Progress Report on 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

abstract here

CCS CONCEPTS

• **Mathematics of computing** → *Mathematical software*; • **Software and its engineering** → *Software development techniques*; *Automatic programming*;

KEYWORDS

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

ACM Reference format:
Daniel Szymczak, Spencer Smith, Jacques Carette, and Steven Palmer. 2017. Progress Report on 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),* 5 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Every developer should strive towards creating the highest possible quality software. As scientists, we should be leading the community in this regard 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. Improving these software qualities is especially important where correctness can have an impact on safety, for example: nuclear safety analysis or medical imaging.

Often considered too high a cost in terms of time and effort for SCS developers, particularly when dealing with rapid changes in development, improved documentation is an important aspect of improving overall software quality. Carver [1] observed that scientists do not view rigid, process-heavy approaches, favourably. SCS developers tend to dislike producing documentation and often consider reports for each stage of software development as counterproductive [4, p. 373].

Well-maintained documentation provides numerous advantages including:

- Improved software qualities
  - Verifiability
  - Reusability
  - Reproducibility
  - etc.
- From Parnas [3]:
  - Easier reuse of old designs
  - Better communication about requirements
  - More useful design reviews
  - etc.

Previous work by Smith & Koothoor [6] found 27 errors in an existing software project when creating new documentation. Developers have become aware of these advantages of documentation [5].

However, documenting software is typically felt to be:

- Too long
- Too difficult to maintain
- Not amenable to change
- Too tied to the waterfall process
- Counterproductive when reporting on each stage of development [4]

The Solution?

Drasil – a framework, utilizing a knowledge-based approach to software development, proposed in a position paper [7]. The goal of the approach is to capture scientific and documentation knowledge in a reusable way, then generate the source code and other software artifacts (documentation, build files, tests, etc).

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

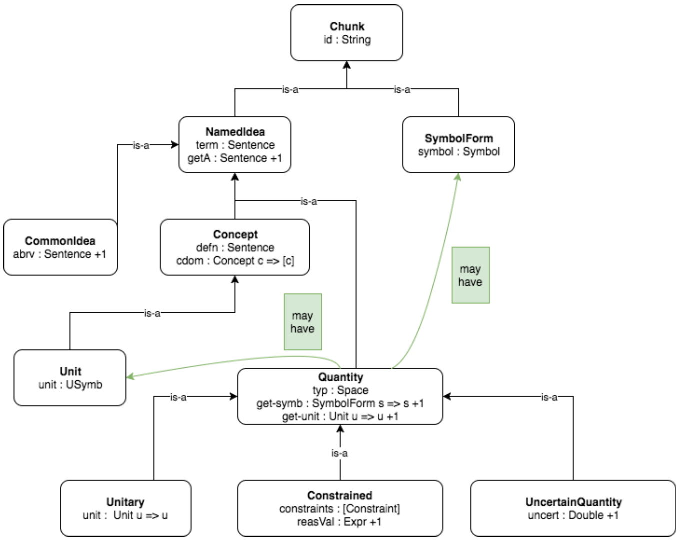

Figure 1: Drasil chunk hierarchy

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

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 chunk (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 *Quantity* is a *NamedIdea* (a chunk containing an id, as well as a term which represents the idea and a potential abbreviation for that term) which also has a *Space* (integer, boolean, vector, etc.), and symbol representation/units (if applicable).

