Be Consistent

A Knowledge-Based Approach to Software Engineering

Dan Szymczak
McMaster University
szymczdm@mcmaster.ca

Jacques Carette Spencer Smith
McMaster University
carette@mcmaster.ca / smiths@mcmaster.ca

Abstract

Every developer should aim to create high-quality software. In many instances, software artifacts are not kept consistent or up-to-date. This lack of consistency negatively impacts the overall quality of the software, particularly its maintainability, traceability, and verifiability. We have combed through the work of others to find ideas we believe will contribute to better consistency and overall higher quality software. In combining these ideas, we have created a knowledge-based approach for software development. We believe that if there is a strong theoretical background in a given problem domain, and if it can be effectively operationalized, then our method will lead to long-term improvements in software quality for a (high) short-term investment. We have created a framework (Drasil) to demonstrate these ideas.

Keywords Literate software, knowledge capture, traceability, software engineering,scientific computing, artifact generation, software quality

1. Introduction

We want to make better software. In particular, we are interested in increased maintainability, traceability, reproducibility, verifiability, and reusability. Our approach is to invest (much) more in the short-term to provide outstanding long-term benefits. This is the fundamental trade-off in our work: we expect our methods to work very well for domains which are well understood with known (but potentially enormous) design spaces. We do not claim to tackle areas of software development where methodologies such as agile are well-suited. But we firmly believe that there are domains – such as safety-critical applications – where agile is not only ill-suited, it would be downright unprofessional to use such a methodology.

Rather than talk in vague generalities, we will pick Scientific Computation (SC) for illustrative purposes throughout this paper. But the fundamental ideas (and, in fact, our framework) should be applicable to any software domain which has well-established theoretical underpinnings, and a well understood translation of the theory into effective code. SC is particularly well-suited as it is also replete with *program families*.

We believe that Knuth's Literate Programming (LP) (Knuth 1984) contains some fundamental insights, but is too restricted (i.e. just to code). Rather than restrict LP to just source code, we want to apply it to all software artifacts (requirements and design documents, source files, test cases, build instructions, user manuals, etc.). In other words, our basic starting points is to "chunk" all aspects of the software construction process, and then weave the actual artifacts from those chunks. We will discuss the details of how we do this in Section 3.

The LP idea of putting everything into a single source file will not scale in this way. Instead, we introduce the idea of libraries of "common knowledge" for concepts which tend to re-occur. A low-level example would be the units from the Système International (SI) Units as "common knowledge" that really ought to get defined only once, and then re-used. A higher-level example would be the various components of the theory of heat transfer, including the conservation of energy equations.

We should make it quite clear that none of the individual ideas presented in this paper are new. Knuth himself said that "I have simply combined a bunch of ideas that have been in the air for a long time" when he coined LP (Knuth 1984). We are, however, taking these ideas and re-assembling them in novel ways. Our principal aim is not a new software framework for doing literate software, but rather a framework which, over the long term, will allow us to do our own work (largely in scientific computation, but also in mechanized mathematics) much more effectively. Our design is thus largely example-driven; we indeed have an industrial client who is funding this work, with the common aim of

reducing the long-term cost of re-certifying scientific computation software as fit-for-use for certain safety critical applications. While the details of that are beyond the scope of this paper, they are never far from our minds as we design and implement our framework.

Others have done similar work (which we will get into in Section 2), but they did not achieve the results we are envisioning. They either set their aims on other targets, spent too long creating a grand design and ended up without any real, practical results, or they simply were not brave enough to break things down into the smallest necessary chunks.

We believe that we have assembled the right ideas to achieve our vision. The overall success or failure of our approach hinges on the idea of a stable, well understood knowledge base. If such knowledge does not exist, or cannot be adequately captured, our approach will not work. Our claim then is that there are areas of application where the knowledge not only exists, it can be adequately captured and operationalized in very effective ways.

As it stands, we are on a (not so) humble, practical route to achieving our goals and improving the overall quality of (some scientific computation) software.

2. Requirements and inspiration

As we mentioned previously, none of the ideas behind our approach are new. There has been a lot of work done by a lot of smart people and we are taking advantage of it. Let us first take a look at the current state of SC software development, then we will discuss our goals and requirements, followed by the tools and techniques that inspired us. Finally we will conclude this section by discussing which ideas we intend to borrow and how we intend to bring everything together.

