TITLE

DAN SZYMCZAK, McMaster University, Canada
JACQUES CARETTE and SPENCER SMITH

CONTEXT: Software (re-)certification requires the creation and maintenance of many different software artifacts. Manually creating and maintaining [and reusing? —SS] them is tedious and costly. [and error prone —SS]

OBJECTIVE: Improve software (re-)certification efforts by automating as much of the artifact creation process as possible, while maintaining full traceability within – and between – artifacts.

METHOD: Start by analyzing the artifacts themselves from several case studies to understand what (semantically) is being said in each. Capture the underlying knowledge and apply transformations to create each of the requisite artifacts through a generative approach.

RESULTS: Case studies – GlassBR to show capture and transformation. SWHS and NoPCM for reuse (Something about Kolmogorov complexity / MDL here?). Captured knowledge can be re-used across projects as it represents the "science". Maintenance involves updating the captured knowledge or transformations as necessary. Creation of our tool – Drasil – facilitates this automation process using a knowledge-based approach to Software Engineering. [Maybe add something about the infrastructure now being in place to reuse/grow the scientific and computing knowledge base to cover new case studies? It would be nice if we could make the connection between Drasil's knowledge and existing scientific knowledge ontologies [like which? —DS], but maybe it is too early for that connection? —SS]

CONCLUSIONS: With good tool support and a front-loaded time investment, we can automate the generation of software artifacts required for certification. (fill in later)?????

[The abstract doesn't have anything from Jacques "bottom up" viewpoint. Can you work in something about consistency by construction? —SS]

Additional Key Words and Phrases: ??

ACM Reference Format:
Dan Szymczak, Jacques Carette, and Spencer Smith. 2018. TITLE. *ACM Trans. Softw. Eng. Methodol.* 1, 1 (October 2018), 27 pages. https://doi.org/0000001.0000001

1 INTRODUCTION

Writing non-executable software artifacts (requirements and design documents, verification & validation plans, etc.) can be tedious work, but is ultimately necessary when attempting to certify software. Similarly, maintenance of these artifacts, as necessary for re-certification as improvements are made, typically requires a large time investment.

Why, in a world of software tools, do we continue to undertake these efforts manually? Literate programming had the right idea, but was too heavily focused on code.

We want to aid software (re-)certification efforts by automating as much of the artifact creation process as possible. By generating our software artifacts – including code – in the right way, we can implement changes much more quickly and easily for a modest up-front time investment. By

Authors' addresses: Dan Szymczak, McMaster University, 1280 Main St. W., Hamilton, ON, L8S 4K1, Canada, szymczdm@mcmaster.ca; Jacques Carette; Spencer Smith.

1049-331X/2018/10-ART $15.00
https://doi.org/0000001.0000001

front-loading the costs of maintenance and rolling them into the development cycle, we can save time and money in the long run.

[The above is true, but only after the investment in building the generator infrastructure and incorporating the required knowledge. Based on a discussion with JC, and some of his original text, I recently wrote a little blurb about this effort. You could probably modify this to work in your paper somewhere. "The proposed work can improve the quality of SCS and reduce the cost of its development, but this will require a transformative change in development practices. Using Drasil for an SCS project will require a willingness to encode all of the knowledge, both declarative and procedural, about the relevant topics inside Drasil. Although encoding this information is a significant undertaking, the payoff is potentially enormous – We get to generate fully-documented families of programs at an incrementally tiny cost. The large up-front investment has a large long-term payoff. Current software development, has a small up-front cost, and ever-increasing maintenance costs to continue. Drasil will completely change this effort curve." —SS]

[The bottom up justification should also be part of the introduction. —SS]

[bottom up justification can go here? —DS] [You mention a table below (Problems and Proposed Solutions). That would be the spot to discuss the bottom up justification. To make sure we are communicating about the same thing, when I say bottom up justification, I'm talking about consistency. —SS]

[Also note: Should point out maintainability/traceability/design for change somewhere in here, since it is relevant later on. —DS]

We [and others – citation ? —DS] have observed a number of issues arise throughout a standard software development and maintenance life-cycle. Information duplication is a major concern as several problems occur therein. The first, most trivial, and incredibly tedious is that of the necessity to duplicate information across software artifacts. A piece of knowledge that appears in the Software Requirements Specification (SRS) will undoubtedly appear in the design and implementation artifacts (documents/source code). This knowledge *may* be transformed, but in many cases is an exact copy (Ctrl-C & Ctrl-V anyone?) and a massive waste of time for a human to replicate throughout multiple artifacts.

Duplication also becomes a time-sink for maintenance, particularly as the duplicated information tends to fall out of sync across artifacts over the course of a piece of software's lifetime. For example, a common problem is that of updating the source code, but not updating any of the design documents to reflect the change. These inconsistencies are at best a minor headache and at worst, in the case of software certification, an expensive lesson. If the artifacts are inconsistent when submitted for (re-)certification, the software will not be (re-)certified and an extra cost will be incurred (both in labour, and from the certifying body) to attempt certification again.

[Fill in this part about consistency, variabilities, and commonalities —DS]

Table 1 gives a brief rundown of our proposed solutions to the problems mentioned above. We are leveraging our experience with generative programming to provide a solution that is applicable to a wide range of problems. [This blurb about the table isn't enough to explain its content. —SS]

[We can certainly wait to do it, but I like a "roadmap" paragraph at the end of the introduction. The roadmap explains how the paper is organized. More importantly, it provides an opportunity to present your "story," assuming that the organization of the paper follows the story. —SS]

1.1 Software (Re-)certification

When we talk about software certification, we are specifically discussing the goal of determining "based on the principles of science, engineering and measurement theory, whether an artifact satisfies accepted, well defined and measurable criteria" [17]. Essentially, we are ensuring that

Table 1. Problems and Proposed Solutions

Problem	Technical Solution
Duplication	Databases (global & system-specific) single-source & generation *
Consistency (incl. Source/ generated content)	Sanity checkers
Variability & Commonality	Software families & Generation

** Also provides free & easy reuse*

software, or some piece of it, performs a given task to within an acceptable standard and can potentially be reused in other systems.

Software certification is necessary by law in certain fields. This is particularly evident in safety-critical applications such as control systems for nuclear power plants or X-ray machines.

Different certifying bodies exist across domains and each has their own list of requirements to satisfy for certifying software. Looking at some examples [5, 7, 8, 46] there are many pieces of requisite documentation including, but not limited to:

- Problem definition
- Theory manual
- Requirements specification
- Design description
- Verification and Validation (V&V) report

We should keep in mind that we require full traceability – inter- and intra-artifact – of the knowledge contained within these artifacts. That is, we should be able to find an explicit link between our problem definition and theory manual, down to our requirements, design, and other development planning artifacts. From there, we should be able to continue through our proposed verification and validation plans, and should eventually end up in the V&V report. [Why do we require full traceability. I agree, but you should tell the reader why this documentation quality is so important. —SS]

[rework the following paragraph —DS] [I agree - you say it is expensive two different ways. You only need to say it once. :-) —SS] Ensuring this traceability and, in fact, getting anything certified has many costs associated with it. There is a massive time investment, fees, and costs associated with contracting out a third-party verifier. Overall it is a very expensive process.

Re-certification of software following any change, no matter how minor, incurs a similar level of costs; all the artifacts must be updated to reflect the new change, and everything must be re-checked and verified to ensure no new errors have been introduced. We have an implicit burden of ensuring the consistency of related information across our artifacts.

We intend to alleviate some of this cost-burden through a strategic, generative approach to Software Engineering (SE). With the automated generation of artifacts we can ensure they are *consistent by construction*, implement changes quickly, and automatically update relevant and/or dependent artifacts. [I was looking for consistent by construction here, and I found it. :-) —SS]

[Work the following paragraph in somewhere related to consistency. —DS] Consistent documentation has its own advantages while developing or maintaining software [18, 25]. Similarly, there are many downsides to inconsistent documentation [25, 45]. [I haven't looked at the papers you

cite, but that is great to have scholarly support for the importance of consistency. Rather than just cite, you could explicitly point out the advantages of consistency. —SS]

1.2 Scope

We limit our scope to Scientific Computing Software (SCS) for our proof of concept. The SCS domain is first and foremost a very well understood domain, by which we mean there are many theories underpinning the work being done. This is a necessity if we wish to be able to explain the science to a machine for the purposes of generating our software artifacts. It also gives us a foundation from which to build our knowledge-base.

