Drasil: A Framework for Scientific Knowledge Capture and Artifact Generation

Daniel Szymczak
Computing and Software Department, McMaster University
Hamilton, Ontario
szymczdm@mcmaster.ca

Spencer Smith
Computing and Software Department, McMaster University
Hamilton, Ontario
smiths@mcmaster.ca

Jacques Carette
Computing and Software Department, McMaster University
Hamilton, Ontario
carette@mcmaster.ca

Steven Palmer
Computing and Software Department, McMaster University
Hamilton, Ontario
palmes4@mcmaster.ca

ABSTRACT

What if building *well-documented* software was not tedious and unrewarding? We are implementing a framework, Drasil, to greatly alleviate these issues. Here we document the design and development process of Drasil. We show, through a case study (GlassBR – for computing the safety of glass panes with respect to explosions), how Drasil can be used. While software like GlassBR is algorithmically straightforward, it is used in safety related applications, which makes its correctness quite important. For Drasil, the context we are most interested in is when correctness is established within the context of *software certification*.

Our principal means of producing documentation is through weaving together standardized knowledge, as captured in Drasil. We illustrate how this makes scientific software development more transparent, and increased but reusability and reproducibity.

CCS CONCEPTS

• **Applied computing** → *Document preparation*; Physical sciences and engineering; • **Software and its engineering** → *Source code generation*; *Software development techniques*; *Reusability*; *Automatic programming*; *Traceability*; • **Social and professional topics** → Testing, certification and licensing;

KEYWORDS

scientific computing, software quality, software engineering, document driven design, code generation, software certification

ACM Reference format:
Daniel Szymczak, Spencer Smith, Jacques Carette, and Steven Palmer. 2017. Drasil: A Framework for Scientific Knowledge Capture and Artifact Generation. In *Proceedings of 2017 International Workshop on Software Engineering for High Performance Computing in Computational and Data-Enabled Science and Engineering, Denver, Colorado, USA, November 2017 (SE-CSE_SE-CoDeSE),* 8 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

SE-CSE_SE-CoDeSE, November 2017, Denver, Colorado, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Every developer should strive towards creating the highest possible quality software. This is especially true in safety-related applications where incorrect software could cost lives. As scientists, we should be leading the community in developing high quality software, as it is our duty to ensure the reusability, reproducibility, and replicability of our work.

Our team is focused on improving the quality of Scientific Computing Software (SCS). We have chosen large, multi-year, multi-developer projects where the end users do much of the development as our target scope. For these projects, we pay particular attention to improving the qualities of reusability, reproducibility, and certifiability.

The goal for software certification is to: "...systematically determine, based on the principles of science, engineering and measurement theory, whether a software product satisfies accepted, well-defined and measurable criteria" [5, p. 12]. We aim to streamline the certification process, by simplifying the creation of the requisite software artifacts (documentation and code) and by ensuring these artifacts are always consistent. Tracking down the source of any information contained in our artifacts should be trivially easy, yet this is not always the case, since documentation is often not a high priority.

Improved documentation is often considered too high a cost in terms of time and effort for SCS developers, particularly when dealing with rapid changes in development. However, it is an important aspect of improving overall software quality. Carver [2] observed that scientists do not view rigid, process-heavy approaches, favourably. Moreover, others have observed that SCS developers tend to dislike producing documentation [9, p. 373].

Previous work by Smith & Koothoor [11] found 27 errors in existing safety-related software (nuclear safety analysis software) when creating new documentation. Had they attempted to certify the software, most, if not all, of these errors would have led to a failure.

Well-maintained documentation provides numerous advantages including:

- Improved software qualities:
  - Verifiability
  - Reusability
  - Reproducibility
  - etc.
- Other improvements (from Parnas [7]):
  - Easier reuse of old designs
  - Better communication about requirements
  - More useful design reviews

Developers are being made aware of the advantages of documentation [10], yet it can still be difficult to ensure it becomes a high priority because documenting software is typically felt to be:

- Too time consuming
- Too difficult to maintain
- Not amenable to change
- Too tied to the waterfall process
- Counterproductive when reporting on each stage of development [9]

There are also other issues with document-rich, certifiable software development. Namely the vast amount of knowledge duplication across software artifacts. Consider a simple piece of software with the following artifacts:

- Software Requirements Specification (SRS)
- Design Document
- Source Code
- Tests

Knowledge, such as the functional requirements, from the SRS is duplicated in both the source code and tests. That same knowledge appears in the design document as well, albeit in a transformed state.

