Position Paper: A Literate Framework for Scientific Software Development *

[Extended Abstract] †

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.

CCS Concepts

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

Keywords

ACM proceedings; LaTeX; text tagging

1. INTRODUCTION

Scientific computing (SC) was the first application of computers. It is still used today for a wide variety of tasks: constructing mathematical models, performing quantitative analyses, creating simulations, solving scientific problems, etc. SC software has been developed for increasingly safety and security critical systems (nuclear reactor simulation, satellite guidance) as well as predictive systems. It has applications including (but not limited to) predicting weather patterns and natural disasters, and simulating economic fluctuations. As such, it is an incredibly important part of an increasing number of industries today.

In the medical, nuclear power, aerospace, automotive, and manufacturing fields there are many safety critical systems in play. With each system, there is the possibility of a catastrophic failure endangering lives. It is incredibly important then to have some means of certifying and assuring the quality of each software system. As Smith et al. [**?**] stated "Certification of ScientiiňĄc Computing (SC) software is official recognition by an authority or regulatory body that the software is iňĄt for its intended use." These regulatory bodies determine certain certification standards that must be met for a system to become recognized as certified. One example of a certification standard is the Canadian Standards Association (CSA) requirements for quality assurance of scientific software for nuclear power plants.

The main goal of software certification is to "... systematically determine, based on the principles of science, engineering, and measurement theory, whether a software product satisfies accepted, well-defined and measurable criteria" [**?**]. As such, certification would not only involve analyzing the code systematically and rigorously, but also analyzing the documentation. Essentially, this means the software must be both valid and verifiable, reliable, usable, maintainable, reusable, understandable, and reproducible.

Developing certifiable software can end up being a much more involved process than developing uncertified software: it takes more money, time, and effort on the part of developers to produce. These increased costs lead to reluctance from practitioners to develop certifiable software [**?**]. However, in our opinion, cost is not the only contributing factor for the developers. As it stands in the field, scientists seem to prefer a more agile development process [**?**]. However, this is not necessarily the best process as typically this would lead to problems maintaining the documentation, meaning that not all aspects of the software would be traceable through the design process and making certification more difficult (if not impossible).

Given proper methods and tools, scientists would be able to follow a more structured approach (while capitalizing on frequent feedback and course correction) to meet documentation requirements for certification as well as improve their overall productivity. This is where our work comes in: our goal is to eat our cake and have it too. We want to improve the qualities (verifiability, reliability, understandability etc.) of SC software and at the same time improve performance. Moreover, we want to improve developer productivity; save time and money on SC software development, certification and re-certification. To accomplish this we need to do the following:

1. Remove duplication between software artifacts for SC software [**?**]

*(Produces the permission block, and copyright information). For use with SIG-ALTERNATE.CLS. Supported by ACM.

†A full version of this paper is available as *Author's Guide to Preparing ACM SIG Proceedings Using LaTeX2ε and BibTeX* at `www.acm.org/eaddress.htm`

WOODSTOCK '97 El Paso, Texas USA

DOI: 10.475/123_4

2. Provide complete traceability between all software artifacts

To achieve the above goals, we propose the following:

1. Provide methods, tools and techniques to support developing scientific software using a literate process

2. Automate software artifact generation

Section 2 will give a more in-depth look at SC software, specifically focusing on SC software quality and literate programming. Section 3 focuses on our framework for improving SC software development using a structured approach. It will introduce the framework, discuss the advantages to our approach, and show a short example of the framework in action. Section 4 will 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

Throughout the history of computing, specifically scientific computing, there have been many challenges towards assuring the quality of software. Many attempts have been made (some successful, others not) at improving software quality. In this section we will discuss those 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 [**?**].

The *technique selection challenge* is a challenge that comes up often when dealing with continuous mathematical equations. Since these cannot be solved directly (due to computers being discrete), some technique must be chosen which can produce a solution which is "good enough" for the user. Choosing the technique to use is typically 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 in the production of 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 existing library routines (slightly modified) 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 that the interface needs to be as complicated as it appears [**?**].

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. As changing requirements can mean completely modified systems, it 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 [**?**]. The main idea behind it is writing programs in a way that allows us to explain (to humans) what we want the computer to do, as opposed to simply telling the computer what to do. There is a focus on ordering the material to promote better human understanding.

In a literate program, the documentation and code are kept together in one source. The program is an interconnected web of code pieces, presented in any sequence. As part of the development process, the algorithms used in literate programming are broken down into small, easy to understand parts (known as "sections" [**?**] or "chunks" [**?**]) which are explained, documented, and implemented in a more intuitive order for better understanding.

To extract the source code or documentation from the literate program, certain processes must be run. To get working source code, the *tangle* process would be run, essentially extracting the code from the literate document and reordering it into an appropriate structure for the computer to understand. To get a human-readable document, the *weaving* 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 incredibly 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 [**?**] and "Physically Based Rendering: From Theory to Implementation" [**?**] (a literate program which is also a full 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.

