Generating Software for Well-Understood Domains

Jacques Carette
Department of Computing and Software, McMaster University, 1280 Main Street West, Hamilton, Ontario, L8S 4L8, Canada

Spencer W. Smith
Department of Computing and Software, McMaster University, 1280 Main Street West, Hamilton, Ontario, L8S 4L8, Canada

Jason Balaci
Department of Computing and Software, McMaster University, 1280 Main Street West, Hamilton, Ontario, L8S 4L8, Canada

Abstract

Current software development is often quite code-centric and aimed at short-term deliverables, due to various contextual forces (such as the need for new revenue streams from many individual buyers). We're interested in software where different forces drive the development. **Well understood domains** and **long-lived software** provide one such context.

A crucial observation is that software artifacts that are currently handwritten contain considerable duplication. By using domain-specific languages and generative techniques, we can capture the contents of many of the artifacts of such software. Assuming an appropriate codification of domain knowledge, we find that the resulting de-duplicated sources are shorter and closer to the domain. Our encodings also seem to do well with respect to increasing traceability and change management. We're hopeful that this could lead to long-term productivity improvements for software that fit this context.

2012 ACM Subject Classification Software and its engineering → Application specific development environments; Software and its engineering → Requirements analysis; Software and its engineering → Specification languages; Software and its engineering → Automatic programming

Keywords and phrases code generation, document generation, knowledge capture, software engineering

Digital Object Identifier 10.4230/OASIcs.CVIT.2016.23

1 The Context

Not all software is the same. In fact, there is enough variation in the context in which developers create various software products to warrant exploring and using different processes, depending on the forces [1] at play. Here we explore two such forces: "well understood" and "long lived".

1.1 "Well-understood" software?

▶ **Definition 1.** *A software domain is **well understood** if*

1. *its Domain Knowledge (DK) [5] is codified,*
2. *the computational interpretation of the DK is clear, and*
3. *writing code to perform said computations is well understood.*

By *codified*, we mean that the knowledge exists in standard form in a variety of textbooks. For example, many engineering domains use ordinary differential equations as models, the quantities of interest are known, given standard names and standard units. In other words, standard vocabulary has been established over time and the body of knowledge is uncontroversial.

We can refine these high level ideas, using the same numbering, although the refinement should be understood more holistically.

1. Models in the DK *can be* written formally.
2. Models in the DK *can be* turned into functional relations by existing mathematical steps.
3. Turning these functional relations into code is an understood transformation.

Most importantly, the last two parts deeply involve *choices*: What quantities are considered inputs, outputs and parameters to make the model functional? What programming language? What software architecture data-structures, algorithms, etc.?

In other words, *well understood* does not imply *choice free*. Writing a small script to move files could just as easily be done in Bash, Python or Haskell. In all cases, assuming fluency, the author's job is straightforward because the domain is well understood.

1.2 Long-lived software?

For us, long-lived software [13] is software that is expected to be in continuous use and evolution for 20 or more years. The main characteristic of such software is the *expected turnover* of key staff. This means that all tacit knowledge about the software will be lost over time if it is not captured.

1.3 Documentation

Well understood also applies to **documentation** aimed at humans [16]. Explicitly:

1. The meaning of the models is understood at a human-pedagogical level, i.e. it is explainable.
2. Combining models is explainable. Thus, the act of combining models must simultaneously operate on mathematical representations and on explanations. This requires that English descriptions also be captured in the same manner as the formal-mathematical knowledge.
3. Similarly, the refinements steps that are performed due to making software oriented decisions should be captured with a similar mechanism, and also include English explanations.

▶ **Definition 2.** *We dub these* ***triform theories****, as a nod to* biform theories *[8]. They are a coupling of a concepts*

1. *an axiomatic description,*
2. *a computational description, and*
3. *an English description*

of a concept.

1.4 Software artifacts

Software currently consists of a whole host of artifacts: requirements, specifications, user manual, unit tests, system tests, usability tests, build scripts, READMEs, license documents, process documents, as well as code.

Whenever appropriate, we use standards and templates for each of the generated artifacts. For requirements, we use a variant [19] of the IEEE [10] and Volere templates [18].