2.1 The current state of SC software development

Most developers in SC software tend to put an emphasis on their science and not on good software development (Kelly 2007). Consequently these developers tend to naturally use an agile philosophy (Ackroyd et al. 2008; Carver et al. 2007; Easterbrook and Johns 2009; Segal 2005) or an amethododical (Kelly 2013), or a knowledge acquisition driven (Kelly 2015) process. These developers are scientists first and foremost and they do not view rigid, process-heavy approaches favourably (Carver et al. 2007).

More than half of these scientists do not have a good understanding of software testing (Merali 2010). In fact, there is very limited use of automated testing for scientific software (Patrick et al. 2015) and quality assurance has "a bad name among creative scientists and engineers" (Roache 1998, p. 352).

Another problem in SC is that knowledge reuse is not fulfilling its potential. For instance, for mesh generators a large number of similar programs have been written. A survey (Owen 1998) shows 81 different mesh generator packages, with 52 of them generating triangular meshes, and 37 of these using the same algorithm of Delaunay triangulation.

Tool use in SC software development is also limited, especially the use of version control software (Wilson 2006). However, code generation has been successful in SC, but thus far the focus has been on the creation of only one software artifact: the code. Examples include

The lack of emphasis on more artifacts is disadvantageous to SC software developers. The value of documentation and a structured process is illustrated by a survey of statistical software for psychology (Smith et al. 2015, Submitted December 2015). A case study on the impact of a document driven process for SC software was performed on legacy software for thermal analysis of a fuel pin in a nuclear reactor (Smith and Koothoor 2016; Smith et al. 2013). The redeveloped version discovered 27 issues, ranging from trivial to substantive, within the previous documentation. If rational documentation were used more often, some previous errors in scientific code may have been uncovered. Examples include the Sleipner oil rig, protein retraction, the Patriot missile, and seismic data processing (Hatton and Roberts 1994).

2.2 Goals and requirements for our approach

The current state of SC software development informs our requirements. We want to tackle many of the problems stated therein, without requiring scientists to become professional software developers. Instead, we should simplify the development process by allowing them to focus on the science.

First and foremost we want to improve software qualities, specifically maintainability, traceability, reproducibility, verifiability, and reusability. We believe the first step towards these improved qualities is a better understanding of the underlying knowledge behind the software itself.

Being able to communicate this improved understanding is also necessary, as it will help with maintainability, reproducibility, and more down the road. What that means to us is ensuring that all software artifacts (requirements, design, source code, etc.) must be consistent and *always* up to date.

If we can essentially *force* the developer to always have consistent and up to date artifacts, without impeding their ability to develop (or wasting their time), then we believe we have already succeeded in improving the software. The less time a developer needs to spend maintaining and updating software, the more time they have to do other important things.

We also want to avoid one of the most classical software engineering mistakes: duplication. Knowledge should not be duplicated, regardless of how many times it must be displayed. If we can capture knowledge in an appropriate manner, then we can simply reuse that knowledge wherever necessary without needing to reimplement it.

Finally, software (and results) should be reproducible. We want to ensure that given all of the appropriate knowledge surrounding a piece of software, scientists will be able to

reproduce that piece of software and any results obtained therein.

To tackle our requirements we need to understand a lot of what has come before. This includes (but is not limited to) methods for ensuring consistency across software artifacts and work done in the field of reproducible research.

2.3 Literate Programming

Literate programming is a programming methodology that was introduced by Knuth. The LP methodology changes the focus from writing programs which simply instruct the computer how to perform a task. Instead, the focus is now on explaining (*to humans*) what we want the computer to do (Knuth 1984).

In a literate program, the documentation and source code is all kept together in one source. Literate programs are developed by breaking down algorithms into small, easy to understand parts referred to as *chunks* (Johnson and Johnson 1997) or *sections* (Knuth 1984). These chunks are then explained, documented, and implemented in a way that promotes understanding – they are not necessarily written in the order that the computer would read them, but rather a "psychological order" (Pieterse et al. 2004). To get the working source code for a program written in the LP style, you run the *tangle* process. Similarly, to extract and typeset the documentation you run the *weave* process.

