propose to include the code itself in the peer review process (Ince et al., 2012; Morin et al., 2012; Barnes, 2010). In the software engineering community, for example, several large conferences recently started employing a peer-reviewed artifact evaluation process (Krishnamurthi and Vitek, 2015).

These suggestions and efforts are certainly important steps in the right direction and could help to increase the appreciation of software and its quality in the computational science community. However, publishing source code alone does not adequately address the fundamental problem that the scientists lack the software engineering skills to tackle the underlying problems of both the credibility and the productivity crisis.

A second attempt to a solution is to try to have software engineers implement the software for the scientists. The experiences of Segal (2005), who put this approach to test, and the considerations given in Section 3.3 a) suggest that this is not a practicable way.

So far, the most promising attempt to solve the dual scientific software crisis seems to be education via workshop-based training programs focusing on PhD students, such as the ones organized by Wilson (2014) and Messina (2015). While the education approach does address the skill gap that is central to the "software chasm," it does so with inadequate means. Our analysis in Section 3 clearly indicates that just exposing scientists to software engineering methods will not be enough because these methods often fail to consider the specific characteristics and constraints of scientific software development. We therefore conclude that we have to select suitable software engineering techniques and *adapt* them specifically to the needs of computational scientists.

4.2 Adapting Domain-Specific Engineering Approaches

The results of our literature study clearly show that computational scientists are only "accidentally" involved in software development: ultimately, their goal is *not* to create software but to obtain novel scientific results (Section 3.3 c)). At the same time, however, they are very concerned about having full control over their applications and how these actually compute their results, which is why many prefer "older" programming languages with a relatively low level of abstraction from the underlying hardware (Section 3.2 b)).

Among the techniques and tools that software engineering has to offer, so-called Domain-Specific Languages (DSLs) (Fowler, 2010) are a promising starting point for addressing the needs of computational scientists. Like General-Purpose Languages (GPLs), such as C or Java, DSLs are programming languages. However, unlike GPLs, which are designed to be able to implement any program that can be computed with a Turing machine, DSLs limit their expressiveness to a particular application domain. By featuring high-level domain concepts that enable to model phenomena at the abstraction level of the domain and by providing a notation close to the target domain, DSLs can be very concise. The syntax of a DSL can be *textual* or *graphical* and DSL programs can be executed either by means of *interpretation* or through *generation* of source code in existing GPLs. A popular example of a textual DSL are regular expressions, which target the domain of text pattern matching and allow to model search patterns independently from any concrete matching engine implementation.

Since DSLs are designed to express solutions at the abstraction level of the domain, they allow the scientists to care about what matters most to them: doing science without having to deal with technical, implementation-specific details. While they use high-level domain abstractions, they still stay in full control over their development process as it is *them* who directly implement their solutions in formal and executable (e.g., through generation) programming languages. Additionally, generation from a formal language into a low-level GPL permits to examine the generated code to trace what is actually computed.

DSLs can also help to overcome the conflict between the quality requirements of performance on the one hand and portability and maintainability on the other hand, which is responsible for many of the difficulties experienced in scientific software development (Section 3.2 d)). DSL source code is maintainable because it is often pre-structured and much easier to read than GPL code, which makes it almost self-documenting. This almost self-documenting nature of DSL source code and the fact that it can rely on an—ideally—well-tested generator for program translation ensure the reliability of scientific results based on the output of the software. Portability of DSL code is achieved by just replacing the generator for the language with one that targets another hardware platform. With DSLs, the high abstraction level does not have to result in performance losses because the domain-specificity first of all enables to apply—at compile time—domain-specific optimizations and greatly simplifies automatic parallelization (Stahl and Völter, 2006).

In the way described above, DSLs integrated into a appropriate software engineering approach could help to overcome both the productivity and the credibility crisis of computational science. A first indicator that supports this hypothesis can be found in the survey report of Prabhu et al. (2011), who find that those scientists who program with DSLs "report higher productivity and satisfaction compared to scientists who primarily use general purpose, numerical, or scripting languages."

