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:	Literate Software and the Drasil Framework
Submitted by:	Dan Szymczak
Student Number:	0759661
	szymczdm@mcmaster.ca
Date of Submission:	June 15, 2016
Expected Date of Completion:	August 2018
Supervisors:	Jacques Carette
	Spencer Smith

Abstract

The current state of Scientific Computing software development leaves room for improvement. There is a need for improved software quality, more consistent use of software engineering practices, and reduced information duplication. As it stands, information duplication creates a number of problems, especially when it comes to software qualities such as traceability and maintainability. Traceability is hard to achieve, as finding the source of a piece of information, as opposed to transformations of that source (ex. an abstract theory and an instance model contain transformations of the same knowledge), is difficult among all of the possibilities. Over time this impacts the maintainability of a piece of software, as developers may not be able to determine how a change will affect the rest of the software. Also, with many duplicated information sources, creating documentation becomes more tedious, reducing a developer's desire to produce and maintain high-quality documentation. I want to facilitate the improvement of qualities (maintainability, traceability, reproducibility, etc.) across all artifacts of a piece of software. I believe a knowledge-based approach, which borrows ideas from literate programming, and involves a larger short-term investment from developers is the first step to improving software as a whole. I am creating a framework (Drasil) to facilitate this approach, focusing on avoiding knowledge duplication, while improving traceability and automating as much of the development process as possible. I believe with the proper tools for automation we can ensure good software engineering practices are followed (regardless of the developer's background) and improve the overall quality of software.

Contents

1 Introduction

Scientists and engineers rely heavily on models that share common mathematical/scientific knowledge. These are shared, in textbook form, across many projects. However, when it comes to implementing software, this information is then duplicated for each project, as well as being duplicated within the project (in requirements, design documents, code and tests, for example). It should be possible to capture this knowledge once, and then re-use it, first within the same project but in different views (documents/contexts), then across many projects. I want to create a system that deals with this issue of information duplication, especially for the knowledge contained in scientific models.

As it stands, information duplication can have a direct impact on qualities such as maintainability, traceability, verifiability, and reproducibility. Consider a project containing (at least) these four software artifacts: requirements document, design document, source code, and a testing document. As the project evolves, some of the knowledge contained within it will inevitably change due to changing requirements, new design decisions, technical or resource limitations, etc. Updating the software artifacts to reflect these changes is tedious; a developer must trace their way through each of the artifacts to ensure new information has been added (new requirements and/or design decisions, code implementation, and test cases), the change does not cause conflicts, and any existing test cases or assumptions are still applicable (or are appropriately removed/updated).

Now consider a software project which follows a rational design process as seen in Figure 1. It consists of even more software artifacts, thus each change to the knowledge becomes even more tedious to propagate. It is not uncommon for developers to lack the resources or motivation to update each artifact to accommodate every minor change.

As artifacts in a software project begin to fall out of sync with each other, the qualities of the project as a whole suffer. Consider traceability: if the artifacts in a software project are inconsistent with each other, how can one accurately analyze a change or validate a requirement? On a similar note, how easy is it to maintain a system if one cannot track where a problem came from? Also, certification becomes impossible until all of the artifacts have been updated, as there is no way to verify/validate the software's correctness. I want to create a system that allows developers to maintain and update software artifacts at very little cost.

Up-to-date software artifacts are also incredibly important for scientists as a whole. Good science relies on reproducible results; being able to reproduce the exact piece of software, as well as any inputs used in running a simulation is vital. Due to library version changes, minor hacks, and undocumented modifications, it is often difficult for other scientists to reproduce the exact results from a given simulation[11]. Would it not lead to higher confidence if we could reproduce and analyze the results of the original experiment? I want to help scientists ensure their work is easily reproducible by giving them a means of sharing how their software was built, as well as any necessary input or configuration data.