LP has advantages beyond understandability. Using LP has been experimentally found to lead to better consistency between code and documentation (Shum and Cook 1993). While the program is being developed, the documentation tends to be updated simultaneously as it surrounds the source code. We should also keep in mind that proper, consistent documentation leads to many advantages while developing or maintaining software (Hyman 1990; Kotula 2000). When documentation does not match the system, it is a major detractor to software quality (Kotula 2000; Thimbleby 1986). Taken together, we can see how LP (when used properly) leads to more elegant, effective, maintainable, and understandable code (Pieterse et al. 2004).

With all of the benefits of LP it is fairly astounding that it has not been very popular (Shum and Cook 1993). There are very few successful examples of LP in SC, however, two that come to mind are VNODE-LP (Nedialkov 2006) and "Physically Based Rendering: From Theory to Implementation" (Pharr and Humphreys 2004). The latter being both a literate program and textbook on the subject. Shum and Cook discuss the topic of LP's lack of popularity a fair bit and present the idea that it comes from two main issues:

1. Output language / text processor dependency.
2. Lack of flexibility on what to present or suppress in the output.

There have been attempts to address these issues. Many focused on changing or removing the output language or text processor dependency. For example CWeb (for the C language), noweb (programming language independent), javadoc (for Java), DOC++ (for C++), Doxygen (for multiple languages), and more. Many of these new tools came with new and interesting features, examples include:

1. A "What You See Is What You Get" (WYSIWYG) editor (Fritzson et al. 2002).
2. Graphicdoc rules (using a word processor to compose graphics in LP) (Shum and Cook 1993).
3. Phantom abstracting (Shum and Cook 1993).
4. Movement away from the idea of "one source" (Simonis 2001).

The development of so many tools for LP has helped drive the understanding behind exactly what LP tools should be doing. On the other hand, the multitude of tools was preventing LP from becoming mainstream (Ramsey 1994). Nowadays, however, we can see parts of LP becoming standard in certain domains (for example: Haskell, Agda, and R support LP).

2.4 Expanding LP: Literate Software

A new methodology was proposed in 2002 by Al-Maati and Boujarwah called "Literate Software Development" (LSD). It was designed as a combination and evolution of LP and Box Structure (Mills et al. 1986). Box Structure is the idea of views (system specifications, design, code) with each view being an abstraction communicating the same information in different levels of detail, for different purposes (Al-Maati and Boujarwah 2002).

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 (Deck 1996); a lack of ability to implement the high-level analysis and design created using box structures.

WebBox, when implemented, expanded LP and box structures to include new chunk types; the ability to refine chunks into more chunks (and eventually Java code); the ability to specify interfaces and communication between boxes; and the ability to decompose boxes at any level.

2.5 Reproducible research

The term "reproducible research" is typically used to mean embedding executable code in research papers in order to allow readers to reproduce the results described (Schulte et al. 2012). Reproducibility of results is fundamental to the idea of good science.

Gentleman and Temple Lang introduced a means for combining research reports with the relevant code, data, etc. This is called a "compendium" and it encapsulates the full work of the author, as opposed to just the publication version of their work. It also allows the reader to re-run

computations (while explicitly showing the computational details) (Gentleman and Lang 2012). The authors proposed that peer review and distribution of scientific work should be done using compendia.

Several tools have been developed for reproducible research (Sweave, SASweave, Statweave, Scribble, Org-mode, etc.) with the most popular being Sweave (Schulte et al. 2012).

2.6 Bringing the ideas together

LP brings many great ideas to the table for helping us to keep our software artifacts consistent. First off, we love the idea of chunks. However, to achieve our goals we believe chunks should not be just pieces of code, but representations of knowledge. If we can "capture" knowledge about a concept within a chunk, then that knowledge can be reused readily. On that note, chunks can also decrease the amount of duplication. If the knowledge regarding one concept is stored in a chunk, we never need to duplicate it.

The knowledge we would like to store should also help us increase the verifiability of the program – we should be able to see all of the assumptions and derivations that the chunk relies on. As such, we want to be storing specification-level knowledge, not code-level. If anything, we should be able to easily get to a code-level representation from the higher level knowledge.

