Beyond Software Carpentry

Spencer Smith
Computing and Software Department, McMaster University
1280 Main Street West
Hamilton, Ontario L8S 4K1
smiths@mcmaster.ca

ABSTRACT

About 20 years ago the need for scientists and engineers to have basic knowledge of software development skills and tools became apparent. Without these so-called software carpentry skills, developers were wasting time and compromising the quality of their work. Since that time great progress has been made with software carpentry, as evidenced by the growing understanding of the importance of tools, and by the growth of the namesake Software Carpentry foundation and other similar projects. With scientific software developers now prepared to move forward, we should turn our attention to the next logical step after carpentry: Software Engineering (SE) applied to Scientific Computing Software (SCS). Past attempts with SE for SCS have not always been successful; therefore, this paper proposes a vision for future success, including SE specifically adapting ideas to SCS, SCS recognizing the value of software artifacts other than the code, and all parties increasing the emphasis on empirical evidence and the quality of replicability. Several ideas are proposed for turning the proposed vision into a reality, including promoting requirements documentation for replicability, building assurance cases for correctness (and other qualities), and automatic generation of all documentation and code.

CCS CONCEPTS

• **Mathematics of computing** → *Mathematical software*; • **Software and its engineering** → *Requirements analysis*; *Software verification and validation*;

KEYWORDS

software engineering, scientific computing, requirements analysis, assurance cases, document generation

ACM Reference format:
Spencer Smith. 2018. Beyond Software Carpentry. In *Proceedings of SE4Science'18: IEEE/ACM International Workshop on Software Engineering for Science, Gothenburg, Sweden, June 2, 2018 (SE4Science'18),* 8 pages.
https://doi.org/10.1145/3194747.3194749

In 1998 the volunteer organization Software Carpentry was formed with the goal of improving the productivity of scientists by teaching them basic computing skills, such as version control, programming and task automation (https://software-carpentry.org/). In this paper, the term software carpentry, in lower case letters, will be used to denote these skills, while the same term with leading capital letters will reference the Software Carpentry foundation. Thanks to the efforts of this organization, and others, the situation has greatly improved over the last 20 years. For instance, in 2006 only 11% of Wilson's software development skills graduate class had prior experience with version control software [46], but in a similar course that I taught in the Fall of 2017, almost 70% of the class had prior experience with version control. Furthermore, in my class the students from outside of computer science (physics, biology etc.) were just as likely to be familiar with software development tools as their computer science colleagues.

The work of the Software Carpentry volunteers, and the efforts of scientists, should be commended. Although there is still room for improvement in the basic software development skills for scientists, the time seems ripe to ask the question: What is the next step for improving the quality Scientific Computing Software (SCS) and the productivity of SCS developers? What should software engineers and tool developers aim for as the scientific community prepares to move beyond software carpentry?

The need to move past carpentry is obvious if we explore the carpentry/engineering analogy. For instance, carpentry is adequate for framing residential homes, but engineering is required to construct large bridges, institutional buildings, nuclear power plants, etc. Just as for physical construction projects, as SCS problems get larger and more complex, carpentry alone is not adequate. As Parnas defined it, Software Engineering (SE) is necessary for "multi-person construction of multi-version programs" [21]. The need for engineering is even more pronounced when one considers the software developer's responsibility to the public. We need to ensure that our confidence in SCS software is not misplaced, especially when it is employed for design, analysis and decision making in areas that impact health and safety, such as nuclear safety analysis and medical imaging. Ensuring confidence will generally involve documentation and mathematical specification. Historically, these tasks have not always been popular with SCS developers, so the path forward needs to be carefully planned.

An example that highlights the need for increasing the emphasis of SE is the recent white paper proposal for future High Energy Physics (HEP) software development [41]. HEP software is used to support the Large Hadron Collider (LHC) facility at CERN as researchers attempt to answer fundamental questions about the structure of the universe. The HEP software is huge, with over 5 million lines of code [20], and yet the white paper says very little about SE. The software that has been created is an impressive accomplishment, and the writers of the paper identify important challenges,

ACM ISBN 978-1-4503-5748-7/18/06…$15.00
https://doi.org/10.1145/3194747.3194749

like the enormous amounts of data that will need to be processed, technology transitions and training. However, the emphasis in the report is almost completely on the code and tools. Although words like sustainability, maintainability and reproducibility are used, little is mentioned about how these qualities will be achieved, or how the theories and algorithms will be documented and inspected so that others outside the immediate community can build confidence in the software and its results.

Section 1 presents evidence for the success of software carpentry and thus lays the foundation for the transition to a future emphasis on SE. Although SE for SCS has been attempted in the past, the efforts have often not been successful; therefore, Section 2 outlines a potential vision for future success in research in SE for SCS. Specific ideas for future SE for SCS research, consistent with the presented recommendations, are given in Section 3.

1 SUCCESS OF SOFTWARE CARPENTRY

The success of the concept of software carpentry is evident by the success of the foundation for Software Carpentry. In a four year period between 2011 and 2015, Software Carpentry delivered over 500 workshops, impacted over 16 000 learners and engaged over 450 instructors [48]. Since 2015, the number of learners impacted has grown to over 22 000 (https://software-carpentry.org/about/). Software Carpentry has not only engaged the scientific community; it has improved the productivity of software developers. Although difficult to measure [48], positive evidence of impact is available. Software Carpentry workshops participants have shown a two-fold (130%) improvement in performance on a computing skills test [2] and an overwhelming number (95%) of workshop participants have reported that they would recommend the workshop to others [2]. Pre and post workshop surveys have shown that participants end the workshop with higher perceptions of their computational ability, computational understanding and Python coding skills [19]. Moreover, Simperler and Wilson [30] show evidence that a majority of workshop participants were able to use their new skills to improve their productivity. The data in a recent report of the Software Carpentry's post-workshop surveys [12] shows, for instance, that pre-workshop 72% of participants have no experience with Git, while post-workshop 88% said their confidence increased by at least a bit, and almost 50% said their confidence increased greatly.