Existing research regarding the application of DSLs in computational science includes the design of several individual DSLs. Examples of this are Liszt (DeVito et al., 2011), which is a DSL for mesh-based partial differential equation solvers with a focus on automatic parallelization, and SESSL (Ewald and Uhrma-

cher, 2014), which allows to model simulation experiments to ensure their reproducibility. Schnetter et al. (2015) describe a framework called Chemora for solving partial differential equations on modern HPC architectures. They use DSLs to separate the concerns of compute model design, model discretization, and the mapping to hardware resources (including performance optimization).

In the examples named so far, DSLs are viewed as more or less isolated tools that scientists can employ to make software development easier for them. Other researchers integrate the use of DSLs into more holistic approaches that directly address the productivity crisis and/or the credibility crisis discussed above. For example, Palyart et al. (2012a) introduce a software engineering approach called MDE4HPC that uses the DSL HPCML (Palyart et al., 2012b) to help scientists with efficiently implementing HPC applications that are independent of any specific HPC hardware architecture. Almorsy et al. (2013) propose to employ suites of graphical DSLs to use graphical modeling in all aspects of the scientific software development process. They provide a web-based tool which aims at enabling scientists to define DSLs by themselves. Another software engineering approach for computational science is Sprat (Johanson et al., 2017; Johanson and Hasselbring, 2014), which integrates multiple DSLs in a hierarchical fashion to facilitate the collaboration of scientists from different disciplines in the development of complex simulation software. In this approach, every developer role in a software project is assigned a separate DSL, which is intended to lead to a clear separation of concerns, well-maintainable code, and a high productivity because the scientists only have to work with abstractions that they are already familiar with from their respective domain.

Similar to the design of GPLs, the design of DSLs faces the problem of not knowing the requirements in advance. Thus, it is important to develop DSLs with agile methods (Gunther et al., 2010) and to involve end users into the design and evaluation process (Johanson and Hasselbring, 2017).

4.3 Software Performance Engineering

Another possible research direction would be to include techniques developed by Software Performance Engineering (SPE) (Bondi, 2014) into software engineering approaches for computational science. Often, performance optimization requires considerable changes in software design. Therefore, performance should already be considered in the design phase of software on an architectural level. However, as can be seen from Section 3.1 c), scientists typically develop software in a highly iterative manner with a focus on the scientific problems at hand, which implies that there usually is no distinct software design phase. This problem can be circumvented by employing DSLs to construct models of the scientific software to be implemented as discussed in the previous section. If the software is implemented using domain-level abstractions, model-based performance prediction and optimization techniques (Balsamo et al., 2004) can be employed without forcing the scientists to adopt rigid software processes.

Since hardware resources are always limited and since especially in HPC *performance really matters*, the application of such SPE approaches to systematically optimize the runtime efficiency of scientific software is a promising area for future work.

4.4 Testing Scientific Software

As discussed in Section 3.1 b), software engineering for computational science should draw the programmers attention to the important role of the correctness of the software (Hinsen, 2015). Software testing usually requires an oracle, which is a mechanism for checking whether the program under test produces the expected output when executed using a set of test cases. However, obtaining reliable oracles for scientific programs is challenging, because the requirements mostly are unclear up front due to the exploratory nature of scientific software development. Model-based testing (Schieferdecker, 2012) requires well-defined and stable requirements to develop the model; thus, model-based testing is not readily applicable to scientific software. Instead, approaches on performing effective testing without pre-defined oracles are required (Kelly et al., 2011). New approaches such as the so-called *metamorphic testing* intend to solve the challenge of testing non-deterministic programs that lack oracles (Guderlei and Mayer, 2007), for instance via machine learning techniques to automatically detect metamorphic relations (Kanewala and Bieman, 2013).

