McMaster University
Department of Computing and Software

Proposal for Thesis Research in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy

Title:	TODO: Title Here
Submitted by:	Dan Szymczak
Student Number:	0759661
	szymczdm@mcmaster.ca
Date of Submission:	April 27, 2016
Expected Date of Completion:	August 2018
Supervisors:	Jacques Carette
	Spencer Smith

Abstract

TODO: ABSTRACT

Contents

1 Introduction

TODO: Introduce the main ideas

2 State of the Art

To pursue this line of research, first one needs to understand the history of research in several closely-related areas. Since I will be focusing on the improvement of SC software, the obvious starting point would be to look at the current state of SC software development and its challenges. Also, as I am intending to expand the ideas of literate programming, it is necessary to delve into sufficient depth on LP and why it has not been widely adapted as yet, as well as any attempts to expand or build upon it. Finally, with the idea of long-term maintainability, full traceability, and reproducibility in mind, I would be remiss if I did not look into the field of reproducible research.

2.1 Current State of SC Software Development

In many instances of SC software development, the developers are the scientists. These developers tend to put the most emphasis on their science instead of good software development practices [13]. Rigid, process-heavy approaches are typically considered unfavourable to these developers [4]. We see the developers choose to use an agile philosophy [1, 4, 6, 28], an amethododical [11], or a knowledge acquisition driven process [12] instead.

There are several clear problems with the current state of SC software development. The first, fairly obvious problem is that knowledge reuse is not being utilized to the fullest extent possible. As an example, a survey [21] showed that of 81 different mesh generator packages, 52 generated triangular meshes. Now that in itself may not show a lack of knowledge reuse, however, looking deeper we see that 37 of those packages used the same Delaunay triangulation algorithm for generating those meshes. There is no reason that the exact same algorithm should be implemented 37 separate times when it could simply be reused.

Another problem in SC software development is the lack of understanding of software testing. More than half the scientists developing SC software lack a good understanding of software testing [18]. It is in such a bad state that quality assurance has "a bad name among creative scientists and engineers" [26, p. 352], not to mention the very limited use of automated testing [22].

It should be obvious that some of these issues could be solved through the use of certain tools. However, it should be noted that tool use by SC software developers is also very limited, especially the use of version control software [36].

Not everything about SC software development today is a negative. For example advanced techniques like code generation have been quite successful in SC. Some generation examples that come to mind are FEniCS [17], FFT [14], Gaussian Elimination [3], and Spiral. The focus of generation techniques, thus

far, have been solely on one software artifact: the source code. Focusing solely on code is a disadvantage to SC software developers as the value of documentation, as well as a structured (or rational) process, have been repeatedly illustrated [31, 32, 33, 34].

2.2 Literate Programming

The LP methodology introduced by Knuth changes the focus from writing programs that simply instruct the computer on how to perform a task to explaining (*to humans*) what we want the computer to do [15].

Developing literate programs involves breaking algorithms down into *chunks* [10] or *sections* [15] which are small and easily understandable. The chunks are ordered to promote understanding, a "psychological order" [24] if you will. They do not have to be written in the same order that a computer would read them. It should also be noted that in a literate program, the code and documentation are kept together in one source. To extract working source code, a process known as *tangle* must be run on the source. A similar process known as *weave* is used to extract and typeset the documentation from the source.

There are many advantages to LP beyond understandability. As a program is developed and updated, the documentation surrounding the source code tends to be updated simultaneously. It has been experimentally found that using LP ends up with more consistent documentation and code [29]. Having consistent documentation has its own advantages while developing or maintaining software [9, 16]. Similarly, there are many downsides to inconsistent documentation [16, 35]. Keeping both of those in mind we can see that more effective, maintainable code can be produced when (properly) using LP [24].

Even with all of the benefits of LP, it has not been very popular [29]. Though it has not been popular, there are still several successful examples of LP's use in SC; two that come to mind are VNODE-LP [20] and "Physically Based Rendering: From Theory to Implementation" [23]. The latter being a literate program as well as a textbook. Shum and Cook discuss the topic of LP's lack of popularity and present the idea that it comes from a couple of main issues: dependency on a particular output language or text processor, the lack of flexibility on what should be presented/suppressed in the output.