The success of the Software Carpentry organization is far from the only evidence of the growing realization of the importance of computing skills for scientific software developers. Since the time of Software Carpentry's inception in 1998, many researchers have investigated software engineering practices applied to scientific programming. A recent survey of this topic [43] has 194 entries in the bibliography and a recent book [4] provides advice, research results and case studies of software engineering applied to SCS. Other organizations besides Software Carpentry have cultivated the importance of sustainable research software, such as the SSI (Software Sustainability Institute) (https://www.software.ac.uk/) and WSSSPE (Working toward Sustainable Science for Software: Practice and Experiences) (http://wssspe.researchcomputing.org.uk/). Governments have invested in research to improve scientific software for research, such as the recent Software Infrastructure for Sustained Innovation (S2I2) grants for the NSF (National Sciences Foundation) in the United States and the Research Software Program for Canarie in Canada.

The proliferation of the use of Concurrent Versioning Systems (CVS) is further evidence of the growth in the use of software carpentry tools. The poor adoption rate that Wilson lamented in 2006 [49] has greatly improved in the intervening years, as shown by Table 1. This table shows CVS usage in several scientific domains both in 2014 and in 2018. The percentages are based on software quality studies for domains such as mesh generation [36], seismology software [39], Geographic Information Systems [18] and statistical software for psychiatry [38]. The studies each identified approximately 30 software packages from their domain using authoritative lists produced by the respective communities. In Table 1 Alv and Ded stand for Alive and Dead, respectively. A software package is considered dead if it has not been updated in over 18 months. CV stands for Concurrent Versioning and the suffix A and D, refer to the alive and dead software, respectively. If CVS usage for a particular project cannot be gleaned from the available information, it is conservatively assumed to not employ a CVS. With the exception of the seismology domain (in both years) and the statistics domain in 2014, the percentages in the CVA column are quite high, especially for 2018.

Table 1: Usage of Version Control Software

	2014				**2018**			
Dom	**Alv**	**CVA**	**Ded**	**CVD**	**Alv**	**CVA**	**Ded**	**CVD**
Mesh	59%	75%	41%	27%	41%	100%	59%	38%
Seis	73%	41%	27%	0%	37%	36%	63%	37%
GIS	63%	89%	37%	9%	63%	95%	37%	9%
Stats	80%	17%	23%	0%	47%	100%	53%	0%

Although there is certainly still work to do in promoting software carpentry to computational scientists and engineers, as shown by lower than expected adoption rates for CVS for seismology software, the situation is certainly much improved since the start of Software Carpentry. The work of organizations such as Software Carpentry should certainly continue, but it also seems appropriate to investigate a transition from software carpentry to software engineering. Previous attempts with applying SE ideas to SCS have not always been successful; therefore, in the next section, we discuss a vision for future software engineering research.

2 RECOMMENDED VISION

Although the reality is more nuanced, for the purpose of discussion, a simplified version of the history of SE for SCS starts with software engineers attempting to apply "textbook" SE methods to SCS. As the case study of Segal [27] highlights, traditional SE failed to meet the expectations of scientists. Roache was also negative toward at least some aspects of SE, considering reports for each stage of software development as counterproductive [25, p. 373]. Based on reports and experiences like these, the gap between SE and SCS has been characterized as a chasm [16].

In our simplified history, the bad experiences with early applications of SE led to an increased emphasis by SE researchers on understanding SCS developers. Multiple studies have been conducted to understand how SCS software is developed [5, 7–9, 14]. At times the failure of traditional SE and the results of the studies have led to conclusions that SCS should possibly be developed using an agile approach [1, 5, 7, 27], or an amethododical process [13], or a knowledge acquisition driven process [14]. Although the current SCS approach may turn out to be the best, the fact that it is usually (unconsciously) used by developers is not evidence on its own that it is the ideal approach. Attempts should at least be made to adapt SE knowledge that has been successfully applied in other domains, like safety critical real time systems, to SCS software.

Our admittedly simplified view of the current relationship between SE and SCS motivates two recommendations:

(1) **SE methods, tools and techniques should be specifically tailored to SCS.**
(2) **SCS developers should recognize the value of documentation. SE is not just programming languages and tools; it is a true engineering discipline.**

Documentation, written before and during development, can provide many benefits [23]: 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.

An argument could be made that previous attempts at collaboration between SE and SCS was premature. Maybe SCS developers needed to master software carpentry first, and software engineers needed to learn the nuances of SCS? Success going forward requires a true collaboration, one that is only possible by working together. Therefore, the next recommendation is:

(3) **Software engineers and scientific software developers should work together on real projects of mutual interest, from the start of the project.**

Everyone is busy, so to achieve full engagement, the projects have to have something of value for everyone. This lesson was reinforced in a study of document driven design for scientific software [32]. Many of the conclusions were uncertain because the software developers did not have time to review the full documentation for a redeveloped version of their project; they had moved on to other projects.