1.5 Instances of our context

When are these well understood and long lived conditions fulfilled? One example is *research software* in science and engineering. While the results of running various simulations is

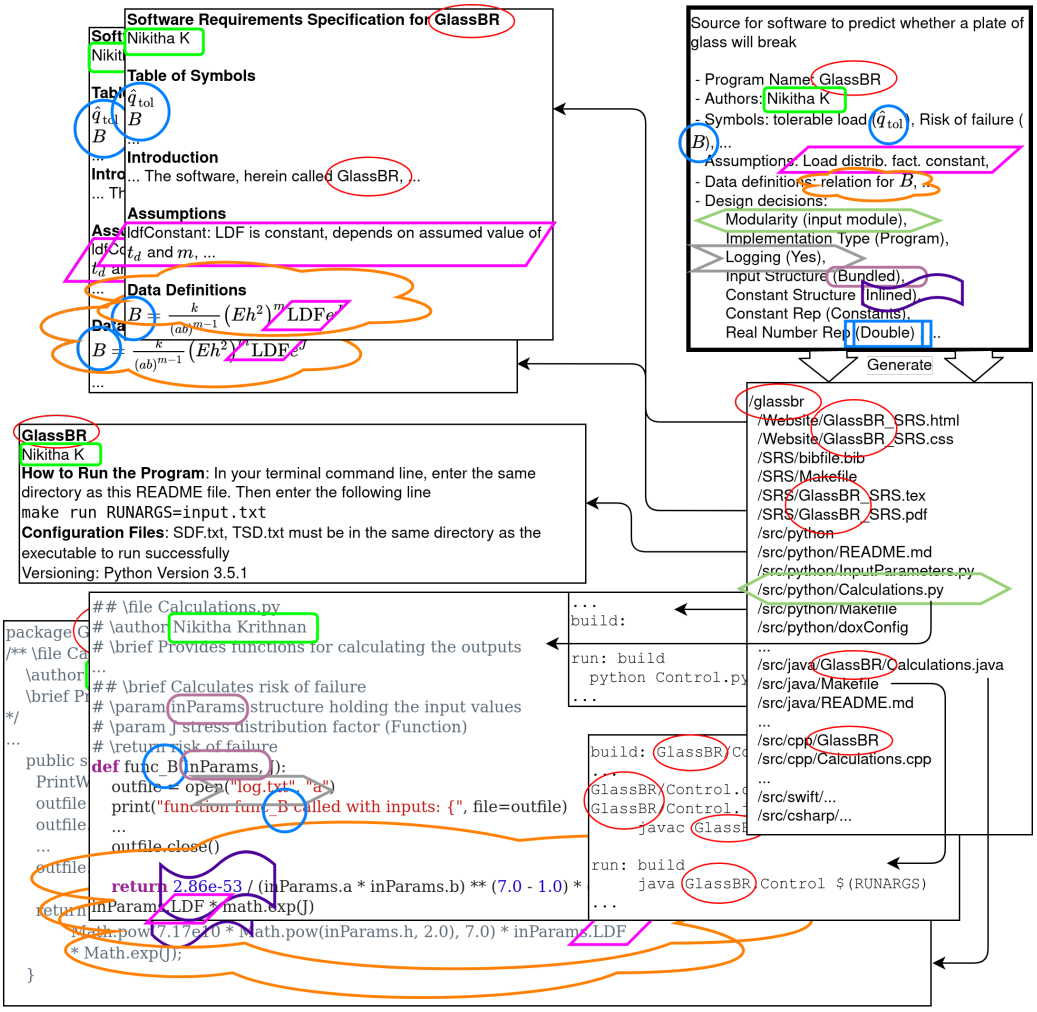

Figure 1 Colors and shapes mapping from captured domain knowledge to generated artifacts.

entirely new, the underlying models and how to simulate them are indeed well known. One particularly long-lived example is embedded software for space probes (like Pioneer 10). Another would be ocean models, such as NEMO [9] which can trace its origins to OPA 8 [15]. In fact, most sub-domains of science and engineering harbour their own niche long-lived software.

2 An Example

We have built infrastructure[1] to carry out these ideas. It consists of 60KLoc of Haskell implementing a series of interacting Domain Specific Languages (DSLs) for knowledge encodings, mathematical expressions, theories, English fragments, code generation and document generation. A full description would take too much space. Instead, we provide an illustrative example.

