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, 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 (July 2018), 15 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/2018/7-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 bottom up justification should also be part of the introduction. —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" [16]. Essentially, we are ensuring that 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, 44] 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.

[rework the following paragraph —DS] 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 [17, 24]. Similarly, there are many downsides to inconsistent documentation [24, 43].

1.2 Scope

- Scientific Computing Software - Why? Many highly specialized SCS require certification. Ex. Control sfwr in nuclear power, x-ray machines, and other safety-critical contexts. - Well understood domain -> theories underpinning the work being done.

[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]

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 or potentially improve. We then explore previous efforts in automating software artifact generation, as well as look at the current state of Model Driven Engineering, scientific knowledge ontologies, and other areas that may be 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 [21]. These developers tend to prefer an agile [1, 4, 11, 40] or knowledge-acquisition driven process [20] 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 [45]. There is also a limited use of automated testing [33] and lack of understanding of what good software testing entails [29] with quality assurance having "a bad name among creative scientists and engineers" [38, p. 352].

With good science relying on replication and reuse, we see a lack of reuse on the software side. A survey [32] 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 simply be reused.

Some SC software developers have used advanced techniques, like code generation, quite successfully. Examples that come to mind are FEniCS [28], FFT [22], Gaussian Elimination [3], and Spiral[36]. The focus of generation techniques, thus far, have been solely on a single software artifact: the source code.

2.2 Software Artifact Reuse and Generation

Previous attempts at generating software artifacts were primarily focused on reusability. 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 [18].

2.2.1 Reproducible Research. The term *reproducible research* means embedding executable code in research papers to allow readers to reproduce the results described [39]. 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 [15].

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 [25], SASweave [27], Statweave [26], Scribble [13], and Org-mode [39]. These tools maintain a focus on specific computational details and code in general. Sweave (the most popular of the aforementioned examples [39]) 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 [23].

Developing literate programs involves breaking algorithms down into *chunks* [19] or *sections* [23] which are small and easily understandable. These chunks are organized into a "psychological order" [35] 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 as a program is developed and updated. There is also some reduction in knowledge duplication through chunking. Adopting proper usage of LP ends up with more consistent documentation and code [41]. Keeping in mind the benefits of artifact consistency we can see that more effective and maintainable code can be produced when using LP [35].

LP has not been very popular due to a couple of main issues: language/text processor dependency and the lack of flexibility on output presentation/suppression [41]. Still, there are several successful examples of literate programs in SC. Two such examples are VNODE-LP [31] and "Physically Based Rendering: From Theory to Implementation" [34]. 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 [37], 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 Sweave [25], however it is mainly used for dynamically generating reports as mentioned in Section 2.2.1.

New tools led to the introduction of many new features including, but not limited to, a "What You See Is What You Get" (WYSIWYG) editor [14], phantom abstracting [41], and even movement away from the "one source" idea [42].

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 as 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 [30] was proposed as a new method called "Literate Software Development" (LSD) [2].

Box structure can be summarized as using different abstractions (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 [30]. 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 [9].

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

Previous attempts at automating / reducing the artifact burden.

- Previous attempts at automatically generating documentation - LP, tools like javadoc, Haddock, etc. - Too code-centric! - Comments and code still need to be updated in parallel, albeit to a lesser extent in some cases - In general, fairly rigidly structured output (you don't have much say on how it looks, only what information should be included and, sometimes, where - Finish with a focus on the good stuff: - Identified the need for good documentation - Keeps docs and code in the same place - Easier to manually maintain consistency and apply updates - One other problem we've identified: - common underlying knowledge between projects is duplicated as there is no real cross-project reuse mechanism in place with these tools.

[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]

2.3 Knowledge-Based Software Engineering (KBSE)

[Pull apart this section and move the pieces around as necessary. Some of this is intro, some is our approach, etc. —DS]

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" [12]. This is a solid 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" [10].

[Red here is used elsewhere —DS] For our purposes, we extend and tighten this definition 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 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. 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.

For our KBSE approach to succeed, there are two major requirements. First off, we must capture the underlying knowledge in a meaningful way that can be reused across artifacts. We want a single source for our knowledge, regardless where it ends up or how it is viewed. This allows us, using the right transformations, to automatically generate our software artifacts from the underlying knowledge-base.