Another challenge is to integrate automated regression and acceptance testing for scientific software, for instance in continuous integration setups (Meyer, 2014). Regression testing allows to compare the current output to previous outputs to identify faults or performance anomalies introduced when the code is modified. Various tools and approaches are under development that address the challenges of testing and debugging software designed to run on distributed systems. Some unit testing frameworks have direct support for MPI and have been successfully used my multiple communities. Another approach is to mock MPI and test/debug components of a simulation in isolation (Clune et al., 2015).

For scientific software, a major difficulty for automated regression testing is caused by the high computational costs of tests. To ensure high code coverage, a potentially exponential set of test configurations must be executed. A solution to this challenge could be a proper modularization of the software such that the software components become testable in isolation. Modularization approaches such as microservices enable scalability (Hasselbring, 2016), as well as agility and reliability (Hasselbring and Steinacker, 2017). Such a modularization may also facilitate automated regression testing of scientific software.

4.5 Requirements Engineering

Some software engineering approaches for computational science, such as the Advises project by Thew et al. (2009) and the approach by Garcia et al. (2013), focus on requirements engineering techniques. Garcia et al. (2013) introduced a component-based and aspect-oriented method for scientific software development. Their approach focuses on formal requirements engineering to enable the reuse of existing software components and their integration via aspect-oriented programming techniques. The idea is that

once the requirements of a scientific application are known, it can be constructed merely by identifying suitable functional components that already exist and the dependency relations between them.

The Advises project (see also Thew et al., 2008; Sutcliffe et al., 2007) acknowledges that in scientific computing, it is usually impossible to specify detailed software requirements up front. In light of this situation, they propose a requirements engineering process in which software engineers use techniques such as unstructured interviews and user observation to iteratively derive detailed requirements. On the basis of these requirements, the software engineers are supposed to develop the software for the domain scientists.

Despite the positive evaluation of the Advises project in the domain of epidemiology, it remains unclear whether such approaches, which focus on requirements engineering and software engineers implementing the software for domain scientists, are applicable to other branches of computational science. The results of the study of Segal (2005), which we cited above, indicate that this may not be the case for branches such as HPC.

5 Concluding Remarks

Based on an examination of the historical development of the relationship between software engineering and computational science (the **Past**), we identified 13 key characteristics of scientific software development by reviewing published literature (the **Present**). We found that the unique characteristics of scientific software development prevent scientists from using state-of-the-art software engineering tools and methods. This situation created a "chasm" between software engineering and computational science, which resulted in a productivity and credibility crisis of the latter discipline. We examined attempts at bridging the gap between software engineering and computational science to reveal shortcomings of existing solutions and to point out further research directions, such as the use of domain-specific languages and testing techniques without pre-defined oracles (the possible **Future**). However, more research on this topic is needed, especially to empirically evaluate the actual gains in productivity and quality achieved for scientific software by such software engineering approaches.

Short Biographies

Arne Johanson is as a data scientist at XING Marketing Solutions GmbH, Germany. He received a Ph.D. in computer science from Kiel University, Germany in 2016. For his Ph.D. studies, he was awarded a scholarship from the Helmholtz Research School Ocean System Science and Technology (HOSST). His research focuses on adapting software engineering techniques for

computational science. Contact him at arj@informatik.uni-kiel.de

Wilhelm Hasselbring is professor of Software Engineering at Kiel University, Germany. In the excellence cluster Future Ocean, a large-scale collaborative project of Kiel University, the GEOMAR Helmholtz Centre for Ocean Research Kiel and others, he is principal investigator and coordinator of the research area Digital Ocean. He is principal investigator of the Helmholtz Research School Ocean System Science and Technology (HOSST). His research interests include software engineering and distributed systems. He received his Ph.D. in Computer Science from the University of Dortmund, Germany. He is a member of the ACM, the IEEE Computer Society, and the German Association for Computer Science. Contact him at hasselbring@email.uni-kiel.de

References

Mohamed Almorsy, John Grundy, Richard Sadus, Willem van Straten, David G. Barnes, and Owen Kaluza. A suite of domain-specific visual languages for scientific software application modelling. In *Symposium on Visual Languages and Human-Centric Computing (VL/HCC), 2013*, pages 91–94. IEEE, 2013.

