THE DRASIL FRAMEWORK

SUCCINCTLY VERBOSE: THE DRASIL FRAMEWORK

BY

DANIEL M. SZYMCZAK, M.A.Sc., B.Eng

A THESIS

SUBMITTED TO THE DEPARTMENT OF COMPUTING & SOFTWARE

AND THE SCHOOL OF GRADUATE STUDIES

OF MCMASTER UNIVERSITY

IN PARTIAL FULFILMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

Doctor of Philosophy(2021) McMaster University

Computing & Software Hamilton, Ontario, Canada

TITLE:	Succinctly Verbose: The Drasil Framework
AUTHOR:	Daniel M. Szymczak M.A.Sc. (Software Engineering) McMaster University Hamilton, Ontario, Canada
SUPERVISORS:	Jacques Carette and Spencer Smith
NUMBER OF PAGES:	xii, 25

Lay Abstract

A lay abstract of not more 150 words must be included explaining the key goals and contributions of the thesis in lay terms that is accessible to the general public.

Abstract

Abstract here (no more than 300 words)

Your Dedication

Optional second line

Acknowledgements

Acknowledgements go here.

Contents

List of Figures

List of Tables

Notation, Definitions, and Abbreviations

[TODO: Update this —DS]

Notation

$A \leq B$	A is less than or equal to B

Definitions

Challenge	With respect to video games, a challenge is a set of goals presented to the player that they are tasks with completing; challenges can test a variety of player skills, including accuracy, logical reasoning, and creative problem solving

Abbreviations

AI	Artificial intelligence

Chapter 1

Introduction

Documentation is good [source], yet it is not often prioritized on software projects. Code and other software artifacts say the same thing, but to different audiences - if they didn't, they would be describing different systems.

Take, for example, a software requirements document. It is a human-readable abstraction of *what* the software is supposed to do. Whereas a design document is a human-readable version of *how* the software is supposed to fulfill its requirements. The source code itself is a computer-readable list of instructions combining *what* must be done and, in many languages, *how* that is to be accomplished.

[Put in figures of an example from GlassBR/Projectile here, showing SRS, DD, and code versions of the same knowledge]

[Figure] shows an example of the same information represented in several different views (requirements, detailed design, and source code). We aim to take advantage of the inherent redundancy across these views to distill a single source of information, thus removing the need to manually duplicate information across software artifacts.

Manually writing and maintaining a full range of software artifacts (i.e. multiple

documents for different audiences plus the source code) is redundant and tedious. Factor in deadlines, changing requirements, and other common issues faced during development and you have a perfect storm for inter-artifact syncronization issues.

How can we avoid having our artifacts fall out of sync with each other? Some would argue "just write code!" And that is exactly what a number of other approaches have tried. Documentation generators like Doxygen, Javadoc, Pandoc, and more take a code-centric view of the problem. Typically, they work by having natural-language descriptions and/or explanations written as specially delimited comments in the code which are later automatically compiled into a human-readable document.

While these approaches definitely have their place and can come in quite handy, they do not solve the underlying redundancy problem. The developers are still forced to manually write descriptions of systems in both code and comments. They also do not generate all software artifacts - commonly they are used to generate only API documentation targeted towards developers or user manuals.

We propose a new framework, Drasil, alongside a knowledge-centric view of software, to help take advantage of inherent redundancy, while avoiding manual duplication and synchronization problems. Our approach looks at what underlies the problems we solve using software and capturing that "common" or "core" knowledge. We then use that knowledge to generate our software artifacts, thus gaining the benefits inherent to the generation process: lack of manual duplication, one source to maintain, and 'free' traceability of information.

1.1 Value in the mundane

??

1.2 Scope

We are well aware of the ambitious nature of attempting to solve the problem of manual duplication and unnecessary redundancy across all possible software systems. Frankly, it would be highly impractical to attempt to solve such a broad spectrum of problems. Each software domain poses its own challenges, alongside specific benefits and drawbacks.