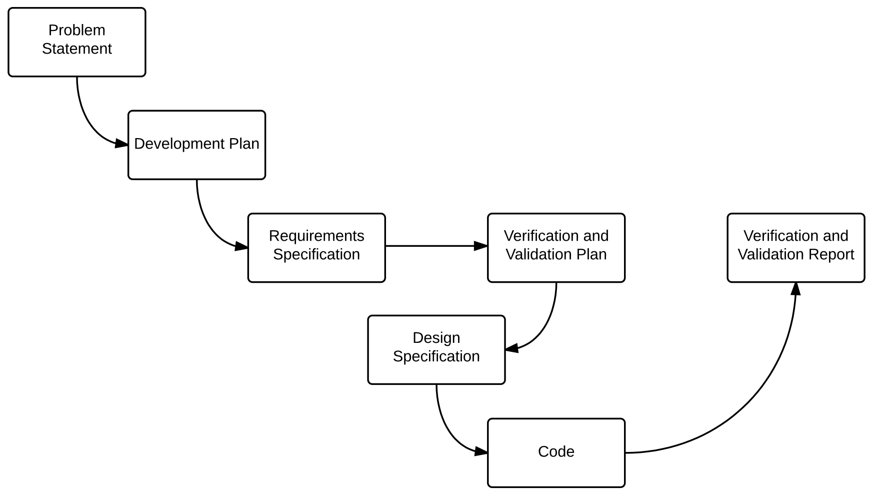

Figure 1: Example Rational Design Process and its Software Artifacts

I think the first step to improving software as a whole is to improve each of the software artifacts. To that end, I would like to expand the ideas of literate programming (which I will discuss more in-depth in Section 2.2). Specifically, I want to keep the idea of *chunks* and expand its application to not only code snippets, but the fundamental knowledge underlying a piece of software. I believe capturing this knowledge will be paramount to creating a new, so-called "knowledge-based" approach to software development.

As improving all software is too broad of a target, I will be focusing on well-understood domains. I believe this is a necessary restriction because a knowledge-based approach will require a thoroughly well-understood background to facilitate the requisite knowledge capture. To that end, I will focus specifically on the domain of Scientific Computing (SC). SC has a very rich, well-understood background, which has already been documented in incredible detail. Also, in many cases, SC developers are scientists first and lack a strong software engineering background. It is my hope that my approach will be well-suited to SC developers as it will allow them to focus more of their energy on the science than the software development.

I do not believe I am creating any individual new ideas. Instead, I want to combine ideas that have existed for some time and use them in a novel way. That is where the idea of a knowledge-based approach to literate software comes from. Not only do I want to create better software using a process similar to literate programming, I want to ensure that process will cover all necessary software artifacts.

To that end, I believe a tool will be necessary to facilitate my approach. A framework known as *Drasil* is currently in development and will be explained in Section 4.

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 (LP), 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 [15]. 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, 34], an amethododical [13], or a knowledge acquisition driven process [14] 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 [26] 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 [23]. It is in such a bad state that quality assurance has "a bad name among creative scientists and engineers" [32, p. 352], not to mention the very limited use of automated testing [27].

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 [42].

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 [22], FFT [16], Gaussian Elimination [3], and Spiral[30]. 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 [37, 38, 39, 40].

2.2 Literate Programming

The LP method 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 [17].

Developing literate programs involves breaking algorithms down into *chunks* [12] or *sections* [17] which are small and easily understandable. The chunks are ordered to promote understanding, a "psychological order" [29] 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 [35]. Having consistent documentation has its own advantages while developing or maintaining software [10, 18]. Similarly, there are many downsides to inconsistent documentation [18, 41]. Keeping both of those in mind we can see that more effective, maintainable code can be produced when (properly) using LP [29].

Even with all of the benefits of LP, it has not been very popular [35]. Still, there are several successful examples of LP's use in SC. Two such examples are VNODE-LP [25] and "Physically Based Rendering: From Theory to Implementation" [28]. 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, and the lack of flexibility on what should be presented or suppressed in the output.