As in any scholarly activity, our path forward should be guided by empirical evidence. Fortunately there is a growing emphasis on empirical studies for software engineering research [47]. This trend should also be applied to SCS software:

(4) **Research on SE methods, tools and techniques for SCS should include empirical evidence whenever possible.**

The final recommendation is:

(5) **SCS software should include documentation that improves the quality of replicability**

Replicability means that the documentation provided for the software, not the software itself, is sufficient for an independent third party to reproduce the computational results. As described by Benureau and Rogier [3], scientific software should achieve five successively more difficult qualities: re-runnable, repeatable, reproducible, reusable and replicable. Although progress has been made up to the level of reproducibility, through the use of Virtual Machines (VMs), VMs do not help when there are problems in the original code. What if our trust in the original code is in doubt? What if we need to reproduce the results not starting from the code, but starting from the original theory? Unfortunately, multiple examples exist [6, 11] where results from research software could not be independently reproduced, due to a need for local knowledge missing from published documents. Examples of missing knowledge includes missing assumptions, derivations, and undocumented modifications to the code. In the future, we wish for it to be easier to independently replicate the work of others, starting from their theory documentation, without needing to rely on their code.

3 POTENTIAL FUTURE DIRECTIONS

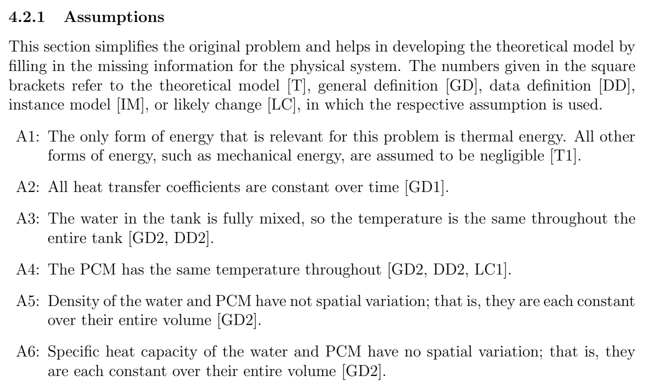
This section provides three potential ideas for future research on SE applied to SCS. There is no effort to be exhaustive. The ideas listed were selected because they appear promising and little prior work exists for them, and because these are areas with which the author is familiar. Ideas like testing and code inspection, were excluded because they have already received attention in the SE for SCS literature. For each of the ideas, it is first described and then potential research steps are listed. Where appropriate reference is made to the motivating recommendations (Rec #) from Section 2.

3.1 Requirements for Replicability

A Software Requirements Specification (SRS) describes the functionalities, expected performance, goals, context, design constraints, external interfaces, and other quality attributes of the software [10]. In a scientific context, an SRS records the necessary terminology, notations, symbol definitions, units, sign conventions, physical system descriptions, goals, assumptions, theoretical models, data definitions, instanced models and data constraints [35]. With this definition, this section's heading "Requirements for Replicability" can be read as both posing a question and providing an answer. The question is: "What are the *requirements for replicability*?" and the answer is: "*Requirements* are needed *for replicability*."

To achieve Rec 5, we need an SRS, since journal articles are inadequate for an independent researcher to replicate computational results. The problem for journals is that the constraints on space and scope make it impossible to record all details. An SRS provides a place to record everything that is necessary. The SRS can systematically be reviewed [33] to check that every symbol is defined; that every symbol is used at least once; that every equation either has a derivation, or a citation to its source; that every general definition, data definition, and assumption is used by at least one other component of the document. When the code is complete, verification can be done to ensure that every line of code either traces back to a description of the numerical algorithm (in the design documentation), to a data definition, to an instance model, to an assumption, or to a value from the auxiliary constants table in the SRS.

An example SRS for a Solar Water Heating System (SWHS) incorporating Phase Change Material (PCM) [26] is available at: https://github.com/smiths/swhs/tree/master/SRS. SWHS simulates the temperature of the water and the PCM in a solar water heated tank over time. The purpose of the PCM is to reduce the tank size, since PCM stores thermal energy as latent heat, which allows higher thermal energy storage capacity per unit weight. Figure 1 shows an excerpt from the SRS for SWHS showing some of the assumptions that are used for the theory and the derived models. This is the kind of information that is necessary for another researcher to verify, and potentially replicate, the computational results.



4.2.1 Assumptions

This section simplifies the original problem and helps in developing the theoretical model by filling in the missing information for the physical system. The numbers given in the square brackets refer to the theoretical model [T], general definition [GD], data definition [DD], instance model [IM], or likely change [LC], in which the respective assumption is used.

A1: The only form of energy that is relevant for this problem is thermal energy. All other forms of energy, such as mechanical energy, are assumed to be negligible [T1].

A2: All heat transfer coefficients are constant over time [GD1].

A3: The water in the tank is fully mixed, so the temperature is the same throughout the entire tank [GD2, DD2].

A4: The PCM has the same temperature throughout [GD2, DD2, LC1].

A5: Density of the water and PCM have not spatial variation; that is, they are each constant over their entire volume [GD2].

A6: Specific heat capacity of the water and PCM have no spatial variation; that is, they are each constant over their entire volume [GD2].

Figure 1: Sample assumptions for SWHS.

An SRS provides many benefits for scientific software, as discussed in [34] and [35]. In addition to *replicability*, discussed above, other benefits include:

- An SRS improves the *verifiability* of the code. If the theory is not explicitly recorded, other researchers may have a different understanding of the assumptions applied. Without a clear statement of what the software is supposed to do, one cannot judge its correctness.
- An SRS improves *maintainability*. Over time the requirements will inevitably change. The SRS facilitates change by capturing the traceability between knowledge in the SRS (as shown in Figure 1) and between the SRS and code modules.
- An SRS improves communication, which in turn improves *training*. In large SCS projects, like software for HEP, getting new developers up to speed is challenging. An SRS will facilitate their training by providing an introduction and answering questions that require understanding definitions, sign conventions, assumptions, etc.