Our work on Drasil is most relevant to software that is well-understood and undergoes frequent change (maintenance). Good candidates for development using Drasil are long-lived (10+ years) software projects with artifacts of interest to multiple stakeholders. With that in mind, we have decided to focus on scientific computing (SC) software. Specifically, we are looking at software that follows the pattern 'input -¿ process -¿ output'.

SC software has a strong fundamental underpinning of well-understood concepts. It also has the benefit of seldomly changing, and when it does, existing models are not necessarily invalidated. For example, rigid-body problems in physics are well-understood and the underlying modeling equations are unlikely to change. However, should they change, the current models will likely remain as good approximations under a specific set of assumptions. For instance, who hasn't heard 'assume each body is a sphere' during a physics lecture?

1.3 Roadmap

1.4 Contributions & Publications

Chapter 2

Background

2.1 Software Artifacts

- (Reference a lot of Parnas)

2.2 Software Reuse and Software Families

- Bring up GNU/Linux and different distros as examples of software families - (Raspbian v Raspbian lite) = Debian–, etc.

2.3 Literate Programming

- Knuth - lots of Knuth - Gentleman and Lang 2012 - Johnson and Johnson 1997 - Literate programming using noweb - VNODE-LP - Physically Based Rendering ... - Al-Maati and Boujarwah 2002

2.4 Generative Programming

- ?

Chapter 3

A look under the hood - Our process

The first step in removing unnecessary redundancy is identifying exactly what that redundancy is, and where it exists. To that end we need to understand what each of our software artifacts is attempting to communicate, who their audience is, and what information can be considered boilerplate versus system-specific. **NOTE: in the breakdown make sure to mention "We actually found that some info is boilerplate, some is system-specific, and some is general to all members of a software family, but more specific than generic boilerplate" **NOTE: Also ensure each artifact has a 'who' (audience), 'what' (problem being solved), and 'how' (specific-knowledge vs boilerplate) Luckily, we have an excellent starting point thanks to the work of many smart people - artifact templates.

Lots of work has been done to specify exactly what should be documented in a given artifact in an effort for standardization. Ironically, this has led to many different 'standardized' templates. However, through the examination of a number

of templates for different artifact types, we have concluded they convey roughly the same overall information for a given artifact. Most differences are stylistic or related to content organization, as we will demonstrate in the following sections. –Or naming conventions

Once we understand our artifacts, we take a practical, example-driven approach to identifying redundancy through the use of existing software system case studies. For each of these case studies, we start by examining the source code and existing software artifacts to understand exactly what problem they are trying to solve. From there, we attempt to distill the system-specific knowledge and generalize the boilerplate.

3.1 A (very) brief introduction to our case study systems

[Potentially canibalize the intro from the next section to specify the templates in use, so it doesn't seem to come out of nowhere.

Summarize the following: – Case study name – Problem being solved – Appendices containing artifacts? – May come in handy for examples]

To simplify the process of identifying redundancies and patterns, we have chosen case studies developed using common artifact templates, specifically those used by [SmithEtAl] [source]. Also, as mentioned in [SCOPE], we have chosen software systems that follow the 'input' -¿ 'process' -¿ 'output' pattern. These systems cover a variety of use cases, to help avoid over-specializing into one particular system-type.

This section is meant to be used as a high-level reference to each case study, providing the general details at a glance. For the specifics of each system, all relevant

case study artifacts can be found in the appendices.