Simonetta Balsamo, Antinisca Di Marco, Paola Inverardi, and Marta Simeoni. Model-based performance prediction in software development: A survey. *Software Engineering*, 30(5):295–310, 2004.

Nick Barnes. Publish your computer code: it is good enough. *Nature*, 467(7317): 753–753, 2010.

Victor R. Basili, Daniela Cruzes, Jeffrey C. Carver, Lorin M. Hochstein, Jeffrey K. Hollingsworth, Marvin V. Zelkowitz, and Forrest Shull. Understanding the high-performance-computing community: A software engineer's perspective. *IEEE Software*, 25(4):29–36, 2008.

Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 2000.

Marc R. Benioff et al. Computational science: ensuring America's competitiveness. Technical report, President's Information Technology Advisory Committee (PITAC), 2005.

A. B. Bondi. *Foundations of Software and System Performance Engineering: Process, Performance Modeling, Requirements, Testing, Scalability, and Practice*. AddisonWesley, 2014.

J. Brown, M.G. Knepley, and B.F. Smith. Run-time extensibility and librarization of simulation software. *Computing in Science & Engineering*, 17(1):38–45, 2015.

Alfredo Buttari, Jack Dongarra, Jakub Kurzak, Julien Langou, Piotr Luszczek, and

Stanimire Tomov. The impact of multicore on math software. In *Applied Parallel Computing. State of the Art in Scientific Computing*, volume 4699 of *LNCS*, pages 1–10. Springer, 2007.

Jeffrey C. Carver and Tom Epperly. Software engineering for computational science and engineering. *Computing in Science & Engineering*, 16(3):6–9, 2014.

Jeffrey C. Carver, Lorin Hochstein, Richard P. Kendall, Taiga Nakamura, Marvin V. Zelkowitz, Victor R. Basili, and Douglass E. Post. Observations about software development for high end computing. *CTWatch Quarterly*, 2(4A):33– 38, 2006.

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

Jeffrey C. Carver, Dustin Heaton, Lorin Hochstein, and Roscoe Bartlett. Selfperceptions about software engineering: A survey of scientists and engineers. *Computing in Science & Engineering*, 15(1):7–11, 2013.

P.E. Ceruzzi. *A History of Modern Computing*. MIT Press, 2 edition, 2003.

Thomas Clune, Hal Finkel, and Michael Rilee. Testing and debugging exascale applications by mocking mpi. In *Proceedings of the 3rd International Workshop on Software Engineering for High Performance Computing in Computational Science and Engineering*, SE-HPCCSE '15, pages 5–8, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-4012-0. doi: 10.1145/2830168.2830173.

Z. DeVito, N. Joubert, F. Palacios, S. Oakley, M. Medina, M. Barrientos, E. Elsen, F. Ham, A. Aiken, K. Duraisamy, et al. Liszt: a domain specific language for building portable mesh-based PDE solvers. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12. ACM, 2011.

Jack Dongarra, Dennis Gannon, Geoffrey Fox, and Ken Kennedy. The impact of multicore on computational science software. *CTWatch Quarterly*, 3(1):1–10, 2007.

S.M. Easterbrook and T.C. Johns. Engineering the software for understanding climate change. *Computing in science & engineering*, 11(6):65–74, 2009.

Roland Ewald and Adelinde M. Uhrmacher. SESSL: A domain-specific language for simulation experiments. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 24(2):11, 2014.

Stuart Faulk, Eugene Loh, Michael L. Van De Vanter, Susan Squires, and Lawrence G. Votta. Scientific computing's productivity gridlock: How software engineering can help. *Computing in Science & Engineering*, 11:30–39, 2009.

Martin Fowler. *Domain-Specific Languages*. Addison-Wesley, 2010.

Samuel H. Fuller and Lynette I. Millett. Computing performance: Game over or next level? *Computer*, 44(1):31–38, 2011.

