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 fields by leveraging its solid, existing knowledge base. By following a rational design process, desirable software qualities such as traceability, verifiability, and reproducibility, are arguably 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 ever changing scientific computing projects, and ensure that software artifacts can be easily and quickly extracted from Drasil. In particular, we are very interested in certifiable software and in easy re-certification.

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

Literate software, knowledge capture, traceability, software engineering, scientific computing, artifact generation.

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 is to use this knowledge to improve the verifiability, reliability, usability, maintainability, reusability and reproducibility of SC Software (SCS).

Some have argued for a rational document-driven design process [13]. 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 [2, 4, 11], or an amethododical [6] process, 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 up-front [2, 12]. 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 diminishing 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 and thus save time and money on SCS development, certification and re-certification. To accomplish this we need to remove duplication between software artifacts [15] as well as provide traceability between all software artifacts. In practice, this means providing facilities for automatic software artifact generation from high level "knowledge." We can accomplish this by having 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 [8] literate programming. Unlike other document generation tools, like doxygen, the focus is on all software artifacts, not just the code and its comments.

In the following section, we focus on SCS 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 SCS and we introduce the ideas behind our approach.

2.1 Challenges

SE4Science'16 Austin, TX, USA
© 2016 ACM. ISBN 978-1-4503-4167-7.
DOI: 10.1145/1235

Label	h_c
Units	$ML^0t^{-3}T^{-1}$
SI equivalent	$\frac{\mathrm{kW}}{\mathrm{m}^{2o}\mathrm{C}}$
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

As the best numerical approach to solve a given problem is usually not known a priori, we have to face the *technique selection challenge* [16]. Experimentation is inevitably necessary to determine the appropriate order of interpolation, the degree of implicitness, etc. Nevertheless, the problem to solve does not change. This implies that we need a separation of concerns between the physical model and the numerical algorithms. Ease of experimentation means that we also need facilities to parameterize algorithmic variabilities.

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; this tends to overwhelm users and cause programmers to not reuse libraries, since they do not believe the interface needs to be as complicated as it appears [3]. This is the *understandability* challenge [16]. An ideal framework would expose an API, as well as generate applications, which use routines which are only as complicated as they need to be for the job at hand.

As requirements change, we face the *maintainability challenge* [16]. The high frequency of change for SCS causes accute 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 directly evaluated.

2.2 Literate Programming

Literate programming (LP) is a 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.

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 acceptable to one's compiler. 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 [9] and "Physically Based Rendering: From Theory to Implementation" [10] (a literate program which is also a successful textbook!).

3. INTRODUCING Drasil

In Drasil, we accomplish our two primary objectives (complete traceability, and eliminating knowledge duplication) by generalizing the literate approach.

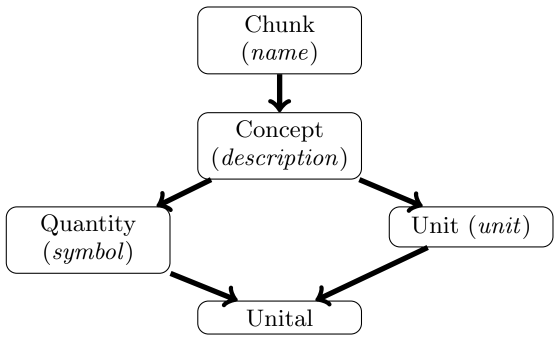

Figure 2: The chunk design

```
srsBody = srs [h_g, h_c] "Spencer Smith" [s1,s2]

s1 = Section (S "Table of Units") [intro, table]

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

intro = Paragraph (S "Throughout this ...")
```

Figure 3: A portion of the SRS recipe

3.1 A Simple Example

We have used a practical, example-driven approach to the development of Drasil. Our first example involves the simplified Software Requirements Specification (SRS) for a fuel pin in a nuclear reactor (see [13] for more details). To get started, let us look specifically at the term h_c (defined in Figure 1). This defines a concept, its units, its defining equation, and the description of the other concepts upon which it depends.

We can currently 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.

3.2 Design

The fundamental task for Drasil is *knowledge capture*, in such a way that this knowledge can be viewed in different ways (code, specification, etc). Each individual piece of knowledge is a named *chunk*; putting chunks together is done via a *recipe*; a *generator* then interprets recipes to produce the desired view.

We have different kinds of chunks. The most basic ones are simply named pieces of information. Most however represent some *concept*, which has a *description*. In the SC context, many concepts are *quantities*, which are represented by a specific *symbol*. Orthogonally, a *unit* is also a concept. Most quantities have units. And so on (as pictured in Fig. 2).

By breaking things down in this way, we can assemble most concepts from pre-existing chunks – see Fig. 3 for an SRS recipe that uses h_c.

Currently, recipes are specified using a collection of DSLs embedded in Haskell. We have a DSL for expressions, expression layout, document layout, C code, and LaTeX code. For example, the expression layout DSL describes how expressions should appear (subscript and superscripts, concatenation of symbols, etc.), whereas the document layout DSL deals with sections, tables, etc.

