Assessing the Impact of MDE and Code Generation on the Sustainability of Scientific Computing Software: A Research Proposal

Spencer Smith

September 17, 2019

Abstract

...

Contents

1 Introduction

Without dramatic intervention, our collective confidence in Scientific Computing Software (SCS) is due for a catastrophic collapse. We are increasingly trusting ever more complex and ambitious computations, but our software foundations are built on sand. Areas of concern include nuclear safety analysis and computational medicine. Successfully building such software requires communication between software developers and experts from multiple domains. Collaboration is difficult at the best of times, and is made worse because developers favour handcrafted solutions over adapting software engineering processes, methods and tools (Faulk et al., 2009). Handcrafted solutions do not account for the inevitable changes in requirements, design and implementation. The twin challenges of changing requirements and inadequate documentation conspire to make computational results notoriously difficult to reproduce, especially in the important endeavour of one researcher independently replicating the results of another (Smith, 2018). Inadequate traceability within documentation means recertification of previously certified engineering software is as expensive and time consuming as the original certification exercise, because of a lack of explicit, effort reducing, traceability information. The existing challenges for modification, maintenance and extension point to problems with the sustainability of SCS.

Documentation can improve software sustainability. For software in general, documentation, written before and during development, provides many benefits Parnas (2010): easier reuse of old designs, better communication about requirements, more useful design reviews, easier integration of separately written modules, more effective code inspection, more effective testing, and more efficient corrections and improvements. For SCS in particular, documentation provides significant benefits. For instance, Smith et al. (2015) shows that the quality of statistical software for psychology is generally improved when developed using the structured CRAN (Comprehensive R Archive Network) process and tools, versus an ad hoc approach. Smith and Koothoor (2016) highlights the value of proper documentation by redeveloping nuclear safety analysis software. Twenty seven (27) worrisome documentation problems were found, including problems with incompleteness, ambiguity, inconsistency, verifiability, modifiability, traceability and a lack of abstraction. A redevelopment experiment with five existing projects (Smith et al., 2016a) enabled the code owners (as ascertained through interviews) to clearly see the value of documentation. However, mirroring other studies

(Carver et al., 2007), the code owners felt documentation takes too much time. Due to the time commitment and resource limitation, documentation is rarely emphasized in SCS.

One promising approach to gain the sustainability benefits of documentation, but significantly reduce the time commitment, is to use a Model Driven Engineering (MDE) process. MDE is an engineering approach that exploits structured representations (metamodels for abstract syntax, and constraints to capture static semantics) to automate repetitive, tedious and error prone tasks. In this research proposal, we will use MDE as a short-hand to include knowledge models, model transformation, Domain Specific Languages (DSLs) and code generation. Several current papers point to DSLs and code/document generation as a transformative technology for documentation, design and verification of SCS (Johanson and Hasselbring, 2018; Smith, 2018). MDE has the potential to dramatically reduce the cost of developing reliable, reproducible and re-certifiable software.

One example of applying MDE to SCS is Drasil, which is introduced in Szymczak et al. (2016) and available at https://github.com/JacquesCarette/Drasil. Drasil is implemented as an open source set of Domain Specific Languages (DSLs). The Drasil approach involves: i) creating infrastructure for a knowledge base of scientific and computing models; and, ii) writing explicit "recipes" that weave together this knowledge to generate theoretical models, design documents, code, test cases and build scripts. The generator can render to multiple languages, such as html, and LaTeX for documentation and Python, Java, C++ and Lua, for code. One source of knowledge, with rules for transformation, means completeness, consistency and traceability can be achieved by construction. Moreover, these qualities can be maintained as requirements are modified, design decisions are changed, documentation standards are varied, and software is recertified. Although creating the knowledge base is time consuming, the knowledge can be built up incrementally. New projects will reuse existing knowledge and expand as necessary.

As described above, sustainability is a concern for SCS; MDE holds promise for addressing this concern. However, a transformative change in the development of SCS requires more than promise. Empirical evidence is necessary to demonstrate that an MDE process can improve the sustainability of SCS. This leads to the long-term objective for this research proposal: *assess and measure the impact of MDE on the sustainability of SCS.*

The scope of SCS software for this proposal includes long-lived software, as implied by the sustainability objective. To gain the full benefits of generating documentation, the scope also includes software that interests multiple stakeholders, with different interests and backgrounds. Finally, the scope of this proposal emphasizes safety related software, such as software for nuclear safety analysis, medical imaging and computational medicine. Safety related software is prioritized because documentation is often expected as part of a certification exercise. Although this study is not specifically restricted to open-source software, for practical reasons (especially the lack of access to the code), commercial software will not be strongly emphasized.

Given the large number of potential MDE processes, techniques and technologies, together with the wide variety in purpose, scope and context for SCS, the long-term objective of measuring the impact of MDE is decomposed into four short-term objectives. For each objective, a reference is given to the corresponding section where details are provided.