I believe there are several other factors which contributed to LP's lack of popularity and slow adoption thus far. LP allows a developer to write their code and its documentation simultaneously. However, that documentation is comprised of a single document which does not necessarily cover the same material as the standard artifacts software engineers expect. LP also does not really simplify the development process: documentation and code must be written as usual, and there is the added effort of re-ordering the chunks. The only major benefits to the development process are that the chunks can be written in any order (allowing a more natural flow in development), the documentation and code are (in theory) updated simultaneously, and chunks can be automatically incorporated into the documentation (reducing some information duplication).

Many attempts to address the issues with LP's popularity have focused on changing or removing the output language or text processor dependency. Several new tools were developed such as: CWeb (for the C language), DOC++ (for C++), noweb (programming language independent), and more. Tools such as javadoc (for Java) and Doxygen (for multiple languages) were also influenced by LP, but differ in that they are merely document extraction tools. They do not contain the chunking features which allow for re-ordering algorithms.

The development of new tools led to the introduction of many new features including, but not limited to, a "What You See Is What You Get" (WYSIWYG) editor [8], phantom abstracting [35], and even movement away from the "one source" idea [36].

While these tools did not bring LP into the mainstream [31], they did help drive the understanding behind what exactly LP tools must do. We can now see LP becoming more standardized in certain domains (for example: Agda, Haskell, and R support LP to some extent), even though it is not yet common practice. R has good tool support, with the most popular being Sweave [19], however it is designed to dynamically create up-to-date reports or manuals by running embedded code as opposed to being used as part of the software development process. I intend to use the knowledge gained from the history of LP to develop the Drasil framework into a full-featured and useful tool.

2.3 Literate Software

A combination of LP and Box Structure [24] was proposed as a new method called "Literate Software Development" (LSD) [2]. Box structure can be summarized as the idea of different views which are abstractions that communicate the same information in different levels of detail, for different purposes. Box structures consist of black box, state machine, and clear box structures. The black box gives an external (user) view of the system and consists of stimuli and responses; the state machine makes the state data of the system visible (it defines the data stored between stimuli); and the clear box gives an internal (designer's) view describing how data are processed, typically referring to smaller black boxes [24]. These three structures can be nested as many times as necessary to describe a system.

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

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. However, literate software (and LSD) remains primarily code-focused with very little support for creating other software artifacts, in much the same way as LP. This is one area where I believe Drasil will outperform WebBox.

2.4 Reproducible Research

Being able to reproduce results, is fundamental to the idea of good science. When it comes to software projects, there are often many undocumented assumptions or modifications (including hacks) involved in the finished product.

This can make replication impossible without the help of the original author, and in some cases reveal errors in the original author's work [11].

Reproducible research has been used to mean embedding executable code in research papers to allow readers to reproduce the results described [33].

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 [9] 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 [9].

Currently, several tools have been developed for reproducible research including, but not limited to, Sweave [19], SASweave [21], Statweave [20], Scribble [7], and Org-mode [33]. The most popular of those being Sweave [33]. The aforementioned tools maintain a focus on code and certain computational details. Sweave, specifically, allows for embedding code into a document which is run as the document is being typeset so that up to date results are always included. However, Sweave (along with many other tools), still maintains a focus on producing a single, linear document. It is my hope that Drasil will outperform these existing tools due to its flexibility and its ability to create multiple artifacts from a knowledge base.

3 Research Objectives and Approach

3.1 Objectives

Looking at the current state of SC software development, there are many areas for improvement particularly the need for applying better software engineering practices and improving knowledge reuse. Domain experts require tools that simplify the software engineering allowing them to do things the "right" way. My first objective, therefore, is to simplify the software development process by allowing scientists to focus on the science. This objective, however, will not be achievable during the time frame I have allotted for my thesis, so I believe a more focused and attainable objective would be to create tool(s) which will enable knowledge-driven SC software development.

My second objective, is the production of better overall software artifacts (requirements and design documents, source code, etc.). It is a worrying trend that software artifacts tend to fall out of sync with each other over time, so ensuring they remain consistent is a means to improve.

Furthermore, developers should not need to spend large swaths of time updating and maintaining their software. I want to automate as much of the software development process as possible. Automation will also allow developers to avoid classical development mistakes such as duplication, previously mentioned in the survey by Owen [26]. It will also ensure reproducibility as, if done right, anyone will be able to automatically create and run the same software easily.