SCS is also a domain rich in specialized software requiring certification. For example, control software in nuclear power plants, x-ray machines, and other safety critical applications require software to be approved by a certifying body. [control software isn't really SCS software - for the nuclear domain we are more focused on nuclear safety analysis software. —SS]

[More to come —DS]

[We definitely want a scope section. What do we do is important, but what we don't do is also important information to convey. We discussed this recently. Many people want us to jump to the more interesting/more challenging problems, but we recognize that there is much to be gained by first tackling the more mundane problems. We allow for many classes of practical changes to be made quickly and easily. This has real value. We can discuss in the future work section the more interesting ideas. —SS]

[My suggestion is to stop writing Section 1 for now and focus on Sections 5 and 6. Section 1 is a good start, but it feels like we need more current information on MDL. We might also want more information on scientific knowledge ontologies. —SS]

[In the scope section we should clarify that our generator will focus on the SRS and some code. We can hopefully find words to express that we are aiming for a big picture, and will discuss the big picture, but the proof of concept implementation has focused on the SRS. This has come up before. We are in the difficult situation of having to express the value of Drasil for generating all things, but also not implying that we are further along than we are. I'm not sure of the right words, but being up front in the scope section should help. —SS] [I'll start that (removing assumptions and trying not to imply things) discussion with something along the lines of "For the sake of transparency..." —DS]

2 BACKGROUND

Reducing the costs of (re-)certification efforts and automating the generation of software artifacts has been attempted in differing scopes by many others. We look to them for insight and in an attempt to combine the fruits of their labours into something more versatile.

In this section we will first look at the state of SC software development, as we have limited our scope to that domain, and identify practices we can learn from and/or potentially improve. We then explore previous efforts in automating software artifact generation, as well as mention other domains we have drawn inspirations from, such as Model Driven Design, scientific knowledge ontologies, and other areas of interest.

2.1 SC Software Development

In many instances of SC software development, the developers are scientists and tend to emphasize their science without necessarily following software development best practices [22]. These developers tend to prefer an agile [1, 4, 12, 41] or knowledge-acquisition driven process [21] instead. They consider process-heavy approaches too rigid and disagreeable [4].

Tool use and adoption is also a problem in the SC software development community, especially the use of version control software [47]. There is also a limited use of automated testing [34] and lack of understanding of what good software testing entails [30] with quality assurance having "a bad name among creative scientists and engineers" [39, p. 352].

Good science relies on replication and reuse, but we see a lack of reuse on the software side. For example, a survey [33] showed that of 81 different mesh generator packages, 52 generated triangular meshes. Of those 52, 37 used the same Delaunay triangulation algorithm. There is no reason for the exact same algorithm to be implemented 37 separate times when it could, and should, simply be reused.

Some SC software developers have used advanced techniques, like code generation, quite successfully. Examples that come to mind are FEniCS [29], FFT [23], Gaussian Elimination [3], and Spiral[37]. The focus of generation techniques, thus far, have been solely on a single software artifact: the source code. This is something we hope to improve upon.

2.2 Software Artifact Reuse and Generation

Previous attempts at generating software artifacts were primarily focused on reusability or reproducibility. One aim of these approaches was to remove the burden of replicating some artifacts as, in many cases, replication can become impossible without the help of the original author. Typically this is due to undocumented assumptions, modifications to the finished product, or errors in the original work [19].

Here we discuss a number of these attempts at improving reusability, however, it should be noted that none of these approaches included non-trivial, cross-project reuse mechanisms.

2.2.1 Reproducible Research. The term *reproducible research* means embedding executable code in research papers to allow readers to reproduce the results described [40]. However, combining research reports with relevant code and data is not easy, particularly when dealing with publication versions of an author's work, thus the idea of *compendia* were introduced [16].

Compendia provide a means of encapsulating the full scope of a work. They allow readers to see computational details and re-run computations performed by the original author. While Gentleman and Lang intended compendia to be used for peer review in scientific journals, we can also see their use in the realm of software certification. Any requisite artifacts for getting software certified could together be considered a type of compendium.

Alongside compendia, several other tools have been created for reproducible research. Examples include Sweave [26], SASweave [28], Statweave [27], Scribble [14], and Org-mode [40]. These tools maintain a focus on specific computational details and code in general. Sweave (the most popular of the aforementioned examples [40]) allows for embedding code into a document to be run during typesetting so up-to-date results are always included. The majority of these tools aim to create a singular, linear document like a research report.

2.2.2 Literate Programming. Introduced by Knuth, Literate Programming (LP) changes the focus from writing programs as a list of instructions to explaining (*to humans*) what we want the computer to do [24].

Developing literate programs involves breaking algorithms down into *chunks* [20] or *sections* [24] which are small and easily understandable. These chunks are organized into a "psychological order" [36] to promote understanding. One key aspect of LP is that chunks do not have to be written in the order necessary for computation, as that may not be the most understandable.

It should also be noted that in a literate program, the code and documentation are kept together in one source. Extracting working source code is done through the *tangle* process. Similarly, *weave* is used to extract and typeset the documentation.

Beyond understandability, LP has some key advantages over traditional development. The intent of LP is to update documentation surrounding a piece of source code during development and maintenance. There is also some reduction in knowledge duplication through chunking (the creation and reuse of chunks). Adopting proper usage of LP ends up with more consistent documentation and code [43]. Keeping in mind the benefits of artifact consistency we can see that more effective and maintainable code can be produced when using LP [36].

Due to some issues, namely language/text processor dependency and the lack of flexibility on output presentation/suppression, LP has not been very popular [43]. Still, there are several successful examples of literate programs in SC. Two such examples are VNODE-LP [32] and "Physically Based Rendering: From Theory to Implementation" [35]. The latter being a textbook that can be run as a literate program.

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. While these tools did not bring LP into the mainstream [38], 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). R has good tool support, with the most popular being the aforementioned Sweave [26], however it is mainly used for dynamically generating reports.

New tools led to the introduction of many new features for LP including, but not limited to, a "What You See Is What You Get" (WYSIWYG) editor [15], phantom abstracting [43], and even movement away from the "one source" idea [44]. Tools such as Haddock (for Haskell), 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.

As a final note, LP does not overly simplify the software development process since documentation and code must be written as usual, barring chunk reuse, but with the additional effort of re-ordering chunks.

2.2.3 Literate Software. A combination of LP and Box Structure [31] was proposed as a new method called "Literate Software Development" (LSD) [2].

Box structure can be summarized as using different abstractions, or views, that communicate the same information in differing levels of detail, for distinctive purposes. Box structures consist of black box, state machine, and clear boxes. 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 [31]. These three structures are nested 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 designs created by box structures; and overcome the lack of tool support for box structure [10].

"WebBox" is a framework for LSD which expands LP and box structures in a variety of ways. It includes new chunk types; chunk refinements; functionality for specifying interfaces and communication between boxes; and the ability to decompose boxes at any level. LSD remains primarily code-focused with little support for other software artifacts.

2.3 Generative Software Development (GSD)

See [9] for a detailed overview.

2.4 Knowledge-Based Software Engineering (KBSE)

Knowledge-Based Software Engineering (KBSE) was originally defined as an "engineering discipline that includes the integration of knowledge into software systems in order to solve complex problems, which would normally require rather high level of human expertise" [13]. This is an apt definition, provided we understand what "knowledge" is. So then, what exactly is knowledge?

Knowledge "presents understanding of a subject area. It includes concepts and facts ... as well as relations ... and mechanisms for how to combine them to solve problems in that area" [11].

[Need to write about how KBSE uses formal methods, particularly for defining the semantics and whatnot for software synthesis. Good stuff in [?] and [?]. —DS]

[This probably will have to move —DS] With that in mind, we have decided to restrict our focus to KBSE for Scientific Computing Software (SCS) as it is a field rich in knowledge we can use.

2.5 Other Influences

There have been many other great ideas and approaches, sadly to cover all of them would not be feasible. Instead, we have chosen a subset and provide a brief overview of each and how their influence has helped us in developing Drasil.

2.5.1 Model Driven Design (MDD). [intro —DS] - Lots of overlap with Generative SFWR Dev (GSD) - Ultimately not our areas of expertise, so while we've kept it on our radars, we have not gone into extensive depth, but there is a lot of good inspiration there that also overlaps with GSD

2.5.2 Grounded Theory. [Really need an intro to the basics of grounded theory here, for section 3.1 (S:IntroCases) and subsequently 4.2 (S:KReUse) if we keep the references to grounded theory —DS]

Had we come across Grounded Theory earlier in our, we would have liked to take a formal grounded theory approach to Drasil's development. As it stands, we can only use it as inspiration to help us analyze and improve upon our work.

3 A RATIONAL ANALYSIS OF SOFTWARE ARTIFACTS

To generate all software artifacts, we must first take stock of how they are composed. We have chosen to focus on the Smith Et Al. templates [**?**] [grab proper citation later —DS]. We made this choice because we have access to a number of case studies which were implemented using the given templates and thus give us a reasonable starting point for our analyses.