1. Assess the current state of the software development practice for SCS. How is the software developed in different sub-domains of SCS? Are there existing development methods that lead to higher software sustainability? (Section 3.1)

2. Assess the impact of an MDE process on end user developed SCS with respect to developer productivity and software sustainability for long-lived lived software. (Section 3.2)

3. Assess the impact on safety when the MDE generated documentation targets a safety assurance case. An assurance case is an explicit structured argument pertaining to specific properties, such as trustworthiness. The evidence for the argument will come from the traceability between artifacts generated by model transformations, expert reviews, test cases, etc. (Section 3.3)

4. Assess the impact on software quality when using MDE to facilitate Computational Variability Testing (CVT). CVT uses code generation to build confidence in the generated code in an analogous way to the use of grid refinement studies. Grid refinement looks at how the solution changes by varying the run-time parameter of grid density and comparing the results to theoretical expectations. CVT, on the other hand, can generate code to "refine" build time parameters, such as order of interpolation, or degree of implicitness. These parameters can

be systematically varied and the results compared against the expected trend. (Section 3.4)

Before presenting the details of the short-term objectives (Section 3), the necessary background information is provided, along with a literature review (Section 2). The literature review covers software quality (Sections 2.1 and 2.2), the literature on the current state of the practice for SCS development (Section 2.3), potential target documentation artifacts for SCS (Section 2.4), MDE and code generation (Section 2.5) and empirical methods for software engineering (Section 2.6). This document ends with a summary of the potential contributions to knowledge from this study (Section 4) and a research schedule (Section 5).

2 Background and Literature Review

To assess the impact of MDE on SCS quality, we need a clear definition of what we mean by quality. Section 2.1 shows how the concept of quality is decomposed into a set of separate qualities. This set of qualities can be applied to the software artifacts (documentation, test cases, etc) and to the software development process itself. Several of the qualities from the list in Section 2.1 cannot be measured directly, such as maintainability. Therefore, Section 2.2 introduces measurable documentation qualities that are believed to contribute to the overall software qualities. In some cases it may be necessary to use the qualities in Section 2.2 to indirectly measure qualities listed in Section 2.1.

The other subsection in this background section provide an overview of the literature on the current approaches to developing SCS software 2.3, recommended document artifacts for SCS 2.4, an overview of MDE and Code Generation in an SCS context 2.5, and a summary of existing literature on empirical methods for software engineering 2.6.

2.1 Software Qualities of Interest

Our analysis is centred around a set of software qualities. Quality is not considered as a single measure, but a collection of different qualities, often called "ilities." These qualities highlight the desirable nonfunctional properties for software artifacts, which include both documentation and code.

Some qualities, such as visibility and productivity, apply to the process used for developing the software. The following list of qualities is based on Ghezzi et al. (2003). To the list from Ghezzi et al. (2003), we have added three qualities important for SC: installability, reproducibility and sustainability.

Installability A measure of the ease of installation.

Correctness Software is correct if it matches its specification.

Verifiability involves "solving the equations right" (Roache, 1998, p. 23); it benefits from rational documentation that systematically shows, with explicit traceability, how the governing equations are transformed into code.

Validatability means "solving the right equations" (Roache, 1998, p. 23). Validatability is improved by a rational process via clear documentation of the theory and assumptions, along with an explicit statement of the systematic steps required for experimental validation.

Reliability is a critical quality for scientific software, since the results of computations are meaningless, if they are not dependable. Reliability is closely tied to verifiability, since the key quality to verify is reliability, while the act of verification itself improves reliability.

Performance considerations can make certification challenging, since QA becomes more difficult for more complex code. However, as Roache (Roache, 1998, p. 355) points out, using simpler algorithms and reducing the number of options in general purpose code, is not always a practical option.

Usability can be a problem. Different users, solving the same physical problem, using the same software, can come up with different answers, due to differences in parameter selection (Roache, 1998, p. 370). To reduce misuse, a rational process must state expected user characteristics, modelling assumptions, definitions and the range of applicability of the code.

Maintainability is necessary in scientific software, since change, through iteration, experimentation and exploration, is inevitable. Models of physical phenomena and numerical techniques necessarily evolve over time Carver et al. (2007); Segal and Morris (2008). Proper documentation, designed with change in mind, can greatly assist with change management.

Reusability provides support for the quality of reliability, since reliability is improved by reusing trusted components Dubois (2005). (Care must still be taken with reusing trusted components, since blind reuse in a new context can lead to errors, as dramatically shown in the Ariane 5 disaster (Oliveira and Stewart, 2006, p. 37–38).) The odds of reuse are improved when it is considered right from the start.

Understandability is necessary, since reviewers can only certify something they understand. Scientific software developers have the view "that the science a developer embeds in the code must be apparent to another scientist, even ten years later" Kelly (2013). Understandability applies to the documentation and code, while usability refers to the executable software. Documentation that follows a rational process is the easiest to follow.

Reproducibility is a required component of the scientific method Davison (2012). Although QA has, "a bad name among creative scientists and engineers" (Roache, 1998, p. 352), the community need to recognize that participating in QA management also improves reproducibility. Reproducibility, like QA, benefits from a consistent and repeatable computing environment, version control and separating code from configuration/parameters Davison (2012).

Productivity [Fill in]

Sustainability A combination of other qualities, likely maintainability and productivity. Reference authorless paper found on-line.

2.2 Desirable Qualities of Documentation

To achieve the qualities listed in Section 2.1, the documentation should achieve the qualities listed in this section. All but the final quality listed (abstraction), are adapted from the IEEE recommended practise for producing good software requirements IEEE (1998). Abstraction means only revealing relevant details, which in a requirements document means stating what is to be achieved, but remaining silent on how it is to be achieved. Abstraction is an important software development principle for dealing with complexity (Ghezzi et al., 2003, p. 40). Smith and Koothoor (2016) present further details on the qualities of documentation for SCS.