Documenting requirements is not particularly popular for scientific software. In part because of the opinion [5, 29] that requirements are impossible to determine up-front. However, there is nothing about the SRS that requires it to be created following a waterfall process. As Parnas and Clements [22] point out, documentation can be "faked" as if a rationale (waterfall) process were followed. Some other arguments debunking the common complaints against requirements documentation include: scientific theories have a high potential for reuse (with the right abstraction); scientific theories are fairly stable over time; variabilities for a family of related theories are predictable; and, the usual design pattern for SCS is the relatively simple pattern: Input $\Rightarrow$ Calculate $\Rightarrow$ Output [31].

If SCS developers are to recognize the value of SRS documents (Rec 1), then the SE approach for collecting, documenting and verifying requirements will need to be tailored to the SCS community (Rec 2). Some specific ideas on how to accomplish this follow.

(1) Investigate the best SRS template for SCS requirements. Filling in the template sections is like following a checklist for providing the standard information. For SCS, a specific template is available [34, 35]. However, this template has not been empirically validated (Rec 4). This template, and others inspired by the required documentation in the nuclear, medical and aeronautical domains, should be studied for their effectiveness in real projects. This study can be facilitated by the document generation techniques in Section 3.3.
(2) As mentioned in Rec 3, the SRS should be applied to real projects. A good starting point is projects that require certification, where certification consists of official recognition by an authority, or regulatory body, that the software is fit for its intended use. Examples domains include nuclear safety, automotive safety, medical imaging, etc.
(3) Although verification and inspection techniques have been developed for SCS code, little attention has been paid to verification of the other design artifacts, including the SRS. One option may be a task-based inspection approach, like that used by Kelly and Shepard [15] for code. One potential option may be to assigned reviewers a set of questions, like "Are the units consistent in each equation?", "Are all symbols defined?" and "Is the information on a given topic sufficient to implement a solution?". The questions could be managed over GitHub (or equivalent issue tracker) to take advantage of the growth of software carpentry skills for SCS developers. It will also be necessary to verify that the implementation provided matches the given requirements. Verifying the pieces, and how they all fit together, can be approached using assurance cases as discussed in Section 3.2.
(4) As the success of software carpentry shows, SCS developers are willing to adopt tools. Therefore, SE tools should be specifically tailored for SCS (Rec 1). For instance, tools should be developed to assist with verification of properties like consistency and completeness of an SRS. Such tools have been developed by the R community, and they have facilitated an improvement of software quality so that a single developer can produce code of comparable quality to a team of developers [38].
(5) A potential high impact tool for SRS documentation is presented in Section 3.3. With this tool, it may be possible to automatically generate and SRS from a scientific knowledge base combined with recipes pulling out the required information and formatting guidelines. This will facilitate automatic verification, in the same sense that a compiler complains when there is missing information. If the SRS can be generated, then many of the complaints the community has about requirements documentation can be removed [32].

3.2 Assurance Case

From [24], an assurance case is "[a] reasoned and compelling argument, supported by a body of evidence, that a system, service or organization will operate as intended for a defined application in a defined environment." Assurance cases have been successfully employed for safety critical systems, but using this technique for

SC software is a new idea. The Goal Structuring Notation (GSN) [40] seems like a good framework for initial investigation.

Scientific software, such as medical software, is often subject to standardization and regulatory approval. While applying such approvals and standards has had a beneficial effect on system quality, it does not provide good tracking of the development stages, as the compliance with the standards is mostly checked after the system development. Once a system is implemented, its documentations must be approved by the regulators. This process is lengthy and expensive. In contrast, assurance case development usually occurs in parallel with the system construction, resulting in a traceable, detailed argument for the desired property. Moreover, since the assurance case is integrated into the system construction, the domain experts are involved from the start.

Putting the argument in the hands of the experts means that they will work to convince themselves, along with the regulators. They will use the expertise that the regulators do not have and they will see the value of documentation (Rec 2); they will be engaged. This engagement will hopefully help bridge the current chasm between software engineering and scientific computing [16], by motivating scientists toward documentation and correcting the problem of software engineers failing to meet scientists' expectations [28]. Significant documentation will still likely be necessary, but now the developers control the documentation. What is created will be relevant and necessary.

For SCS a top level claim could read "Program X delivers correct outputs when used for its intended use/purpose in its intended environment." The next step would be to decompose this claim into sub-claims that will be easier to prove. The sub-claims will likely also be further divided until the bottom of the graph, where the measurable evidence is provided. Typical evidence will consist of documents, expert reviews, test case results etc.

Using the specific example of preparing an assurance case for preexisting medical image analysis software (3dfim+) [37], the top level can be decomposed into four sub-goals, as shown in Figure 2. This example follows the same pattern as used for medical devices [45]. The first sub-goal (GR) argues for the quality of the documentation of the requirements (SRS). The second sub-goal (GD) says that the design complies with the requirements and the third proposes that the implementation also complies with the requirements. The fourth sub-goal (GA) claims that the inputs to 3dfim+ will satisfy the operational assumptions, since we need valid input to make an argument for the correctness of the output.

Preparing an assurance case for the pre-existing 3dfim+ software justifies the value of assurance cases for the certification of SCS [39]. Although no errors were found in the output of the existing software, the rigour of the proposed approach did lead to finding ambiguities and omissions in the existing documentation, such as missing information on the coordinate system convention. In addition, a potential concern for the software itself was identified from the GA argument: running the software does not produce any warning about the obligation of the user to provide data that matches the parametric statistical model employed for the correlation calculations.