3.1 Introducing our case studies

To understand exactly what we are looking at in our software artifacts, we will now introduce the case studies that have driven the development of the Drasil framework. [Should we include links to the appropriate repos? —SS]

[I think it would be helpful to give some of the history of the case studies. They come from SmithJegatheesanAndKelly2016. We redeveloped existing software using a modern software engineering process and documentation. The case studies are manually written; Drasil translates them so that the documentation and code can be generated. By the way, SmithJegatheesanAndKelly2016 can be used to motivate Drasil. I wrote the following in a recent grant proposal: "A recent study by the applicant further highlighted the value of documentation. Five existing SCS projects were redeveloped, with an emphasis on documentation and software engineering best practices [?].

Interviews with the code owners showed agreement that a systematic development process can be beneficial, and they had a positive or neutral response to the software artifacts produced during redevelopment. However, mirroring the comments in other studies [4, 42], the code owners felt that documentation requires too great a time commitment and too much up front effort. To reduce the effort required for documentation, work has begun on a prototype tool called Drasil [?]." —SS]

The first case study we will explore is *GlassBR* – software created for checking whether a given pane of glass will be able to withstand the force of a particular explosion. It has a number of glass and explosion-related parameters which are used to compute the probability of glass breakage. This will be our running example throughout this paper, with minor asides to demonstrate particular features or curiosities found in the other case studies.

[Figure of the GlassBR system for context —DS]

The next two case studies – *SWHS* and *NoPCM* – are members of the same software family. They both model a solar water heating system with one non-trivial difference. SWHS incorporates Phase Change Material (PCM) which allows the system to store more thermal energy than one without PCM such as that modelled in NoPCM.

Our fourth case study, *SSP*, is for modelling a slope stability problem intended to analyze whether a given slope (natural or excavated) can be considered safe. The software identifies the surface most likely to experience slip and gives its factor of safety – an index of its relative stability.

Finally we have *Gamephysics* which is a pared-down version of the Chipmunk2D game physics engine (https://chipmunk-physics.net/). We chose this project specifically for its rich use of physics knowledge, but as it is a very large physics library, we narrowed the scope for our proof of concept in the interest of time.

There is also a trivial case study, affectionately nicknamed *tiny*, which is taken from an example of a fuel-pin inside a nuclear reactor. This toy example helped push the early development of Drasil, but has recently been primarily of use as an additional test-case.

With our case studies in hand we must look for the commonalities between them, particularly those between the software artifacts in each project and what they *mean*. With that in mind we can extrapolate these commonalities to other pieces of SCS. One strikingly obvious commonality across many projects in SCS is that of the use of Système International (SI) and derived units.

3.2 Common software artifacts

As our case studies appear to follow a rational design process, the same software artifacts can be found within each. That is to say, each project includes:

- Software Requirements Specification (SRS)
- Detailed design documents such as:
  - Module Guide (MG)
  - Module Interface Specification (MIS)
- Source code
- Test cases
- User Guide
- Verification and Validation (V&V) Plan
- V&V Report

Let us first look at the similarities and differences found in the SRS, the detailed design documents, and the source code. The specific system knowledge found in the SRS is duplicated throughout the design and source code, though not always verbatim. The knowledge may be transformed to produce different views relevant to the current document.

Fig. 1. SRS & MG showing the same piece of knowledge [name it —DS] (from GlassBR) in different contexts

Fig. 2. Data Definition for !FIXME: Load or something?! from GlassBR SRS

Fig. 3. Data Definition code from GlassBR implementation

Though the knowledge is duplicated, it may only be in part as only certain aspects of a piece of knowledge are necessarily relevant in each of the artifacts. We can see an example of this in Figure 1 where we see the same knowledge from GlassBR's SRS and MG. [or is it MIS? Double-check. —DS]

The SRS displays more knowledge related to the abstract theory, while the [MG —DS] provides a view meant to transition from the abstract to something far more concrete. Both of these *mean* the same thing in the context of our case study, but they are meant to display what is most relevant to the audience of that particular artifact.

[In fact, we may know early on (perhaps during requirements gathering, even before the SRS is complete) that we will – whatever implementation detail is in the figure –, but that is not relevant until we begin to plan and discuss the design. —DS]

[Here you are talking about reuse of knowledge between artifacts in the same problem domain. Is there a place where you talk about reuse between the same artifact, across the case studies? You alluded to this with the discussion of the SI section. The reference section was parameterized so that it would fit with all of the examples. This might be a good example to include in this paper? —SS]

3.3 Emerging structures

As shown in the common software artifacts, we see different ways of representing what are, semantically, the same things (for example, see Figure 1). We are really seeing the pieces of underlying knowledge that have been composed from a variety of components. Each component tells us something about one aspect of that piece of knowledge. Particularly, they give examples of how we can transform, or view, the same semantic knowledge in different contexts.

If we take a look at one particular example [name it —DS] across artifacts from GlassBR (Figures 2,3), we can see that it is an aggregation of the following components:

- Unique Identifier (label) – [id —DS]
- Symbolic (theory) representation – [symb —DS]
- Symbolic (implementation) representation – [csymb —DS]
- Concise natural language description (a term) – ["load or something" —DS]
- Verbose natural language description (a definition) – ["definition of load or something" —DS]
- Equation – [? = ? ? —DS]
- Constraints – [? ≥ 0? (Really relevant. Again show as math and code) —DS]
- Units – [id (If applicable in the example, otherwise unitless) —DS]

[I like where you are going with this, but it is hard to review without the "blanks" being filled in. —SS]

As shown above, the unique identifier is fairly straightforward (!FIXME id!), it is just a label that we associate with this particular piece of knowledge and nothing else. The symbolic representations are just the symbols we use when referring to this particular quantity in an equation (theory) or code (implementation) context. Our natural language descriptions are terms and their

corresponding definitions [remove this if it's above, or leave it in for readability? – (!FIXME! and !FIXME! respectively for this example). —DS]

We also have a defining equation, which incorporates the symbolic representation for various other pieces of knowledge and relates them to !FIXME load or something!. Similarly, we have constraints which are just relationships which must be maintained between !NAME! and some other quantities. Lastly, we have the units which our quantity is measured in, which are derived from the fundamental !SI UNITS!.

Similar examples of knowledge crop up over all the artifacts. Some have the same depth of information, whereas others do not. Regardless, all of our knowledge shares some components in common. We will always have a label, and usually a term and definition. Depending on what we are looking at, there may not be a symbolic representation, or perhaps we have a quantity that is unitless. These special cases help us see the underlying root structure from which our knowledge buds.

[Going through the GlassBR example like this seems like a good motivating example to me. I'm assuming that you will also be able to reuse it later when you discuss the advantages of our approach for consistency and maintainability (with respect to change). —SS]

After comparing knowledge across case studies and analyzing patterns that emerged, we have created a number of "knowledge classes" to categorize and differentiate our various kinds of knowledge. If you refer to Table 2 you can see the breakdown of each class. Items marked with an *X* are necessary for a piece of knowledge to be considered part of a knowledge class, *O* is something we have observed in a number of distinct cases, and anything else is optional. For example, a *Named Idea* must have both a unique identifier and a term (in some cases, these are identical!), but a Named Idea may, in many cases, also have a commonly understood abbreviation. However, there are cases where a piece of knowledge must have an abbreviation, in which case our *Named Idea* would also be considered a *Common Idea*.

[This table looks like a good way to summarize this information to me. —SS]

Certain things stand out in this table. How can a *Named Idea* have a term but not necessarily be defined? This is not an error, but a reflection of the knowledge maintained within the case studies. Many of them rely on additional knowledge from external sources and use commonly understood terminology from their domain to avoid lengthy or tedious definitions. There is also a lot of overlap between classes, some only add one new mandatory piece of information. However, by analyzing where and how these types of knowledge are used, we see they do, in fact, *mean* something different than the existing classes.

It should also be noted that knowledge in those classes with additional mandatory fields are less restrictive in their use. We see *Quantities* used alongside *Named Ideas* in a number of places throughout our case study artifacts, but they also appear in *Quantity*-specific areas. For example, a table of symbols can only display knowledge that falls into the *Quantity* class, as that is the minimal complete class with a mandatory symbol.

4 OUR SOLUTION – A COMBINED APPROACH

With inspiration from similar problem domains, as mentioned in Section 2, we combine a number of ideas to tackle the problems of duplication, inter-/intra-artifact inconsistency, design for change [this is the first time you have mentioned design for change - I think this is an important advantage of our approach. We should highlight it sooner. It is also related to why you want traceability —SS], lack of reusability, and difficulty with (re-)certifiability.