Completeness Documentation is said to be complete when all the requirements of the software are detailed. That is, each goal, functionality, attribute, design constraint, value, data, model, symbol, term (with its unit of measurement if applicable), abbreviation, acronym, assumption and performance requirement of the software is defined. The software's response to all classes of inputs, both valid and invalid and for both desired and undesired events, also needs to be specified.

Consistency Documentation is said to be consistent when no subset of individual statements are in conflict with each other. That is, a specification of an item made at one place in the document should not contradict the specification of the same item at another location.

Modifiability The documentation should be developed in such a way that it is easily modifiable so that likely future changes do not destroy the structure of the document. Also it should be easy to reflect the change, wherever needed in the document to maintain consistency, traceability and completeness. For documentation to be modifiable, its format must be structured in a way that repetition is avoided and cross-referencing is employed.

Traceability Documentation should be traceable, as this facilitates maintenance and review. If a change is made to the design or code of the software, then all the documentation relating to those segments have to be modified. This property is also important for recertification.

Unambiguity Documentation is said to be unambiguous only when every requirement's specification has a unique interpretation. The documentation should be unambiguous to all audiences, including developers, users and reviewers.

Correctness There is no direct tool or method for measuring correctness. One way of building confidence in correctness is by reviewing to ensure that each requirement stated is one that the stakeholders and experts desire. By maintaining traceability, consistency and unambiguity, we can reduce the occurrence of errors and make the goal of reviewing for correctness easier.

Verifiability Every requirement in the documentation must be the one fulfilled by the implemented software. Therefore all the requirements should be clear, unambiguous and testable, so that a person or a machine can verify whether the software product meets the requirements.

Abstract Documented requirements are said to be abstract if they state what the software must do and the properties it must possess, but do not speak about how these are to be achieved. For example, a requirement can specify that an Ordinary Differential Equation (ODE) must be solved, but it should not mention that Euler's method should be used to solve the ODE. How to accomplish the requirement is a design decision, which is documented during the design phase.

2.3 Literature Review on Current State of the Practice for SCS Development

The SCS community is finally realizing that current practices are not sustainable. Faulk et al. (2009) observe, "growing concern about the reliability of scientific results based on ... software." Embarrassing failures have occurred, like a retraction of derived molecular protein structures (Miller, 2006), false reproduction of sonoluminescent fusion (Post and Votta, 2005), and fixing and then reintroducing the same error in a large code base three times in 20 year (Milewicz and Raybourn, 2018). A recent report on directions for SCS research and education states: "While the volume and complexity of [SCS] have grown substantially in recent decades, [SCS] traditionally has not received the focused attention it so desperately needs ... to fulfill this key role as a cornerstone of long-term collaboration and scientific progress" (Rüde et al., 2018). Estimates suggest that the number of released faults per thousand executable lines of code during a given program's life cycle is at best 0.1, and more likely 10 to 100 times worse (Hatton, 2007).

Although reproducibility is the cornerstone of the scientific method, until recently it has not been treated seriously in software (Benureau and Rougier, 2017). Fortunately, in recent years multiple conferences, workshops and individuals are calling for dramatic change (Bailey et al., 2016). The need for action is highlighted by a study of 402 computer systems papers - only

48.3% of the code was both available and compilable (Collberg et al., 2015). (Drasil addresses this problem because as programming languages evolve the code renderers in Drasil can be updated.) Reproducibility problems are even more extreme when the goal is replicability. A third party should be able to repeat a study using only the description of the methodology from a published article, with no access to the original code or computing environment (Benureau and Rougier, 2017). However, replicability is rarely achieved, as shown for microarray gene expression (Ioannidis et al., 2009) and for economics modelling (Ionescu and Jansson, 2012). Drasil addresses completeness and ambiguity problems, since it emphasizes capturing and documenting all of the required knowledge, including derivation of equations and rationales. Crick et al. (2014) point out potential roadblocks for reproducibility, including page length constraints and differing detail needs depending on the audience. Again Drasil addresses these concerns because the recipes used to generate the documentation can be tailored to the level of detail required.

Although scientists recognize the seriousness of their SCS problems, their corrective steps are too incremental. For instance, a recent proposal for the future of High Energy Physics (HEP) software Stewart et al. (2017) uses words like sustainability, maintainability and reproducibility, but is almost completely silent on how these qualities are to be achieved. The proposal mentions developing new and improved algorithms (including parallel computing and machine learning), programming tools, recruitment and training, but there is little on documentation, design or verification techniques (other than unit testing). Similarly, a recent proposal on future directions for SCS research and education (Rüde et al., 2018) recognizes the desperate need for change, but then only suggests training on project management tools, open sharing and ethics. *Incremental change is not adequate; we need transformative change.*

Thankfully SCS leaders recognize that an interdisciplinary approach provides the path forward. They believe that the solution to SCS quality problems is applying, adapting and developing SE methods, tools and techniques. However, typical software processes are a barrier to progress. "To break the gridlock, we must establish a degree of cooperation and collaboration with the [SE] community that does not yet exist" (Faulk et al., 2009). "There is a need to improve the transfer of existing practices and tools from ... [SE] to scientific programming. In addition, ... there is a need for research to specifically develop methods and tools that are tailored to the domain" (Storer, 2017). This tailored research will require teams with a multidisciplinary background.