Making sure the developer always keeps their artifacts consistent is difficult in the standard LP style. Documentation and chunks are kept together in one source, but changing one does not necessarily mean the other will be updated accordingly. As such, we want to ensure that all of our artifacts are being automatically updated each time the knowledge base is updated. This also brings us to the idea of traceability: we should (at all times) be able to find where any concept or piece of code came from.

Instead of the previous `tangle` and `weave` processes, we want to introduce the idea of recipes and a standard generator. Instead of having two processes (one for documentation, one for source code) we effectively want to have one recipe for each software artifact, all of which can be used by a single standard generator. Recipes will allow us to trace exactly what knowledge is coming from which sources, as well as allow us to reproduce any software artifacts trivially. In fact, anyone with access to the recipes and knowledge base should then also be able to reproduce these artifacts.

With the idea of chunks and recipes working together to create all of our software artifacts, we can expect software that is much easier to maintain. All of our artifacts will consistently be up to date.

3. Drasil

We are currently developing a framework called Drasil (shortened from *Yggdrasil* from Norse Mythology, which is also known as the *world tree*). To avoid getting lost in the design phase, we have opted to take a very practical and example-driven approach in implementing Drasil. There are three fundamental ideas behind our design thus far:

1. Organize the knowledge base – We want a knowledge base that we can structure conceptually (i.e. keep knowledge for a certain class of problems together). This is where chunks come into play: each chunk encapsulates a single piece of knowledge like a concept or a quantity. We want to keep knowledge in the smallest possible pieces while simultaneously organizing it into knowledge libraries.
2. Use recipes to create artifacts – We can think of each artifact in our software project as a different view of our knowledge base. We want to use recipes to specify exactly what information from the knowledge base is necessary for each artifact, and how that information should be displayed. For many artifacts we would like to have a standard recipe which can be quickly customized for the current problem and that is how we will avoid duplicating knowledge.
3. Remove technology constraints – We want to be able to create our software without worrying about the underlying technical constraints of our display or specification technology, programming language(s), etc. Anyone using Drasil should be able to work with the knowledge base and their recipes, then simply set their output technology and have the generator take care of all the technical details.

We argue that by implementing Drasil around these ideas, we can see drastic improvements in software quality. In fact, using a generative approach we can avoid certain problems altogether. One obvious and recurring problem that comes to mind when upgrading software is "feature creep" or "software bloat" (Amsel et al. 2011). With Drasil, a software upgrade will actually be a completely new piece of software that includes the previously desired functionality as well as the new upgrades. We will discuss further improvements in more detail after our example in Section 3.2.

3.1 Drasil's current implementation

Fundamentally, Drasil must be able to capture knowledge and produce different views of that knowledge. With our current implementation we have each individual piece of knowledge as a named *chunk*. We are then able to manipulate our chunks through the use of a *recipe*. Our *generator* interprets the recipes to produce the final desired view. This view represents one of the many software artifacts mentioned in Section 1.

There are several varieties of named chunks. In fact, we have a hierarchy of chunk types where each new chunk encapsulates more than the previous one(s) (see Figure 1 for an idea, the labels in parentheses are added knowledge). The most basic *chunk* represents a named piece of information.

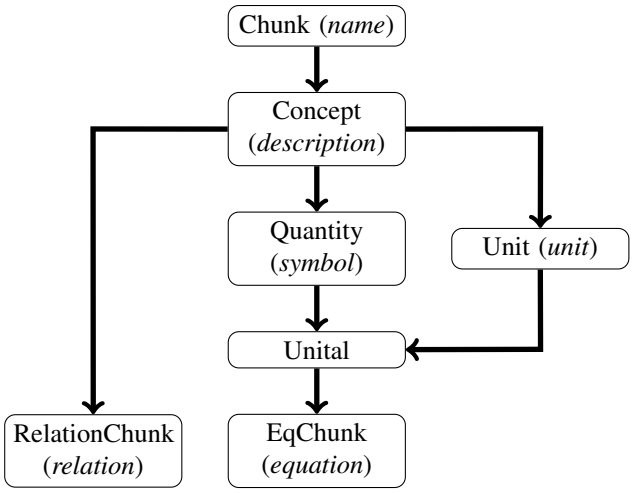

Figure 1. Our current chunk design (parentheses indicate newly added knowledge)

Above that we have named *concepts* which introduce some descriptive information.