Javier Corral Garcia, César Gómez Martin, José Luis González Sánchez, and David Cortés Polo. Development of scientific applications with high-performance computing through a component-based and aspect-oriented methodology. *International Journal of Advanced Computer Science*, 3(8):400–408, 2013.

Ursula Goltz, Ralf Reussner, Michael Goedicke, Wilhelm Hasselbring, Lukas Märtin, and Birgit Vogel-Heuser. Design for future: managed software evolution. *Computer Science – Research and Development*, 30(3):321–331, August 2015. doi: 10.1007/s00450-014-0273-9.

R. Guderlei and J. Mayer. Statistical metamorphic testing testing programs with random output by means of statistical hypothesis tests and metamorphic testing. In *Seventh International Conference on Quality Software (QSIC 2007)*, pages 404–409, 2007.

S. Gunther, M. Haupt, and M. Splieth. Agile engineering of internal domainspecific languages with dynamic programming languages. In *2010 Fifth International Conference on Software Engineering Advances*, pages 162–168, August 2010. doi: 10.1109/ICSEA.2010.32.

J.E. Hannay, H.P. Langtangen, C. MacLeod, D. Pfahl, J. Singer, and G. Wilson. How do scientists develop and use scientific software? In *Software Engineering for Computational Science and Engineering, 2009. SECSE'09. ICSE Workshop on*, pages 1–8. IEEE, 2009.

Wilhelm Hasselbring. Microservices for scalability: Keynote talk abstract. In *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering (ICPE 2016)*, pages 133–134, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4080-9. doi: 10.1145/2851553.2858659.

Wilhelm Hasselbring and Guido Steinacker. Microservice architectures for scalability, agility and reliability in e-commerce. In *Proceedings 2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, pages 243–246, Gothenburg, Sweden, April 2017. IEEE. doi: 10.1109/ICSAW.2017.11.

Les Hatton and Andy Roberts. How accurate is scientific software? *Software Engineering, IEEE Transactions on*, 20(10):785–797, 1994.

Dustin Heaton and Jeffrey C. Carver. Claims about the use of software engineering practices in science: A systematic literature review. *Information and Software Technology*, 67:207–219, 2015.

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, Alan Williams, and Kendall S. Stanley. An overview of the Trilinos project. *ACM Trans. Math. Softw.*, 31(3):397–423,

September 2005. doi: 10.1145/1089014.1089021.

Konrad Hinsen. The approximation tower in computational science: Why testing scientific software is difficult. *Computing in Science & Engineering*, 17(4):72–77, 2015.

Lorin Hochstein, Jeffrey Carver, Forrest Shull, Sima Asgari, Victor Basili, Jeffrey K. Hollingsworth, and Marvin V. Zelkowitz. Parallel programmer productivity: A case study of novice parallel programmers. In *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference*, pages 35–43. IEEE, 2005.

James Howison and James D. Herbsleb. Scientific software production: incentives and collaboration. In *Proceedings of the Conference on Computer supported cooperative work 2011 (CSCW'11)*, pages 513–522. ACM, 2011.

Darrel C. Ince, Leslie Hatton, and John Graham-Cumming. The case for open computer programs. *Nature*, 482(7386):485–488, 2012.

ISO 25010. Systems and software engineering—systems and software quality requirements and evaluation (SQuaRE)—system and software quality models. ISO 25010:2011, International Organization for Standardization, Geneva, Switzerland, 2011.

Arne N. Johanson and Wilhelm Hasselbring. Hierarchical combination of internal and external domain-specific languages for scientific computing. In *Proceedings of the 2014 European Conference on Software Architecture Workshops*, ECSAW'14, pages 17:1–17:8. ACM, 2014.

Arne N. Johanson and Wilhelm Hasselbring. Effectiveness and efficiency of a domain-specific language for high-performance marine ecosystem simulation: a controlled experiment. *Empirical Software Engineering*, 22(4):2206–2236, August 2017. ISSN 1382-3256. doi: 10.1007/s10664-016-9483-z.