** Should this be a table / series of cards?

— Case Study Name: GlassBR — — Problem being solved: We need to efficiently and correctly predict whether a — glass slab can withstand a blast under given conditions. — — Relevant artifacts in Appendix

— Case Study Name: SWHS — — Problem being solved: Solar water heating systems incorporating phase change — material (PCM) use a renewable energy source and provide a novel way of — storing energy. A system is needed to investigate the effect of employing PCM — within a solar water heating tank. — — Relevant artifacts in Appendix

— Case Study Name: NoPCM — — Problem being solved: Solar water heating systems provide a novel way of — heating water and storing renewable energy. A system is needed to investigate — the heating of water within a solar water heating tank. — — Relevant artifacts in Appendix

— Case Study Name: SSP — — Problem being solved: A slope of geological mass, composed of soil and rock — and sometimes water, is subject to the influence of gravity on the mass. — This can cause instability in the form of soil or rock movement which can — be hazardous. A system is needed to evaluate the factor of safety of — a slope's slip surface and identify the critical slip surface of the slope, — as well as the interslice normal force and shear force along the critical — slip surface. — — Relevant artifacts in Appendix

— Case Study Name: Projectile — — Problem being solved: A system is needed to efficiently and correctly predict — the landing position of a projectile. — — Relevant artifacts in Appendix

— Case Study Name: GamePhys — — Problem being solved: Many video games need physics libraries that simulate — objects acting under various physical conditions, while simultaneously being — fast and efficient enough to work in soft real-time during the game. — Developing a physics library from scratch takes a long period of time and is — very costly, presenting barriers of entry which make it difficult for game — developers to include physics in their products. — — Relevant artifacts in Appendix

The majority of the aforementioned case studies were developed to solve real problems, though there are a couple of exceptions. **NOTE: trying to find a good way to say 'these are not just things we cooked up to make Drasil look good'

The NoPCM case study was created as a software family member for the SWHS case study. It was manually written, removing all references to PCM and thus remodeling the system.

The Projectile case study, however, was the first example of a system created solely in Drasil (there was no manually created version to compare and contrast). As such, it will not be referenced often until DRASILSECTION since it did not inform Drasil's design or development until much further in the process. The Projectile case study was created so we'd have a simple, understandable example for a general audience (it requires, at most, a high-school level understanding of physics).

With our carefully selected case studies in hand we were able to begin our practical approach to finding and removing redundancies.

3.2 Breaking down artifacts

As noted earlier, for our approach to work we must understand exactly what each of our artifacts are trying to say and to whom. By selecting our case studies from those developed using common artifact templates, we have given ourselves a head start on that process, however, there is still much work to be done.

To start, we look at the Software Requirements Specification (SRS). The SRS (or some incarnation of it) is one of the most important artifacts for any software project as it specifies what problem the software is trying to solve. There are many ways to state this problem, and [SmithEtAl] have given us a strong recommendation of what to use as a starting point in their template. [Figure] shows the table of contents for an SRS using the [SmithEtAl] template.

[Figure showing the ToC of SmithEtAl template]

With the structure of the document in mind, let us look at several of our case studies' SRS documents to get a deeper understanding of what each section truly represents. [Figure] shows the first section of [one/several of our case studies, rest in Appendices]. [The case studies themselves will be introduced in more detail in later sections, but keep in mind at this point, we don't care about the superficial differences. – Not exactly true if we introduce them in previous section] We are strictly looking for patterns! Patterns will give us insight into the root of *what* is being said in each section.

[Figure showing the Ref Section of at least 3 (preferably 4-6) case studies, may need to be split into multiple, or have them somehow cropped/overlayed/arranged to be not terrible to look at - TBD]

Looking at the Table of Symbols, Table of Units, and Table of Abbreviations and

Acronyms from [Previous Figure] we can see that, barring the table values themselves, they are almost identical. The Table of Symbols is simply a table of values, essentially akin to a glossary, specific to the symbols that appear throughout the rest of the document. For each of those symbols, we see the symbol itself, a brief description of what that symbol represents, and the units it is measured in (if applicable). Similarly, the Table of Units lists the Systme International dUnits (SI) Units used throughout the document, their descriptions, and the SI name. Finally, the table of Abbreviations and Acronyms merely lists the abbreviations and their full forms, which are essentially the symbols and their descriptions.