Now consider that software certification often requires a great many more artifacts than the four listed above. Much of the same knowledge, possibly in a transformed representation, will appear in all of the artifacts and will need to be maintained. Manually maintaining these artifacts is a lesson in tedium and the time could be better spent elsewhere, if it were easier to deal with automatically.

The Solution?

How can we enjoy the benefits of documentation, without the usually associated drudgery? Our proposed solution is *Drasil* – a framework that utilizes a knowledge-based approach to software development, as originally proposed in a position paper by Szymcak et al [13]. The goal of the approach is to capture scientific and documentation knowledge in a reusable way, then generate the source code and all other software artifacts (documentation, build files, tests, etc).

Work on Drasil has continued steadily since the original position paper, as described below. We begin with a brief overview of the design of the Drasil framework in Section 2, then describe the development process that has been adopted in Section 3. Following this, we show an example of Drasil in action (Section 4) and the results we have seen to date (Section 5). Finally, we lay out some of the work that still needs to be done (Section 6) before concluding.

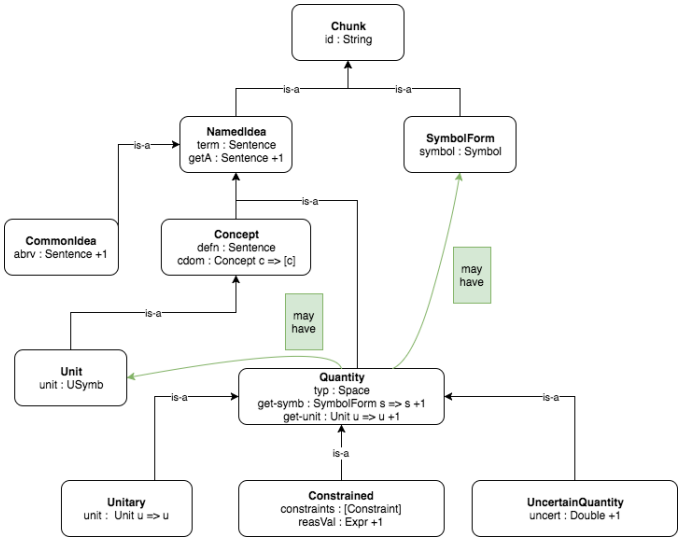

Figure 1: Drasil chunk hierarchy

2 DESIGN OF DRASIL

Drasil's design is based around three main components:

(1) Knowledge capture mechanisms (*Chunks*)
(2) Artifact generation language(s) (*Recipes*)
(3) Knowledge-base (*Data.Drasil*)

Chunks are the primary knowledge-capture mechanism. There are many flavours of chunks (as shown in Figure 1). The most basic chunk is simply a piece of data with an id. From there, all other chunks can be created. For example, a *NamedIdea* is a chunk containing an id, a term representing its principle idea, and a potential abbreviation for that term. A *Quantity* is a *NamedIdea* which also has a *Space* (integer, boolean, vector, etc.), and symbol representation/units (if applicable). The notation in Figure 1 uses a +1 to indicate when a value may or may not be present (implemented using **`Maybe`** types in Haskell).

We can think of chunks as our building blocks of knowledge; they are the ingredients to be used in our *Recipes*. Our language of recipes is a Domain-Specific Language (DSL) embedded in Haskell that is used to define what we would like to generate, and in what order. A small snippet of recipe language code for our Software Requirements Specification (SRS) can be seen in Figure 2. This code is used to generate the *Reference Materials* section of our SRS, which contains an introduction followed by the table of units, table of symbols, and table of abbreviations and acronyms subsections.

The document generation language is highly abstracted, but allows for a fairly high degree of customization. Drasil also contains a code-generation language integrating GOOL [3] – a Generic Object-Oriented Language – which can generate code in a number of different target languages including Python, Lua, and C++. We will discuss code generation in more depth through the example in Section 4.

