Position Paper: A Literate Framework for Scientific Software Development

Dan Szymczak
McMaster University
1280 Main Street W
Hamilton, Ontario
szymczdm@mcmaster.ca

Spencer Smith
McMaster University
1280 Main Street W
Hamilton, Ontario
smiths at mcmaster.ca

Jacques Carette
McMaster University
1280 Main Street W
Hamilton, Ontario
carette at mcmaster.ca

ABSTRACT

Awesome abstract that makes readers fall in love with the research and throw grant money at us in droves goes here. I'm going to just fill in some more words so we can get an idea of how much space our abstract will most likely take up. First we start by introducing ideas. Here are some ideas: compostable javelins with seeds in the points, which are somewhat hollow (or absorb water well) to allow for easy irrigation. These ideas come with problems. Now we'll mention that we're hoping to solve these by utilizing our LSS framework. Finally we'll discuss how the examples we've used have already shown clear advantages from using this process.

CCS Concepts

•**Computer systems organization** → **Embedded systems;** *Redundancy;* Robotics; •**Networks** → Network reliability;

Keywords

ACM proceedings; LATEX; text tagging

1. INTRODUCTION

A rational document driven design process for Scientific Computing (SC) software potentially improves the software qualities of verifiability, reliability, usability, maintainability, reusability and reproducibility [13]. However, many researchers have reported that a document driven process is not used by, nor suitable for, SC software; they argue that scientific developers naturally use an agile philosophy [1, 2, 4, 11] or an amethododical [6], or a knowledge acquisition driven [7], process. The arguments are that scientists do not view rigid, process-heavy approaches, favourably [2] and that in SC requirements are impossible to determine upfront [2, 12]. Rather than abandon the benefits of a rational document driven process, these arguments can be addressed

through the development of appropriate tools that support change and allow the scientist to focus on their science.

Our goal is to eat our cake and have it too. We want to improve the qualities (verifiability, reliability, usability etc.) of SC software and at the same time improve, or at least not diminish, performance. Moreover, we want to improve developer productivity to save time and money on SC software development, certification and re-certification. To accomplish this we need to remove (by hand generated) duplication between software artifacts for SC software [14] and provide complete traceability between all software artifacts. Practical accomplishment of these objectives means we need to provide facilities for automatic software artifact generation. We aim to accomplish this by having one "source" that captures all of the relevant knowledge for a given SC problem. From this source we can generate all required documents and views. That is, we aim to provide methods, tools and techniques to support developing scientific software using a literate process, analogous to Knuth's [8] literate programming.

In the following sections we will look at SC software, specifically focusing on software quality and literate programming. Then we introduce our framework for improving SC software development using a structured approach. We will show a short example of the framework in action and discuss its advantages. We will then discuss how we want the framework to evolve and what future work we intend to do. Finally, Section 5 will provide some concluding remarks.

2. BACKGROUND

There have been many challenges towards assuring the quality of software throughout the history of scientific computing. Many attempts have been made (some successful, others not) at overcoming these challenges and improving software quality. In this section we will discuss some of the challenges, as well as introduce the ideas behind our proposed approach.

2.1 Challenges for Scientific Computing Software Quality

SC software has certain characteristics which create challenges for its development. We will not discuss them all in depth, however, those of interest to us are the technique selection, input output (or understandability), and modification (or maintainability) challenges as described by Yu [15].

The *technique selection challenge* is a challenge that comes up often when dealing with continuous mathematical equa-

tions. Since these cannot be solved directly (as computers are discrete), some technique must be chosen which can produce a solution which is "good enough" for the user. Typically this choice is left to domain experts as each technique will (not) satisfy certain non-functional requirements.

The *understandability* challenge impacts the usability of SC software and typically comes up where considerable amounts of input data are used to produce large volumes of output. The main issue in this case is the complicated nature of the input data and the output, leading developers to recreate (slightly modified) existing library routines to deal with the nature of the input and output. Programmers do not reuse libraries as often as they could because they do not believe the interface needs to be as complicated as it appears [3].

Finally, the *maintainability challenge* tends to come up as requirements change. As SC software is used at the forefront of scientific knowledge, there can be a high frequency of requirements changes. Changing requirements poses a problem for scientists: commercial software can be difficult/expensive to modify and noncommercial programs are not often flexible enough to change.

2.2 Literate Programming

Literate programming (LP) is a programming methodology introduced by Knuth [8]. The main idea is to write programs in a way that explains (to humans) what we want the computer to do, as opposed to simply giving the computer instructions. There is a focus on ordering the material to promote better human understanding.