Potential objectives for research on assurance cases applied to SCS are as follows:

(1) Creation of an assurance case template will require looking at a variety of SCS projects, which will require engaged project partners (Rec 3); therefore, the initial approach should be to target domains where certification is required, or domains where verification is critical, but challenging, like the High Energy Physics (HEP).
(2) Evidence is provided at the bottom of an assurance case, but what is the best form for this evidence? As mentioned for the SRS, we need an approach for verification. We also need an approach to verify the other software artifacts, the traceability between them, the team of developers, the team of reviewers, etc. The question for the future is what combination of tools and processes will provide convincing evidence?
(3) Creation of assurance cases is challenging with existing tools. In the future, the goal should be automatic generation (see Section 3.3) of the visual assurance case documentation from the arguments and evidence
(4) Creating a revelation on the value of documentation (Rec 2) may require a pseudo adversarial approach. SCS developers should be explicitly challenged to present their verification efforts so that their work can be independently verified. Although SCS developers have developed many successful theories, techniques, and testing procedures for verification, the evidence is generally presented in an ad hoc manner. Developers should be asked to connect all the pieces of their evidence in a coherent argument. As a bonus, the act of creating the assurance case may also lead developers to discover subtle edge cases, which would not have been noticed with a less rigorous and systematic approach.

3.3 Generate All Things

A knowledge-based approach for scientific software development holds promise. The SRS (Section 3.1) and other software artifacts are considered useful to developers, but many believe that the imposed workload is too onerous [32]. Ideally, developers should be able to create high quality documentation without the drudgery of writing and maintaining it. A potential solution is to generate the documentation automatically by using Domain Specific Languages (DSLs) over a base of scientific knowledge. This is the approach that is proposed for a new scientific software development framework called Drasil [44]. The Drasil framework will be described below as one potential approach for software artifact generation. The presentation focuses on what a tool like Drasil could accomplish, although the Drasil framework itself is far from complete. The current version of Drasil can be found at: https://github.com/JacquesCarette/Drasil.

The principal perceived drawbacks of document-driven design methodologies are [44]:

- information duplication,
- synchronization headaches between artifacts,
- an over-emphasis on non-executable artifacts.

Successfully improving software qualities, such as verifiability, reliability, and usability, while also improving, or at least not diminishing performance, requires finding a way to simultaneously deal with the above drawbacks [44]. A generative frameworks also adds the goal of improving developer productivity, and saving time and money for certification and re-certification. To accomplish this,