Finally, there is the knowledge-base for Drasil (located in Data.Drasil). We are creating a database of reusable scientific knowledge that can be applied across a number of different applications across multiple

```
mkSRS :: DocDesc
mkSRS = [RefSec (RefProg intro
        [TUnits,
        tsymb [TSPurpose, SymbOrder],
        TAandA])]
        ...
```

Figure 2: The reference material section for an SRS written in Drasil's Document Language

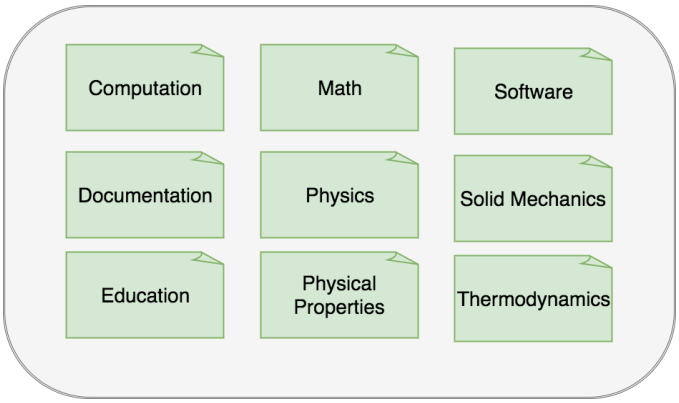

Figure 3: Data.Drasil knowledge domains

domains. As the Drasil framework grows, we hope to continue to expand this database into an ontology of scientific knowledge for a number of disciplines. See Figure 3 for an example of some of the domains in which we have started to capture knowledge.

3 DEVELOPMENT PROCESS FOR DRASIL

Drasil is being developed using a practical, example-driven process. There are currently five different examples being developed concurrently within (and driving the development of) Drasil:

- Chipmunk2D Game Physics Engine
- Solar Water Heating System Incorporating Phase Change Material (PCM)
- Solar Water Heating System (No PCM)
- Slope Stability Analysis
- Glass Breakage Analysis (GlassBR)

The original (non-Drasil) documentation for these examples came out of a study on Document Driven Design (DDD) for SCS [10].

Our practical design approach allows us the flexibility to prototype without over-designing. As a new feature becomes necessary to continue the implementation of a given example, only then do we design, test, implement, and re-test it. We occasionally implement features we may need in the future, but only in those instances when it is obvious that we are taking the right approach.

The five examples provided an ideal means to bring additional members on to the project. Summer student research assistants joined the project and were responsible for completing most of the work on translating the original examples into Drasil, and then later transforming the examples, as the Drasil DSLs have been extended and refactored. Through their work, the summer students have demonstrated that using Drasil does not require prior knowledge of Haskell and code generation techniques. The student members of the Drasil project cover disciplines ranging from Chemical Engineering, Applied Mathematics, Computer Science and Software Engineering, with levels of education ranging from first to third year. None of the students had prior knowledge of Haskell.

The current incarnation of the Drasil framework can be found on GitHub at https://github.com/JacquesCarette/literate-scientific-software. We utilize peer-review of code throughout development to correct missteps early on, and keep an up-to-date issue tracker for any bugs, feature requests, or other "to-do" tasks.

Progressive development of Drasil is achieved by not only looking for new features that must be implemented, but also through a cyclic approach towards improvement. This approach relies on finding new (extractable) patterns in the framework through refactoring, de-embedding and extracting knowledge from the example materials, and reducing knowledge duplication by capturing it in a highly reusable way.

By adopting a practical, example-driven, "bottom-up," approach for the development of Drasil, we have avoided the problems inherent in a more theoretical, or "top-down," approach. The size, scope and complexity of SCS means that attempting to imagine a priori the DSLs that we would need would be exceedingly difficult. We would invariably leave something out and likely provide too simple a framework. Rather than try to arbitrarily determine the initial scope of Drasil, we let the examples tell us what they need. The "bottom-up" approach was previously successfully used for generating members of the family of Gaussian elimination algorithms [1].