Arne N. Johanson, Andreas Oschlies, Wilhelm Hasselbring, and Boris Worm. SPRAT: a spatially-explicit marine ecosystem model based on population balance equations. *Ecological Modelling*, 349:11–25, 2017. doi: http://dx.doi.org/ 10.1016/j.ecolmodel.2017.01.020.

Lucas N. Joppa, Greg McInerny, Richard Harper, Lara Salido, Kenji Takeda, Kenton O'Hara, David Gavaghan, and Stephen Emmott. Troubling trends in scientific software use. *Science*, 340(6134):814–815, 2013.

Upulee Kanewala and James M. Bieman. Using machine learning techniques to detect metamorphic relations for programs without test oracles. In *IEEE 24th International Symposium on Software Reliability Engineering (ISSRE 2013)*, pages 1–10, November 2013.

Upulee Kanewala and James M. Bieman. Testing scientific software: A systematic literature review. *Information and software technology*, 56(10):1219–1232, 2014.

Diane Kelly. A software chasm: Software engineering and scientific computing. *IEEE Software*, 24(6):120–119, 2007.

Diane Kelly, Spencer Smith, and Nicholas Meng. Software engineering for scientists. *Computing in Science & Engineering*, 13(5):7–11, 2011.

Richard Kendall, Jeffrey C. Carver, David Fisher, Dale Henderson, Andrew Mark, Douglass Post, Clifford E. Rhoades, and Susan Squires. Development of a weather forecasting code: A case study. *Software, IEEE*, 25(4):59–65, 2008.

Sarah Killcoyne and John Boyle. Managing chaos: Lessons learned developing software in the life sciences. *Computing in Science & Engineering*, 11(6):20–29, 2009.

Shriram Krishnamurthi and Jan Vitek. The real software crisis: repeatability as a core value. *Communications of the ACM*, 58(3):34–36, 2015.

Randall J. LeVeque, Ian M. Mitchell, and Victoria Stodden. Reproducible research for scientific computing: Tools and strategies for changing the culture. *Computing in Science & Engineering*, 14(4):13, 2012.

Z. Merali. Computational science: Error, why scientific programming does not compute. *Nature*, 467(7317):775–777, 2010.

Paul Messina. Gaining the broad expertise needed for high-end computational science and engineering research. *Computing in Science & Engineering*, 17(2): 89–90, 2015.

M. Meyer. Continuous integration and its tools. *IEEE Software*, 31(3):14–16, 2014.

A. Morin, J. Urban, P.D. Adams, I. Foster, A. Sali, D. Baker, and P. Sliz. Shining light into black boxes. *Science*, 336(6078):159–160, 2012.

Peter Naur and Brian Randell. *Software Engineering: Report of a Conference Sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 Oct. 1968, Brussels, Scientific Affairs Division, NATO*. NATO Scientific Affairs Division, 1969.

Newell, Perlis, and Simon. letter to the editor. *Science*, 157:1373–1374, 1967.

Marc Palyart, David Lugato, Ileana Ober, and Jean-Michel Bruel. MDE4HPC: An approach for using model-driven engineering in high-performance computing. In *Proceedings SDL'11: Integrating System and Software Modeling*, volume 7083 of *LNCS*, pages 247–261, 2012a.

Marc Palyart, Ileana Ober, David Lugato, and Jean-Michel Bruel. HPCML: a modeling language dedicated to high-performance scientific computing. In *Proceedings of the 1st International Workshop on Model-Driven Engineering for High Performance and Cloud computing*, pages 1–6, 2012b.

Roger D. Peng. Reproducible research in computational science. *Science*, 334 (6060):1226–1227, 2011.

Douglass E. Post. The changing face of scientific and engineering computing. *Computing in Science & Engineering*, 15(6):4–6, 2013.

Douglass E. Post and Lawrence G. Votta. Computational science demands a new paradigm. *Physics today*, 58(1):35–41, 2005.