For our purposes, we extend and tighten the definition of knowledge introduced in Section 2.4 to include the additional constraint that a piece of knowledge has a structured encoding, as opposed to natural language encoding, which then allows it to be automatically reused. For example, the

[Figure out how to landscape this table if necessary, it's probably going to be big —DS]

Table 2. Knowledge Classes

Class	ID	Term	Abbrev.	Defined	Domain	Symbol	Units	Equation	Constraints
Labeled	X								
Named Idea	X	X	O						
Common Idea	X	X	X						
Concept	X	X	O	X	X				
Quantity	X	X	O	O	X	X			
Unitary	X	X	O	O	X	X	X		
Defined by Expression	X	X	O	O	X	X	O	X	O
Constrained	X	X	O	O	X	X	O	O	X

X → Mandatory; O → Commonly observed; all others optional

[Why aren't Unital/Unitary or constrained classes in Drasil? —DS]

first law of thermodynamics is a piece of knowledge that can be simply expressed as "total energy within a closed system must be conserved", but this is not a structured encoding. One such encoding would allow us to view the knowledge in those relatively simple terms, or just as easily, we could view it as:

$$\Delta U = Q - W$$

where we define a *closed system* as one which cannot exchange matter with its surroundings, but energy can be transferred (we express this as a constraint). ΔU is then the change in internal energy of a closed system, Q is the amount of energy supplied to the system, and W is the amount of energy lost to the system's surroundings as a result of work.

Regardless of our view, we have not changed the underlying structured knowledge encoding – we merely project out what is relevant to our current audience.

Our approach involves using this underlying idea in the creation and maintenance of a singular knowledge-base from which we can pull to implement our software. With knowledge coming from a single source, we have guaranteed consistency. We are able to mix and match knowledge as needed, which allows for greater reuse across projects, and we can use generators to produce the software artifacts we need.

For our approach to succeed we must satisfy two major requirements. First off, we must capture the underlying knowledge in a meaningful and reusable (artifact-independent) way. We want a single source for our knowledge, regardless where it ends up or how it is used. Thus, using the right transformations and/or projections, we can automatically generate our software artifacts from the knowledge-base.

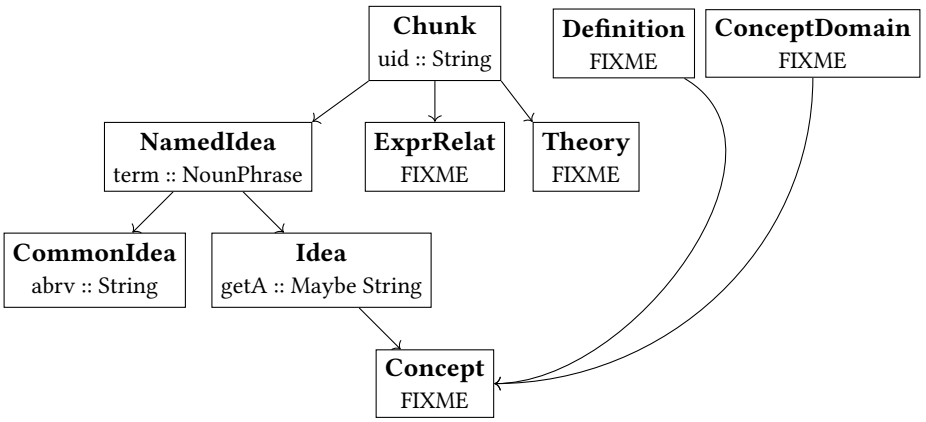

Fig. 4. Chunk hierarchy in Drasil Today

The second requirement is that we restrict our scope to well-understood domains since we need a solid theoretical underpinning of knowledge. Both mathematics and the physical sciences are good examples of well-understood domains as the knowledge has already been formalized and, to an extent, structured. These are also good candidate domains since we need to explain the underlying knowledge to computers in a nontrivial way, which from our experience is harder than it sounds. Hence the reduction in scope we mentioned in Section 1.2

4.1 Capturing Knowledge

From our work in Section 3.3 we can create a knowledge-capture mechanism for encoding the requisite underlying science into a machine-usable form. By laying out the structure, we can see which information must be captured for each piece of knowledge. Recall Table 2 where we classified our different types of knowledge. This is the start of our knowledge-capture structure.

Different types of information are required for encoding each of the various pieces of knowledge we intend to use. Some types of knowledge lack specific information bindings, for example a *Named Idea* does not necessarily have a symbol associated with it, however, a *Quantity must* have a symbol alongside its *term* – the fundamental information in a named idea.

To facilitate our knowledge-capture we borrow, and expand, the idea of *Chunks* from Literate Programming (LP) [24]. A chunk in its most rudimentary sense is simply a labeled piece of information. Given our understanding of how the knowledge should be structured, we have created a hierarchy of classes built up from the simplest of chunks, to fulfill our knowledge-capture requirements. This hierarchy as implemented in Drasil can be seen in Figure 4. It mimics the structure mentioned in Section 3.3. We will delve deeper into the specifics of our hierarchy in Section 5.

When we capture knowledge, we try to encode all of the information surrounding that piece of knowledge in an artifact-agnostic manner. We are not concerned with which views will be used by our artifacts, only what the underlying knowledge is and how it should be captured. [Great point. –SS]

Provided we have properly captured the relevant knowledge, we should not have to capture it again to reuse it in a different project. Any given piece of knowledge should only be added to the knowledge-base once! This follows from our intuition that different representations of knowledge are not expressing different ideas, only saying them in different ways. Recall the first law of thermodynamics example from earlier, once captured we do not need to re-capture that law

Fig. 5. Similarity between NoPCM and SWHS from SRS

Fig. 6. Difference between NoPCM and SWHS from SRS

regardless of how we wish to view it, we only need apply the appropriate transformation. [It would be great if we could show an example for this, maybe from SWHS and NoPCM? It looks like you are planning on doing this below. —SS] [Something like that, yes. I'd like to show off (particularly when dealing with Kolmogorov complexity) that NoPCM is using a lot of things from SWHS —DS]

4.2 (Re-)Using Knowledge

Capturing knowledge alone helps us improve our understanding of the underlying theory by laying things out in a structured way. That is a benefit in itself, however, when we can actually use the captured knowledge we see many advantages to this approach.

The most obvious perk is that we no longer need to manually copy knowledge across artifacts, we can simply pull what we need from our knowledge-base. While this seems trivial, the ramifications are huge – we have guaranteed *consistency by construction*.

At this point you may be wondering, "what if I want to do more than just copy information around?" Recall the example from the beginning of Section 2.4, the view of our knowledge can change without affecting our encoding. To project these views, we use transformations.

Transformations represent the different 'views' of the knowledge we want based on how abstract they need to be, what audience we are targeting, and potentially a host of other factors. We use transformations to translate knowledge into its requisite forms, whether they be equations, descriptions, code, or something else entirely. It should be noted that we view transformations as unidirectional – we cannot necessarily recover the knowledge as it exists in our knowledge-base from a given view as it is often the case a transformation will be lossy. However, that does not mean it is impossible, as given a multitude of views of the same piece of knowledge, we could potentially reconstruct it. Regardless, that should not be necessary provided the knowledge-base is well-maintained.

Transformations can also be used to expose variabilities. These are what define project families – projects which solve the same general problem, but with differences in the specific goals and/or implementations of those solutions.

For example, our case studies (introduced in Section 3.1) for SWHS and NoPCM are members of the same *software family* as they solve the same general problem with a variation on whether phase-change material is present in the system. A correct solution for each problem will look different, but there is a non-trivial amount of overlap in the fundamental knowledge being shared by both solutions.

Figure 5 shows similarities from the NoPCM and SWHS case studies. Note that they are identical with regard to [TODO: Fill this is —DS]. On the other hand, Figure 6 shows the key difference between NoPCM and SWHS in the form of the [equation to be solved -?- —DS]. It is fairly obvious that NoPCM uses the same equation seen in SWHS, only simplified to remove the effects of PCM.

Manually transforming knowledge in this way is tedious and would likely not end up cutting costs or saving time. If, on the other hand, we had a framework or tool to support the automation of these transformations for our software artifacts, those particular disadvantages disappear.