A *quantity* is a concept that has a symbolic representation (we can refer to it by either its name or symbol). In a similar vein are *units*, and *relation chunks* (which add the idea of a relation between some other pieces of knowledge).

In an SC context, most of the knowledge we work with is represented as a quantity with some units, in other words a *unital* chunk. Expanding on that, we can actually calculate values for many of these unital concepts. As such, we have *equation chunks (EqChunks)* which allow us to capture the equation along with everything else included in a unital chunk.

Now with the means to encapsulate knowledge, we can turn our attention to our recipes. The recipes are implemented using a combination of embedded Domain Specific Languages (DSLs) in Haskell. We have currently implemented the following DSLs:

1. Expression language – A simple expression language that allows us 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. LaTeXRepresentation Language – A DSL for representing LaTeXcode inside of Drasil. As with the C representation, it is used by the generator to produce working LaTeXcode.
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.

Number	**DD2**
Label	h_c
Units	$ML^0t^{-3}T^{-1}$
SI Units	$\frac{\text{kW}}{\text{m}^{2\circ}\text{C}}$
Equation	$h_c = \frac{2k_c h_b}{2k_c + \tau_c h_b}$
Description	h_c is the 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 2. Data definition for h_c from the fuel pin SRS

Of the DSLs mentioned, we actually only require three of them to write our recipes. Each of the representation languages are used strictly by the generator as an intermediary in the production of our desired views. We write our recipes using the document layout language, expression layout language, and expression language. We will discuss the particulars of each of these in more depth during our example (Section 3.2).

The last piece of the puzzle is the generator. We use it to interpret the recipes, create intermediary representations of our desired views, and then pretty-print them.

3.2 Drasil in action

For this section we will take a look at a simplified version of a Software Requirements Specification (SRS) for a fuel pin in a nuclear reactor (for more information on that particular SRS see (Smith and Koothoor 2016)).

Starting off we will look at one specific term: h_c. In this example, h_c represents the convective heat transfer coefficient between the clad and coolant. The data definition for h_c from the original SRS can be seen in Figure 2.

The data definition of h_c displays some interesting knowledge. It gives us the name for a concept, its description, a symbol to use for easier reference, its (SI) units, and its defining equation. Encapsulating all of this knowledge into a chunk is not very difficult as shown in Figure 3. Note that the equation for h_c is written in our expression language (Expr) and it also includes references to chunks that h_c depends on.

```
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 3. The h_c chunk in Drasil

These will come into play when we create the description for h_c in the generated view, but we do not concern ourselves with them now as their knowledge is stored elsewhere. The units for h_c are defined by a piece of common knowledge known as `heat_transfer`.

On the topic of common knowledge, we can show an example alluded to previously, that of the SI unit library. The seven base SI units captured in Drasil are shown in Figure 4. The SI unit library also contains several derived units (for example: centigrade which is derived from kelvin) which we make use of later on in our example. The knowledge behind each unit and (in the case of derived units) their relation to the base units is captured in the relevant chunks.

Now that we've shown a bit of knowledge capture, you may be wondering how to use it. This is where the recipes come in. Figure 5 shows a portion of the recipe for creating our intended SRS view. The body of the simplified SRS is composed of three sections: `s1`, `s2`, and `s3`. We then show the definition of the first section (omitting the others for brevity) which is titled "Table of Units" and includes an introductory paragraph and a table. This table simply extracts the symbol and description information from the SI units to display them in our view. It should be fairly obvious at this point, but these macro-scale layout objects (sections, tables, paragraphs, etc.) are specified using our document layout language.

With our recipe in place, we are now able to run it through the generator and see what it spits out. We've included the three sections of this simplified SRS in Appendix A, and what you see there is the typeset version of the generated LaTeXcode. We are also able to output our view as an HTML document (Figure 6).

Now if you recall the data definition from Figure 2, we will show the painstaking amount of work it takes to create (almost) that exact same table:

```
s3_dd2 = Definition (Data h_c)
```

Again looking to the appendix, or to the HTML output (Figure 7 for the data definition), it should be easy to see that the recipe handled all of the layout details. The recipe actually goes one step further in that it found all of the chunk's dependencies and built the description by finding the appropriate knowledge related to the other chunks. This is configurable in that we can include short descriptions or verbose descriptions at this point in time.

One last thing we'd like to show is the source code generation. The recipe is fairly straightforward in that we have:

```
cSample = CodeBlock (toCode CLang Calc h_c)
```

All that says is `cSample` is a block of code in the C language that calculates the value of h_c. Currently the only implemented output language is C, but we are planning to implement more and are designing around that coming change. The generator output for `codesample` ends up looking like:

```
double calc_h_c(double k_c, double h_b, double tau_c){
    return (2 * k_c * h_b / (2 * k_c + tau_c * h_b));
}
```

This is a fairly trivial piece of code, but it is a good example of how the source can be generated from the high-level knowledge encapsulated in a chunk.

3.3 Advantages

We have already experienced some of the advantages of using Drasil. During the knowledge capture phase of our example a seemingly trivial problem came up regarding the simplified SRS. One of the concepts was being referred to with multiple different descriptions. By capturing only the right description, the generator now ensures internal consistency.

With that in mind, look at all of the previous figures (or Appendix A) and how many times knowledge has been copied into different places within one software artifact. Now imagine how many times that knowledge appears across all software artifacts. By using a generator, we avoid the problem of manually copying all of that information. It also ensures that should a piece of knowledge need to be updated, those updates will propagate throughout all of the artifacts *automatically*.

Another great benefit to knowledge capture, especially in the SC domain, is that it promotes reuse. The SI units are a trivial example of a completely reusable knowledge library that will come up across many different problem domains (within and outside of SC). If we can build these common knowledge libraries for a multitude of problem domains, we believe it will aid in the creation of higher quality software overall and will allow developers and scientists to spend their time on more important things.

Speaking of more important things, what about software certification? Many types of safety- or security-critical software must be certified. Depending on the certification process and regulatory body that means including different types of high-quality documents along with the source code. As we all know, requirements or algorithm choices can and do change throughout the software development process leading to a need for updated documentation. With Drasil we are able to ensure all of the software artifacts always remain consistent both internally and with each other.

```
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 4. The seven funamental 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 5. A portion of our simplified SRS recipe

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 6. Section 1 of the generated SRS (HTML version)

Label	DD: h_c
Units	$°C^{-1}s^{-3}kg$
Equation	$h_c = \frac{2h_c h_b}{2h_c + \tau_c h_b}$
Description	h_c is the convective heat transfer coefficient between clad and coolant h_c is the clad conductivity h_b is the initial coolant film conductance τ_c is the clad thickness

Figure 7. Generated SRS data definition (HTML version)

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%

Drasil is meant to design for change. We mentioned previously (Section 1) that we intend to make working on program families trivial, and Drasil does just that. Implementing a likely change is as simple as adjusting a configuration file. If the knowledge has been properly captured, any likely changes will involve trivial modifications. In the event of an unforeseen change, modifying the recipe or (worse) the captured knowledge is still possible and should end up simplifying the development process.

Finally, we need to verify our software and Drasil can help with that as well. We can include so-called "sanity" checks in the knowledge we capture that can be reused throughout the entire development process. A simple example of these types of checks can be seen in the `Constraints` column of Table 1.

If we encapsulate the knowledge regarding thse constraints into the appropriate chunks, we can have Drasil automatically test them for us. For example: a chunk that contains the dimensions of a physical object should also constrain those dimensions to positive values.

Another need in verification is traceability. With Drasil we have introduced complete traceability: every piece of knowledge can be tracked (through all software artifacts) to its source! We can also check that all chunks are actually necessary, since the recipes can automatically report which chunks they use.

With all knowledge coming from unique chunks, we get one last added benefit (which can also be seen as a disadvan-

tage): pervasive bugs. Any error in the knowledge base will spread through every artifact and even a single bug could break everything. However, we still see this as an advantage since the bugs tend to be fairly shallow, easy to fix, and now much harder to miss.