The idea that scientific software forms a program family [12] is helpful for the current work. We can imagine that each of our five examples provides one specific instance of the more general family of SCS. With five examples we feel confident that enough of the commonalities and variabilities between members of the family of SCS are present that the initial implementation of Drasil will provide a reasonable starting point for implementing a representative subset of the full family of SCS.

4 A PRACTICAL EXAMPLE (GLASSBR)

GlassBR is a piece of software used by structural engineers to predict whether or not a slab of glass will be able to withstand a given explosion without breaking. The two classes of input are: glass geometry and blast type. Each of these input classes has a number of fields (glass type, dimensions, TNT equivalent factor, standoff distance, etc.) used as input to the analysis. The user must also specify their tolerable probability of breakage.

The output of GlassBR is whether or not the glass slab is considered safe. This is determined by calculating the probability of breakage, which involves interpolation from several tabulated values, then that probability is compared to the tolerable probability and a safety evaluation is made.

To understand the Drasil implementation, we will follow one specific piece of knowledge through from requirements to code. We intend to show how this knowledge is captured and used in Drasil to produce our software artifacts (documentation and code).

Refname	DD:sdf.tol
Label	J_{tol}
Units	
Equation	$J_{tol} = \log\left(\log\left(\frac{1}{1-P_{btol}}\right)\frac{\left(\frac{a}{1000}\frac{b}{1000}\right)^{m-1}}{k\left((E*1000)\left(\frac{h}{1000}\right)^2\right)^m * LDF}\right)$
Description	J_{tol} is the stress distribution factor (Function) based on Pbtol P_{btol} is the tolerable probability of breakage a is the plate length (long dimension) b is the plate width (short dimension) m is the surface flaw parameter k is the surface flaw parameter E is the modulus of elasticity of glass h is the actual thickness LDF is the load duration factor

Figure 4: J_{tol} from GlassBR Requirements

Let us start by taking a look at a data definition for the tolerable stress distribution factor (J_{tol}) for GlassBR. Figure 4 shows J_{tol} in pdf format, which was compiled from the Drasil-generated LaTeX for the J_{tol} quantity chunk. We can also generate the documents in HTML. The data definition is part of the requirements for the GlassBR software, and as such, we will eventually need code that can be used to calculate J_{tol}. A python version of this code can be seen in Figure 5. From the J_{tol} quantity chunk, we can generate this code as well as the documentation! Not only that, but thanks to the incorporation of GOOL, we can also generate Java, Lua, C#, etc.

The source knowledge for generating both the documentation and the code has been captured using chunks, as shown in Figure 6. The value of J_{tol} is calculated from the `tolStrDisFac_eq` expression, which is part of the `tolStrDisFac` chunk. The equation (`tolStrDisFac_eq`) is defined using other quantity chunks, like `plate_len` and `mod_elas`. These chunks themselves have their own definitions, which includes their description, symbol and units. When the knowledge from J_{tol} is rendered in some form, the knowledge from its constituent chunks is also available. This means the *Description* field in Figure 4 can be generated automatically from the knowledge implicit in the `tolStrDisFac_eq` equation.

To facilitate potential future change, Figure 6 shows that the symbol for J_{tol} is defined as `sub (stressDistFac ^. symbol) (Atomic "tol")`. This means the symbol for J_{tol} is the symbol for the `stressDistFac` with the subscript `"tol"`. If the symbol for `stressDistFac` should change from J to some other letter, the change in the base symbol and its variations will automatically be consistent throughout all artifacts.

During the process of translation from the original documentation [10] to the Drasil code, we noticed an error in the code and documentation. The formula for J_{tol} should not include the unit conversion implied by dividing by 1000 in a number of places. Luckily, with one quick change to `tolStrDisFac_eq` (shown in Figure 7), we have corrected the error in our knowledge-base. After re-running the generator, our code and documentation has now been fixed and remains consistent. With Drasil, we are able to correct an error in one place (in the knowledge-base) and every artifact that depends on that knowledge is automatically fixed.

Table 1: Constraints on Quantities – Used To Verify Inputs

Var	Constraints	Typical Value	Uncertainty
L	$L > 0$	1.5 m	10%
ρ_P	$\rho_P > 0$	1007 kg/m^3	10%

Similarly to the knowledge for J_{tol}, we have captured all of the knowledge pertaining to GlassBR in chunks. This knowledge can be assembled and extracted in different ways to produce a multitude of views, i.e. our artifacts, including the SRS. For a sense of what we can generate from this knowledge, see the table of contents for the currently generated GlassBR SRS (Figure 8). Again, our documents can be generated in TeX and/or HTML for more flexibility. Both contain automated internal referencing.