Many attempts to address the issues with LP's popularity have focused on changing or removing the output language or text processor dependency. Many new tools were developed such as: CWeb (for the C language), javadoc (for Java), DOC++ (for C++), Doxygen (for multiple languages), noweb (programming language independent), and more. The development of new tools led the introduction of many new features including, but not limited to, a "What You See Is What You Get" (WYSIWYG) editor [7], phantom abstracting [29], and even movement away from the "one source" idea [30].

While these tools did not bring LP into the mainstream [25], they did help drive the understanding behind what exactly LP tools must do. Although LP is not yet mainstream, we can see it becoming more standardized in certain domains (for example: Agda, Haskell, and R support LP).

2.3 Literate Software

A combination of LP and Box Structure [19] was proposed as a new methodology called "Literate Software Development" (LSD) [2]. Box structure can be summarized as the idea of different views (ex. system specifications, design, code) which are abstractions that communicate the same information in different levels of detail, for different purposes.

LSD was developend with the intent to overcome the disadvantages of both LP and box structure. Specifically, it was intended to overcome LP's inability to specify interfaces between modules, box structure's inability to decompose boxes and the inability to implement the design created by box structures, as well as the lack of tools to support box structure [5].

The main idea behind LSD was to overcome the disadvantages of LP and box structure. These disadvantages included: the inability of LP to specify interfaces between modules; the lack of ability to decompose boxes; a lack of tools to support box structure [5]; a lack of ability to implement the high-level analysis and design created using box structures.

The framework developed for LSD, "WebBox", expanded LP and box structures in a variety of ways. It included new chunk types, the ability to refine chunks, the ability to specify interfaces and communication between boxes, and the ability to decompose boxes at any level.

2.4 Reproducible Research

Being able to reproduce results is fundamental to the idea of good science. Reproducible research has been used to mean embedding executable code in research papers to allow readers to reproduce the results described [27].

Combining research reports with relevant code, data, etc. is not necessarily easy, especially when dealing with the publication versions of an author's work. As such, the idea of *compendia* were introduced [8] to provide a means of encapsulating the full scope of the work. Compendia allow readers to see computational details, as well as re-run computations performed by the author. Gentleman and Lang proposed that compendia should be used for peer review and distribution of scientific work [8].

Currently, several tools have been developed for reproducible research including, but not limited to, Sweave, SASweave, Statweave, Scribble, and Org-mode. The most popular of those being Sweave [27].

3 Research Objectives and Approach

3.1 Objectives

-Want to tackle SC software dev current problems by simplifying the dev process

-Improve software qualities (maint. trace. reproduc. verify. reus.). How? Better understand underlying knowledge.

-Communicate the understanding of the knowledge -¿ better documentation (software artifacts) that's always up to date.

-Devs should spend less time maintaining.

-Avoid a classical software mistake (shown to be prevalent in SC thanks to Owen): duplication. Reuse, don't reimplement.

-Everything should be reproducible.

-Remove technology constraints

3.2 Approach

-Use ideas from LP, but expand them. -chunks: not just pieces of code; representations of knowledge. -capture knowledge: easily reuse it, never duplicate it.

-Knowledge stored should be spec-level. Assumptions, derivations, etc. - Pretty easy to get code from there. - Automate code generation.

-Move away from LP for maintainability, consistency, traceability, and reproducibility. -Recipes, standard generator, thematically linked knowledge-bases. -Given the above, anyone should be able to reproduce results. -All knowledge is easily traced to a source chunk. -Running the gen will ensure all artifacts are consistent.

-Make sure everything from the previous section is addressed.

4 Current Work and Preliminary Results

4.1 The Drasil Framework

-Introduce Drasil; eDSL(s) in Haskell

-chunks, recipes, generator

-multiple types of chunks (diagram)

-Recipe DSLs - Expr, Spec, LayoutObj.

-Gen specific ASTs - ASTC, ASTTeX, ASTHTML

4.2 Using Drasil

-h_g and h_c example overview.

-Water heater example overview.

-show some of the simple example impl.

-talk about common knowledge and (example) reusable library SI Units.

-discuss the output being nearly identical to the original (exceptions being specific design decisions such as ignoring the initial "Units" and only using "SI Equivalent").

-Show off some of the output -¿ Screencap the HTML page for water heater or FP?

-Discuss adv and disadv of method.

-Adv: No inconsistencies. Avoid manual copying of knowl through artifacts. Auto-propagate updates to knowl. Mentioned before: reusable knowl (SI). Supports design for change (properly captured knowledge means we just tweak a config to change our software). 100% traceability. Pervasive bugs!