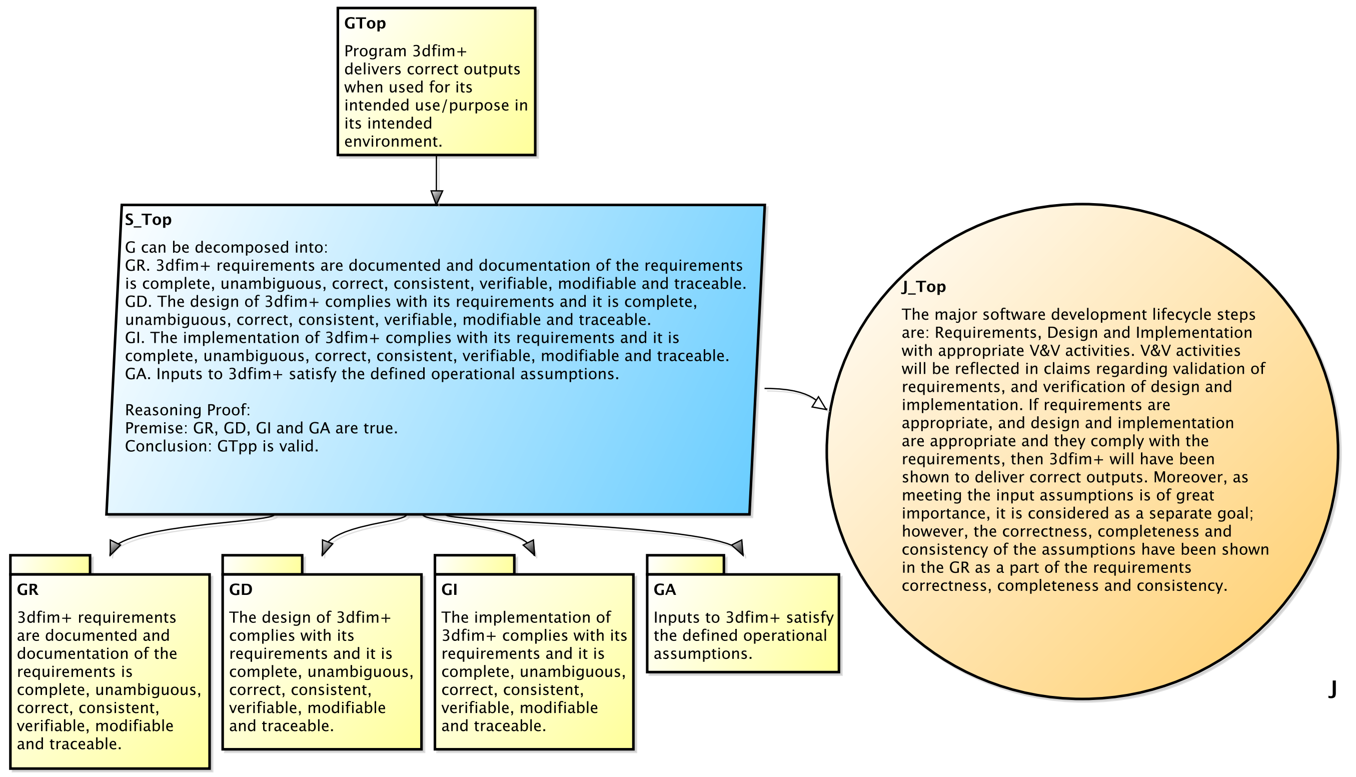

GTop
Program 3dfim+ delivers correct outputs when used for its intended use/purpose in its intended environment.
S_Top
G can be decomposed into:
GR. 3dfim+ requirements are documented and documentation of the requirements is complete, unambiguous, correct, consistent, verifiable, modifiable and traceable.
GD. The design of 3dfim+ complies with its requirements and it is complete, unambiguous, correct, consistent, verifiable, modifiable and traceable.
GI. The implementation of 3dfim+ complies with its requirements and it is complete, unambiguous, correct, consistent, verifiable, modifiable and traceable.
GA. Inputs to 3dfim+ satisfy the defined operational assumptions.
Reasoning Proof:
Premise: GR, GD, GI and GA are true.
Conclusion: GTpp is valid.
J_Top
The major software development lifecycle steps are: Requirements, Design and Implementation with appropriate V&V activities. V&V activities will be reflected in claims regarding validation of requirements, and verification of design and implementation. If requirements are appropriate, and design and implementation are appropriate and they comply with the requirements, then 3dfim+ will have been shown to deliver correct outputs. Moreover, as meeting the input assumptions is of great importance, it is considered as a separate goal; however, the correctness, completeness and consistency of the assumptions have been shown in the GR as a part of the requirements correctness, completeness and consistency.
J
GR
3dfim+ requirements are documented and documentation of the requirements is complete, unambiguous, correct, consistent, verifiable, modifiable and traceable.
GD
The design of 3dfim+ complies with its requirements and it is complete, unambiguous, correct, consistent, verifiable, modifiable and traceable.
GI
The implementation of 3dfim+ complies with its requirements and it is complete, unambiguous, correct, consistent, verifiable, modifiable and traceable.
GA
Inputs to 3dfim+ satisfy the defined operational assumptions.

Figure 2: Top Goal of the assurance case and its sub-goals

tools like Drasil need to remove duplication between artifacts and provide traceability between all artifacts. In practice, this means providing facilities for automatic software artifact generation from high level "knowledge." Drasil accomplishes this by having a single "source" for each relevant piece of information, as illustrated by the roots of the tree in Figure 3. From this source Drasil generates, via recipes, all required documents and views, as shown by the branches of the tree. Drasil aims to provide methods, tools and techniques to support a literate process for developing scientific software analogous to Knuth's [17] Literate Programming, but more general. Unlike other document generation tools, like LP and Doxygen, the focus is on all software artifacts (SRS, design documents, test plans, build scripts, etc), not just the code and its comments.

As mentioned in the requirements discussion (Section 3.1), generative frameworks can also potentially check for completeness and consistency. For example, every symbol could be checked for a definition, along with a verification that the same symbol is not given more than one definition, at least not within the same name space. In the case where different symbols have the same meaning, like during the transition from a typeset document to ASCII code, the traceability can be explicitly identified between the symbols and the concept. The generation process can also check that the specification is complete and disjoint. That is, if behaviour is only given for $x > 0$, the tool can ask about the case $x \leq 0$. The tool could also highlight cases of overlap, say when behaviour is specified for $x \geq 0$ and $x \leq 0$.

Drasil works by building a representation of the scientific and computing knowledge, including scientific theories (like the conservation laws), data definitions (like the definition of a coordinate system), assumptions (like dimensionality), and algorithms (like interpolation or solving systems of equations). As an example, the following equation is part of the knowledge for software to calculate the risk of failure (B) for a plane of glass with dimensions a by b; thickness h; material properties m, k and E; load duration factor LDF; and, stress distribution factor J.

$$B = \frac{k}{(a \times b)^{m-1}}((E \times 1000)(h)^2)^m \times \mathrm{LDF} \times e^J$$

In Drasil this knowledge can be represented in a generic form using the following code.

```
risk_eq :: Expr
risk_eq = ((C sflawParamK) / (Grouping ((C plate_len) *
  (C plate_width))) :^ ((C sflawParamM) - 1) *
  (Grouping (C mod_elas * 1000) *
  (square (Grouping (C act_thick)))):^ (C sflawParamM) *
  (C lDurFac) * (exp (C stressDistFac)))
```

Drasil can then take this captured knowledge and generate the corresponding data definition for the SRS. The SRS itself can be in formats such as LaTeXor html. The same knowledge can then be translated into different programming languages, like Python, Java or C#. Once the investment is made in capturing the knowledge,