Prakash Prabhu, Thomas B. Jablin, Arun Raman, Yun Zhang, Jialu Huang, Hanjun Kim, Nick P. Johnson, Feng Liu, Soumyadeep Ghosh, Stephen Beard, Taewook Oh, Matthew Zoufaly, David Walker, and David I. August. A survey of the practice of computational science. In *State of the Practice Reports*, SC'11, pages 19:1–19:12. ACM, 2011.

M Ragan-Kelley, F Perez, B Granger, T Kluyver, P Ivanov, J Frederic, and M Bussonnier. The Jupyter/IPython architecture: a unified view of computational research, from interactive exploration to communication and publication. In *AGU Fall Meeting Abstracts*, 2014.

Winston W. Royce. Managing the development of large software systems. In *Proceedings of WESCON'70*, pages 328–338. IEEE, 1970.

Rebecca Sanders and Diane F. Kelly. Dealing with risk in scientific software development. *Software, IEEE*, 25(4):21–28, 2008.

Ina Schieferdecker. Model-based testing. *IEEE Software*, 29(1):14–18, 2012.

Erik Schnetter, Marek Blazewicz, Steven R. Brandt, David M. Koppelman, and Frank Löffler. Chemora: A PDE-solving framework for modern high-performance computing architectures. *Computing in Science & Engineering*, 17(2):53– 64, 2015.

Judith Segal. When software engineers met research scientists: A case study. *Empirical Software Engineering*, 10(4):517–536, 2005.

Judith Segal. Some problems of professional end user developers. In *Symposium on Visual Languages and Human-Centric Computing, 2007. VL/HCC 2007.*, pages 111–118. IEEE, 2007.

Judith Segal. Models of scientific software development. In *Proceedings of the First International Workshop on Software Engineering for Computational Science and Engineering, SECSE'08*, pages 1–7, 2008.

Judith Segal. Some challenges facing software engineers developing software for scientists. In *Proceedings of the 2009 ICSE Workshop on Software Engineering for Computational Science and Engineering*, pages 9–14. IEEE, 2009.

Judith Segal and Chris Morris. Developing scientific software. *Software, IEEE*, 25 (4):18–20, 2008.

Forrest Shull, Jeffrey Carver, Lorin Hochstein, and Victor Basili. Empirical study design in the area of high-performance computing (HPC). In *Proceedings of the International Symposium on Empirical Software Engineering 2005*, pages 1–10. IEEE, 2005.

Thomas Stahl and Markus Völter. *Model-Driven Software Development: Technology, Engineering, Management*. Wiley, 2006.

Tim Storer. Bridging the chasm: A survey of software engineering practice in scientific programming. *ACM Comput. Surv.*, 50(4):47:1–47:32, August 2017. doi: 10.1145/3084225.

A.G. Sutcliffe, Sarah Thew, Colin Venters, Oscar De Bruijn, John Mcnaught, Rob Procter, and Iain Buchan. Advises project: Scenario-based requirements analysis for e-Science applications. In *UK e-Science All Hands Meeting*, pages 142–149. UK e-Science, 2007.

Sarah Thew, Alistair Sutcliffe, Oscar De Bruijn, John McNaught, Rob Procter, Colin Venters, and Iain Buchan. Experience in e-Science requirements engineering. In *16th International Requirements Engineering Conference, 2008. RE'08.*, pages 277–282. IEEE, 2008.

Sarah Thew, Alistair Sutcliffe, Rob Procter, Oscar De Bruijn, John McNaught, Colin C. Venters, and Iain Buchan. Requirements engineering for e-Science: experiences in epidemiology. *IEEE Software*, 26(1):80, 2009.

Iris Vessey. Problems versus solutions: the role of the application domain in software. In *Papers presented at the seventh workshop on Empirical studies of programmers*, pages 233–240. ACM, 1997.

Gregory V. Wilson. Where's the real bottleneck in scientific computing? *American Scientist*, 94(1):5–6, 2006.

Gregory V. Wilson. Software carpentry: lessons learned. *F1000Research*, 3:1–11, 2014.