The second requirement is that we restrict our scope to well-understood domains as we need a solid theoretical underpinning. 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.

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.

[Should MDL show up here? —SS]

3 A RATIONAL ANALYSIS OF SOFTWARE ARTIFACTS

- This section exists to show how we get from problem to solution.

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.

- We introduce our case studies in a bit more depth here - GlassBR - what it's for, if it'll - SWHS and NoPCM - Software family members with a twist. - The rest (tiny, Gamephysics, and SSP) for additional examples and to give us a bit more credibility in our analysis. - Looking for commonalities between types of artifacts and what they are really saying. - An obvious commonality across many projects in SCS – SI and derived Units.

3.2 Common software artifacts

- Compare and contrast different software artifacts. - SRS vs. detailed design vs. code - same knowledge, different 'views' - only some of that knowledge is necessarily relevant in those views - !FIGURE!: SRS & DD showing the same piece of knowledge in diff contexts. Use a few different !FIGURE!here. - !FIGURE!: Attempt to show generalized overlap via Venn diagram?

3.3 Emerging structures

~~[want to fit this into the analysis but it would make more sense after the intro to KBSE —DS]~~

[Makes more sense now that intro to KBSE has been moved. —DS]

- As shown above,

Fig. 1. Data Definition for !FIXME! from GlassBR SRS

Fig. 2. Data Definition code from GlassBR implementation

In the common software artifacts we see different ways of representing what are, semantically, the same things (for example, see Figure 3.2). 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 across artifacts from GlassBR (Figures 1,2), we can see that it is an aggregation of the following components:

- Unique Identifier (label)
- Symbolic (theory) representation
- Symbolic (implementation) representation
- Concise natural language description (a term)
- Verbose natural language description (a definition)
- Equation
- Constraints
- Units?

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 (!FIXME! and !FIXME! respectively for this example).

We also have a defining equation, which incorporates the symbolic representation for various other pieces of knowledge and relates them to !FIXME name!. 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 unit-less. 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]

-Discuss the breakdown of knowledge into classes. Refer to Table 1 for more. [This table looks like a good way to summarize this information to me. —SS]

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, lack of reusability, and difficulty with (re-)certifiability.

Table 1. Knowledge Classes

Knowledge Class	ID	Term	Abbreviation	Definition	Symbol	Equation	Constraints	Units
Labeled	X							
Named Idea	X	X	O					
Common Idea	X	X	X					
Concept	X	X	X	X				

Legend: X - Mandatory; O - Optional

For our purposes, we extend and tighten the definition of knowledge introduced in Section 2.3 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 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. 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 approaches involves using this underlying idea in the creation and maintenance of a singular knowledge-base that we can pull from 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 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 viewed. Thus, using the right transformations and projections, we can automatically generate our software artifacts from the knowledge-base.

The second requirement is that we restrict our scope to well-understood domains as we need a solid theoretical underpinning. 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

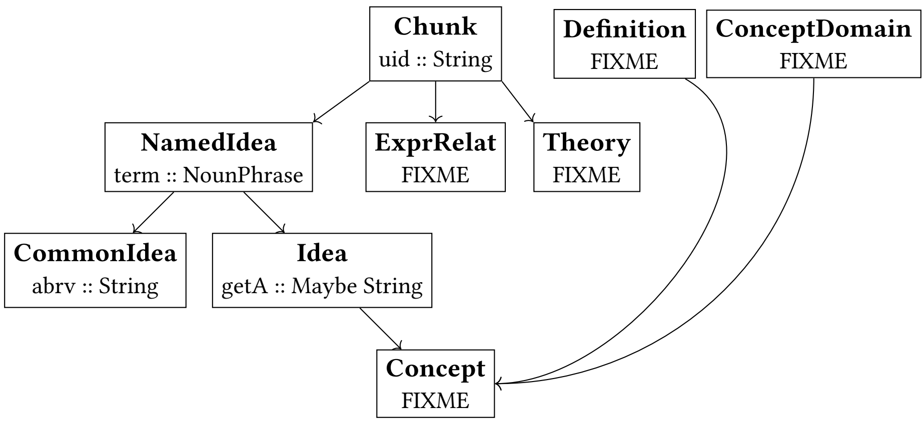

Fig. 3. Chunk hierarchy in Drasil Today

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.