Another thing to note in our SRS is that the traceability information between definitions, assumptions, theories, and instance models is automatically generated, including the traceability graph shown in Figure 9.

5 QUALITY IMPROVEMENTS

Throughout Drasil's development lifecycle, we have already noticed a number of non-trivial quality improvements to the software examples being re-developed. These improvements come in a wide range of areas, but for the sake of brevity we will focus on certifiability, reusability, and reproducibility.

5.1 Certifiability

Certifying (or re-certifying) software can be a lengthy and expensive process where a single error can cause the software to fail the certification process. Consider the following equation from our solar water heating example which states that energy in the water must be conserved between the heating coil and water and water and PCM (Phase Change Material):

$$E_W = \int_0^t h_C A_C (T_C - T_W(t))dt - \int_0^t h_P A_P (T_W(t) - T_P(t))dt$$

Where:

- E_W is the change in heat energy in the water
- A_C is the heating coil surface area
- A_P is the PCM surface area
- h_C is the convective heat transfer coefficient between coil and water
- h_P is the convective heat transfer coefficient between PCM and water
- T_C is the temperature of the heating coil
- T_P is the temperature of the PCM
- T_W is the temperature of the water

This is trivial for a human to check, and writing code to automatically check it during the software execution is also very simple. However, it is still time-consuming and should it ever need to change, such as if there is a change in the assumption that the tank is a perfect insulator, it would require developers to make a number of modifications to many artifacts in the future.

```
def calc_j_tol(inparams):
    j_tol = math.log((math.log(1.0/(1.0 - inparams.pbtol))) *
      ((((inparams.a / 1000.0) * (inparams.b / 1000.0)) ** (inparams.m - 1.0)) /
      ((inparams.k * (((inparams.E * 1000.0) *
      ((inparams.h / 1000.0) ** 2.0)) ** inparams.m)) * inparams.ldf)))
    return j_tol
```

Figure 5: Python code to Calculate $J_{\mathbf{tol}}$

```
stressDistFac = makeVC
  "stressDistFac" (nounPhraseSP $ "stress distribution" ++ " factor (Function)") cJ

sdf_tol = makeVC
  "sdf_tol" (nounPhraseSP $ "stress distribution" ++ " factor (Function) based on Pbtol")
  (sub (stressDistFac ^. symbol) (Atomic "tol"))

tolStrDisFac_eq :: Expr
tolStrDisFac_eq = log (log ((1) / ((1) - (C pb_tol))) * ((Grouping (((C plate_len) / (1000)) *
  ((C plate_width) / (1000)))) :^ ((C sflawParamM) - (1)) / ((C sflawParamK) *
  (Grouping (Grouping ((C mod_elas) * (1000)) *
  (square (Grouping ((C act_thick) / (1000)))))) :^ (C sflawParamM) * (C loadDF))))

tolStrDisFac :: QDefinition
tolStrDisFac = mkDataDef sdf_tol tolStrDisFac_eq
```

Figure 6: Drasil (Haskell) code for $J_{\mathbf{tol}}$ Knowledge

```
tolStrDisFac_eq :: Expr
tolStrDisFac_eq = log (log ((1) / ((1) - (C pb_tol))) * ((Grouping ((C plate_len) *
  (C plate_width))) :^ ((C sflawParamM) - (1)) / ((C sflawParamK) *
  (Grouping ((C mod_elas) * (square (C act_thick)))) :^ (C sflawParamM) * (C loadDF))))
```

Figure 7: Modified Drasil (Haskell) code for $J_{\mathbf{tol}}$ equation

Thanks to Drasil's knowledge-capture mechanisms, we can capture the above equation in one place, thus any changes can be made very quickly. Not only that, but sanity checks (such as ensuring constraints on inputs) can also be captured and reused. An example of these sorts of sanity checks can be seen in Table 1. They include hard physical constraints, ex. $L > 0$, as well as typical values. Capturing this information delivers a wider array of advantages, including:

- Generating guards against invalid input automatically
- Generating certain test cases automatically
- Generating views suitable for inspection
- Warning the user if they choose a value that is order(s) of magnitude different than the typical value

We can also see that it is extremely easy to trace changes and verify that they are correct, thanks to our one-source, generative approach. We can produce documents similar to those commonly found in literate programming where we display the equations and code implementations (example: the J_{tol} equation and the python code used to calculate it) side-by-side, allowing a human to quickly verify the code is correct. This literate technique proved valued with the nuclear safety analysis software mentioned earlier [11].

We have also touched on one of the greatest advantages (implied earlier): *If there is an error somewhere, it is wrong everywhere.* This may seem like a disadvantage, but ensuring every artifact contains the same errors means there is a much greater chance of those errors being spotted. In our experience, many examples of software include code that does not match the design (due to hacks, last-minute changes, etc.), and an error which would hamper certification efforts can fly under the radar.

For (re-)certification purposes, we want to be able to find any and all errors, trace them to their source, and fix them quickly. Drasil has shown great promise in that respect.

5.2 Reusability

Reusability is a core tenet of Drasil's development. This can be seen most obviously in the knowledge-capture mechanisms we have created. These mechanisms facilitate reuse through common knowledge databases for each domain (recall Figure 3). The reuse

Figure 8: Table of Contents for Generated SRS for GlassBR

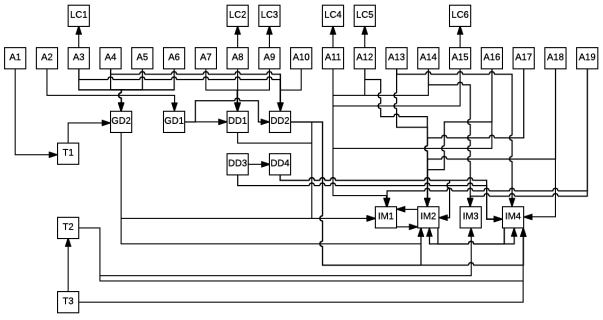

Figure 9: Traceability Graph

can occur within a single document, with a series of documents, and even between projects.

The value of reuse within a given document is seen in Figure 4 and Figure 8. Information on variables, including their definitions, their symbols and their units, is reused in multiple locations throughout the requirements document (SRS). For instance, lists of variables appear in the table of symbols, as part of each data definition and as part of the instance models. Maintaining the consistency between these different renderings would be very challenging manually, but by reusing the knowledge whenever necessary, the documentation is significantly more maintainable.

Consider the following software artifacts typically found in a rational design process for software development [8]:

- Software Requirements Specification (SRS)
- Module Interface Guide (MIS)
- Source Code
- Test cases

Within these artifacts, there is considerable duplication of specific knowledge related to the software being designed. Knowledge from the SRS will appear, possibly transformed, throughout each of the other artifacts. We seek to de-embed specific knowledge from within any one artifact to easily reuse it throughout them all. This knowledge includes, but is not limited to:

- Scientific Knowledge
- Models
- Units
- Symbols
- Descriptions
- Traceability information

While implementing our case studies in Drasil, we have found this de-embedding process to be invaluable. Previously "dead" information found in the software artifacts is now captured, reusable, and transformable. For example, an equation such as the one for defining J_{tol} can now be reused across artifacts in whatever form we deem necessary. This could be the equation written mathematically, described in English, or converted into executable code. Every form is still being generated from the same source, we simply reuse it in a slightly different way.

Reusing knowledge across artifacts in one software project is already incredibly useful, however, we have taken it a step further. Drasil allows us to reuse between projects. We can reuse a family of related models, or reuse pieces of knowledge from a given domain as necessary. Consider the vast number of software projects that rely on interpolation, or must ensure conservation of thermal energy. We can capture this knowledge once and reuse it through all of these projects.

We have already begun using knowledge across projects with great results. Our two implementations of the solar water heating systems (one with PCM and one without) have a vast amount of overlap in the requisite knowledge. The same governing equations apply, but in a simplified form when one does not have to worry about heat transfer to and phase change of, the PCM. As such, these two implementation utilize many of the same chunks from our knowledge-base, simply reordered and transformed as necessary. If we find an error in the theory, or need to make a change to improve the validity of our calculations, fixing the Drasil source will update both projects simultaneously.