We have broken down our example into common knowledge, specific knowledge and a recipe for an SRS (see Figure 3). Here the fundamental SI units are common knowledge. Each is contained within its own chunk in the SI unit library – see Figure 4 for a taste.

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

Figure 4: Segment of the SI unit library

The h_c chunk (Figure 5) is specific knowledge: it contains the name, description, symbol, units, and equation for h_c.

The internal expression language Expr allows for the straighforward generation of source code. We utilize methods similar to those found in [1, 14] wherein the expression language is converted to an 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 some advantages over traditional SC development. How Drasil addresses the specific challenges of Section 2.1 will be explored below.

3.3.1 Knowledge Capture

At the appropriate abstraction level, many SC problems have significant commonality, since a large class of physical models are instances of a relatively small number of conservation equations (conservation of energy, mass and momentum). For instance, the theoretical model for conservation of thermal energy for a fuel pin in [13] is written generally, without reference to a specific coordinate system. The exact same theoretical model can be reused in any thermal model. The variation in the final instanced model will come from the refinement of the theory using assumptions appropriate to the problem.

Our approach aims to build libraries of knowledge 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 *Système international d'unités*, also known as SI units (Figure 4). They are commonly used throughout all of SC, so why should they be redefined for each project? Once the knowledge has been captured, it can simply be reused. With Drasil this is possible with minimal effort, allowing developers and scientists to spend their valuable time on more important tasks.

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

h_c :: EqChunk
h_c = fromEqn "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 Software Certification

Current software certification processes require high-quality documentation. Depending on the regulatory body and the certification standards, many types of documents may be required, such as the requirements specification, verification plans, design specification and code. However, creating these should not impede a scientist's work. As requirements and numerical algorithmic decisions change, documentation and code must be updated. This creates issues with traceability and maintainability. Drasil aims to generate these documents alongside the code, while accounting for any changes. When properly used, singular knowledge will be embodied in a single chunk, and the generation process will take care of baking this information into all appropriate places in the documentation and code. Thus any change is guaranteed to propagate throughout all of the artifacts.

Recipes are useful too: 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 accommodate the new view. Traceability has the advantage of also improving reproducibility.

3.3.3 "Everything should be made as simple as possible, but not simpler." — Einstein

Take finite element methods as one example: while there exist many powerful, general commercial programs for this, they are not often used to develop new "widgets" because they are hard to understand. Thus engineers often resort to building and testing prototypes, instead of performing simulations, due to a lack of clearly relevant tools.

By using recipes that generate versions of general algorithms specific to a particular problem, we can generate applications suited to the needs of the engineers. Changes in specifications can be reflected in the code in (essentially) real-time at trivial cost. For example, if an engineer were designing parts for strength, they could start from a general stress analysis knowledge base. This could then be specialized to (say) plane stress/strain, depending on which assumption would be most appropriate at the time. The generated program could even be customized to the parameterized shape of the part the engineer is interested in. Importantly, the simulation process is made simpler because an engineer is not required to interact with the source code; they simply modify those degrees of freedom (ex. material properties or specific dimensions of the part), that are exposed to them by the generator.

3.3.4 Verification

When it comes to verification, requirements documents typically include so-called "sanity" checks (see 2nd column of Table 1) 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

Table 1: Constraints on quantities

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%

latter to guard against invalid inputs.

With Drasil, these sanity checks can also be captured and re-used. Each chunk can maintain its own sanity checks (as constraints) and recipes can incorporate them into generated systems, either for testing or for in-field use, to ensure all inputs and outputs (including intermediaries) are valid.

Through recipes, complete traceability is achievable: as knowledge must be drawn together explicitly, one can automatically list all chunks that were used by a recipe.

Finally, because all knowledge has a unique source, 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 see, and thus easier to find and fix.

4. FUTURE WORK

Our prototype is still in its early stages, producing only one document type (the SRS) and only one type of code (C code for calculations), but has already been a great source of inspiration to us. We plan to expand Drasil in several ways, including at least:

1. Generate more artifact types.
2. Generate different document views.
3. More types of information in chunks (see Table 1).
4. Use these constraints to generate test cases.
5. Implement much larger examples.

For the 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 development of high-quality SCS, especially those which can be (re-)certified, leave much to be desired. The burden of keeping documentation and code synchronized, due to hand-duplicated information, leads to many problems. The end result tends to involve either extremely high development costs (and bored developers), or software of dubious quality which is difficult to maintain.

With Drasil, we hope to achieve complete traceability between all software artifacts, while causing as little inconvenience to the developers as possible. We hope that this will lead to higher quality software, at a lower *long term* cost.

6. REFERENCES

[1] L. Beyak and J. Carette. SAGA: A DSL for story management. In O. Danvy and C. chieh Shan, editors, *DSL*, volume 66 of *EPTCS*, pages 48–67, 2011.

[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] D. Szymczak. Generating Learning Algorithms: Hidden Markov Models as a Case Study. Master's thesis, McMaster University, Hamilton, ON, Canada, 2014.

[15] 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.

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