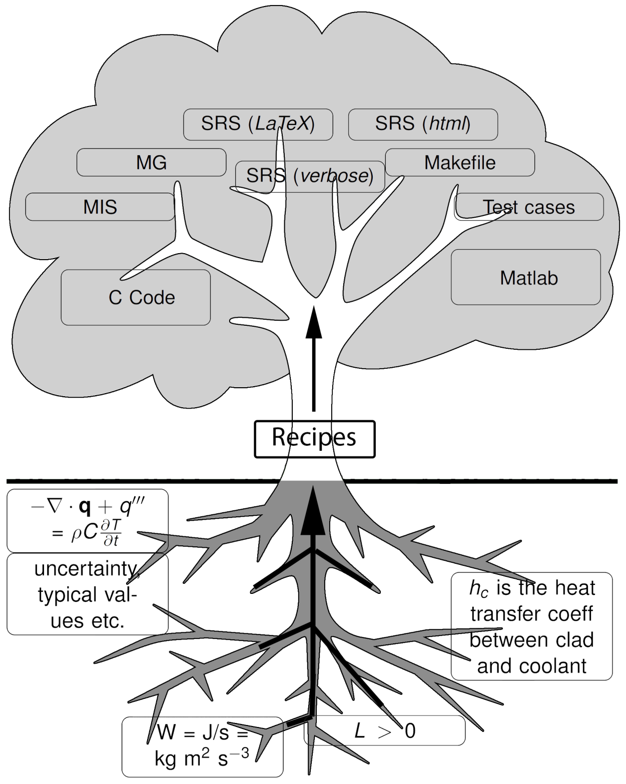

SRS (*LaTeX*)
SRS (*html*)
MG
SRS (*verbose*)
Makefile
MIS
Test cases
C Code
Matlab
Recipes
$-\nabla \cdot \mathbf{q} + q''' = \rho C \frac{\partial T}{\partial t}$
uncertainty, typical values etc.
h_c is the heat transfer coeff between clad and coolant
W = J/s = kg m^2 s^{-3}
$L > 0$

Figure 3: Drasil Knowledge Tree

regeneration for changes, or reuse for new problems, is a mechanical/automated, process. For instance, if a change is made in the original Drasil code, regeneration will fix the error in all generated targets. The generator can also include selection of design choices, like whether to include logging information in the generated code.

Initial development of Drasil began in 2014 with 5 case study applications, including glass breakage, solar water heating, slope stability analysis, etc. Scientific software developers that have learned about the project are excited about the documentation, but they do not have the time or expertise to work with the current Domain Specific Languages, written in Haskell. Future work is necessary to create a tool that will be more complete and practically useful. Specific steps going forward are listed below.

(1) Although a framework like Drasil could start by building an ontology of scientific knowledge, a more bottom up, example driven, approach seems more fruitful. This means that going forward more examples from the SCS community (Rec 3) will need to be converted into the Drasil language, and the language refactored to work with the new knowledge. A Grounded Theory (GT) approach could be used for building the Drasil language, and organizing the scientific knowledge. GT is appropriate because the problem scope is too large for a top down approach, but in GT the theory is inductively generated from data [42].
(2) In the future, the utility of document and code generation can be illustrated by exploring Computational Variability Testing (CVT). For numerical techniques that rely on a grid, like partial differential equation solvers, CVT can use code generation to build confidence in the generated code, analogous to how grid refinement is currently employed. Grid refinement calculates solution changes by varying run-time parameters like grid density. 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.
(3) In the future, code generation can be employed to optimize over build-time parameters. Optimization is generally done over run-time parameters, but code generation allows for variations that usually occur at build time. If the variations are computational parameters, then the idea of CVT can be extended to improve performance, since the different combinations of variabilities can be profiled to optimize performance. Another option is variation of modelling parameters through code generation, including variations that change the underlying assumptions. For instance, an optimization could be done over the constitutive equation to allow variation from elastic to elastoviscoplastic behaviour.

4 CONCLUDING REMARKS

The arguments in this paper suggest that, thanks to the efforts of the Software Carpentry Foundation, and others, software carpentry practices have improved the quality of SCS and the productivity of its developers. The improvements have started with basic software tools, like version control and task automation. If their construction is made a priority, more powerful tools can be available in the future, like tools for document and code generation. If the carpentry analog for the current tools are hammers and hand saws, then an appropriate analog for future tools will be 3D printers. Moreover, with current and future software carpentry tools, it should be possible to move beyond software carpentry to SE. Staying with our construction analogy, just like engineers focus on the "big picture" of solving problems, future computational scientists will be able to focus on their science, rather than worry about documentation and programming details.

The paper recommends some goals and priorities for future research on SE applied to SCS, such as SE specifically adapting ideas to SCS, SCS recognizing the value of software artifacts other than the code, and all parties increasing the emphasis on empirical evidence and the quality of replicability. Several ideas are proposed for turning the proposed vision into a reality, including promoting requirements documentation for replicability, building assurance cases for correctness (and other qualities), and automatic generation of all documentation and code.

ACKNOWLEDGEMENTS

The section on Drasil benefits greatly from collaboration with several colleagues, including Jacques Carette, Daniel Szymczak and Steven Palmer. The financial support of the Natural Sciences and Engineering Research Council (NSERC) is gratefully acknowledged.

REFERENCES

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

[2] Jorge Aranda. Software carpentry assessment report. https://software-carpentry.org/files/bib/aranda-assessment-2012-07.pdf, July 2012.

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

[4] Jeffrey C. Carver, Neil P. Chue Hong, and George K. Thiruvathukal, editors. *Software Engineering for Science*. Chapman & Hall/CRC Computational Science. Chapman and Hall/CRC, 2016.

[5] 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.

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

[7] Steve M. Easterbrook and Timothy C. Johns. Engineering the software for understanding climate change. *Comuting in Science & Engineering*, 11(6):65–74, November/December 2009.

[8] Jo Erskine Hannay, Carolyn MacLeod, Janice Singer, Hans Petter Langtangen, Dietmar Pfahl, and Greg Wilson. How do scientists develop and use scientific software? In *Proceedings of the 2009 ICSE Workshop on Software Engineering for Computational Science and Engineering*, SECSE '09, pages 1–8, Washington, DC, USA, 2009. IEEE Computer Society.