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 which 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 [?] – 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 later on.

```
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

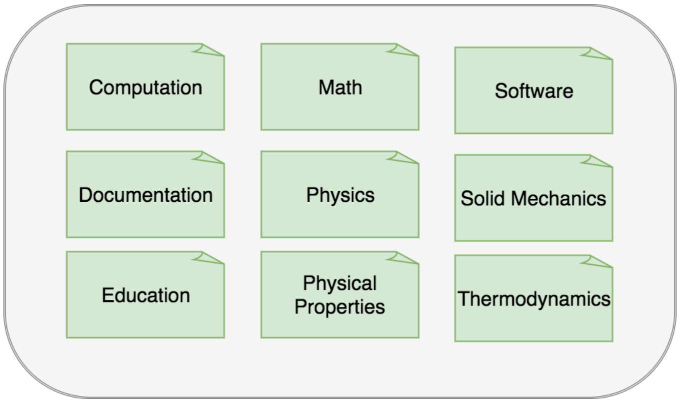

Figure 3: Data.Drasil knowledge domains

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 domains. As the Drasil framework grows, we hope to continue to expand this database into an ontology of scientific information 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 Engine
- Solar Water Heating System Incorporating Phase Change Material (PCM)
- Solar Water Heating System (No PCM)
- Slope Stability Analysis
- Glass Breakage Analysis

These examples overlap with those found in [5].

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.

Refname	**DD:sdf.tol**
Label	J_{tol}
Units	
Equation	$J_{tol} = \log\left(\log\left(\frac{1}{1-P_{btol}}\right)\frac{\left(\frac{a}{1000}\frac{b}{1000}\right)^{m-1}}{k\left(\left(E*1000\right)\left(\frac{h}{1000}\right)^{2}\right)^{m}*LDF}\right)$
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

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.

4 A PRACTICAL EXAMPLE (GLASSBR)

GlassBR is a piece of software used in Civil Engineering to predict whether or not a slab of glass will be able to withstand a given blast without breaking. It has two classes of input: 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 simulation. Also, a tolerable probability of breakage is given by the user.

The output of GlassBR is whether or not the glass slab is considered safe. This is based on a probability that is calculated through interpolation being compared to the tolerable probability.

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).

Let us start by taking a look at a data definition for the tolerable stress distribution factor (J_{tol}) from GlassBR. Figure 4 shows the Drasil-generated TeX version of the data definition for J_{tol}, however we can also generate the documents in HTML. This figure is part of the requirements for the GlassBR software, and as such, we will eventually need code (like that in Figure 5) that can be used to calculate J_{tol}. We can generate this code as well! Not only that, but thanks to the incorporation of GOOL we can also generate Java, Lua, etc.

The source knowledge for generating both the documentation and the code has been captured using chunks as shown in Figure 6.

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%

The value of J_{tol} is calculated from the expression tolStrDisFac_eq , which is part of the tolStrDisFac chunk.

Notice there is actually an error in the code and documentation. We should not be 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. Thus, after re-running the generator, our code and documentation has now been fixed and remains consistent.

-All of the knowledge on GlassBR can be put together to generate the software requirements specification. Can point to a figure showing the table of contents for the SRS. Explain that it can be generated in tex (pdf) or html.

Part of SRS is automatically generated traceability information between definitions, assumptions, theories and instanced models.

5 QUALITY IMPROVEMENTS

5.1 Certifiability

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

- *If wrong, wrong everywhere*
- Sanity checks captured and reused
- Generate guards against invalid input
- Generate test cases
- Generate view suitable for inspection
- Traceability for verification of change

5.2 Reusability

- De-embed knowledge
- Reuse throughout document
  - Units
  - Symbols
  - Descriptions
  - Traceability information
- Reuse between documents
  - SRS
  - MIS
  - Code
  - Test cases
- Reuse between projects
  - Knowledge reuse
  - A family of related models, or reuse of pieces
  - Conservation of thermal energy
  - Interpolation
  - Etc.

5.3 Reproducibility

- Usual emphasis is on reproducing code execution

```
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_{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_{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_{tol}

- However, [2] show reproducibility challenges due to undocumented:
  - Assumptions
  - Modifications
  - Hacks
- Shouldn't it be easier to independently replicate the work of others?
- Require theory, assumptions, equations, etc.
- Drasil can potentially check for completeness and consistency

6 FUTURE WORK

7 CONCLUDING REMARKS

8 ACKNOWLEDGEMENTS

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

REFERENCES

[1] 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

[2] 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

[3] 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

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

[5] 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.

[6] 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

[7] 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.

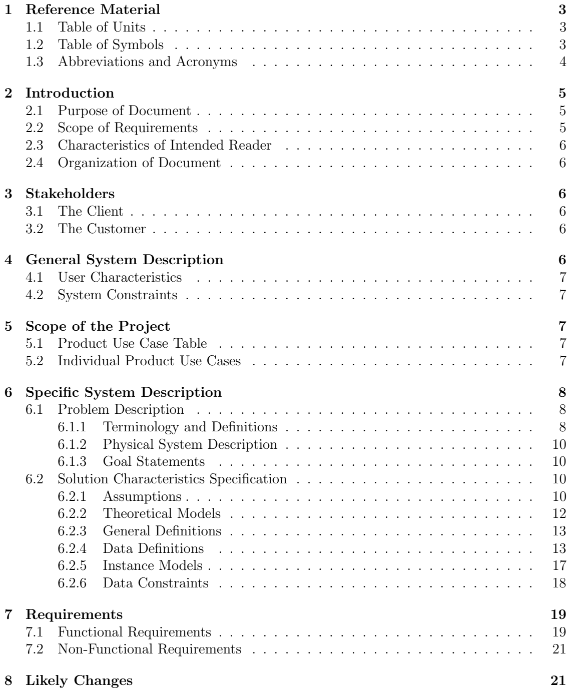

Figure 8: Table of Contents for Generated SRS for GlassBR

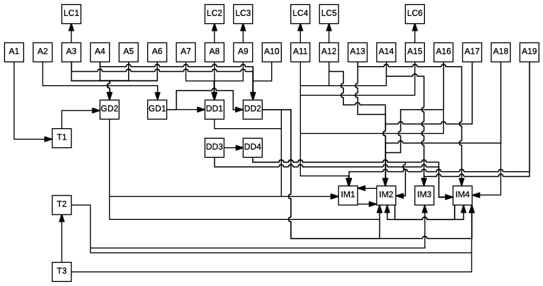

Figure 9: Traceability Graph