The reference section of the SRS provides a lot of knowledge, in a very straightforward and organized manner. The basic units provided in the table of units give a prime example of fundamental / global knowledge. Nearly any system involving physical quantities will use some of these units. On the other hand, the table of symbols provides system/problem-domain specific knowledge that will not be useful across unrelated domains. For example, the stress distribution factor (J) from GlassBR may appear in several related problems, but would be unlikely to be seen in something like SWHS/NoPCM or Projectile. Finally, acronyms are very context-dependant. They are often specific to a given domain and, without a coinciding definition, it can be very difficult for even the target audience to understand what they refer to. Within one domain, there may be several acronyms meaning different things (for example: PM can refer to Product Manager, Project Manager, Program Manager, etc).

From the Reference section alone,

- Practical approach to design - Let's use some case studies - Understanding what's really going on

3.3 Artifact Summary

See Table [NUM] for a summary of each software artifact's 'what', 'who', and 'how'.

TABLE — Artifact — Who (Audience) — What (Problem) — How (Specific vs boilerplate) —

3.4 Patterns and repetition and patterns and repetition – (OR – Repeating patterns and patterns that repeat –)

From the above sections, we see many emerging patterns in our software artifacts. Ignoring, for now, the organizational patterns from the [SmithEtAl] templates we can already see simple patterns emerging. For example, we see the same concept being introduced in multiple areas within a single artifact and across artifacts in a project. [Example from one of the figures in the previous section. Preferably something like a DD or TM that shows up within a single doc multiple times]. We also see patterns of commonality across software family members (The SWHS and NoPCM case studies) as they have been developed to solve similar, or in our case nearly identical, problems.

- inter-project (repetition throughout different views + other patterns.) vs intra-project knowledge (repetition across projects/family members, minor modifications, but fundamentally the same + other patterns.) - Hint at chunkifying/parceling out the fundamental (system/view-agnostic) knowledge vs the specific knowledge

3.5 Organizing knowledge - a fluid approach

**Subsec roadmap: - We see the patterns above, we can generalize a lot of that - Direct repetition (copy-paste) vs indirect repetition (view-changes) require us to pull together knowledge from all artifacts into one place - Some can be derived automatically, the rest must be explicitly stated - We need to create a categorization system (hint at chunks) that is both robust and extensible to cover a wide variety of use cases. - Finally the templates give us structure

**NOTE: Under the hood section should explain the process of how we determined what we needed to do. What we ended up doing should come in the following section(s) - no 'real' implementation details, only conceptual stuff here.

3.6 The seeds of Drasil

**Subsec roadmap: – Summarize the above subsections and lead into next section – Add relevant information that doesn't quite fit above and isn't implementation related – 'Relevant buckshot section'

Chapter 4

Drasil

**Section Roadmap: – This is where the real meat of Drasil is discussed (implementation details) – Intro to our knowledge-capture mechanisms - Chunks/hierarchy - Break down each with examples from the case studies. - Look for 'interesting' examples (synonyms, acronyms, complexity, etc.) – Intro to the DSL - Captured knowledge is useless without the transformations/rendering engine - DSL for each softifact

4.1 What Drasil is and isn't

* Basically just restating some things from the intro in more depth - Not a silver bullet - Built around a specific set of assumptions, for a particular class of problems - NOT an ontology / ontology builder

4.2 Our Ingredients: Organizing and Capturing Knowledge

- Organization of knowledge implies a need for knowledge-capture mechanisms at different levels (different levels of abstraction / specificity) - segues right into chunks/hierarchy - project-specific vs DB of knowledge - Make it clear this is NOT an ontology - Look for interesting examples (synonyms, acronyms, complexity) from case studies and refer to them here (again if was covered in Organizing Knowledge)

4.3 Recipes: Codifying Structure

- Organized knowledge is fine, but is essentially just a collection of (collections of) definitions. Pretty meaningless on its own so we need the structure (in our case from the templates / case studies) to have meaning. - Each softifact has its own recipe for combining knowledge - As we consider softifacts "views" of the knowledge, we need to combine/transform/manipulate the knowledge into a meaningful form for the given view - ex: Math formula for human-readable doc, Function/method for code (show examples). - Recipes define the "how" and "where" of putting together the knowledge. The rendering engine reads the recipe and follows its instructions.