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. [Should this information go closer to the GlassBR example above? —SS]

We borrow, and expand, the idea of *Chunks* from Literate Programming (LP) [23] to facilitate our knowledge-capture. 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 3. 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]

Once 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!

4.2 (Re-)Using Knowledge

Capturing knowledge in itself 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 2.3, 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 we need it to be, what audience we are targeting, and 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. [Can we call what we are doing transformation? I've always thought of what we are doing as model transformation, but in a conversation Wolfram Kahl implied that it isn't. I didn't quite catch his point, but I think there was a concern that we do not do bidirectional transformation? —SS]

We can also use transformations 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 fundamental knowledge being shared by both solutions.

!FIGURE! Show portion of each SRS, one similarity, one difference?

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.

5 DRASIL

- To use KBSE to its potential we need a strong support framework - Intro to Drasil !FIGURE! Knowledge tree - What it is and does - Domain Specific Language(s) - Generate all the things! - Dev to date. - How is Knowledge Capture handled in Drasil? - chunks! - What do transformations look like? Recipes! !FIGURE! SmithEtAl template for SRS = Drasil.DocumentLanguage - Key components of the generator / renderer

- Haddock

5.1 Developing Drasil - A grounded theory

- Following grounded theory (ish). Using data from case studies to guide development and implement new features. - !FIGURE!: Before and after System Information. - !FIGURE!: Before and after mini-DBs - Majority of features developed after analyzing commonalities in the case studies and abstracting them out. - Allows for rapid progress -> constant iteration based on what we find in the data.

5.2 Packaging Drasil

The current version (0.1.1 as of this writing) of Drasil is built as a group of Haskell packages (see Figure 4). 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/LaTeX printers in *drasil-printers*. We maintain our current knowledge-base in *drasil-data* and our case study examples in *drasil-example*.

5.2.1 *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`.

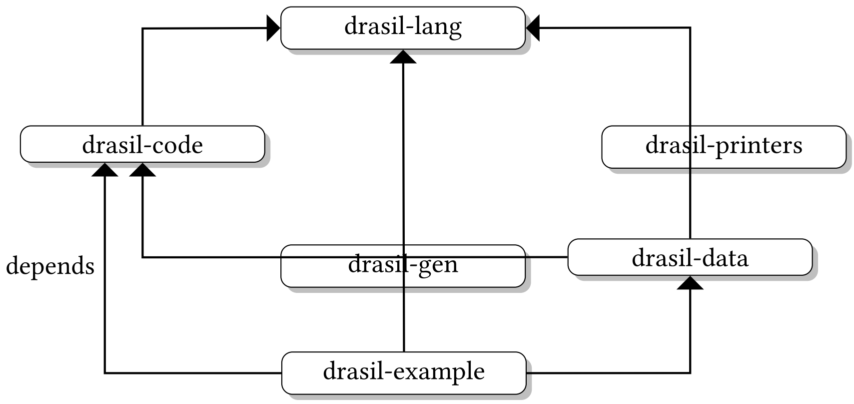

Fig. 4. Drasil package dependencies

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

5.2.2 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.

5.2.3 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.

5.2.4 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 use 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.

5.2.5 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` will be given in Section 6.1.

5.2.6 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 2.

The *drasil-example* package also contains an example recipe, found in `Drasil.DocumentLanguage` targeted at recreating the SmithEtAl SRS template[**?**]. Keeping the recipe with the examples is less than ideal, hence why it will soon be moved out of *drasil-example* and into its own package soon,

Table 2. 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`

along with other recipes as they are created. We will discuss the recipe and document language specifics in Section 5.3.

[I like the explanation of the packages —SS]

5.3 Drasil Today

- Sentence and Document - Explain the chunk hierarchy (refer to Section 4.1 figure) - Data.Drasil !FIGURE! Knowledge areas we have started to capture (See: SE-CSE paper) - Recipe Language(s) – Refer to: !FIGURE! Drasil.DocumentLanguage - The generator - HTML and TeX rendering - GOOL for code - System Information -> Get into it

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]

6.1 Data.Drasil

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

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

6.5 Freebies - Compliments of System Information

- Thanks to the recipe language and the way we structure out system information we can get - Table of Symbols - Table of Units - Table of Abbreviations and Acronyms - Bibliography

- All tedious to do by hand, but are free to automatically generate - Generator includes sanity-checking -> Can't use something that isn't defined! - Sanity-checks are 'free' -> we can check for errors with our symbols, ensure units are consistent, guard against constraints, and ensure we only reference those things which are defined in our system. - Sanity-checks are run every time artifacts are generated.