In a literate program, the documentation and code are together in one source. While developing a literate program, the algorithms used are broken down into small, understandable parts (known as "sections" [8] or "chunks" [5]) which are explained, documented, and implemented in an order which promotes understanding. To get working source code, the *tangle* process is run, which extracts the code from the literate document and reorders it into an appropriate structure for the computer to understand. Similarly, the *weave* process must be run to extract and typeset the documentation.

By adhering to the LP methodology, a literate program ends up with documentation that is expertly written and of publishable quality. The code also ends up being well documented and high quality. There are several examples of SC programs being written in LP style, and two that stand out are VNODE-LP [9] and "Physically Based Rendering: From Theory to Implementation" [10] (a literate program which is also a textbook).

3. INTRODUCING LSS

Our framework, LSS, is being developed with the goals of providing complete traceability throughout the development process for SC software and reducing the amount of knowledge duplication across software artifacts. Both of these goals can be accomplished by following a (somewhat modified) literate approach.

3.1 The current state of LSS

LSS has been developed to this point using a practical, example-driven approach. The first example used in guiding the development of LSS involves the incredibly simplified Software Requirements Specification (SRS) for a fuel pin in a nuclear reactor (See Appendix A). It is a simple example, but the knowledge gained from it has been invaluable.
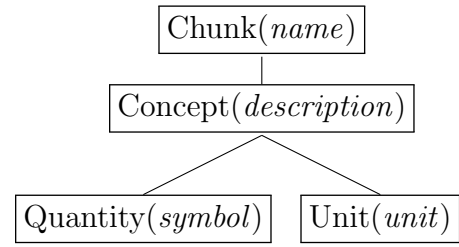

Figure 1: The chunk hierarchy design

```
srsBody = Document ((S "SRS for ") :+:
    (N $ h_g ^. symbol) :+:
    (S " and ") :+: (N $ h_c ^. symbol))
    (S "Spencer Smith") [s1,s2]

s1 = Section (S "Table of Units")
    [s1_intro , s1_table]

s1_table = Table [S "Symbol", S "Description"] $ mkTable
    [(\x -> Sy (x ^. unit)),
     (\x -> S (x ^. descr))
    ] si_units

s1_intro = Paragraph (S "Throughout this ...
```

Figure 2: A portion of the SRS recipe

As of the time of this writing, we are able to generate much of the SRS for the fuel pin as well as the source code for the calculations contained within (See Appendix B).

3.2 How it works

The current framework is composed of several components including *chunks*, *recipes*, and a *generator*. The generator produces views of the source material, where these views are the software artifacts we require. Recipes are descriptions of these views and chunks contain the source material.

Each chunk needs to represent some concept, quantity, etc. Our current design introduces a chunk hierarchy (as seen in Figure 1), where the most basic chunk has only one field: a *name*. A "Concept" adds a *description*, and so on.

Each chunk is built from one or more of the existing chunks. This can be seen in the `h_c` definition in Figure 4. Thus, all of the information related to any one concept in a software project will be stored in a single chunk.

Recipes are specified using a micro- and macro-layout language embedded in Haskell. Each describes how the generated artifacts should appear: The micro-language handles the small-scale layout details (subscript and superscripts, concatenation of symbols, etc.), whereas the macro-language handles the larger layout details (sections, tables, etc.).

The source for our example has been broken down into the recipe, common knowledge, and specific knowledge. A portion of the recipe can be seen in Figure 2.

In this example, the fundamental SI units are common knowledge. Each is contained within its own chunk in the SI unit library. As Figure 3 shows, each chunk has a name, description, and symbolic representation.

The h_c chunk (Figure 4) is a piece of specific knowledge. It contains the name, description, symbol, units, and equation (written in an internal expression language) for h_c.

3.3 Advantages

We can already see the beginning of some advantages over traditional SC development. Certain advantages are rele-

```
metre , kilogram , second , kelvin , mole :: FundUnit
metre    = fund "Metre"     "length (metre)"        "m"
kilogram = fund "Kilogram"  "mass (kilogram)"       "kg"
second   = fund "Second"    "time (second)"         "s"
kelvin   = fund "Kelvin"    "temperature (kelvin)"  "K"
```

Figure 3: Segment of the SI unit library

```
h_c_eq :: Expr
h_c_eq = ((Int 2):*(C k_c):*(C h_b)) :/
    ((Int 2):*(C k_c) :+ ((C tau_c):*(C h_b)))

h_c :: EqChunk
h_c = EC (UC (VC "h_c"
    "convective heat transfer coefficient
        between clad and coolant"
    (sub h c) ) heat_transfer) h_c_eq
```

Figure 4: The h_c chunk

vant to specific challenges (Section 2.1) and will be explored further in the following sections.