3.4 Disadvantages

We mentioned one disadvantage in Section 1: the inability to create a local hack to get things working. With Drasil we have to take an all-or-nothing approach. This can, however, be viewed as an advantage: it prevents the developer from adding undocumented knowledge to the project. Any hacks added to the generated artifact sources will be overwritten the next time anything is updated.

With an all-or-nothing approach we also remove the ability to iterate in a meaningful way. We *must* do everything right (or extremely close to it) the first time. Yes, we can still modify the program family member that we are generating and adapt it as necessary, but adding new knowledge becomes a much more involved process.

Adding new common knowledge libraries is also non-trivially difficult. These must be created by (or with the help of) domain experts to ensure the right knowledge is being captured in the right way.

4. Future work

Drasil is still in its early stages and currently is only producing one type of code (in the C language) and one type of document (the SRS). However, we have been greatly inspired and will continue to expand Drasil's implementation. Currently, we plan to:

1. Generate code in more languages – The next planned programming language implementation is MATLAB, however, we plan to implement a variety of languages as potential outputs.
2. Generate more artifact types – We would like to see not only requirements and code, but also test cases, design documents, build instructions, user manuals and more! To do this we will be implementing new "default" recipes.
3. Generate different document views – We are currently able to generate the SRS with verbose or simplified data descriptions, but we would like to add a lot more configurability to the view. For example: being able to have a view of the requirements with full derivations of all equations from their sources vs. having requirements that simply state the necessary equations to solve. Each view is intended for a different audience.
4. More types of information in chunks – See Table 1 for some ideas. We want to ensure we can encapsulate anything that comes our way.
5. Use constraints to generate test cases – Typical values are considered "reasonable" values for realistic scenarios (if the user chooses values outside of the "reasonable" range, warn them but do not stop them) whereas physical constraints can be seen as hard limits on values (ex. density must always be positive, otherwise throw an error).
6. Implement much larger examples – We want to prove that Drasil can be scaled up to usable examples and not just toy problems. At the time of this writing, a larger example is currently being implemented. However, it is not complete enough to show off (yet).
7. External syntax – The current implementation of Drasil requires at least a basic understanding of Haskell syntax on top of each of the DSL's syntax. We would like to lower the barrier of entry by creating a singular external syntax to encompass all aspects of Drasil.

5. Concluding Remarks

Many ideas are out there for improving the quality of software. We have proposed a combination of these ideas which we believe will lead to vast improvements in application domains where knowledge can adequately be captured and is well understood.

With Drasil we hope to improve the qualities of traceability, maintainability, and verifiability (among others) by leveraging a higher short-term investment for long term gains. These gains can be most appreciated the longer a piece of software is expected to be in use, especially when dealing with (re-)certifiability and maintenance.

Drasil is already showing interesting results in its infancy. We hope to develop it into a framework that can be used across many different problem domains with many reusable common knowledge libraries.

A. Simplified SRS

Here we see the typeset output of the LaTeX code for the simplified SRS example:

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)

Table of Symbols

The table that follows summarizes the symbols used in this document along with their units. The choice of symbols was made with the goal of being consistent with the nuclear physics literature and that used in the FP manual. The SI units are listed in brackets following the definition of the symbol.

Symbol	Description	Units
h_g	effective heat transfer coefficient between clad and fuel surface	$^\circ C^{-1}s^{-3}kg$
h_c	convective heat transfer coefficient between clad and coolant	$^\circ C^{-1}s^{-3}kg$

Data Definitions

Refname	**DD:h.g**
Label	h_g
Units	$^\circ C^{-1}s^{-3}kg$
Equation	$h_g = \frac{2h_c h_p}{2h_c + \tau_c h_p}$
Description	h_g is the effective heat transfer coefficient between clad and fuel surface

Refname	**DD:h.c**
Label	h_c
Units	$^{\circ}\mathrm{C}^{-1}\mathrm{s}^{-3}\mathrm{kg}$
Equation	$h_c = \frac{2h_c h_b}{2h_c + \tau_c h_b}$
Description	h_c is the convective heat transfer coefficient between clad and coolant

References

K. S. Ackroyd, S. H. Kinder, G. R. Mant, M. C. Miller, C. A. Ramsdale, and P. C. Stephenson. Scientific software development at a research facility. *IEEE Software*, 25(4):44–51, July/August 2008.