3.2 Approach

To meet my objectives I am using a combination of existing ideas, which I have dubbed a *knowledge-based* approach to software development. First off, I want to expand and use ideas from LP. Namely, I want to keep the brilliant idea of chunks, however, it should be expanded to represent encapsulated knowledge, not just code. If the knowledge behind an underlying concept can be properly encapsulated, then it can easily be reused and duplication can be avoided.

Knowledge encapsulation should be done at the specification level instead of the code level as was previously done in LP. It will include assumptions, derivations, equations, etc. and from there it will be a fairly simple and straightforward process to get a code representation. Specifically, the high-level knowledge can be used for the automatic generation of code. I believe this will simplify the development of software artifacts by automatically allowing captured knowledge to be transformed and reused as necessary.

The standard LP style helps to increase consistency in documentation, but it does not go far enough. It is easy for a developer to update the code and/or documentation, however, there is no way to ensure that when the code is changed the documentation is also updated accordingly. I want to ensure that all artifacts are updated any time there is a change (no matter how trivial) with trivial effort.

This automatic updating is achieved through generation. The idea is to have a standard generator which uses *recipes* for the creation of artifacts. Each recipe essentially defines a different view of our knowledge-base (chunks). Since each recipe represents a different software artifact and pulls appropriate knowledge from reusable chunks (see Figure 2), the artifacts will remain consistent and fully traceable; finding the source of a piece of information will be trivial. All artifacts will automatically be updated any time the generator is run, thus ensuring consistency is maintained. I believe this is one of the key features that will set my approach apart from those that have come before and will make it all the more useful for (SC) software developers.

I also believe my approach will be useful for scientists because anyone who is given access to the recipes and chunks used in creating a piece of software will be able to reproduce that software exactly.

Currently, I am taking a practical, example-driven approach to create a framework, Drasil, to exemplify this knowledge-based approach to software development. Thus far, the implementation of Drasil has been guided by some small and fairly specific case studies. However, I plan to continue expanding the framework through the use of larger, more varied case studies. More information regarding Drasil can be found in the next section.

4 Current Work and Preliminary Results

4.1 The Drasil Framework

Drasil is currently being implemented as a combination of embedded Domain Specific Languages (eDSLs) in Haskell. Currently, six eDSLs have been imple-