3.3.1 Software Certification

High-quality documentation is required to certify software. The creation of said documentation should not impede scientists' work. One of the major problems in creating documentation for SC software stems from the maintainability challenge (Section 2.1). As requirements change, documentation must be updated. Changes to requirements result in greater changes to documentation as a project progresses. This creates issues with traceability and maintainability.

Depending on the regulatory body and set of standards for certification, many types of documents may be required. LSS aims to generate all of these documents alongside the code, while accounting for any changes. As the changes will affect chunks and those chunks will be used to generate the documentation and code, there is a guarantee that changes will propagate throughout all of the artifacts.

The following are examples of software artifacts that we wish to generate (we assume software engineers will be familiar with most, if not all, of these):

1. Software Requirements Specification (SRS)

2. Verification and Validation (V&V) plan

3. Design Specification

4. Code

5. V&V Report

The artifacts listed above share non-trivial amounts of information, which is where traceability comes into play. To make any changes to the documentation, there must be some way to determine how a change will affect **all** of the artifacts. This is especially important for (re-) certification, as the documents and source must remain consistent.

In the context of re-certification, if a piece of software were developed using LSS and some changes needed to be made, updating the artifacts and submitting them for re-certification would be completely trivial. All documentation would be generated from the (newly modified) chunks, according to their existing recipes. Or in the case of new information being added, a new chunk would be created and

the recipe slightly modified. During the implementation of our example, we have already seen how LSS allows us to make changes at trivial cost.

On another note, if a document standard were to be changed during the development cycle, it would not necessitate rewriting the entire document. All of the information in the chunks would remain intact, only the recipe would need to be changed to accomodate the new view.

3.3.2 Knowledge Capture

In SC software there are many commonly shared theorems and formulae between different applications. For an example consider the conservation of thermal energy equation:

$$-\nabla \cdot \mathbf{q} + g = \rho C \frac{\partial T}{\partial t}$$

This equation is widely used in a variety of thermal analysis applications, with each solving a different problem, or modeling a different system. Each application tends to reimplement this equation, as opposed to reusing it [?].

Our approach aims to build libraries of chunks that can be reused in many different applications. Each library should contain common chunks relevant to a specific application domain (ex. thermal analysis) and each project should aim to reuse as much as possible during development.

A more abstract, yet more common example of a reused knowledge source is the concept of Systéme International (SI) units. They are used in applications throughout all of SC, so why should they be redefined for each project? Once the knowledge has been captured, they can simply be reused wherever necessary.

3.3.3 "Everything should be made as simple as possible, but not simpler." (Einstein quote)

Currently there exist many powerful, general commercial programs for solving problems using the finite element method. However, they are not often used to develop new "widgets" because of the cost and complexity of doing so [?]. As it stands, engineers often have to resort to building prototypes and testing them instead of performing simulations due to a lack of tools that can assist with their exact set of problems.

Our approach could change that by generating source code suited to the needs of the engineers. For example, if an engineer were designing parts for strength, they could have a general stress analysis program. This program could be 3D or specialized for plane stress or strain, depending on which assumption would be most appropriate. The program could even be customized to the parameterized shape of the part that the engineer is interested in. The new program would also only expose the degrees of freedom necessary for the engineer to change (ex. material properties or specific dimensions of the part), making the simulation process simpler and safer.

LSS will tackle the understandability challenge (Section 2.1) by allowing developers to build components which will provide exactly what is needed, no more and no less.

3.3.4 Verification

When it comes to verification, requirements documents typically include so-called "sanity" checks that can be reused throughout subsequent phases of development. For instance, a requirement could assume conservation of mass or constrain lengths to be always positive. The former would be

Table 1: Future knowledge to capture

Var	Constraints	Typical Value	Uncertainty
L	$L > 0$	1.5 m	10%
D	$D > 0$	0.412 m	10%
V_P	$V_P > 0$ (*)	0.05 m^3	10%
A_P	$A_P > 0$ (*)	1.2 m^2	10%
ρ_P	$\rho_P > 0$	1007 kg/m^3	10%

used to test the output and the latter to guard against some invalid inputs.