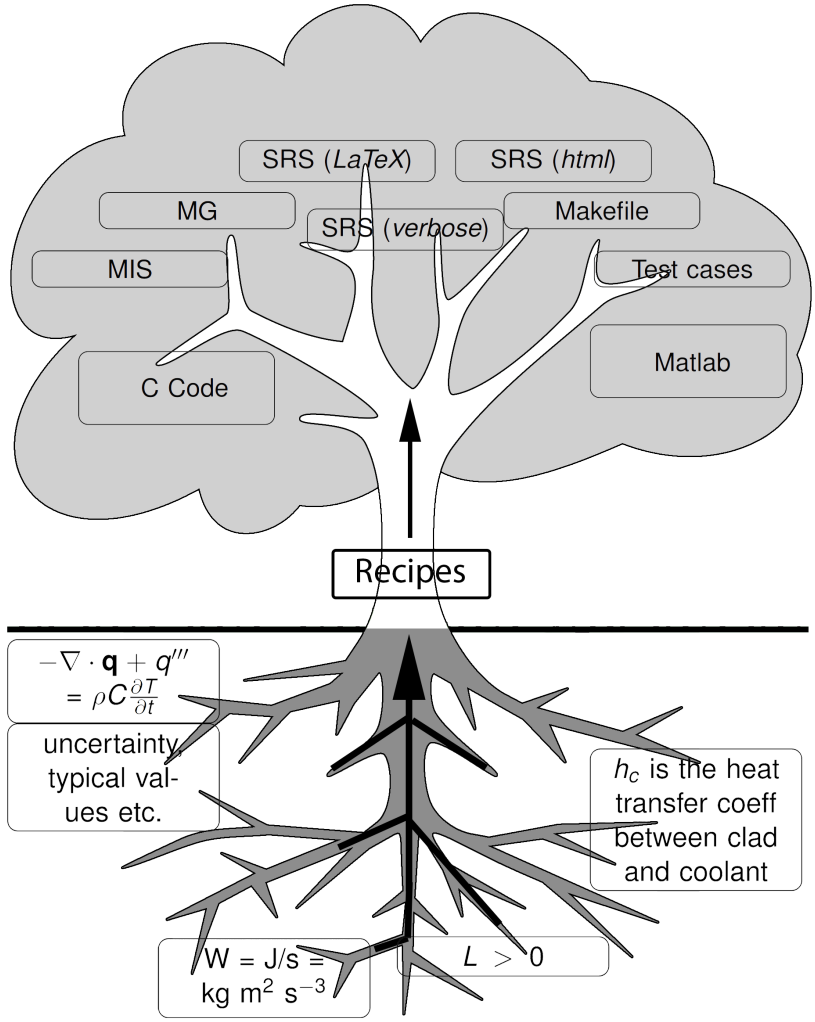

Fig. 7. Drasil concept

5 DRASIL

Drasil is a framework created to support our combined approach from Section 4. Figure 7 gives a conceptual look at what Drasil should do. The tree is our generation framework. The leaves are our generated artifacts. For each project, we can have many spring from a single source. The trunk is our *recipe*, it determines which of those artifacts we want to generate and what they should describe. Its roots extend into the knowledge-base, pulling out the relevant knowledge for a given project.

To clarify, Drasil is a framework for capturing knowledge and generating software from that knowledge. Our aim with Drasil is to generate *all the things* – i.e. all software artifacts including, but not limited to, the SRS, MG, MIS, test cases, build instructions, and source code – from a single knowledge-base. To that end we have leveraged our experience with generative programming to

```
dispEqn :: Expr
dispEqn = deriv (apply1 QP.position QP.time) QP.time
```

Fig. 8. Linear displacement equation $\mathbf{r}(t) = \frac{d\,\mathbf{p}(t)}{d\,t}$ in Drasil

create a number of Domain-Specific Languages (DSLs) to help achieve our goals. In the simplest terms, we use a DSL for knowledge-capture, document layout, printing, code-generation, and more. These will be discussed in more depth in Section 5.1.

Drasil's development has followed a cyclic approach of analysis and abstraction to reach its current state. We use our case studies as a guide, treating them like data, to search for commonalities. Once a commonality is found, we abstract it out and repeat the process. This approach allows for rapid progress due to constant iteration, while also helping us to determine what is "meaningful" knowledge in the case studies as opposed to boilerplate information. This process of analysis and abstraction is what led us to our first real recipe for generating documents using the SmithEtAl template: `Drasil.DocumentLanguage`. Again, we will discuss that in more detail in the coming sections.

One last thing to note regarding Drasil is our liberal use of Haddock. The vast majority of Drasil is documented within the comments of the source code and this documentation can be extracted by running `make docs` on the Drasil repository at https://github.com/JacquesCarette/Drasil.

[I like the above description of Drasil. It felt more focused than some of the other sections. —SS]

5.1 The Generator & Back-end DSLs

Drasil is composed of a number of DSLs working together to generate software artifacts. While there are other DSLs within Drasil, namely the recipe language (See Section 5.3), they are not exactly necessary for generating documentation and source code – they provide useful high-level abstractions. The required DSLs for generating documentation and source code using Drasil are the:

- Expression language
- Sentence specification language
- Knowledge-capture language
- Document layout language
- (Modified) Generic Object Oriented Language (GOOL) framework
- Printing translation language
- LaTeX and HTML printers

The expression language is used for capturing mathematic expressions, such as defining equations. With it we can capture equations, relations, and more. We have included a host of standard unary, binary, and n-ary arithmetic, trigonometric, boolean, and matrix operations. For example, the equation captured in Drasil for linear displacement (from the game physics example) $\mathbf{r}(t) = \frac{d\,\mathbf{p}(t)}{d\,t}$ can be seen in Figure 8. The equation is simply defined as a derivative of the position function with respect to time. At this point in time, the expression language is untyped, however, we recognize the advantages of using a typed expression language and intend to implement expression typing in the future.

The sentence specification language is used for capturing knowledge at the "sentence" level. These are typically English-language sentences which combine captured knowledge in different ways. We can build mixtures of strings with knowledge-level or document-layout-level references, expressions, symbols and special characters, units, and any other captured information. An example

```
inputGlassPropsDesc :: Sentence
inputGlassPropsDesc = ... :+:
  S "Note" :+: ch plate_len sAnd ch plate_width :+:
  S "will be input in terms of" :+: plural millimetre sAnd :+:
  S "will be converted to the equivalent value in" :+: plural metre
```

Fig. 9. A sentence captured in Drasil

sentence (shortened for the sake of brevity) can be seen in Figure 9. Note the use of references to existing pieces of knowledge (`plate_len`, `plate_width`, `millimetre`, and `metre`) allowing us to look up the relevant symbols (*a* and *b* for `plate_len` and `plate_width` respectively) and other information on generation to maintain consistency. The binary infix constructor `:+:` is used for sentence concatenation.

The knowledge-capture language is the collection of data structures, smart constructors, lenses, transformations, and other functions related specifically to capturing and operationalizing knowledge. The knowledge capture language uses the expression and sentence specification languages as building blocks for capturing different properties. For example, a defining equation like that seen in Figure 8 will be captured as one property of a chunk using the expression language, whereas the English-language term referring to the same chunk will be captured using the sentence specification language. See Section 5.2 for more of the implementation details.

The document layout language is where things really start to come together. This is the lowest-level specification for how a document artifact should be pieced together. A *Document* is simply defined as a data structure which includes a title, author(s), and a list of (one or more) sections. The sections are then composed of (sub-)sections and contents (tables, figures, paragraphs, lists, equations, etc.) which are in turn composed of sentences and expressions.

The document layout language is *not* meant to be used to generate artifacts directly – it is far too naive and very tedious to use. We show evidence of this in Figure 10 which shows the effort required to create a document given a simple title, authors, and an introduction section composed of an introductory paragraph and an empty “Scope" subsection.

It seems clear that this approach is not much better than using tools like LaTeX macros, other than our ability to print in both LaTeX and HTML. The only real benefits of assembling documents at this level are the highly-specific control of the layout and the consistent-by-construction output when (re-)using captured knowledge. Alone, that is not enough, hence why we need higher level abstractions (see Section 5.3).

The printing translation language and printers are very closely related. The translation language is a mid-level representation between the higher-level semantic constructs in the document, sentence, and expression languages and the lower-level printable HTML and LaTeX syntax. The printers themselves deal with printing the actual HTML and LaTeX found in the generated artifacts, based solely on the translation-level constructs.

Finally, GOOL [6] is a generic object-oriented language used to generate source code for a number of different programming languages (Python, Java, Objective C, C++, C#, and Lua) from a more abstract specification. As we like to practice what we preach, we have chosen to incorporate a modified version of GOOL into Drasil to facilitate code generation without reinventing the wheel.

5.2 Knowledge Capture & Transformations in Drasil

Before we get into the specifics of transforming knowledge in Drasil, we should first explain how knowledge is captured. We originally borrowed the idea of *Chunks* from Literate Programming, but

```
myDocument :: Document
myDocument = Document myTitle authors sections

myTitle :: Sentence
myTitle = S "Sample document title"

author :: Sentence
authors = S "Szymczak, Carette, and Smith"

sections :: [Section]
sections = [introduction, filler, conclusion]

introduction, filler, conclusion :: Section
introduction = Section (S "Introduction")
  [Con $ ULC $ Paragraph $
    S "This is the introduction paragraph for our tedious document."
    Sub Section (S "Scope")
      [{- Scope section contents go here -}]
  ]
... omitted for brevity
```

Fig. 10. Assembling a document using only the low-level document language is tedious

```
data QuantityDict = QD { _id' :: IdeaDict
                       , _typ' :: Space
                       , _symb' :: Stage -> Symbol
                       , _unit' :: Maybe UnitDefn
                       }
...
instance HasUID         QuantityDict where uid = id' . uid
instance NamedIdea      QuantityDict where term = id' . term
instance Idea           QuantityDict where getA  qd = getA (qd ^. id')
instance HasSpace       QuantityDict where typ = typ'
instance HasSymbol      QuantityDict where symbol = view symb'
instance Quantity       QuantityDict where
instance Eq             QuantityDict where a == b = (a ^. uid) == (b ^.
    uid)
instance MayHaveUnit    QuantityDict where getUnit = view unit'
```

Fig. 11. The *QuantityDict* data structure definition and class instances

have since expanded our knowledge capture mechanism to more closely resemble the classification scheme mentioned in Section 4.1, specifically following the property-based knowledge classes seen in Figure 4.

