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)?????

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), 11 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.

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" [6]. 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 [1–3, 8] 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.

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.

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.

[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 explore previous efforts in automating software artifact generation, as well as take a look at the current state of Model Driven Engineering and scientific knowledge ontologies.

2.1 Software Artifact Reuse and Generation

Previous attempts at generating software artifacts were primarily focused on artifact reuse. One aim of these approaches was to remove the burden of replicating some artifacts.

We will start by looking at Compendia [**?**]

Previous attempts at automating / reducing the artifact burden.

- Compendia - Trying to solve the problem of reproducibility - Fits with goals of certification - focused on good science and being able to re-run experiments exactly - Not focused on DDD or its benefits, moreso

- 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.

[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

- 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're 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.

-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 KNOWLEDGE-BASED SOFTWARE ENGINEERING (KBSE)

Knowledge-Based Software Engineering (KBSE) was originally defined as an "engineering discipline that includes the integration of knowledge into software systems in order to solve complex problems, which would normally require rather high level of human expertise" [5]. 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" [4].

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 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.

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.

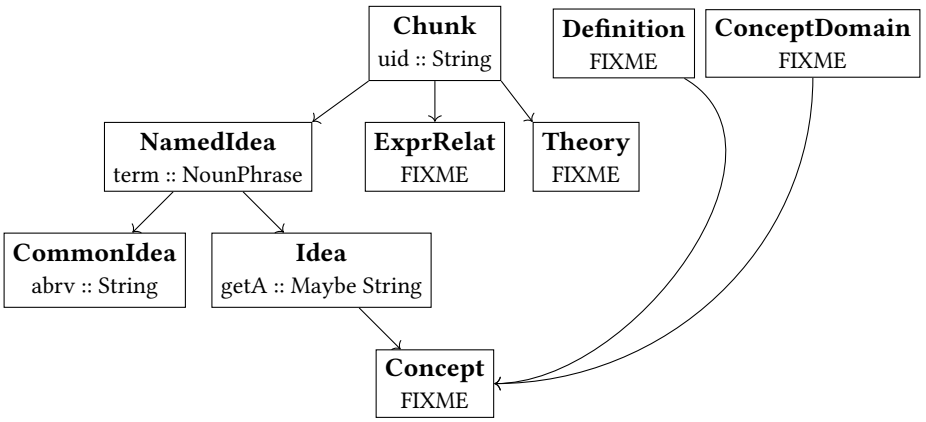

Fig. 3. Chunk hierarchy in Drasil Today

We borrow, and expand, the idea of *Chunks* from Literate Programming (LP) [7] 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.

Once we have properly captured the relevant knowledge, we shouldn't 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 4, the view of our knowledge can change without affecting our encoding. To project these views, we use transformations.

Transformations represent the different 'views' of the knowledge we want based on how abstract 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.

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.

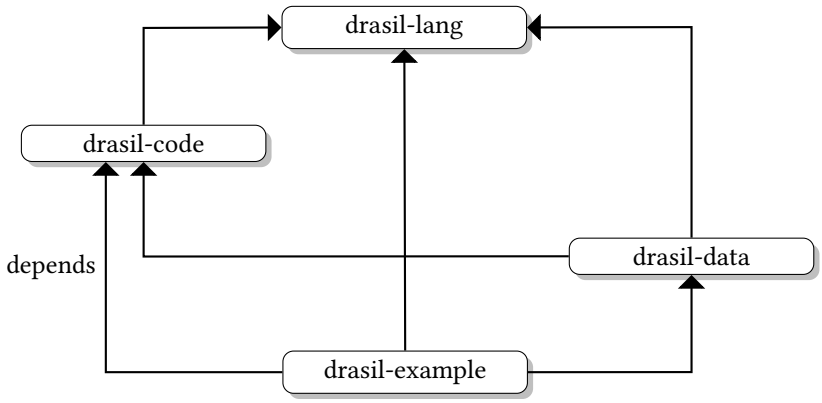

Fig. 4. Drasil package dependencies

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 generation, are stored in a package named *drasil-lang* and code-generation related components are in *drasil-code*. 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

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`

`Language.Drasil`. This package also exports a second module with functionality targeted to developers of Drasil known as `Language.Drasil.Development`.

`Language.Drasil` presently contains the expression DSL, document layout DSL, printing DSL, and the generator back-end. Every other *drasil-** package relies on, and builds off, this core. Some of these, notably the printing DSL and generator back-end, may be moved to their own packages in the near future.

With `Language.Drasil` alone we can capture knowledge and generate 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-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 [**?**], to give us the ability to target multiple languages – C++, C#, Objective C, Java, Lua, and Python.

5.2.3 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.

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

5.3 Drasil Today

- Sentence and Document - Explain the chunk hierarchy (refer to Section 4.1 figure) - Data.Drasil !FIGURE! Knowledge areas we've 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)

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] Center for Devices and Radiological Health, CDRH. General principles of software validation; final guidance for industry and FDA staff. Tech. rep., US Department Of Health and Human Services Food and Drug Administration Center for Devices and Radiological Health Center for Biologics Evaluation and Research, York, England, January 2002.

[2] 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.

[3] 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.

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

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

[6] 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.

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

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