2.4 Documentation for SCS

Table 1 shows the recommended documentation for a scientific software project. The documents are typical of what is suggested for scientific software certification, where certification consists of official recognition by an authority, or regulatory body, that the software is fit for its intended use. For instance, the Canadian Standards Association (CSA) requires a similar set of documents for quality assurance of scientific programs for nuclear power plants CSA (1999).

Table 1: Recommended Documentation

Problem Statement	Description of problem to be solved
Development Plan	Overview of development process/infrastructure
Requirements	Desired functions and qualities of the software
V&V Plan	Verification that all documentation artifacts, including the code, are internally correct. Validation, from an external viewpoint, that the right problem, or model, is being solved.
Design Specification	Documentation of how the requirements are to be realized, through both a software architecture and detailed design of modules and their interfaces
Code	Implementation of the design in code
V&V Report	Summary of the V&V efforts, including testing
User Manual	Instructions on installation, usage; worked examples

[Add figure showing the V-model of the documentation. —SS]

2.5 MDE and Code Generation

DSLs and code/document generation provide a transformative technology for documentation, design and verification (Johanson and Hasselbring, 2018; Smith, 2018). DSLs allow scientists their preferred approach of focusing on science not software (Kelly, 2007). A generative approach removes the maintenance nightmare of documentation duplicates and near duplicates (Luciv et al., 2018), since knowledge is only captured once and automatically transformed as needed. Code generation has previously been applied to improve SCS (Whaley et al., 2001; Veldhuizen, 1998; Püschel et al., 2001). For instance, ATLAS (Automatically Tuned Linear Algebra Software) (Whaley et al., 2001) and Blitz++ (Veldhuizen, 1998) produces efficient and portable linear algebra software. Spiral (Püschel et al., 2001) uses software/hardware generation for digital signal processing. Carette and Kiselyov (2011) shows how to generate a family of efficient, type-safe Gaussian elimination algorithms. FEniCS (Finite Element and Computational Software) (Logg et al., 2012) uses code generation when solving differential equations. Unlike previous work on SCS code generation, Drasil (https://github.com/JacquesCarette/Drasil) focus on generating all software artifacts (requirements, design etc.), not just code.

Drasil, as presented in Szymczak et al. (2016) removes excuses for avoiding documentation by providing transformative SCS development technology. Drasil provides an infrastructure for knowledge capture and document/code generation. Drasil is implemented via Domain Specific Languages (DSLs) embedded in Haskell. Drasil generates requirements documentation and code for several case studies, including simulating the temperature of a solar water heating system, glass breakage and two-dimensional game physics. The full documentation (requirements, design etc), code and test cases have been created manually for each case study. These manual case studies (many from Smith et al. (2016a)) provide the 'gold standard' against which Drasil is tested. All of these artifacts are available publicly. The generation techniques for Drasil began to take shape during work on generating geometric data structures and functions (Carette et al., 2011).

2.6 Empirical Methods for Software Engineering

The empirical methods in this study will follow the guidelines given in Kitchenham et al. (2002). Since case studies will be part of the methodology, the guidelines for conducting and reporting case study research in software engineering (Runeson and Höst, 2009) will be followed.

3 Research Design and Methodologies

roadmap - fit the different pieces together

3.1 State of the Practice

A survey of existing SCS software will identify patterns, find opportunities for improvement and identify candidate program families for future implementation. The survey will require developing terminology and a classification system to answer such questions as: What are the current best practises? What software packages are reused and why these packages? What do practitioners look for in the tools that support their work?

To achieve objective S2, a measurement template, possibly using pairwise comparisons of quality metrics between software packages, will be developed for assessing the quality of the documentation, code, test cases and development processes for existing SCS program families. The quality measurement template will be used to assess the quality of approximately 30 different SCS families. About half of the families will be model oriented and the other half tool oriented. Additional details on best practices will be collected by students interviewing code owners. Based on preliminary discussions, Dr. Jeffrey Carver from the University of Alabama and some of his colleagues may also participate in collecting interview data. Once all the surveys and interviews are completed, a meta-analysis will draw conclusions for each domain and between domains. Knowing the state of practice for family development will highlight which domains are using best practices that should be emulated in LSS.

The plan here is to return to the task of measuring/assessing the state of software development practice in several scientific computing domains. We will update the work that was done previously for domains such as Geographic Information Systems Smith et al. (2018a), Mesh Generators Smith et al. (2016b), Seismology software Smith et al. (2018d), and Statistical software for psychology Smith et al. (2018c). We could return to these domains and/or introduce new domains. Potential domains/collaborators are listed in the subsections.

MEng students will be asked to assess the state of practice in each domain. Last time each student measured two domains, so we would start off with this model again. If the scope becomes large enough, we might switch to one domain per student.

In the previous project, we measured 30 software projects for each domain. With the increased scrutiny required in this re-boot, we won't likely be able to measure the details on this high a number of software projects. We may still start with 30 software examples in each domain, but then use some criteria to create a shorter sub-list for detailed measurement. The details of how to proceed here would be determined and documented in the measurement protocol.