[9] Dustin Heaton and Jeffrey C. Carver. Claims about the use of software engineering practices in science. *Inf. Softw. Technol.*, 67(C):207–219, November 2015.

[10] IEEE. Recommended practice for software requirements specifications. *IEEE Std 830-1998*, pages 1–40, Oct 1998.

[11] 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.

[12] Kari L. Jordan, Ben Marwick, Jonah Duckles, Naupaka Zimmerman, and Erin Becker. Analysis of software carpentry's post-workshop surveys, July 2017.

[13] 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.

[14] Diane Kelly. Scientific software development viewed as knowledge acquisition: Towards understanding the development of risk-averse scientific software. *Journal of Systems and Software*, 109:50–61, 2015.

[15] Diane Kelly and Terry Shepard. Eight maxims for software inspectors. *Software Testing, Verification and Reliability*, 14(4):243–256, 2004.

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

[17] D. E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, 1984.

[18] Adam Lazzarato, W. Spencer Smith, and Jacques Carette. State of the practice for remote sensing software. Technical Report CAS-15-03-SS, McMaster University, January 2015. 47 pp.

[19] J. Libarkin. Software carpentry workshop evaluation. https://software-carpentry.org/files/bib/libarkin-assessment-report-2012-06.pdf, June 2012.

[20] Emil Obreshkov and the ATLAS Collaboration. Software release build process and components in atlas offline. *Journal of Physics: Conference Series*, 331(4):042017, 2011.

[21] D. L. Parnas. Software engineering or methods for the multi-person construction of multi-version programs. In Clemens E. Hackl, editor, *Programming Methodology*, pages 225–235, Berlin, Heidelberg, 1975. Springer Berlin Heidelberg.

[22] David L. Parnas and P.C. Clements. A rational design process: How and why to fake it. *IEEE Transactions on Software Engineering*, 12(2):251–257, February 1986.

[23] David Lorge Parnas. Precise documentation: The key to better software. In *The Future of Software Engineering*, pages 125–148, 2010.

[24] David J. Rinehart, John C. Knight, and Jonathan Rowanhill. Current practices in constructing and evaluating assurance cases with applications to aviation. Technical Report CR-2014-218678, National Aeronautics and Space Administration (NASA), Langley Research Centre, Hampton, Virginia, January 2015.

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

[26] Padideh Sarafraz. Thermal optimization of flat plate PCM capsules in natural convection solar water heating systems. Master's thesis, McMaster University, Hamilton, ON, Canada, 2014.

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

[28] Judith Segal. Models of scientific software development. In *Proceedings of the First International Workshop on Software Engineering for Computational Science and Engineering (SECSE 2008)*, pages 1–6, Leipzig, Germany, 2008. In conjunction with the 30th International Conference on Software Engineering (ICSE).

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

[30] Alexandra Simperler and Greg Wilson. Software carpentry get more done in less time. *CoRR*, abs/1506.02575, 2015.

[31] W. Spencer Smith. A rational document driven design process for scientific computing software. In Jeffrey C. Carver, Neil Chue Hong, and George Thiruvathukal, editors, *Software Engineering for Science*, Chapman & Hall/CRC Computational Science, chapter Examples of the Application of Traditional Software Engineering Practices to Science, pages 33–63. Taylor & Francis, Boca Raton, FL, 2016.

[32] 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 2016. 8 pp.

[33] 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.

[34] W. Spencer Smith and Lei Lai. A new requirements template for scientific computing. In J. Ralyté, P. Ȧgerfalk, and N. Kraiem, editors, *Proceedings of the First International Workshop on Situational Requirements Engineering Processes – Methods, Techniques and Tools to Support Situation-Specific Requirements Engineering Processes, SREP'05*, pages 107–121, Paris, France, 2005. In conjunction with 13th IEEE International Requirements Engineering Conference.

[35] W. Spencer Smith, Lei Lai, and Ridha Khedri. Requirements analysis for engineering computation: A systematic approach for improving software reliability. *Reliable Computing, Special Issue on Reliable Engineering Computation*, 13(1):83–107, February 2007.

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

[37] 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 2018. 2 pp.

[38] 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.

[39] W. Spencer Smith, Zheng Zeng, and Jacques Carette. Seismology software: State of the practice. *Journal of Seismology*, 2018. 33 pp.

[40] John Spriggs. *GSN - The Goal Structuring Notation*. Springer-Verlag, London, 2012.

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

[42] Klaas-Jan Stol, Paul Ralph, and Brian Fitzgerald. Grounded theory in software engineering research: A critical review and guidelines. In *Proceedings of 38th International Conference on Software Engineering*, 05 2016.

[43] 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.

[44] 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.

[45] Alan Wassyng, Neeraj Kumar Singh, Mischa Geven, Nicholas Proscia, Hao Wang, Mark Lawford, and Tom Maibaum. Can product-specific assurance case templates be used as medical device standards? *IEEE Design & Test*, 32(5):45–55, 2015.

[46] Greg Wilson. Software carpentry: Getting scientists to write better code by making them more productive. *Computing in Science Engineering*, 8(6):66–69, Nov 2006.

[47] Greg Wilson. Bits of evidence: What we actually know about software development, and why we believe it's true. https://www.slideshare.net/gvwilson/bits-of-evidence-2338367, October 2009.

[48] Greg Wilson. Software carpentry: lessons learned [version 2; referees: 3 approved]. *F1000Research*, 3(62):1–12, 2016.

[49] Gregory V. Wilson. Where's the real bottleneck in scientific computing? Scientists would do well to pick some tools widely used in the software industry. *American Scientist*, 94(1), 2006.