5.3 Reproducibility

Typically when we refer to reproducibility, the emphasis is on reproducing the calculations from compiled code. The problem is that over time programming languages, compilers and operating environments change. Virtual Machines (VMs) can be a significant help here, but VMs do not help when there are problems in the original code. What if our trust in the original code is in doubt? What if we need to reproduce the results not starting from the code,

but starting from the original theory? Unfortunately, multiple examples exist [4, 6] where results from research software could not be independently reproduced, due to a need for local knowledge missing from published documents. Examples of missing knowledge includes missing assumptions, derivations, and undocumented modifications to the code. Should it not be easier to independently replicate the work of others without relying on the code they have written?

We want to replicate not only the execution of the code, but the whole of the software from theory to implementation. Given the appropriate theoretical knowledge, assumptions, equations, etc. we should be able to take this high-level knowledge and reproduce an implementation which will return consistent results. Drasil allows us to do exactly this; we can capture the high-level science knowledge and use it to reproduce a piece of software. We can also package our projects in a Drasil-ready format, such that the software can be generated on-the-fly by any who wish to replicate the results or double-check the science.

Drasil can also potentially check for completeness and consistency, ensuring the little things that are tedious, if trivial, to manually verify. For example: every symbol should be defined and used and should not change throughout the artifacts (inter- and itra-artifact consistency). This may seem minor, but one of our case studies, which had been reviewed by humans on numerous occasions, introduced a new symbol for an existing concept in a derivation. This new symbol was never defined and did not appear in the table of symbols. When implementing the project in Drasil, however, this error was made obvious almost immediately and was easily fixed.

Previously, the manual checks for verifying documentation have been listed [11]:

- check that every symbol is defined
- check that every symbol is used at least once
- check that every equation either has a derivation, or a citation to its source
- check that every general definition, data definition, and assumption is used by at least one other component of the document
- check that every line of code either traces back to a description of the numerical algorithm, to a data definition, to an instance model, to an assumption, or to a value from the auxiliary constants table in the SRS.

Going forward Drasil can be used to automate these checks and provide error or warning messages when they are violated.

Overall, Drasil lends itself incredibly well to ensuring reproducibility in a much larger sense than we normally consider. It allows us to focus on the science, (re-)building our projects from the fundamental knowledge instead of relying on a lone developer's code. Drasil also forces all assumptions to be documented, and disallows local hacks, since the generated code should never be modified by a human, it should only be (re-)generated.

6 FUTURE WORK

Drasil has come a long way in the last year, yet there is still much to be done. With the current code-generation mechanisms being fairly naive, we have begun work on a language of design which allows us to apply design decisions to the theory knowledge to produce more flexible code. This language allows us to classify design and implementation choices in a reusable manner and will allow us to progress with the generation of design documents, which are currently difficult to produce in Drasil.

The language of design will also facilitate more robust automated test case generation. By utilizing the captured design decisions, we will have a deeper knowledge of specific choices being made in our generated code. For example, in GlassBR, we will know which interpolation algorithm to use, and we will be able to generate test cases relevant to said algorithm.

Another large feature that will be a necessity in the long-run, but has thus far been relegated to the back-burner, is a much more user-friendly syntax, or even a tool to aid in the creation of Drasil projects. Drasil is currently fairly esoteric to use, even with all our documentation to date. Yet, with each iteration of the language we introduce new, simplifying abstractions, which have already made it easier to use. Our experience with our summer student research assistants is a testament to that. We are aware, however, of the gap that still needs to be closed before it will be ready for our target demographic.

We are also continuing work on:

- Generating new types of software artifacts
- Implementing much larger examples

Thanks to our practical approach to Drasil's design, there will likely be many other minor features to add and changes to make that are not yet entirely clear to us.

7 CONCLUDING REMARKS

While Drasil is still a work-in-progress, it shows great promise as a tool to aid SCS developers. The ability to capture scientific knowledge, document it, and transform it into usable code from a single, fully-traceable source should not be underestimated.