In the previous state of practice assessing exercise, a series of questions and simple metrics was devised to measure the quality of the documentation and adherence to best software development practices. The new project will critically assess the previous set of questions and revise as necessary. In addition, the following data will be collected/developed for each domain:

- Characterization of the functionality provided by the software in the domain. What services does each member of the domain provide? Ideally this information would be summarized in a commonality analysis document. The document will summarize the domain software via its commonalities, variabilities and parameters of variation.

- Usability testing of each software in the domain. Specific metrics for assessing usability still need to be researched, but measures that involve the time to complete a task and the users overall impression would be considered.

- Collect empirical software engineering related data, such as the number of files, number of lines of code, cyclomatic complexity, number of open issues, etc. As a starting point for tool support, HubListener (`https://github.com/pjmc-oliveira/HubListener`) could be used.

- Collect at least one empirical measure of the quality of the documentation - the number of lines of documentation. [Has anyone previously looked at this metric?]

The above data will be combined to rank the software in the domain. The specifics are yet to be determined, but in the past the Analytic Hierarchy Process proved helpful in this.

Key to the success of this exercise will be to involve and engage a partner for each state of practice project. In the previous effort we didn't involve domain experts with the rationale of excluding potential bias, but this advantage is not worth not being able to evaluate the functionality/usability of the software. Moreover, not having an expert makes publication more difficult, since there is no one to advice on how best to approach journals and publishers. The domain expert will be asked to help in the following ways:

- Review the protocol for assessing/measuring quality. This same protocol will be used in each domain.
- Provide their expertise on potential publication. Specifically, they will recommend a suitable journal and act as the corresponding author for any paper submissions.
- Provide an authoritative list of domain software, possibly augmented by existing on-line lists.
- Provide some assistance from their own team of supervised students to facilitate software testing and domain characterization. Assessing the functionality and measuring usability will require an individual with domain knowledge. Multiple measurements will be necessary to have confidence. The measurements will likely take a few hours over a few days of time from each student volunteer.

There is no budget for this project, but the student volunteers will be considered co-authors in the resulting paper. Having them as co-authors also means that ethics approval should not be necessary.

GitHub will be used to coordinate the work of the large team of people that will be involved in this project. In addition to the project record left on GitHub, the final data will be exported to Mendeley.

Determine criteria for what makes a domain a good fit.

Potential collaborators/domains are listed in the following subsections.

Rule 4: Keep knowledge up to date and findable (Sholler et al., 2019).

3.1.1 Medical Imaging

Medical imaging and analysis software (Dr. Michael Noseworthy)?

3.1.2 Climate Modelling Software

Climate modelling software (Dr. Zoe Li or Dr. Xander Wang)?

3.1.3 Computational Medicine

Extraction of 3D geometries from medical images (Dr. Zahra Motamed)? Lattice Boltzmann Solver?

3.1.4 Finite Element Analysis

Finite element solvers and/or mesh generators (Dr. Dieter Stolle)? Potential software includes FeNiCS, MOOSE, FreeFEM++, Dolphin, etc. Install the packages and use them to do something non-trivial.

3.1.5 Psychology Software

Statistics software for psychology (Dr. Karin Humphreys)?

3.1.6 Chemical Engineering Software

Li Xi (https://www.eng.mcmaster.ca/chemeng/people/faculty/li-xi)?

3.1.7 Geographic Information Systems

Geographic Information Systems (Jason Brodeur and/or Patrick DeLuca)?

3.1.8 Physical Metallurgy Software

Solidification and casting software (Dr. André Phillion)?

3.1.9 Quantum Chemistry Software

Quantum chemistry software (Dr. Paul Ayers)?

3.2 Impact of MDE on SCS

roadmap

Research Question: What tools and methods from MDE are perceived by end user developers to improve their productivity and the sustainability of their software.

3.2.1 Fitting Drasil into the Scientific Software Development Process

Scientists generally use an approximation of agile methods for their software development. In addition, documentation is generally not emphasized. How should Drasil be fit into the scientific software development process? The Drasil process should encourage an early intensity of thought and effort, but at the same time, it shouldn't require waterfall development. Drasil supports frequent chance and incremental development, but where should developers start and how should the work proceed? How does one create an almost empty project in Drasil and then borrow from existing knowledge and create new knowledge? This project could start with something like the Projectile example currently in Drasil.

Drasil process should move importance of requirements and domain analysis earlier in the process. The quality of the resulting software depends on the quality of the requirements. As is known from historical data, the greatest return on investment is at the requirements state (Boehm papers). Unfortunately, developers don't like requirements. Quite possibly they don't like the intensity of thought that is necessary to get the requirements right. Inspection techniques, such as task based inspection (Kelly et al) hold promise for getting the scientist to think deeply at an earlier stage in the development process.

3.2.2 etc.

Experimentally Determine Appropriate Documentation Template: The best approach for documenting requirements and design is an open question. Drasil could be used to attempt to address this question. Different recipes for documentation could be developed and compared. Some comparison could be automated between the documents, such as empirical measures of their quality. Other measures would require some kind of usability experiments.

Empirical Measurement of Documentation: Considerable effort has been invested into developing metrics, like cyclomatic complexity, to attempt to quantify code quality, but no measures (as far as I know) exist for quantifying the quality of documentation. (Khedri does have some work with Bhahati (sp?) Sanga that quantifies the connectivity of the documentation via the traceability graph.) Likely the reason that effort has not been invested to quantifying the quality of documentation is that is not represented formally.