7 RESULTS

- Here we discuss the results we've seen so far. - Had some of these case studies attempted to be certified, they would (should) have failed. - A number of common problems.

7.1 Common issues across case studies

- A number of undefined symbols even after multiple passes by humans. (Auto-generating the symbol table and including sanity-checking revealed them) [You are giving specific examples below, right? —SS]

7.2 NoPCM and SWHS

- Along with the common errors, there was some sharing of PCM-related knowledge - Found because PCM symbols were not in the ToS and the sanity-check caught it. - No way to specifically exclude knowledge that shouldn't 'exist' in a project - Work in Kolmogorov complexity / MDL for NoPCM + SWHS? - Kolmogorov/MDL implies less writing for the same artifacts -> less to sift through = maybe better?

7.3 SSP

- Symbols for given quantities changed throughout the documentation - Went unnoticed by a human for years! Found almost instantly by Drasil - the new symbols were undefined.

7.4 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 if it were hiding in the 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 exact chunk causing the problem 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.

8 FUTURE WORK

[*SS* - 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* - Related to the previous 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.]

- 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
- Create drasil-gen package
- Design language
- Open issues (as of writing there are ### issues currently open on the Drasil repository).

9 CONCLUSION

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

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] 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).

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

[11] 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.

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

[13] 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.

[14] 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).

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

[16] 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.

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

[18] 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.
[19] Johnson, A., and Johnson, B. Literate programming using noweb. *Linux Journal 42* (October 1997), 64–69.
[20] 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.
[21] Kelly, D. F. A software chasm: Software engineering and scientific computing. *IEEE Softw. 24*, 6 (2007), 120–119.
[22] 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.
[23] Knuth, D. E. Literate programming. *The Computer Journal 27*, 2 (1984), 97–111.
[24] Kotula, J. Source code documentation: an engineering deliverable. In *tools* (2000), IEEE, p. 505.
[25] Leisch, F. Sweave: Dynamic generation of statistical reports using literate data analysis. In *Compstat* (2002), Springer, pp. 575–580.
[26] Lenth, R. V. Statweave usersâĂŹ manual. *URL http://www. stat. uiowa. edu/˜ rlenth/StatWeave* (2009).
[27] Lenth, R. V., Højsgaard, S., et al. Sasweave: Literate programming using sas. *Journal of Statistical Software 19*, 8 (2007), 1–20.
[28] 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.
[29] Merali, Z. Computational science: ...error. *Nature 467* (2010), 775–777.
[30] Mills, H. D., Linger, R. C., and Hevner, A. R. Principles of information systems analysis and design.
[31] 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.
[32] Owen, S. J. A survey of unstructured mesh generation technology. In *INTERNATIONAL MESHING ROUNDTABLE* (1998), pp. 239–267.
[33] Patrick, M., Elderfield, J., Stutt, R. O., Rice, A., and Gilligan, C. A. Software testing in a scientific research group.
[34] Pharr, M., and Humphreys, G. *Physically Based Rendering: From Theory to Implementation.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.
[35] 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.
[36] 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.
[37] Ramsey, N. Literate programming simplified. *IEEE software 11*, 5 (1994), 97.
[38] Roache, P. J. *Verification and Validation in Computational Science and Engineering.* Hermosa Publishers, Albuquerque, New Mexico, 1998.
[39] 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.
[40] Segal, J. When software engineers met research scientists: A case study. *Empirical Software Engineering 10*, 4 (October 2005), 517–536.
[41] Shum, S., and Cook, C. Aops: an abstraction-oriented programming system for literate programming. *Software Engineering Journal 8*, 3 (1993), 113–120.
[42] Simonis, V. Progdoc–a program documentation system. *Lecture Notes in Computer Science 2890* (2001), 9–12.
[43] Thimbleby, H. Experiences of 'literate programming'using cweb (a variant of knuth's web). *The Computer Journal 29*, 3 (1986), 201–211.
[44] U.S. Food and Drug Administration. Infusion pumps total product life cycle: Guidance for industry and fda staff. on-line, December 2014.
[45] 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).