Position Paper: A Knowledge-Based Approach to 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@mcmaster.ca

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

ABSTRACT

As a relatively mature field, scientific computing has the opportunity to lead other software by leveraging its solid, existing knowledge base. By following a rational design process, desirable software qualities such as the qualities of traceability, verifiability, and reproducibility, are arguable easier to reach than for other classes of software.

We have begun development of a framework, Drasil, to put this into practice. Our aims are to ensure complete traceability, to facilitate agility in the face of the ever changing nature of scientific computing projects, and ensure that software artifacts can be easily and quickly be extracted from Drasil.

Using an example-based approach to our prototype implementation, we have already seen many benefits. Drasil keeps all software artifacts (requirements, design, code, tests, build scripts, documentation, etc) synchronized with each other. This allows for reuse of common concepts across projects, and aids in the verification of software. It is our hope that Drasil will lead to the development of higher quality software at lower cost over the long term.

Keywords

1. INTRODUCTION

We believe that, because of the solid scientific knowledge base built up over the last 6+ decades of work in Scientific Computing (SC), it is feasible for SC to once again take a leadership position as regards the development of high quality software. More precisely, our goal s to use this knowledge to improve the verifiability, reliability, usability, maintainability, reusability and reproducibility SC Software (SCS).

Some have argued for a rational document-drive design process[12]. However, many researchers have reported that a document driven process is not used by, nor suitable for, SCS; they argue that scientific developers naturally use either an agile philosophy [1, 3, 10], or an amethododical [5] process, or a knowledge acquisition driven [6] process. The arguments are that scientists do not view rigid, process-heavy approaches, favourably [1] and that in SC, requirements are impossible to determine up-front [1, 11]. But rather than abandon the benefits of a rational document-driven process, we argue that the appropriate tools can in fact let the scientists focus even more of their time on science.

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

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

Thus, to successfully achieve our goal of improving various software qualities (verifiability, reliability, usability etc.) of SCS, whilst also improving, or at least not diminish performance, we must also find a way to simultaneously deal with the above drawbacks. In fact, we are even more ambitious: we want to improve developer productivity to save time and money on SCS development, certification and recertification. To accomplish this we need to remove duplication between software artifacts [13] as well as provide traceability between all software artifacts. In practice, this means providing facilities for automatic software artifact generation from high level "knowlege". We can accomplish this by have a single "source" for each relevant piece of information which makes up an SC problem and its solution. From this, we can generate all required documents and views. That is, we aim to provide methods, tools and techniques to support a literate process for developing scientific software that generalizes the idea behind Knuth's [7] literate programming.

In the following section, we focus on SC software quality and literate programming. Then we introduce our framework, Drasil. We show a short example of the framework and discuss its advantages. We then discuss how we want the framework to evolve. The last section provides concluding remarks.

2. BACKGROUND

In this section we discuss challenges for developing SC software and we introduce the ideas behind our approach.

2.1 Challenges for Scientific Computing Software Quality

The *technique selection challenge* [14] arises in SC because the best numerical approach to solve a given problem is not

SE4SC ?, 2016, Seattle, WA, USA

DOI: 10.1145/1235

known a priori. Experimentation is inevitably necessary to determine the appropriate order of interpolation, the degree of implicitness, etc. For a framework for developing SC software to be successful, it should support a separation of concerns between the physical model and the numerical algorithm. Moreover, the framework should provide facilities for parameterizing the expected algorithmic variabilities for easy experimentation with different options.

The *understandability* challenge [14] effects the source code and the executable application. In an effort to make scientific libraries and software as widely applicable as possible, most packages provide a generic interface with a large number of options. The number of options overwhelms users and causes programmers to not reuse libraries, since they do not believe the interface needs to be as complicated as it appears [2]. To improve understandability, an ideal framework will generate applications and libraries that are only as complicated as they need to be for the job at hand.

The *maintainability challenge* [14] comes up as requirements change. The high frequency of change for SC software especially causes problems for certification. If the expense and time required for re-certification is on the same order of magnitude as the original certification, changes will not be made. To be effective in this environment, a framework needs to provide traceability, so the consequences of change can be evaluated.

2.2 Literate Programming

Literate programming (LP) is a programming methodology introduced by Knuth [7]. 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.

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" [7] or "chunks" [4]) 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 is run to extract and typeset the documentation. There are several examples of SC programs being written in LP style, such as VNODE-LP [8] and "Physically Based Rendering: From Theory to Implementation" [9] (a literate program which is also a textbook).

3. INTRODUCING Drasil

Our framework, Drasil, is being developed with two goals: complete traceability throughout the development process and reduced knowledge duplication. Both of these goals can be accomplished by generalizing a literate approach.

3.1 The current state of Drasil

Drasil has been developed to this point using a practical, example-driven approach. The first example used in guiding the development of Drasil involves the simplified Software Requirements Specification (SRS) for a fuel pin in a nuclear reactor (See [12] for more details). For brevity we look specifically at the term h_c (defined in Figure 1).