With Drasil much of the documentation is formal. This should facilitate automatic measures of documentation quality, or at least metrics that might be correlated with quality.

Measuring the Impact of Drasil on Developing Scientific Software: This project is challenging to design, but would be HUGE if we were successful. We assume that Drasil will improve the quality of scientific software and the efficiency of its production, but data to back this up would go a long way to convincing the scientific community. Experiments where a given project is developed in parallel by two independent teams come to mind, but this would be expensive, and difficult to control. Some additional thought, and resources, are necessary here. If we could do a project where the scientist, or a student under their supervision, is a true partner in the development, we might get more convincing anecdotal evidence.

To quantify the quality of the generated documentation, measures of information compression will be used. The measure will be based on the length of the (family of) generated documentation, code and test cases, compared to the length of the generator itself. Our measure will be based on the Minimum Description Length (MDL) Principle, which states that "the more we are able to compress the data, the more we have learned about the data" (Grünwald, 2004). Another important measure will be whether the generated documentation achieves reproducibility. The proposed measure is reproducibility depth (Soergel, 2014). A depth of 1 means that an independent researcher can start from the generated code and calculate the same results. Achieving deeper reproducibility will mean the same results can be calculated for an independent researcher starting with the design documentation. Each higher measure gets further from the source code, until the highest depth (replicability), which starts from the requirements and leads to identical (within acceptable error) results.

3.3 Impact of Assurance Cases for Building Confidence

Some SCS needs to be verifiably safe. We will achieve this by building a safety assurance case. An assurance case is an explicit structured argument pertaining to specific properties, such as trustworthiness

Assurance cases have potential to be used in SC to improve our collective confidence in the developed software Smith et al. (2018b). Further work is necessary to demonstrate the value of assurance cases, but they certainly hold promise. Drasil could go a long way to supporting development of assurance cases. Some potential Drasil related support ideas include the following:

- Automatic generation of visual depiction of the assurance case from the goals, claims, evidence etc.
- Automated reporting of missing evidence, incomplete arguments, etc.
- Coordination of requirements, design decisions, etc. Assurance cases use the same information as recommended in "document driven" design, but the traceability and connection between the data is even more important. A tool like Drasil could go a long way in facilitating the presentation and navigation of an assurance case.

Assurance cases in Drasil could potentially go beyond the scientific context. For instance, they could also incorporate safety cases for control systems. This could potentially be of interest to McSCert.

3.4 Impact of Computational Variability Testing

This activity supports objective S3. The applicant's students will apply LSS and code generation to Computational Variability Testing (CVT), using the example of a family of finite element programs. As proposed by the applicant, CVT uses code generation to build confidence in the generated code in an analogous way to the use of grid refinement studies. Grid refinement looks at how the solution changes by varying the run-time parameter of grid density and comparing the results to theoretical expectations. CVT, on the other hand, can generate code to "refine" build time parameters, such as order of interpolation, or degree of implicitness. These parameters can be systematically varied and the results compared against the expected trend. Varying computational variabilities is not a new idea, but it is only recently that the option has become available of coupling this with code generation, so that the validity of an entire program family can be assessed at the same time. The experimentation with CVT will initially use the FEniCS package (Logg et al., 2012) for elliptical partial differential equations to compare elements with different orders of interpolation to verify that the convergence to the solution agrees with the theoretical expectations.

Design and Documentation For Family of Data Fitting Algorithms: The fitting used in GlassBR and in SFS (Software for Solidification) could be made much more generic. We could have a family of fitting algorithms that could be used in any situation where fitting is required. A proper commonality analysis of this domain could potentially show the potential design decisions that bridge between the requirements and the design. In the SFS example many different fitting routines were tried. If the experiments could have been done easily via a declarative specification, considerable time would have been saved. If the experiments are combined with automated testing and "properties of a correct solution" the human involvement could be reduced, so that we have partially automated algorithm selection.

4 Implications and Contributions to Knowledge

We are currently putting too much trust in SCS. Examples such as nuclear safety analysis and computational medicine show the significance of SCS for health and safety. Although SCS developers do excellent work, we do not currently have enough checks and balances in place for full confidence. More should be expected for documentation, design and testing. Fortunately, SE offers techniques, like DSLs and code generation, that can be employed to not only improve SCS quality, but also reduce development costs. Now is the time for change. SCS practitioners have recognized problems with the status quo and multidisciplinary researchers are needed to bridge the gap between SE and SCS.

Codifying medical, biomechanical and computational knowledge is challenging, but success will completely transform the development of safety-related software in medicine, science and engineering. MDE will remove the drudgery of generating and maintaining documentation, code, test cases and build environments, enabling scientists to focus on science, engineers to focus on engineering and medical professionals to focus on medicine. MDE will detect inconsistencies between models via inter-model consistency constraints. As a consequence, mistakes like inconsistent units or assumptions cannot occur. MDE will raise the bar so that we can expect an explicit argument for safety, not just code that we are expected to blindly trust. With the right up-front investment of knowledge capture, we can have software that is long lived because the stable knowledge is separated from the rapidly changing modelling assumptions and design decisions.

5 Research Schedule

References

David H. Bailey, Jonathan M. Borwein, and Victoria Stodden. *Reproducibility: Principles, Problems, Practices*, chapter Facilitating reproducibility in scientific computing: principles and practice, pages 205–232. John Wiley and Sons, New York, 2016.