S. A. Al-Maati and A. A. Boujarwah. Literate software development. *Journal of Computing Sciences in Colleges*, 18(2):278–289, 2002.

N. Amsel, Z. Ibrahim, A. Malik, and B. Tomlinson. Toward sustainable software engineering (nier track). In *Proceedings of the 33rd International Conference on Software Engineering*, pages 976–979. ACM, 2011.

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. ISBN 0-7695-2828-7. doi: http://dx.doi.org/10.1109/ICSE.2007.77.

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

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. ISSN 0740-7475. doi: http://dx.doi.org/10.1109/MCSE.2009.193.

P. Fritzson, J. Gunnarsson, and M. 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.

R. Gentleman and D. T. Lang. Statistical analyses and reproducible research. *Journal of Computational and Graphical Statistics*, 2012.

L. Hatton and A. Roberts. How accurate is scientific software? *IEEE Trans. Softw. Eng.*, 20(10):785–797, 1994. ISSN 0098-5589. doi: http://dx.doi.org/10.1109/32.328993.

M. S. Hyman. Literate c++. *COMP. LANG.*, 7(7):67–82, 1990.

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

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. URL `http://dl.acm.org/citation.cfm?id=2555523.2555555`.

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. doi: 10.1016/j.jss.2015.07.027. URL `http://dx.doi.org/10.1016/j.jss.2015.07.027`.

D. F. Kelly. A software chasm: Software engineering and scientific computing. *IEEE Softw.*, 24(6):120–119, 2007. ISSN 0740-7459. doi: http://dx.doi.org/10.1109/MS.2007.155.

D. E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, 1984. doi: 10.1093/comjnl/27.2.97. URL `http://comjnl.oxfordjournals.org/content/27/2/97.abstract`.

J. Kotula. Source code documentation: an engineering deliverable. In *tools*, page 505. IEEE, 2000.

Z. Merali. Computational science: ...error. *Nature*, 467:775–777, 2010.

H. D. Mills, R. C. Linger, and A. R. Hevner. Principles of information systems analysis and design. 1986.

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.

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

M. Patrick, J. Elderfield, R. O. Stutt, A. Rice, and C. A. Gilligan. Software testing in a scientific research group. 2015.

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

V. Pieterse, D. G. Kourie, and A. 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. URL `http://dl.acm.org/citation.cfm?id=1035053.1035054`.

N. Ramsey. Literate programming simplified. *IEEE software*, 11 (5):97, 1994.

P. J. Roache. *Verification and Validation in Computational Science and Engineering*. Hermosa Publishers, Albuquerque, New Mexico, 1998.

E. Schulte, D. Davison, T. Dye, C. Dominik, et al. A multi-language computing environment for literate programming and reproducible research. *Journal of Statistical Software*, 46(3):1–24, 2012.

J. Segal. When software engineers met research scientists: A case study. *Empirical Software Engineering*, 10(4):517–536, October 2005. ISSN 1382-3256. doi: 10.1007/s10664-005-3865-y. URL `http://dx.doi.org/10.1007/s10664-005-3865-y`.

S. Shum and C. Cook. Aops: an abstraction-oriented programming system for literate programming. *Software Engineering Journal*, 8(3):113–120, 1993.

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

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.

S. Smith, Y. Sun, and J. Carette. Comparing psychometrics software development between CRAN and other communities. Technical Report CAS-15-01-SS, McMaster University, January 2015. 43 pp.

S. Smith, Y. Sun, and J. Carette. Statistical software for psychology: Comparing development practices between CRAN and other communities. *Software Quality Journal*, Submitted December 2015. 33 pp.

W. S. Smith, N. Koothoor, and N. 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.

H. Thimbleby. Experiences of literate programmingusing cweb (a variant of knuth's web). *The Computer Journal*, 29(3):201–211, 1986.

G. 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. URL `http://www.americanscientist.org/issues/pub/wheres-the-real-bottleneck-in-scientific-computing`.