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 them is tedious and costly.

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. Creation of our tool – Drasil – will facilitate this automation process using a knowledge-based approach to Software Engineering.

METHOD: Use grounded theory in the creation of a tool for software artifact generation. Start by analyzing the artifacts themselves from several case studies to understand what (semantically) is being said in each. Generate all the things! Capture the underlying knowledge and apply transformations to create each of the requisite artifacts. Captured knowledge can be re-used across projects as it represents the "science". Maintenance involves updating the captured knowledge or transformations as necessary.

RESULTS: Case studies – GlassBR to show capture and transformation. SWHS and NoPCM for reuse (Something about Kolmogorov complexity / MDL here?).

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 (April 2018), 6 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 front-loading the costs of maintenance and rolling them into the development cycle, we can save time and money in the long run.

Authors' addresses: Dan Szymczak, McMaster University, 1280 Main St. W., Hamilton, ON, L8S 4K1, Canada, szymczdm@mcmaster.ca; Jacques Carette; Spencer Smith.

1049-331X/2018/4-ART $15.00
https://doi.org/0000001.0000001

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 satisïňĄes accepted, well deïňĄned and measurable criteria" [?]. 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 (!CITE STANDARDS!) there are several common required artifacts:

- Requirements document
- !OTHER JUNK!

- The cost(s) of re-certification: - Expensive! $$$ and time

1.2 Document-Driven Design

- Document-driven design and its advantages/disadvantages. - Quality improvements - Large time investment for initial artifact creation - Large time investment for every (non-trivial) update - Often artifacts fall out of sync with each other and are inconsistent - All but necessary for certification - Need to overcome the disadvantages somehow. - Most obvious solution would be to automate if possible.

1.3 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.

2 BACKGROUND

- We are not the first to try and deal with certification / artifact creation.

2.1 Previous efforts

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.

3 A RATIONAL ANALYSIS OF SOFTWARE ARTIFACTS

- This section exists to show how we get from problem to solution. - We introduce our case studies in a bit more depth here - GlassBR - what it’s for, if it’ll - SWHS and NoPCM - Program 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.

3.1 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?

3.2 The structure of knowledge

- As shown above, in the artifacts we see different ways of representing what are semantically the same things. - There are many pieces to a single piece of underlying knowledge. - Each of these components tells us about the way our knowledge can be used and transformed. - Take a look at one particular example across artifacts in GlassBR. Has: - identifier (label) - symbolic (theory) representation (shown in SRS, DD - !FIGURE!) - symbolic (implementation) representation (!FIGURE!) - Concise natural language description (term) - Verbose natural language description (definition) - equation (!FIGURE!– show as math and code; same thing in a transformed view) - units? - constraints (Really relevant. Again show !FIGURE!as math and code)

- Similar examples crop up all over the artifacts, some with the same depth of information, and some without. - Those lacking depth still contain some of the pieces - term, defn, etc.

4 KNOWLEDGE-BASED SOFTWARE ENGINEERING (KBSE)

- Start with a note on terminology? How "knowledge" means a structured encoding that allows for automatic reuse vs. natural language encoded information which does not?

- The main ideas: - Capture underlying science knowledge in a meaningful way - Reuse wherever appropriate (inter- and intra-project) thanks to knowledge-base and transformations. - Maintain a single source of knowledge and generate the software artifacts from there. - Scope - Well-understood domains - There is a solid theoretical underpinning (science/math). - We can explain it to a computer! (not as easy as you'd think) - Particularly we focus on KBSE for Scientific Computing Software (SCS) as it is rich in knowledge

4.1 Capturing Knowledge

- Based on Section 3.2 we can create a knowledge-capture mechanism for encoding the science into a computer-usable form.

- We like chunks *(nod to LP)* for knowledge-capture. - A chunk is a labeled piece of information. - !FIGURE!: Chunk hierarchy in Drasil (to be explained more in the coming sections) mimics the structure mentioned in Section 3.2 - We have to capture all of the information surrounding a piece of knowledge to create our artifacts, regardless of whether that information is relevant to any one particular artifact. - Once knowledge is properly captured, we shouldn't have to capture it again if we want to reuse it in a different project.

4.2 (Re-)Using Knowledge

- Most obvious benefit -> no more copy/paste! Just reuse the chunk you need. - Transformations - Represent different 'views' of the knowledge based on how abstract, what audience, etc. - Translate the knowledge into its requisite form (eqns, descriptions, code) - Variabilities -> different projects in the same family. - Easy to specialize to different family members - Example: SWHS vs NoPCM

!FIGURE! Show portion of each SRS, one similarity, one difference? - Requires a framework / tool support to automate rendering of these transformations, otherwise it is even more work for humans.

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 - 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

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 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!?)

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.

6.6 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.

6.7 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)

6.8 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?

6.9 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.

6.10 Pervasive Bugs

- Mistakes in knowledge can be found in all artifacts - more likely to be caught! - Easy to track down errors (smart error messages point to the exact chunk causing the problem).

7 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

8 CONCLUSION

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