Knowledge is captured in a number of various data structures representing the common combinations of properties we have observed. For instance, a *QuantityDict* (Quantity dictionary), is shown in Figure 11.

To understand exactly what is being captured, we need to first look at the definition of an *IdeaDict*, which is itself composed of a *NamedChunk* and may have an abbreviation. A *NamedChunk* has a

Fig. 12. Knowledge areas we have started to capture (See: SE-CSE paper) [I can't find this in the old papers, hmm, oh well, I know what to draw —DS]

Unique IDentifier (UID) and a term – it is the minimal complete definition for the *NamedIdea* class of properties. The *IdeaDict* then is the minimal complete definition for the *Idea* class. Finally, the *QuantityDict* also has a space (domain/type), symbol, and may have units. Thus it is the minimal complete definition of the *Quantity* class.

Since Haskell is strictly-typed, having a number of different data structures and expecting them to behave with each other can be difficult. As such, we have implemented each class property using lenses. For those unaware, lenses are composable functional references – they allow us to read and write data deep within our data structures. Combining lenses with Haskell's type classes allows us to capture our knowledge in a variety of data structures, then use the lenses to project out only that which is relevant in a given context, or related to a specific property. Classes and lenses work in tandem to give us data-structure-agnostic setters and getters.

We have already started to capture knowledge from several areas into our knowledge base. A brief, high-level overview can be seen in Figure 12. While we have started capturing knowledge, we have not strayed far from the specifics of our case studies. Our knowledge-base should by no means be considered an ontology or even somewhat comprehensive collection of knowledge at this time. It is quite interesting even, to see how little we have captured, yet how much we can achieve.

What about transformations? In high-level terms, they are our recipes. More specifically, recipes combine atomic knowledge transformations to describe software artifacts to our generator. Atomic transformations come in two flavours, projections or translations. A projection typically involves using a lens to extract the relevant portion of knowledge, while a translation can take knowledge in one form and convert it to another. For example, we can translate an equation into an English description by projecting out each of its symbols' definitions and combining them into a sentence or description block. An example of the generated output can be seen in Figure 2.

5.3 The Recipe Language

The recipe language in Drasil is known under the name `Drasil.DocumentLanguage`. While it is not necessary to generate documentation artifacts, it provides a number of higher-level semantic constructors for combining document-specific layout information alongside document meta-knowledge (captured in `Data.Drasil`).

The recipe language essentially defines new custom data-types specific to the documentation artifact we wish to generate. We are creating representations which combine (document) layout information with system information (captured knowledge) in a meaningful way. Alongside the recipe language, the system information is used to define exactly what knowledge is relevant to a particular software family member.

Within our *SystemInformation* data structure we maintain a database of 15 different chunk classes (all the quantities, concepts, inputs, outputs, constraints, constants, etc.) required to generate (all) our artifacts for the given piece of software. Without a recipe, however, these chunks are essentially useless – Drasil can not do anything with them without a recipe. We need at least one recipe for each of the artifact types we wish to create. Let us consider the SRS as our running example for the rest of this section.

`Drasil.DocumentLanguage` currently contains one recipe for generating the SmithEtAl SRS template. We have created custom, meaningful constructors for the different document sections relevant to the current context of the SRS, see Figure 13 for the top-level constructors. Each of

```
data DocSection = Verbatim Section
                | RefSec RefSec
                | IntroSec IntroSec
                | StkhldrSec StkhldrSec
                | GSDSec GSDSec
                | ScpOfProjSec ScpOfProjSec
                | SSDSec SSDSec
                | ReqrmntSec ReqrmntSec
                | LCsSec LCsSec
                | UCsSec UCsSec
                | TraceabilitySec TraceabilitySec
                | AuxConstntSec AuxConstntSec
                | Bibliography
                | AppndxSec AppndxSec
                | ExistingSolnSec ExistingSolnSec
```

Fig. 13. Our top-level SRS recipe section constructors

these top-level sections, with the exception of Verbatim, can be decomposed in a meaningful way, thus allowing us to specify the subsections we would like to include as well.

With our recipe, we enforce certain conventions, for example subsections can only belong to the appropriate parent sections as outlined in the original template. We have attempted to enforce the bare minimum necessary to maintain a cohesive document, allowing for a greater level of customization for the user. Options we give to our users include such trivial things as customizing the order sections should appear (for example, should the reference information be up-front, somewhere in the middle, or at the end?). We also include less trivial decisions such as the verbosity of description fields (should all relevant symbols be defined in the context of a defining equation, or only the symbol(s) defined by that equation?), or whether to include model derivations.

An example of the SRS recipe in use can be seen in Figure **??** and is explained further in Section 6.2.

5.4 Packaging Drasil

Since Drasil has a lot of moving parts, it may be difficult to see where to start or how things fit together. We have attempted to simplify this by splitting the current version (0.1.23 as of this writing) of Drasil into a group of Haskell packages (see Figure 14).

The core components, including those for document concepts, are stored in a package named *drasil-lang* and code-related components are in *drasil-code*. The main generator code is in *drasil-gen*, with the HTML and LaTeX printers in *drasil-printers*. We maintain our current knowledge-base in *drasil-data* and our case study examples in *drasil-example*. Finally, we have our document recipes in *drasil-docLang*.

5.4.1 drasil-lang. The core language used within the Drasil framework, including all of the building blocks for our knowledge-base are stored within *drasil-lang* under the exported module `Language.Drasil`. This package also exports a second module with functionality targeted to developers of Drasil known as `Language.Drasil.Development`.

`Language.Drasil` presently contains the expression DSL, document layout DSL, and knowledge capture classes & data types. Every other *drasil-** package relies on, and builds off, this core.

With `Language.Drasil` alone we can capture knowledge for generating artifacts nearly identical to those shown in Section 3.1. However this is a much lower-level approach than we would like to

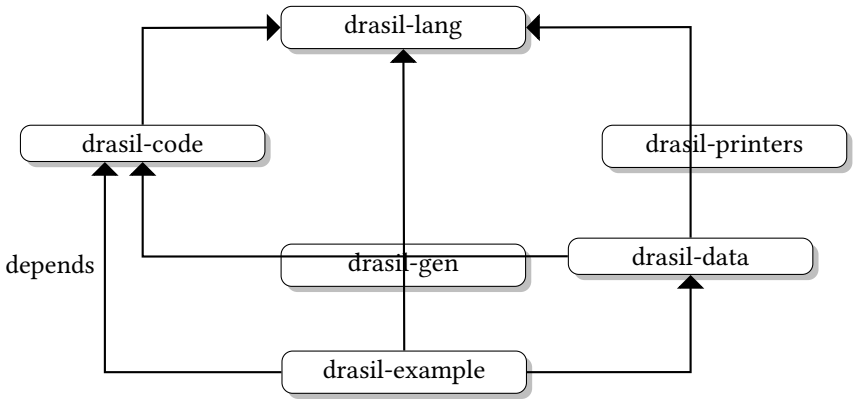

Fig. 14. Drasil package dependencies

use and could be seen as being akin to programming in a lower-level language like C, or perhaps even assembly.

5.4.2 drasil-printers. The printing DSL for HTML and LaTeX are stored here under the namespace `Language.Drasil.Printers`. These are the DSLs which translate Drasil specifications to the target language(s). Please note that the printers for source-code are kept in a separate location.

5.4.3 drasil-code. The code generation DSL used within the Drasil framework is stored here under the namespace `Language.Drasil.Code`. The code generation framework incorporates *GOOL*, a Generic Object-Oriented Language [6], to give us the ability to target multiple languages – C++, C#, Objective C, Java, Lua, and Python.

5.4.4 drasil-gen. The main generator functions used by Drasil are stored here under the `Language.Drasil.Generate` namespace. These functions are used to take a Drasil specification and transform them for use by the appropriate printers to generate the final output file(s). Both code and document generation are handled through function-calls found here.

The actual body of *drasil-gen* is very small, consisting of only approximately 60 lines of Haskell code, but generation would fail without it.

5.4.5 drasil-data. The knowledge-base common to all Drasil programs is curated and maintained within this package under the `Data.Drasil` name. Currently we have captured knowledge in a range of domains including, but not limited to, Computation, Education, Math, Physics, and Software. We have also captured meta-knowledge related to documentation, physical properties, and more.

A more detailed breakdown of `Data.Drasil` is given in Section 6.1.