-Disadv: No (consistent) local hacks. All-or-nothing approach removes meaningful iteration. Modifying knowledge over the lifetime of a project can become a very involved process. Creating common-knowledge libraries is difficult (requires domain expert).

5 Work Plan and Next Steps

A full schedule of my work plan can be found in Section 5.1. There are still many features I would like to add to Drasil as well as improvements to the overall implementation. Currently anticipated additions and changes (in no particular order) are as follows:

- Encapsulate more types of information in chunks. Some of the next additions should be physical constraints and reasonable values.
- Use constraints to generate test cases.
- Implement much larger examples.
- Generate code in more languages. Specifically MATLAB is the next planned output language implementation.
- Generate more artifact types. As it stands, the current recipes only create requirements documents or code. New recipes should be included to cover design documents, test cases, build instructions, user manuals, and more.
- Generate different document views. This is partially implemented in the requirements document recipe by allowing simplified or verbose data descriptions. The ability to simplify parts of the document that are unnecessary for a target audience should be expanded.
- Create an external syntax for Drasil.

Over the summer (2016), three undergraduate students will be working on translating existing implementations of large examples into Drasil implementations. It is my hope that this experiment will provide insight onto any lacking features of Drasil, as well as how well the new approach works. A second PhD student will also be joining the project over the summer. His first task will involve implementing a large example and helping to expand Drasil.

5.1 Detailed Schedule

Table 1 shows a current breakdown of what I will be doing for the rest of my tenure as a PhD student.

Table 1: A detailed work schedule for the next twenty-eight months

Summer 2016	Summer student experiment. Implement multiple (3-4) large examples using Drasil, updating the framework as new needs for features arise.
May 2nd 2016	Introduce summer students to Drasil and begin teaching them.
May 2016	Improve C output and implement MATLAB output in Drasil.
June/July 2016	Implement more artifact types in Drasil
August 2016	Submit results of summer experimentation to SEHPCCSE'16
September 2016	Overhaul the Drasil back-end to solidify necessary features and redesign parts of the implementation.
September 2016	Begin the last course for my PhD requirements.
October 2016	Meet with Ernie from OPG.
October 2016	Begin implementation of more knowledge capture.
November 13, 2016	(Hopefully) Present work at SEHPCCSE'16.
November 2016	Finish implementation of more KC.
December 2016	Finish coursework.
December 2016	Write a paper to submit to ??.
January 2016	Begin implementation of auto-gen'd test cases.
January 2017	Begin work on external syntax.
February 2017	Finish work on external syntax.
February 2017	Write a paper to submit to a Journal ??
March 2017	Re-evaluate current Drasil implementation for usability. Work on making it as user-friendly as possible
April 2017	Find and collect more large-scale examples to be implemented and begin implementation of one.
April 2017	Meet with my committee.
May 2017	Write a paper to submit to ??
Summer 2017	Attempt to get summer students for second round of experimentation. Complete implementation of multiple large-scale examples.
August 2017	Write a paper to submit to (Journal ??) detailing results.
September 2017	Meet with Ernie from OPG.
Sept - Nov 2017	Update Drasil and perform a final evaluation.
Dec 2017	Begin writing up full PhD Thesis.
February 2018	Complete first draft of thesis and send to supervisors for comments.
March 2018	Begin editing thesis.
May 2018	Complete final thesis draft before defense.
June 1-14	Vacation.
June 21, 2018	Defend thesis.
July-August 2018	Make any necessary revisions to the thesis, if major: defend again; if minor: submit the finished copy.

References

[1] Karen S. Ackroyd, Steve H. Kinder, Geoff R. Mant, Mike C. Miller, Christine A. Ramsdale, and Paul C. Stephenson. Scientific software development at a research facility. *IEEE Software*, 25(4):44–51, July/August 2008.

[2] Shereef Abu Al-Maati and Abdul Aziz Boujarwah. Literate software development. *Journal of Computing Sciences in Colleges*, 18(2):278–289, 2002.

[3] Jacques Carette. Gaussian elimination: a case study in efficient genericity with metaocaml. *Science of Computer Programming*, 62(1):3–24, 2006.

[4] Jeffrey C. Carver, Richard P. Kendall, Susan E. Squires, and Douglass 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.