F. Benureau and N. Rougier. Re-run, Repeat, Reproduce, Reuse, Replicate: Transforming Code into Scientific Contributions. *ArXiv e-prints*, August 2017.

Jacques Carette and Oleg Kiselyov. Multi-stage programming with Functors and Monads: Eliminating abstraction overhead from generic code. *Sci. Comput. Program.*, 76(5):349–375, 2011.

Jacques Carette, Mustafa ElSheikh, and W. Spencer Smith. A generative geometric kernel. In *ACM SIGPLAN 2011 Workshop on Partial Evaluation and Program Manipulation (PEPM'11)*, pages 53–62, January 2011.

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 *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 550–559, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2828-7. doi: http://dx.doi.org/10.1109/ICSE.2007.77.

Christian Collberg, Todd Proebsting, and Alex M Warren. Repeatability and benefaction in computer systems research. Technical Report TR 14-04, Department of Computer Science, University of Arizona, Tucson, AZ, 2015. URL http://repeatability.cs.arizona.edu/v2/RepeatabilityTR.pdf.

Tom Crick, Benjamin A. Hall, and Samin Ishtiaq. "Can I implement your algorithm?": A model for reproducible research software. *CoRR*, abs/1407.5981, 2014. URL http://arxiv.org/abs/1407.5981.

CSA. Quality assurance of analytical, scientific, and design computer programs for nuclear power plants. Technical Report N286.7-99, Canadian

Standards Association, 178 Rexdale Blvd. Etobicoke, Ontario, Canada M9W 1R3, 1999.

A. P. Davison. Automated capture of experiment context for easier reproducibility in computational research. *Computing in Science & Engineering*, 14(4):48–56, July-Aug 2012.

P. F. Dubois. Maintaining correctness in scientific programs. *Computing in Science & Engineering*, 7(3):80–85, May-June 2005. doi: 10.1109/MCSE.2005.54.

S. Faulk, E. Loh, M. L. V. D. Vanter, S. Squires, and L. G. Votta. Scientific computing's productivity gridlock: How software engineering can help. *Computing in Science Engineering*, 11(6):30–39, Nov 2009. ISSN 1521-9615. doi: 10.1109/MCSE.2009.205.

Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall, Upper Saddle River, NJ, USA, 2nd edition, 2003.

Peter Grünwald. A tutorial introduction to the minimum description length principle. *ArXiv e-prints*, June 2004.

Les Hatton. The chimera of software quality. *Computer*, 40(8), August 2007.

IEEE. Recommended practice for software requirements specifications. *IEEE Std 830-1998*, pages 1–40, October 1998. doi: 10.1109/IEEESTD.1998.88286.

John PA Ioannidis, David B Allison, Catherine A Ball, Issa Coulibaly, Xiangqin Cui, Aedín C Culhane, Mario Falchi, Cesare Furlanello, Laurence Game, Giuseppe Jurman, et al. Repeatability of published microarray gene expression analyses. *Nature genetics*, 41(2):149–155, 2009.

Cezar Ionescu and Patrik Jansson. Dependently-Typed Programming in Scientific Computing — Examples from Economic Modelling. In *Revised Selected Papers of the 24th International Symposium on Implementation and Application of Functional Languages*, volume 8241 of *Lecture Notes in Computer Science*, pages 140–156. Springer International Publishing, 2012. doi: 10.1007/978-3-642-41582-1_9.

Arne N. Johanson and Wilhelm Hasselbring. Software engineering for computational science: Past, present, future. *Computing in Science & Engineering*, Accepted:1–31, 2018.

Diane Kelly. Industrial scientific software: A set of interviews on software development. In *Proceedings of the 2013 Conference of the Center for Advanced Studies on Collaborative Research*, CASCON '13, pages 299–310, Riverton, NJ, USA, 2013. IBM Corp. URL http://dl.acm.org/citation.cfm?id=2555523.2555555.

Diane F. Kelly. A software chasm: Software engineering and scientific computing. *IEEE Software*, 24(6):120–119, 2007. ISSN 0740-7459. doi: http://dx.doi.org/10.1109/MS.2007.155.

Barbara A. Kitchenham, Shari Lawrence Pfleeger, Lesley M. Pickard, Peter W. Jones, David C. Hoaglin, Khaled El Emam, and Jarrett Rosenberg. Preliminary guidelines for empirical research in software engineering. *IEEE Trans. Softw. Eng.*, 28(8):721–734, August 2002. ISSN 0098-5589. doi: 10.1109/TSE.2002.1027796. URL https://doi.org/10.1109/TSE.2002.1027796.

A. Logg, K.-A. Mardal, and G. N. Wells, editors. *Automated Solution of Differential Equations by the Finite Element Method*, volume 84 of *Lecture Notes in Computational Science and Engineering*. Springer, 2012. doi: 10.1007/978-3-642-23099-8. URL http://dx.doi.org/10.1007/978-3-642-23099-8.

D. V. Luciv, D. V. Koznov, G. A. Chernishev, A. N. Terekhov, K. Yu. Romanovsky, and D. A. Grigoriev. Detecting near duplicates in software documentation. *Programming and Computer Software*, 44(5):335–343, Sep 2018. ISSN 1608-3261. doi: 10.1134/S0361768818050079. URL https://doi.org/10.1134/S0361768818050079.

