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, 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 *input output* 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.

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.

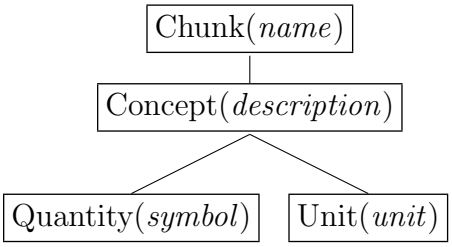

Figure 1: The chunk hierarchy design

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 re-

lated 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 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 **??**). 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. Requirements Specification
4. Verification and Validation plan
5. Design Specification
6. Code
7. Verification and Validation 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)

- although powerful/general commercial finite element programs are available, they are often not used to develop new âĂIJwidgetsâĂİ - reasons are cost, and complexity - rather than use simulation, engineers often resort to building prototypes and testing - engineers would greatly benefit from tools to assist their design efforts that are customized to their exact set of problems - with a literate family approach family members can be generated to fit their needs - if an engineer designs parts for strength, they could have a general stress analysis program - the program could be 3D if needed, or specialized for plane stress or plane strain, if that was the appropriate assumption - the program could even be customized to the parameterized shape of the part they are interested in, with only the degrees of freedom, like material properties, or the specific dimensions, they can change being exposed. [This section is also where we can mention the understandability challenge from Section 2.1. LSS lets us build components that provide exactly what is needed, and no more.]

3.1.4 Verification

- requirements include so-called âĂIJsanityâĂİ checks that can be reused when they come up in subsequent phases - for instance, requirement would state conservation of mass, or the fact that lengths are always positive - the first used to test output, the second to guard against invalid input

- computational variability testing, from Yu (2011), FEM example - usual to do grid refinement tests - same order of interpolation, but more points - code generation allows for increases in the order of interpolation, for the same grid - Yu discusses in section 6.3 of her thesis

3.1.5 Testing

- ???

3.2 Bringing it together with an example

- Introduce fuel pin example. - Use appendix for SRS and generated SRS?

3.2.1 How it works

- Explain the source (recipe, common knowledge, specific knowledge) - Show example recipe - Example of common knowledge (SI Units to tie into section 3.1.2) - Example of specific knowledge (h_c or h_g chunk)

3.2.2 The result?

- Bring it all back to the arguments made in 3.1.1-3.1.4 and justify them.

- 3.1.1: (Ideally) No cost to make changes, no longer impedes the scientists. - Explain in-depth

- 3.1.2: SI Units shows that common knowledge is easily stored to be used elsewhere - No need to reinvent the wheel! Keeps things consistent across projects.

- 3.1.3: (Ideally) Prototyping becomes trivial due to code gen. - Any changes to spec can be seen in (essentially) real-time.

- 3.1.4: Any mistakes that occur in the software artifacts occur everywhere. - Easier to find errors since they propagate through all the artifacts. - Documents are never out of sync with the source.

4. FUTURE WORK

- Generate more document types (more default recipes) - Include more types of information in chunks (PCM SRS example of constraints, etc.) - Auto-generate test cases! Use constraint/typical value info to create tests. - Use constraints for errors and typical values for warnings. - Expand the tool practically through examples.

5. CONCLUDING REMARKS