[5] Michael Deck. Cleanroom and object-oriented software engineering: A unique synergy. In *Proceedings of the Eighth Annual Software Technology Conference, Salt Lake City, USA*, 1996.

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

[7] Peter Fritzson, Johan Gunnarsson, and Mats Jirstrand. Mathmodelica-an extensible modeling and simulation environment with integrated graphics and literate programming. In *2nd International Modelica Conference, March 18-19, Munich, Germany*, 2002.

[8] Robert Gentleman and Duncan Temple Lang. Statistical analyses and reproducible research. *Journal of Computational and Graphical Statistics*, 2012.

[9] Marco S Hyman. Literate c++. *COMP. LANG.*, 7(7):67–82, 1990.

[10] Andrew Johnson and Brad Johnson. Literate programming using `noweb`. *Linux Journal*, 42:64–69, October 1997.

[11] Diane 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.

[12] Diane 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.

[13] Diane F. Kelly. A software chasm: Software engineering and scientific computing. *IEEE Softw.*, 24(6):120–119, 2007.

[14] Oleg Kiselyov, Kedar N Swadi, and Walid Taha. A methodology for generating verified combinatorial circuits. In *Proceedings of the 4th ACM international conference on Embedded software*, pages 249–258. ACM, 2004.

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

[16] Jeffrey Kotula. Source code documentation: an engineering deliverable. In *tools*, page 505. IEEE, 2000.

[17] Anders Logg, Kent-Andre Mardal, and Garth Wells. *Automated solution of differential equations by the finite element method: The FEniCS book*, volume 84. Springer Science & Business Media, 2012.

[18] Zeeya Merali. Computational science: ...error. *Nature*, 467:775–777, 2010.

[19] Harlan D Mills, Richard C Linger, and Alan R Hevner. Principles of information systems analysis and design. 1986.

[20] Nedialko 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.

[21] Steven J. Owen. A survey of unstructured mesh generation technology. In *INTERNATIONAL MESHING ROUNDTABLE*, pages 239–267, 1998.

[22] Matthew Patrick, James Elderfield, Richard OJH Stutt, Andrew Rice, and Christopher A Gilligan. Software testing in a scientific research group. 2015.

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

[24] Vreda Pieterse, Derrick G. Kourie, and Andrew Boake. A case for contemporary literate programming. In *Proceedings of the 2004 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists on IT Research in Developing Countries*, SAICSIT '04, pages 2–9, Republic of South Africa, 2004. South African Institute for Computer Scientists and Information Technologists.

[25] Norman Ramsey. Literate programming simplified. *IEEE software*, 11(5):97, 1994.

[26] Patrick J. Roache. *Verification and Validation in Computational Science and Engineering*. Hermosa Publishers, Albuquerque, New Mexico, 1998.

[27] Eric Schulte, Dan Davison, Thomas Dye, Carsten Dominik, et al. A multi-language computing environment for literate programming and reproducible research. *Journal of Statistical Software*, 46(3):1–24, 2012.

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

[29] Stephen Shum and Curtis Cook. Aops: an abstraction-oriented programming system for literate programming. *Software Engineering Journal*, 8(3):113–120, 1993.

[30] Volker Simonis. Progdoc–a program documentation system. *Lecture Notes in Computer Science*, 2890:9–12, 2001.

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

[32] Spencer Smith, Yue Sun, and Jacques Carette. Comparing psychometrics software development between CRAN and other communities. Technical Report CAS-15-01-SS, McMaster University, January 2015. 43 pp.

[33] Spencer Smith, Yue Sun, and Jacques Carette. Statistical software for psychology: Comparing development practices between CRAN and other communities. *Software Quality Journal*, Submitted December 2015. 33 pp.

[34] W. Spencer Smith, Nirmitha Koothoor, and Nedialko Nedialkov. Document driven certification of computational science and engineering software. In *Proceedings of the First International Workshop on Software Engineering for High Performance Computing in Computational Science and Engineering (SE-HPCCE)*, page [8 p.], November 2013.

[35] Harold Thimbleby. Experiences of 'literate programming'using cweb (a variant of knuth's web). *The Computer Journal*, 29(3):201–211, 1986.

[36] Gregory V. Wilson. Where's the real bottleneck in scientific computing? Scientists would do well to pick some tools widely used in the software industry. *American Scientist*, 94(1), 2006.