Reed Milewicz and Elaine M. Raybourn. Talk to me: A case study on coordinating expertise in large-scale scientific software projects. *ArXiv e-prints*, September 2018.

Greg Miller. SCIENTIFIC PUBLISHING: A Scientist's Nightmare: Software Problem Leads to Five Retractions. *Science*, 314(5807):1856–1857, 2006. doi: 10.1126/science.314.5807.1856. URL http://www.sciencemag.org.

Suely Oliveira and David E. Stewart. *Writing Scientific Software: A Guide to Good Style*. Cambridge University Press, New York, NY, USA, 2006. ISBN 0521858968.

David Lorge Parnas. Precise documentation: The key to better software. In *The Future of Software Engineering*, pages 125–148, 2010. doi: 10.1007/978-3-642-15187-3_8. URL http://dx.doi.org/10.1007/978-3-642-15187-3_8.

D. E. Post and L. G. Votta. Computational Science Demands a New Paradigm. *Physics Today*, 58(1):35–41, January 2005. doi: 10.1063/1.1881898.

Markus Püschel, Bryan Singer, Manuela Veloso, and José M. F. Moura. Fast automatic generation of DSP algorithms. In *International Conference on Computational Science (ICCS)*, volume 2073 of *Lecture Notes In Computer Science*, pages 97–106. Springer, 2001.

Patrick J. Roache. *Verification and Validation in Computational Science and Engineering*. Hermosa Publishers, Albuquerque, New Mexico, 1998.

U. Rüde, K. Willcox, L. McInnes, and H. Sterck. Research and education in computational science and engineering. *SIAM Review*, 60(3):707–754, 2018. doi: 10.1137/16M1096840. URL https://doi.org/10.1137/16M1096840.

Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14(2):131–164, Dec 2009. ISSN 1573-7616. doi: 10.1007/s10664-008-9102-8. URL https://doi.org/10.1007/s10664-008-9102-8.

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

Dan Sholler, Igor Steinmacher, Denae Ford, Mara Averick, Mike Hoye, and Greg Wilson. Ten simple rules for helping newcomers become contributors to open projects. *PLOS Computational Biology*, 15(9):1–10, 09 2019. doi: 10.1371/journal.pcbi.1007296. URL https://doi.org/10.1371/journal.pcbi.1007296.

W. Spencer Smith. Beyond software carpentry. In *2018 International Workshop on Software Engineering for Science (held in conjunction with ICSE'18)*, pages 1–8, 2018.

W. Spencer Smith and Nirmitha Koothoor. A document-driven method for certifying scientific computing software for use in nuclear safety analysis. *Nuclear Engineering and Technology*, 48(2):404–418, April 2016. ISSN 1738-5733. doi: http://dx.doi.org/10.1016/j.net.2015.11.008. URL http://www.sciencedirect.com/science/article/pii/S1738573315002582.

W. Spencer Smith, Yue Sun, and Jacques Carette. Comparing psychometrics software development between CRAN and other communities. Technical Report CAS-15-01-SS, McMaster University, January 2015. 43 pp.

W. Spencer Smith, Thulasi Jegatheesan, and Diane F. Kelly. Advantages, disadvantages and misunderstandings about document driven design for scientific software. In *Proceedings of the Fourth International Workshop on Software Engineering for High Performance Computing in Computational Science and Engineering (SE-HPCCE)*, November 2016a. 8 pp.

W. Spencer Smith, Adam Lazzarato, and Jacques Carette. State of practice for mesh generation software. *Advances in Engineering Software*, 100:53–71, October 2016b.

W. Spencer Smith, Adam Lazzarato, and Jacques Carette. State of the practice for GIS software. https://arxiv.org/abs/1802.03422, February 2018a.

W. Spencer Smith, Mojdeh Sayari Nejad, and Alan Wassyng. Assurance cases for scientific computing software (poster). In *ICSE 2018 Proceedings of the 40th International Conference on Software Engineering*, May 2018b. 2 pp.

W. Spencer Smith, Yue Sun, and Jacques Carette. Statistical software for psychology: Comparing development practices between CRAN and other communities. https://arxiv.org/abs/1802.07362, 2018c. 33 pp.

W. Spencer Smith, Zheng Zeng, and Jacques Carette. Seismology software: State of the practice. *Journal of Seismology*, 22(3):755–788, May 2018d.

David Soergel. Confirmation depth as a measure of reproducible scientific research. http://davidsoergel.com/posts/confirmation-depth-as-a-measure-of-reproducible-scientific-research, October 2014.

Graeme Stewart et al. A Roadmap for HEP Software and Computing R&D for the 2020s. *arXiv*, 2017.

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. ISSN 0360-0300. doi: 10.1145/3084225. URL http://doi.acm.org/10.1145/3084225.

Daniel Szymczak, W. Spencer Smith, and Jacques Carette. Position paper: A knowledge-based approach to scientific software development. In *Proceedings of SE4Science'16 in conjunction with the International Conference on Software Engineering (ICSE)*, Austin, Texas, United States, May 2016. In conjunction with ICSE 2016. 4 pp.

Todd. L. Veldhuizen. Arrays in Blitz++. In *Proceedings of the 2nd International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE'98), Lecture Notes in Computer Science*. Springer-Verlag, 1998.

R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27(1–2): 3–35, 2001.