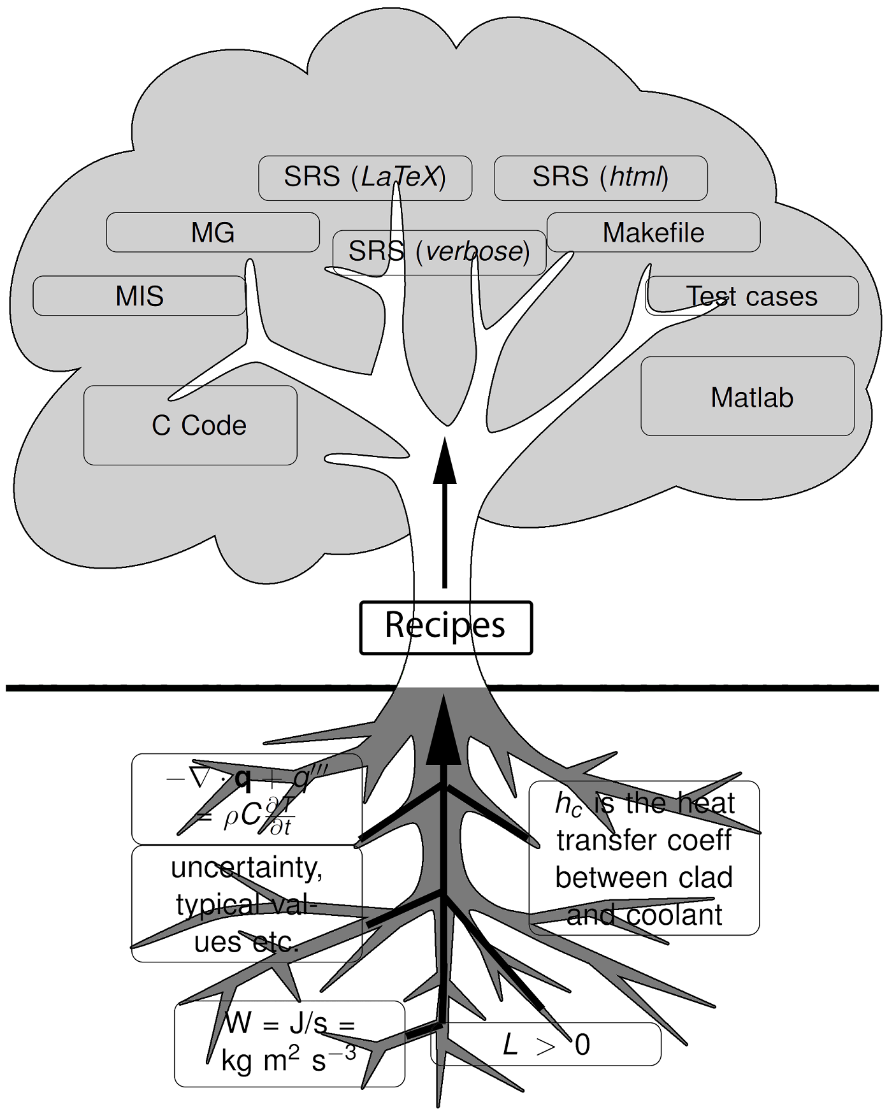

Figure 2: Recipes pull relevant information from chunks (the roots of the tree) and combine them to create software artifacts (the leaves of the tree).

mented as follows:

1. Expression language – A simple expression language that allows one to capture knowledge relating to equations and mathematical operations. It includes operations such as addition, multiplication, derivation, and exponentiation among others.

2. Expression layout language – A micro-scoped language for describing how expressions should appear. Expressions may need to use a variety of inline layout modifiers (subscripts, superscripts, etc.) to be properly displayed.

3. Document layout language – A macro-scoped language for describing how large-scale layout objects (tables, sections, figures, etc.) should appear.

4. C Representation Language – A DSL for representing parts of the C programming language inside the Drasil framework. This allows the generator to produce working C code.

5. LaTeX Representation Language – A DSL for representing LaTeX code inside of Drasil. As with the C representation, it is used by the generator to produce working LaTeX code.

6. HTML Representation Language – A DSL for representing HTML within Drasil. Similar to the other representation languages as it is used by the generator to produce working HTML.

With these eDSLs it is easy to encapsulate knowledge, create recipes, and generate required artifacts.

When it comes to chunks, they come in several varieties. You can see a hierarchy of chunk types in Figure 3.

Each new chunk in the hierarchy adds to the knowledge encapsulated within it. The most basic *Chunk* represents a named piece of information. A *Concept* adds a description to the named information, and so on.

Unital chunks are somewhat special as they do not add any new information in and of themselves, they act as a combination of a *Quantity* (concept with a symbolic representation) and a *Unit*. They are quantities with units.

Equation chunks continue to expand on *Unital* chunks by allowing us to capture equations for calculating quantities. In a similar vein are *Relation* chunks, which allow us to relate two pieces of knowledge to each other.

Now that knowledge has been encapsulated it is time to do something with it. This is where the recipes, mentioned earlier, come into play. Writing a recipe requires using three of the previously mentioned eDSLs – the expression, expression layout, and document layout languages. I will go into more detail on writing and using recipes in the next section.

Finally, the last piece of the implementation is the generator. The remaining three representation languages are specific to the generator. The generator interprets the recipes and creates intermediary representations of artifacts using the representation languages, then pretty-prints the results.