We have already seen non-trivial improvements to existing software during its re-implementation in our current, working-copy of Drasil. The case studies are now easier to verify, maintain, reuse, and replicate than they were before. There is also far less ambiguity on where certain concepts originated, as everything is captured in one source (the knowledge-base).

Not only do we believe that scientists will be able to create higher quality software while focusing more on the science; they will be able to replicate and reuse each other's work in a far more robust manner than we currently see in practice. We hope Drasil will solve the current problems with software result replicability starting from the theory [4, 6].

Software (re-)certification will also become a far less daunting task in a future with Drasil. With full traceability of information and the ability to generate new document views on-the-fly, we will be able to streamline the process. This means that recertification will be significantly less expensive than the original certification exercise.

While developing in Drasil does come with a fairly high, upfront investment of effort, we believe the long-term maintainability improvements, along with the streamlining of verification/certification and the other previously mentioned advantages are well worth the cost.

ACKNOWLEDGMENTS

The assistance of McMaster University's Dr. Manuel Campidelli, Dr. Wael and Dr. Michael Tait with the GlassBR example is greatly appreciated.

REFERENCES

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

[2] Jeffrey C. Carver, Richard P. Kendall, Susan E. Squires, and Douglass E. Post. 2007. 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*. IEEE Computer Society, Washington, DC, USA, 550–559. https://doi.org/10.1109/ICSE.2007.77

[3] Jason Costabile. 2012. *GOOL: A Generic OO Language.* Report. McMaster University, Department of Computing and Software, Hamilton, ON, Canada.

[4] Tom Crick, Benjamin A. Hall, and Samin Ishtiaq. 2014. "Can I Implement Your Algorithm?": A Model for Reproducible Research Software. *CoRR* abs/1407.5981 (2014). http://arxiv.org/abs/1407.5981

[5] John Hatcliff, Mats Heimdahl, Mark Lawford, Tom Maibaum, Alan Wassyng, and Fred Wurden. 2009. A Software Certification Consortium and its Top 9 Hurdles. *Electronic Notes in Theoretical Computer Science* 238, 4 (2009), 11–17. https://doi.org/10.1016/j.entcs.2009.09.002

[6] Cezar Ionescu and Patrik Jansson. 2012. 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 (Lecture Notes in Computer Science)*, Vol. 8241. Springer International Publishing, 140–156. https://doi.org/10.1007/978-3-642-41582-1_9

[7] David Lorge Parnas. 2010. Precise Documentation: The Key to Better Software. In *The Future of Software Engineering*. 125–148. https://doi.org/10.1007/978-3-642-15187-3_8

[8] David L. Parnas and P.C. Clements. February 1986. A Rational Design Process: How and Why to Fake it. *IEEE Transactions on Software Engineering* 12, 2 (February 1986), 251–257.

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

[10] W. Spencer Smith, Thulasi Jegatheesan, and Diane F. Kelly. 2016. Advantages, Disadvantages and Misunderstandings About Document Driven Design for Scientific Software. In *Proceedings of the Fourth International Workshop on Software Engineering for High Performance Computing in Computational Science and Engineering (SE-HPCCE).* 8 pp.

[11] W. Spencer Smith and Nirmitha Koothoor. 2016. A Document-Driven Method for Certifying Scientific Computing Software for Use in Nuclear Safety Analysis. *Nuclear Engineering and Technology* 48, 2 (April 2016), 404–418. https://doi.org/10.1016/j.net.2015.11.008

[12] W. Spencer Smith, John McCutchan, and Fang Cao. 2007. Program Families in Scientific Computing. In 7^{th} *OOPSLA Workshop on Domain Specific Modelling (DSM'07)*, Jonathan Sprinkle, Jeff Gray, Matti Rossi, and Juha-Pekka Tolvanen (Eds.). Montréal, Québec, 39–47.

[13] Daniel Szymczak, W. Spencer Smith, and Jacques Carette. 2016. Position Paper: A Knowledge-Based Approach to Scientific Software Development. In *Proceedings of SE4Science'16 in conjunction with the International Conference on Software Engineering (ICSE).* In conjunction with ICSE 2016, Austin, Texas, United States. 4 pp.