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 essentially descriptions of these views and chunks contain the source material.

Complete traceability is achievable through the use of chunks, providing that all requisite knowledge can be appropriately encapsulated. Each chunk needs to represent some concept, quantity, unit, 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" then adds a *description*, and so on down the tree.

Each more complex chunk is built from one or more of the existing chunks. This will be shown in more detail through the example in Section 3.2. Thus, all of the information related to any one concept in a software project will be stored in a single chunk, allowing for complete traceability to the source of the information.

Recipes are specified using a micro- and macro-layout language currently embedded in Haskell. Each are used to describe how the generated artifacts should appear. The

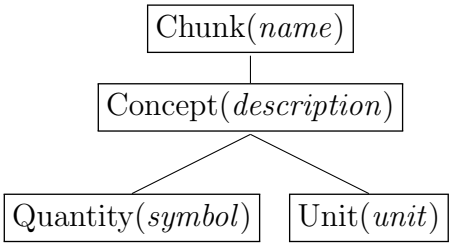

Figure 1: The chunk hierarchy design

micro-layout language handles the small-scale layout details such as those of specially formatted characters (i.e. subscript and superscripts), concatenation of symbols, etc. The macro-layout language, on the other hand, handles the large-scale layout details such as sections, paragraphs, equations, tables, etc.

Recipes also specify which knowledge should be used from which chunks, thus removing knowledge duplication as all of the knowledge is now being referenced from within the specified chunks.

3.1 Advantages

- Move into examples/discussion on advantages. Should write something here.

3.1.1 Software Certification

To certify software there is a need for high-quality documentation. This documentation needs to be created without impeding the work of scientists. One of the major problems with creating documentation for SC software stems from the maintainability challenge (Section 2.1). As the requirements change, the documentation must be updated. The further scientists are into a project, the greater the effect a change in requirements will have on the documentation, leading to maintainability and traceability issues.

The cost of making changes to the documentation should be reasonable. The best way to keep costs low is to ensure the traceability of the documentation. Traceability allows the developers to track which areas of a project will be affected by a change, thus allowing them to ensure that those areas are updated accordingly.

Depending on the regulatory body and set of standards for certification, many types of documents can be required. LSS aims to generate all of these documents alongside the code, while accounting for any changes made. 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 software artifacts that we wish to generate (we assume software engineers will be familiar with most, if not all, of these):

1. Problem Statement
2. Development Plan
3. Software Requirements Specification (SRS)
4. Verification and Validation (V&V) plan
5. Design Specification
6. Code
7. V&V Report
8. User Manual

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 documents.

This is especially important for (re-) certification, as the documents must be consistent.

In the context of re-certification, if a piece of software was 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.

On another note, if a document standard were to be changed during the development cycle, it would not necessitate re-writing 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 standard.

3.1.2 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:

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

This equation is widely used in a variety of thermal analysis applications, where each is solving a different problem, or modeling a different system.

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.

3.1.3 "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, the 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

```
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 a recipe

```
metre, kilogram, second, kelvin, mole, ampere, candela :: 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"
mole     = fund "Mole"     "amount of substance (mole)"
"mol"
ampere   = fund "Ampere"   "electric current (ampere)"
"A"
candela  = fund "Candela"  "luminous intensity (candela)" "cd"
```

Figure 3: Fundamental SI unit library

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.

3.1.4 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, the requirement would state conservation of mass or that lengths are always positive. The former would be 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.

3.1.5 Testing

- ???

3.2 The current state of LSS

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

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 (in C) for the calculations contained within (See Appendix B).

3.2.1 How it works

The source for this example has been broken down into the recipe, common knowledge, and specific knowledge. An example of each source can be seen in Figures 2, 3, and 4 respectively.

The recipe is a description what information is necessary and how it should be arranged (notice in the "Section" declaration from Figure 2, `s1_intro` comes before `s1_table`, even though they are defined in the reverse order).

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

```
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

Figure 4 shows one piece of specific knowledge for this example: the h_c chunk. This chunk contains the equation for calculating h_c (written in an internal expression language which can then be converted to a math representation or source code) as well as the name, description, symbolic representation, and units for h_c.

3.2.2 The result?

LSS allows us to make changes with minimal effort at effectively no cost! This means changing requirements no longer impede scientists as described in Section 3.1.1.

The self-contained SI unit library exemplifies the ease of utilizing common knowledge (as mentioned in Section 3.1.2) 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 3.1.3) for any given situation.

Finally, any mistakes that occur in the software artifacts will occur **everywhere**. This makes it much easier to find errors since they propagate through all artifacts, and the artifacts will never be out of sync of each other (or the source), allowing for ease of verification (Section 3.1.4).

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).

Table 1: Future example variables

Var	Physical 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%

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

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));
}
```