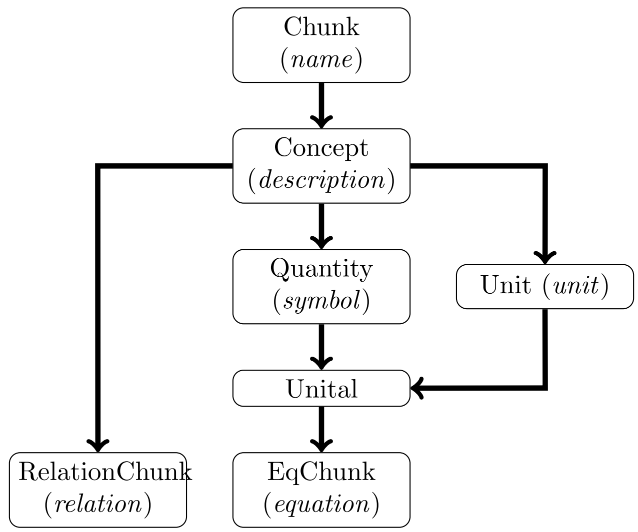

Figure 3: Our current chunk design (parentheses indicate newly added knowledge)

4.2 Using Drasil

For this section I will touch on one of the case studies that I have used to motivate the development of Drasil. It is a simplified version of a Software Requirements Specification (SRS) for a fuel pin in a nuclear reactor [37]. While other case studies are being used to help motivate Drasil's development, they will not be shown explicitly at this time, as their results can be seen using the fuel pin example.

Let's start by looking at a single term from the fuel pin SRS: h_c. This term represents the convective heat transfer coefficient between the clad and coolant. We can see the data definition for this term in Figure 4.

From the data definition, we can glean interesting knowledge. First off there is the name of the concept, its description, the symbol representing it, its SI units, and its defining equation. All of this information can be encapsulated into an EqChunk with ease as shown in Figure 5. Note that the units for h_c are defined in a piece of common knowledge known as `heat_transfer`.

The SI Units are a great example of common knowledge. The SI Unit library contains all seven base SI units and several different derived units (for example degrees Celsius, which are derived from Kelvin). The seven fundamental base SI units implemented in Drasil can be seen in Figure 6.

From the captured knowledge and common knowledge, it is now possible to start putting together a document using recipes. A small portion of the recipe for the simplified fuel pin SRS can be seen in Figure 7. Currently the recipes are fairly clunky and rely heavily on Haskell, but in the future this will be

Number	**DD2**
Label	h_c
SI Units	$\frac{\text{kW}}{\text{m}^{2o}\text{C}}$
Equation	$h_c = \frac{2k_c h_b}{2k_c + \tau_c h_b}$
Description	h_c is the convective heat transfer coefficient between clad and coolant k_c is the clad conductivity h_b is the initial coolant film conductance τ_c is the clad thickness

Figure 4: Data definition for h_c from the fuel pin SRS

```
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 in Drasil

```
metre, second, kelvin, mole, kilogram, ampere, candela :: FundUnit
metre    = fund "Metre"    "length (metre)"                "m"
second   = fund "Second"   "time (second)"                 "s"
kelvin   = fund "Kelvin"   "temperature (kelvin)"          "K"
mole     = fund "Mole"     "amount of substance (mole)"    "mol"
kilogram = fund "Kilogram" "mass (kilogram)"               "kg"
ampere   = fund "Ampere"   "electric current (ampere)"     "A"
candela  = fund "Candela"  "luminous intensity (candela)"  "cd"
```

Figure 6: The seven fundamental SI Units in Drasil

```
srsBody = srs [h_g, h_c] "Spencer Smith" [s1,s2,s3]

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 7: A portion of our simplified SRS recipe

improved. However, even with the current recipes we can see that the output from the generator (Figure 8) is exactly what we want. Note: the output I am showing is from the HTML version being generated, the TeX version looks (almost) identical to the original SRS.

Already several advantages of using Drasil and a knowledge based approach become apparent First off, there are no inconsistencies in the knowledge both within and across artifacts. Manually copying knowledge is no longer necessary, and we can easily trace where a piece of knowledge came from. Should anything need updating, the updates will automatically propagate throughout the artifacts when the generator is run. We also have completely reusable knowledge (for example: the SI Units) and the use of recipes supports design for change. As long as knowledge is captured properly, it is as simple as changing a configuration file to change the software.