5.4.6 drasil-docLang. The *drasil-docLang* package contains an example recipe, found in `Drasil.DocumentLanguage`, targeted at recreating the SmithEtAl SRS template[**?**]. This is currently our only recipe, however, *drasil-docLang* will contain our future recipes as well (once they are created)

5.4.7 drasil-example. All of the code required to generate artifacts for our case study examples is maintained in this one package. Each case study has a unique namespace containing everything, other than common knowledge from *drasil-data*, required to generate that particular case study's artifacts. These namespaces can be seen in Table 3.

[I like the explanation of the packages —SS]

Table 3. Case study namespaces in *drasil-example*

Case Study	Namespace
GlassBR	`Drasil.GlassBR`
GamePhysics	`Drasil.GamePhysics`
SSP	`Drasil.SSP`
SWHS	`Drasil.SWHS`
NoPCM	`Drasil.NoPCM`
Tiny	`Drasil.HGHC`

6 CASE STUDIES - IN MORE DEPTH

- Re-introduce case studies - Our methods for reimplementing - CI for testing - Start showing off re-use and automated generation. - Start with common knowledge (generalized !FIGURE!?) - Then onto GlassBR example to show off the doc lang recipe (!FIGURE!?) - Then let's see SRS vs. NoPCM for reuse (particularly NoPCM) (!FIGURE!?)

[I like how specific this section is. You are highlighting specific lessons/findings from actual examples. When you get stuck with writing other sections, this would be a good place to focus your energy. You should be able to write this material almost independently of the other sections, at least to get started. —SS]

[Somewhere in here (probably in GlassBR) I need to really emphasize the separation of System Information from the DocLang. It comes up later. —DS]

6.1 Data.Drasil

- Common knowledge !FIGURE! SI_Units !FIGURE! Thermodynamics (ConsThermE?)

Important to not we have not captured even a minute fraction of all the domain knowledge, yet the results with out limited knowledge-capture already speak for themselves.

6.2 GlassBR

- Brief intro to problem GlassBR is solving - how it works - Show off the doc language here !FIGURE! GlassBR SRS in (truncated) DocLang format - "Reads like a table of contents, with a few quirks" - Show off some code generation !FIGURE! Side-by-side of Chunk Eqn vs. Doc Eqn vs. Code - "Easy to see that the code matches the equations" - Talk about potential variabilities and how to make this a family - Why is this interesting? - Fairly straightforward example of something a scientist would create/use in their research

6.3 NoPCM & SWHS

- Re-introduce the problems - See how they're a family? - Really drill in the similarities !FIGURE! Figure showing NoPCM import(s) - Lots of knowledge-reuse - Very few 'new' chunks (count them?) - Show example of variability in action !FIGURE! Equation with/without PCM (rendered?) - Why this example is interesting: - ODE solver -> We don't gen, just link to existing good one(s)

6.4 Others

- Mention SSP, Tiny, GamePhysics, but don't go too in-depth. - Useful examples as they give us a wider range of problems for analysis - Testing - Physics is physics -> when we make updates, the underlying knowledge isn't changing, so neither should our output - Refer to CI

Fig. 15. Tiny before (left) and after (right) use of Document Language and System Information
[This might need to be an appendix since I'm not sure how to fit it in here —DS]

7 RESULTS

We will now discuss the results of what we have observed thus far. In general, had the manually written versions of these case studies attempted to be certified, they would (and should) have failed.

Overall there were a number of common problems, alongside some case-study specific issues, many of which had gone unnoticed for years. It was not until the case studies were re-developed in Drasil that they were finally resolved.

We will also discuss other observations and enhancements provided by Drasil that we hope will lead to long-term quality improvements for the generated software.

[I like this Results section - examples like these are something people can understand and hopefully be convinced by. —SS]

7.1 Freebies - Compliments of System Information

[Change the title —DS]

We have reduced the burden of knowledge capture for a project by separating the system information from the layout specification and template boilerplate for our artifacts. While this in itself is a good start, it also gives us a number of convenient, and free, perks.

First off, thanks to the recipe language we can generate a number of sections manually. One of the most tedious sections to write and maintain is that of reference material. Manually maintaining the tables of symbols, units, and abbreviations and acronyms is necessary, but ultimately uninteresting. By separating out the system information we have a smaller, more maintainable data structure from which we can generate all of the previously mentioned tables (as well as many other sections and subsections).

Let us first look at the *tiny* case study. As our first toy-example, it lacked the rigour of the other case studies. Particularly, there was no reference information section, only a table of units. However, since Drasil takes the monotony out of creating a reference information section, one is currently being generated in our software artifacts. See Figure 15 for a comparison of the original version of tiny to the freebie-inclusive one.

Not only does the generator populate the reference material from our system information, it also provides a number of sanity-checks. First and foremost, if a symbol is not defined, it cannot be used anywhere else within the document. This actually extends to cross-referencing as well, only pieces of knowledge that have been properly captured and added to the system context can be referenced.

There are other sanity-checks being implemented, such as guarding against captured constraints and ensuring unit consistency in equations, but those features are as yet incomplete. We can already see instances of where they would be very useful, and an example related to unit consistency will be given in Section 7.4.

With that in mind, we can still see the benefits of running even the most trivial sanity-checks every time we generate our artifacts. We no longer need to worry if the reference material will remain consistent with the rest of the artifact, as if a system is modified, we can ensure that no requisite definitions were accidentally removed.

[Move - Bibliography - to section explaining System Information as a concept? —DS]

7.2 Common issues across case studies

Across all of our case studies we were able to find a number of undefined symbols using Drasil. This was after multiple passes by humans attempting to ensure the documents were consistent and up-to-date had failed to catch these small, but important, missing pieces of knowledge.

Drasil was able to detect and correct the issues through its automated generation of the symbol table and use of sanity-checking to ensure that any symbol used had to have been defined within the table of symbols to be used elsewhere in the artifacts.

The symbols that lacked definitions were typically those with an “understood" meaning in the context of their domain, but not once did that definition appear concretely in any of the software artifacts.

Otherwise, the most common issue to crop up across the case studies was that of implicit information. There is tacit knowledge (for domain experts) and undocumented assumptions in a number of the manually-written case studies. Identifying and capturing that knowledge is necessary when attempting to use Drasil, otherwise, there is no way to properly define the system relative to its domain.

7.3 NoPCM and SWHS

As previously mentioned, NoPCM and SWHS are members of the same software family. They share a lot of the same core knowledge. It should be noted that NoPCM was manually created after SWHS.

When implementing NoPCM in Drasil, we found PCM-related knowledge that should not have existed in the case study. It is likely that in the manual version, some information was copied directly from the SWHS case study and pasted into the NoPCM documentation.

This information was trivial to remove in the Drasil version, as a number of the PCM-related symbols were not defined and ended up causing errors during the sanity-checking portion of artifact generation. The PCM-related concepts were not defined in the context of NoPCM, and as such were not in the table of symbols or other reference material. However, prior to implementing our sanity-checking, the PCM-related concepts did actually occur in Drasil, namely due to the lack of a means to specifically exclude knowledge that *should not* exist in a given project. That may be something worth looking into for specific software family-related knowledge, however, on the whole it is not feasible to force developers to specify excluded knowledge for every project.

Another interesting result was the reduction in Kolmogorov complexity across these two case studies. As of this writing the total number of 80-character lines of code written in Drasil, including import blocks and some deprecated code, needed to generate both SWHS and NoPCM is 4460 (ignoring whitespace and comments). This value ignores all back-end packages that define the languages used to write the case study examples in Drasil, as they are not part of the case study knowledge or recipe itself. However, to ensure we are getting an upper bound on the number of lines of code required, this number also includes the entirety of our knowledge base (*Data.Drasil*).

The generated artifacts for SWHS and NoPCM contain a total of 13695 lines of code (again, ignoring whitespace and comments). This is again, an upper limit, as there are many lines of boilerplate, particularly in the HTML, that could be condensed. However, we also do not limit the length of lines in the generated output, many of which far exceed 80 characters. Overall, we need only write less than one third of the lines of code to produce the same output!

If we consider each of NoPCM or SWHS independent of the other, the results are still interesting, but less so. For SWHS the amount of Drasil code written (again, including *Data.Drasil*) is 3678 and the number of generated lines for that example is 8550. While this is still a reduction of over 50%, we can see there is a substantial benefit when producing multiple software family members.

S_i	N	Mobilized shear force. Shear forces that cause instability in a slice. For slice index i.
P_i	N	Shear resistance. Mohr Coulomb frictional force that describes the limit of mobilized shear force the slice can withstand before failure. For slice index i.

Fig. 16. Definitions of P_i and S_i from the SSP case study

Finally, the code generator is still in the process of being updated, so we are not generating any source code for SWHS. Thus the 13695 figure *will* increase, while the 4460 (or 3678) figure should remain near constant. We can support this claim as our NoPCM example is currently generating source code and it requires less than 100 dedicated lines of code in Drasil to do so (included in the above figure of 4460).