At the time of this writing, we are able to generate the `.tex` file for much of the SRS for the fuel pin as well as the source code (in C) for the required calculations.

Label	h_c
Units	$ML^0t^{-3}T^{-1}$
SI equivalent	$\frac{\text{kW}}{\text{m}^{2\circ}\text{C}}$
Equation	$h_c = \frac{2k_c h_b}{2k_c + \tau_c h_b}$
Description	h_c is the effective heat transfer coefficient between the clad and the coolant τ_c is the clad thickness h_b is initial coolant film conductance k_c is the clad conductivity

Figure 1: SRS data definition of h_c

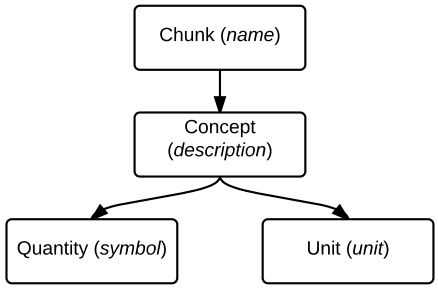

Figure 2: The chunk hierarchy design

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 2), 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` implementation below. 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 3.

In this example, the fundamental SI units are common knowledge. Each is contained within its own chunk in the SI unit library. As Figure 4 shows, each chunk has a name, description, and symbolic representation.

The h_c chunk (Figure 5) is a piece of specific knowledge. It contains the name, description, symbol, units, and equation (written in an internal expression language) for h_c.

The internal expression language used to implement the equation for h_c allows for the straighforward generation of source code. We utilize methods similar to those found in [**?**, **?**] wherein the expression language is converted to an

```
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 3: A portion of the SRS recipe

```
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 4: Segment of the SI unit library

abstract representation of the code and then passed to a pretty-printer to create the final source.

We currently generate C code, however it would be possible to generate any language provided we have an appropriate representation for that language.

3.3 Advantages

We can already see the beginning of some advantages over traditional SC development. How Drasil addresses the specific challenges of Section 2.1 and will be explored below.

3.3.1 Software Certification

High-quality documentation is required to certify software, but its creation should not impede a scientists' work. Major problems in creating SC software stem from the maintainability and technique selection challenges (Section 2.1). As requirements and numerical algorithmic decisions change, documentation and code must be updated. This creates issues with traceability and maintainability.

Depending on the regulatory body and the certification standards, many types of documents may be required, such as requirements specification, verification plans, design specification and code. Drasil aims to generate 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.

In the context of re-certification, if a piece of software were developed using Drasil and some changes needed to be made, updating the artifacts and submitting them for re-certification would be straightforward. 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 Drasil 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 accommodate the new view. Thus, capturing all of the knowledge that a program is based on and improving reproducibility.

```
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 5: The h_c chunk

3.3.2 Knowledge Capture

In SC software there are many commonly shared theorems and formulae across different applications. For example consider the general form of the conservation of thermal energy equation. This equation is widely used in a variety of thermal analysis applications, with each solving a different problem, or modeling a different system.

Our approach aims to build libraries of chunks that can be reused anywhere. 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.

From our example, a common source of reused knowledge is the Systéme International (SI) units (Figure 4). 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. With Drasil they can be reused across projects with minimal effort, allowing developers and scientists to spend their valuable time on more important things.

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 understandability challenge. Engineers often have to resort to building and testing prototypes, instead of performing simulations, due to a lack of tools that can assist with their exact set of problems.

Our approach will change that by ideally making prototyping trivial through generating source code suited to the needs of the engineers. Changes in specifications could be seen in the code in (essentially) real-time at trivial cost. 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/strain, depending on which assumption would be most appropriate at the time. The program could even be customized to the parameterized shape of the part the engineer is interested in. The new program could 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.

Drasil tackles this 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

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%

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

With Drasil, 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 ensure all inputs and outputs (including intermediaries) are valid.

Also, with Drasil, complete traceability is achievable through the use of chunks, provided the requisite knowledge can be appropriately encapsulated.

Finally, any mistakes that occur in the generated software artifacts will occur **everywhere**. Errors propagate through artifacts, and the artifacts will always be in sync with each other (and the source). As a consequence, errors will be much easier to find and only need to be fixed in one place.

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 Drasil in several ways including, but not limited to:

1. Generate more artifact types.
2. Generate different document views.
3. Include more types of information in chunks (see Table4).
4. Auto-generate test cases 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 values (ex. length must always be positive and a negative value would throw an error). Typical values, on the other hand, are "reasonable" values (ex. the length of a beam should be on the order of several metres, but theoretically it could be kilometres, 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] 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.

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

[3] 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.

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

[5] 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.

[6] 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.

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

[8] 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.

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

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

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

[12] 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.

[13] 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.

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