With LSS, these sanity checks can be ensured by the knowledge capture mechanism. Each chunk can maintain its own sanity checks and incorporate them into the final system to make sure that all inputs and outputs (including intermediaries) are valid.

3.3.5 The result?

The self-contained SI unit library exemplifies the ease of utilizing common knowledge (as mentioned in Section 3.3.2) across multiple projects with minimal effort, allowing developers and scientists to spend their valuable time on more important things.

Prototyping ideally will also become trivial due to code generation. Changes in specifications can be seen in the code in (essentially) real-time at trivial cost. Thus we provide the means to create exactly what is needed (Section 3.3.3) for any given situation.

Finally, any mistakes that occur in the software artifacts will occur **everywhere**. Errors propagate through all artifacts, and the artifacts will never be out of sync with each other (or the source), making them much easier to find and allowing for easier document verification (Section 3.3.4).

Complete traceability is achievable through the use of chunks, providing that all requisite knowledge can be appropriately encapsulated.

4. FUTURE WORK

Currently the framework is still very small, producing only one document type (the SRS) and only one type of code (C code for calculations). We plan to expand LSS in several ways including, but not limited to:

1. Generate more artifact types (i.e. have more default recipes).

2. Generate different document views (ex. SRS with equation derivations).

3. Include more types of information in chunks (for example: physical constraints, typical values, and uncertainty as seen in Table4).

4. Auto-generate test cases by using constraints and typical values. The constraints should determine error cases, and typical value ranges give warnings.

5. Continue to expand the tool by implementing larger example systems.

For the auto-generation of test cases, physical constraints will be seen as hard limits on the possible values. For example: length must always be positive. A negative value would cause the code to raise an error and execution would halt.

Typical values, on the other hand, are "reasonable" values. For example: the length of a beam should be on the order of several metres, not centimetres or kilometres. Theoretically, however, it could be either and thus the code will raise a warning instead of an error.

5. CONCLUDING REMARKS

The current standard of using agile approaches to SC software development leave many things to be desired. Documents tend to fall out of sync with the source with each iteration, and the amount of hand-duplicated information leads to errors affecting the quality of the software.

We have begun the creation of a framework to help ensure complete traceability between software artifacts in the development process, while attempting to inconvenience the developers as little as possible. Using our framework will hopefully lead to higher quality software at very little cost.

6. REFERENCES

[1] K. S. Ackroyd, S. H. Kinder, G. R. Mant, M. C. Miller, C. A. Ramsdale, and P. C. Stephenson. Scientific software development at a research facility. *IEEE Software*, 25(4):44–51, July/August 2008.

[2] J. C. Carver, R. P. Kendall, S. E. Squires, and D. 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.

[3] P. F. Dubois. Designing scientific components. *Computing in Science and Engineering*, 4(5):84–90, September 2002.

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

[5] A. Johnson and B. Johnson. Literate programming using `noweb`. *Linux Journal*, 42:64–69, October 1997.

[6] D. 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.

[7] D. 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.

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

[9] N. S. Nedialkov. VNODE-LP — a validated solver for initial value problems in ordinary differential equations. Technical Report CAS-06-06-NN, Department of Computing and Software, McMaster University, 1280 Main Street West, Hamilton, Ontario, L8S 4K1, 2006.

[10] M. Pharr and G. Humphreys. *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

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

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

[13] S. Smith and N. Koothoor. A document driven method for certifying scientific computing software used in nuclear safety analysis. *Nuclear Engineering and Technology*, Accepted, 2016. 42 pp.

[14] G. Wilson, D. Aruliah, C. T. Brown, N. P. C. Hong, M. Davis, R. T. Guy, S. H. Haddock, K. D. Huff, I. M. Mitchell, M. D. Plumblet, B. Waugh, E. P. White, and P. Wilson. Best practices for scientific computing. *CoRR*, abs/1210.0530, 2013.

[15] W. Yu. *FASCS: A Family Approach for Developing Scientific Computing Software*. PhD thesis, McMaster University, Hamilton, ON, Canada, 2011.

APPENDIX

A. SRS FOR H_G AND H_C

B. ARTIFACTS GENERATED BY LSS

B.1 SRS

B.2 Source code

```
double calc_h_g(double k_c, double h_p, double tau_c){
    return (2 * k_c * h_p / (2 * k_c + tau_c * h_p));
}
```