We hope these results will lead to an overall reduction in the time required to create and maintain software artifacts while using Drasil, particularly for software families, since there will be less code to manage. However, we make no claim to that effect without experimental evidence to support it. We will discuss plans for experiments in Section 8.

7.4 SSP

A major inconsistency was found in the SSP case study regarding three symbols: S_i, P_i, and τ_i. In the table of symbols, S_i and P_i were defined as shown in Figure **??** and there was no instance of a definition for τ_i.

The lack of definition was a simple enough problem to find, as mentioned in Section 7.2. However, there was a second, more insidious problem to be solved: τ_i actually represented S_i and at some point S_i and P_i had their definitions swapped! If you are interested in the full details of the problem and solution, they can be found at https://github.com/JacquesCarette/Drasil/issues/348.

This seemingly minor error in consistency was only caught after Drasil alerted us that τ_i had not been defined. We then traced through the equations using τ_i to determine its meaning. It took several passes through the manually-written documentation to determine all of the requisite changes. First, the definition of τ_i was added, then the equations were double-checked for correctness, and any place where S_i and P_i were swapped had to be identified.

Another issue arose while fixing the symbol-mixup situation: there was an implicit assumption in one of the equations which caused inconsistencies in the units of the output. Namely, there is an assumption that a particular force per meter is being applied over one unit of width “into the page", thus changing the expected units from $\frac{N}{m}$ to N.

Overall, the changes required in the Drasil source were minimal. A definition and assumption were added and a few symbols were shuffled. Yet there was an immediate improvement in the quality of the documentation, particularly with regards to consistency!

7.5 Pervasive Bugs

One of the utmost benefits of the knowledge-based approach using Drasil is the introduction of “pervasive bugs". These are typically mistakes made in the captured knowledge which propagate across all generated artifacts wherein that knowledge is used. Calling this a benefit may seem counter-intuitive, but when an error appears in a multitude of locations it is far more likely to be caught then were it hiding in some corner of one artifact.

Not only is it more likely that we will find an error, it is also far easier to track down the source of said error – we need only go to the knowledge base and find the requisite chunk. We can also tune error messages to point us at the offending chunk if we so desire.

Correcting an error in a chunk of knowledge is also trivial. It only needs to be fixed once to be fixed across all of our software artifacts. No need to `grep`, find-and-replace, or the like.

7.6 Improved Understanding of Software Artifacts

- Another benefit to using Drasil for writing recipes and performing knowledge-capture is the increased understanding of software artifacts that follows. To write a recipe, one needs to truly understand what is being said and how each piece of knowledge relates to each other.

[Related to the previous [now next —DS] point, the act of formalizing the knowledge that goes into the requirements documentation forces us to deeply understand the distinctions between difference concepts, like scope, goal, theory, assumption, simplification, etc. With this knowledge we can improve the focus and effectiveness of existing templates, and existing requirements solicitation and analysis efforts. Teaching it to a computer. —SS]

8 FUTURE WORK

[This might end up being really long —DS]

[Once we are capable of true variability in the documentation, we can really start asking the question about what is the "best" documentation for a given context. In the future experiments could be done with presenting the same information in different ways to find which approach is the most effective. —SS]

- Run an experiment to determine how easy it is to create new software with Drasil.
- Run an experiment to see how easy it is to find and remove errors with Drasil
- Experiment to see time saved in maintenance while using Drasil vs. not
- Design language
- Open issues (as of writing there are ### issues currently open on the Drasil repository).
- User friendliness
- More sanity checks
- Expression language -> Make it typed
- Expression language -> Make it easier to extend
- More recipes for current artifacts (ie. not just SmithEtAl template), and for new types of artifacts!

9 CONCLUSION

- Easier to find errors (anecdotally) - future work will tell us if this holds.

It is not (yet) faster or easier to create software using Drasil, however, we front-load the investment of effort and receive a number of benefits. Consistency by construction is paramount!

The next step, experimentation, is pivotal to confirm our intuition holds and hopefully show a knowledge-based approach with strong tool-support is feasible for software family generation.

REFERENCES

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

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

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

[4] Carver, J. C., Kendall, R. P., Squires, S. E., and Post, D. E. 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* (Washington, DC, USA, 2007), IEEE Computer Society, pp. 550–559.

[5] CDRH. General principles of software validation; final guidance for industry and fda staff, 2002.

[6] Costabile, J. Gool: A generic oo language. Master's thesis, McMaster University, Canada, 2012.

[7] CSA. Quality assurance of analytical, scientific, and design computer programs for nuclear power plants. Tech. Rep. N286.7-99, Canadian Standards Association, 178 Rexdale Blvd. Etobicoke, Ontario, Canada M9W 1R3, 1999.

[8] CSA. Guideline for the application of N286.7-99, quality assurance of analytical, scientific, and design computer programs for nuclear power plants. Tech. Rep. N286.7.1-09, Canadian Standards Association, 5060 Spectrum Way, Suite 100, Mississauga, Ontario, Canada L4W 5N6, 1-800-463-6727, 2009.

[9] Czarnecki, K. Overview of generative software development. In *Unconventional Programming Paradigms*. Springer, 2005, pp. 326–341.

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

[11] Durkin, J. *Expert systems: design and development*. Macmillan Coll Div, 1994.

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

[13] Feigenbaum, E. A., and McCorduck, P. *The fifth generation*. Addison-Wesley, 1983.

[14] Flatt, M., Barzilay, E., and Findler, R. B. Scribble: Closing the book on ad hoc documentation tools. In *ACM Sigplan Notices* (2009), vol. 44, ACM, pp. 109–120.

[15] Fritzson, P., Gunnarsson, J., and Jirstrand, M. Mathmodelica-an extensible modeling and simulation environment with integrated graphics and literate programming. In *2nd International Modelica Conference, March 18-19, Munich, Germany* (2002).

[16] Gentleman, R., and Lang, D. T. Statistical analyses and reproducible research. *Journal of Computational and Graphical Statistics* (2012).

[17] Hatcliff, J., Heimdahl, M., Lawford, M., Maibaum, T., Wassyng, A., and Wurden, F. A software certification consortium and its top 9 hurdles. *Electronic Notes in Theoretical Computer Science 238*, 4 (2009), 11–17.

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

[19] Ionescu, C., and Jansson, P. 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* (2012), vol. 8241 of *Lecture Notes in Computer Science*, Springer International Publishing, pp. 140–156.

[20] Johnson, A., and Johnson, B. Literate programming using noweb. *Linux Journal 42* (October 1997), 64–69.

[21] Kelly, D. Scientific software development viewed as knowledge acquisition: Towards understanding the development of risk-averse scientific software. *Journal of Systems and Software 109* (2015), 50–61.

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

[23] Kiselyov, O., Swadi, K. N., and Taha, W. A methodology for generating verified combinatorial circuits. In *Proceedings of the 4th ACM international conference on Embedded software* (2004), ACM, pp. 249–258.

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

[25] Kotula, J. Source code documentation: an engineering deliverable. In *tools* (2000), IEEE, p. 505.

[26] Leisch, F. Sweave: Dynamic generation of statistical reports using literate data analysis. In *Compstat* (2002), Springer, pp. 575–580.

[27] Lenth, R. V. Statweave usersâĂŹ manual. *URL http://www. stat. uiowa. edu/˜ rlenth/StatWeave* (2009).

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

[29] Logg, A., Mardal, K.-A., and Wells, G. *Automated solution of differential equations by the finite element method: The FEniCS book*, vol. 84. Springer Science & Business Media, 2012.

[30] Merali, Z. Computational science: ...error. *Nature 467* (2010), 775–777.

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

[32] Nedialkov, N. S. VNODE-LP — a validated solver for initial value problems in ordinary differential equations. Tech. Rep. CAS-06-06-NN, Department of Computing and Software, McMaster University, 1280 Main Street West, Hamilton, Ontario, L8S 4K1, 2006.

[33] Owen, S. J. A survey of unstructured mesh generation technology. In *INTERNATIONAL MESHING ROUNDTABLE* (1998), pp. 239–267.

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

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

[36] Pieterse, V., Kourie, D. G., and Boake, A. 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 (Republic of South Africa, 2004), SAICSIT '04, South African Institute for Computer Scientists and Information Technologists, pp. 2–9.

[37] Püschel, M., Moura, J. M., Johnson, J. R., Padua, D., Veloso, M. M., Singer, B. W., Xiong, J., Franchetti, F., Gačic, A., Voronenko, Y., et al. Spiral: Code generation for dsp transforms. *Proceedings of the IEEE 93*, 2 (2005), 232–275.

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

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

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

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

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

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

[44] Simonis, V. Progdoc–a program documentation system. *Lecture Notes in Computer Science 2890* (2001), 9–12.

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

[46] U.S. Food and Drug Administration. Infusion pumps total product life cycle: Guidance for industry and fda staff. on-line, December 2014.

[47] Wilson, G. V. 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).