4.4 Cooking it all up: Generation/Rendering

- Recipes are little programs - Each recipe can be rendered a number of ways, based on parameters fed to the generator. - Implicit parameters vs explicit: Ex. an SRS will always be rendered based on the recipe, but its output will either be LaTeX or

HTML based on an explicit choice. Implicit params fed to gen table of symbols/A&A / ToU.

4.5 Reimplementing the case studies in Drasil

[TODO: move this / rearrange it into a "Iteration and refinement" section that doesn't discuss the specific results with the case studies. —DS] - Practical approach to iron out kinks / find holes in Drasil - Find places to improve upon the existing case studies - update as you go mindset - Observe the amount of effort required to correct errors - show examples - Most of the code well show off should be in here.

Chapter 5

Results

5.1 Originals vs. Reimplementations

Link to original case studies and reimplementations Highlight key (important) changes with explanations Show off major errors/oversights Original softifacts LoC vs number of Drasil LoC to reimplement (and compare the output of Drasil - multiple languages, etc.)

5.2 Pervasive Bugs

- Sounds bad, but actually really good. - If a bug shows up in more places, it's more likely to be found. - Softifacts remain consistent, even if wrong.

5.3 Design for change

GlassBR /1000 example

5.4 Usability

One of Drasil's biggest issues is that of usability. Unless one reads the source code or has a member of the Drasil team working with them, it can be incredibly difficult, or even impossible, to create a new piece of software in Drasil.

As seen in the examples from [SECTION], while the recipe language is fairly readable, the knowledge-capture mechanisms are arcane and determining which knowledge has already been added to the database can be very difficult. As our living knowledge-base expands, this will become even more difficult, particularly for those concepts with many possible names.

- As the above mentions, not great, but CS students / summer interns picked it up fast enough to make meaningful changes in a short time period.

Chapter 6

Future Work

[This can probably be a section at the end of results / conclusion instead? —DS]

Development of Drasil is ongoing and the framework is still being iterated upon to date. In this section we present areas we believe have room for improvement along with plans for additional features to be added in the long-term.

6.1 Typed Expression Language

6.2 Model types

6.3 Usability

As mentioned in the results section, usability remains a great area for improvement for Drasil. Work to create a visual front-end for the framework has been planned and we hope to eventually get to the point of usability being as simple as drag-and-drop or similar mechanisms.

Work on usability will address each of the core areas of development with Drasil: knowledge-capture of system-agnostic information, knowledge-capture of system-specific information, and recipe creation and modification.

6.4 Many more artifacts

6.5 More display variabilities

6.6 More languages

Chapter 7

Conclusion

[Should Future work be collapsed into here? —DS]

Every thesis also needs a concluding chapter

Appendix A

Your Appendix

Your appendix goes here.

Appendix B

Long Tables

This appendix demonstrates the use of a long table that spans multiple pages.

Col A	Col B	Col C	Col D
A	B	C	D
A	B	C	D
A	B	C	D
A	B	C	D
A	B	C	D
A	B	C	D
A	B	C	D
A	B	C	D

Continued on the next page

Continued from previous page

Col A	**Col B**	**Col C**	**Col D**
A	B	C	D
A	B	C	D
A	B	C	D
A	B	C	D
A	B	C	D
A	B	C	D
A	B	C	D
A	B	C	D
A	B	C	D
A	B	C	D
A	B	C	D
A	B	C	D

Bibliography

Hudlicka, E. (2002). This time with feeling: Integrated model of trait and state effects on cognition and behavior. *Applied Artificial Intelligence*, **16**(7-8), 611–641.

Popescu, A., Broekens, J., and van Someren, M. (2014). GAMYGDALA: An emotion engine for games. *Affective Computing, IEEE Transactions on*, **5**(1), 32–44.