One advantage that may seem like a disadvantage (but is not) is that of pervasive bugs. Thanks to the full traceability and reuse of knowledge, a single bug will be reproduced anywhere that piece of knowledge is necessary. This improves the odds of finding it and makes it simple to fix. A single change should update all of the artifacts and remove the bug.

There are a few disadvantages to using the knowledge based approach with Drasil. The most obvious is the inability to include local hacks. Any human-modified generated file will lose its modifications the next time the generator is run. Creating common-knowledge libraries is also fairly difficult as it requires the involvement of domain experts.

5 Work Plan and Next Steps

A full schedule of my work plan can be found in Table 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.

Table of Units

Throughout this document SI (Système International d'Unités) is employed as the unit system. In addition to the basic units, several derived units are employed as described below. For each unit, the symbol is given followed by a description of the unit with the SI name in parentheses.

Symbol	**Description**
m	length (metre)
kg	mass (kilogram)
s	time (second)
K	temperature (kelvin)
mol	amount of substance (mole)
A	electric current (ampere)
cd	luminous intensity (candela)
°C	temperature (centigrade)
J	energy (joule)
W	power (watt)
cal	energy (calorie)
kW	power (kilowatt)

Figure 8: Section 1 of the generated SRS (HTML version)

- 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 on 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.

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. Write up results of experiment as a paper and submit to SEH-PCCSE'16.
Fall 2016	Overhaul the Drasil back-end to solidify necessary features and redesign parts of the implementation. Finish PhD course requirements. Meet with Ernie from OPG.
Winter 2016	Implement external syntax for Drasil. Write a paper for ICSR (International Conference on Software Reuse). Write a journal paper for Automated Software Engineering.
Spring 2017	Re-evaluate current Drasil implementation for usability. Work on making it as user-friendly as possible. Submit a paper to Onward! Submit a paper to ASE (International Conference on Automated Software Engineering) Meet with committee.
Summer 2017	Write a paper. Attempt to get summer students for second round of experimentation. Write a journal paper detailing results.
Fall 2017	Meet with Ernie from OPG. Update Drasil and perform a final evaluation.
Winter 2017	Begin writing up full PhD Thesis.
Spring 2018	Complete first draft of thesis and send to supervisors for comments. Complete first round of edits.
Summer 2018	Complete final thesis draft before defense. Make any necessary revisions to the thesis and submit it.

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] Matthew Flatt, Eli Barzilay, and Robert Bruce Findler. Scribble: Closing the book on ad hoc documentation tools. In *ACM Sigplan Notices*, volume 44, pages 109–120. ACM, 2009.

[8] 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.

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

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

[11] Cezar Ionescu and Patrik Jansson. Dependently-Typed Programming in Scientific Computing — Examples from Economic Modelling. In *Revised Selected Papers of the 24th International Symposium on Implementation and Application of Functional Languages*, volume 8241 of *Lecture Notes in Computer Science*, pages 140–156. Springer International Publishing, 2012.

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

[13] 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.

[14] 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.

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

[16] 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.

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

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

[19] Friedrich Leisch. Sweave: Dynamic generation of statistical reports using literate data analysis. In *Compstat*, pages 575–580. Springer, 2002.

[20] Russell V Lenth. Statweave users manual. *URL http://www. stat. uiowa. edu/˜ rlenth/StatWeave*, 2009.

[21] Russell V Lenth, Søren Højsgaard, et al. Sasweave: Literate programming using sas. *Journal of Statistical Software*, 19(8):1–20, 2007.

[22] 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.

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

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

[25] 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.

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

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

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

[29] 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.

[30] Markus Püschel, José MF Moura, Jeremy R Johnson, David Padua, Manuela M Veloso, Bryan W Singer, Jianxin Xiong, Franz Franchetti, Aca Gačic, Yevgen Voronenko, et al. Spiral: Code generation for dsp transforms. *Proceedings of the IEEE*, 93(2):232–275, 2005.

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

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

[33] 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.

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

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

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

[37] 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.

[38] 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.

[39] 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.

[40] 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.

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

[42] 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.