[1] `https://github.com/JacquesCarette/Drasil`

2.1 GlassBR

We will focus on information capture and the artifacts we can generate. For concreteness, we'll use a single example from our suite: GlassBR, used to assess the risk for glass facades subject to blast loading. The requirements are based on an American Standard Test Method (ASTM) standard [2, 3, 4]. GlassBR was originally a Visual Basic code/spreadsheet created by colleagues in a civil engineering research lab. We added their domain knowledge to our framework, along with recipes to generate relevant artifacts. Not only can we generate code for the necessary calculations (in C++, C#, Java, Python and Swift), we also generated documentation that was absent in the original (Software Requirements Specification, doxygen, README.md and a Makefile). Moreover, our implementation is actually a family of implementations, since some design decisions are explicitly exposed as changeable variabilities, as described below.

The transformation of captured domain knowledge is illustrated in Figure 1. This is read starting from the upper right box. Each piece of information in this figure has its own shape and colour (orange cloud, pink lozenge, etc). It should be immediately clear that all pieces of information reappear in multiple places in the generated artifacts. For example, the name of the software (GlassBR) ends up appearing more than 80 times in the generated artifacts (in the folder structure, requirements, README, Makefile and source code). Changing this name would traditionally be extremely difficult; we can achieve this by modifying a single place, and regenerating.

The first box shows the directory structure of the currently generated artifacts; continuing clockwise, we see examples of Makefiles for the Java and Python versions, parts of the fully documented, generated code for the main computation in those languages, user instructions for running the code, and the processed LaTeX for the requirements.

The name GlassBR is probably the simplest example of what we mean by *duplication*: here, the concept "program name" is internally defined, and its *value* is used throughout.

In general, we capture more complex knowledge. An example is the assumption that the "Load Distribution Factor" (LDF) is constant (pink lozenge). If this needs to be modified to instead be an input, the generated software will now have LDF as an input variable.

We also capture design decisions, such as whether to log all calculations, whether to inline constants rather than show them symbolically, etc. These different pieces of knowledge can also be reused in different projects.

2.2 The Steps

We describe an "idealized process" that we could have used to produce GlassBR, following Parnas' idea of faking a rational design process [17].

Understand the Program's Task Compute the probability that a particular pane of (special) glass will break if an explosive is detonated at a given distance. This could be in the context of the glass facade for a building.

Is it well understood? The details are extensively documented in [2, 3, 4].

Record Base Domain Knowledge A recurring idea is the different types of `Glass`:

Concept	Term (Name)	Abbrev.	Domain
`fullyT`	Fully Tempered	FT	`[Glass]`
`heatS`	Heat Strengthened	HS	`[Glass]`
`iGlass`	Insulating Glass	IG	`[Glass]`
`lGlass`	Laminated Glass	LG	`[Glass]`
`glassTypeFac`	Glass Type Factor	GTF	`[Glass]`

The "Risk of Failure" is definable:

Label	Risk of Failure
Symbol	B
Units	Unitless
Equation	$B = \frac{k}{(ab)^{m-1}}(Eh^2)^m LDF e^J$
Description	B is the Risk of Failure (Unitless) k is the surface flaw parameter ($\frac{m^{12}}{N^7}$) a & b are the plate length & width (m) m is the surface flaw parameter ($\frac{m^{12}}{N^7}$) E is the modulus of elasticity of glass (Pa) h is the minimum thickness (m) LDF is the load duration factor (Unitless) J is the stress distribution factor (Unitless)
Source	[2], [4]

Some concepts, such as those of *explosion*, *glass slab*, and *degree* do not need to be defined mathematically – an Enlish description is sufficient.

The descriptions in GlassBR are produced using an experimental language using specialized markup for describing relations between knowledge. For example, the goal of GlassBR ("Predict-Glass-Withstands-Explosion") is to "Analyze and predict whether the *glass slab* under consideration will be able to withstand the **explosion** of a certain **degree** which is calculated based on *user input*.", where italicized names are "named ideas", and bold faced names are "concept chunks" (named ideas with a domain of related ideas). We call this goal a "concept instance" (a concept chunk applied in some way). This language lets us perform various static analyses on our artifacts.

This doesn't build a complete ontology of concepts, as we have not found that to be necessary to generate our artifacts. In other words, we define a *good enough* fraction of the domain ontology.

The most important results of this phase are descriptions of *theories* that link all the important concepts together. For example, the description of all the elements that comprise Newton's Law $F = ma$, i.e. what is a force, a mass, acceleration (and how is it related to velocity, position and time), what are the units involved, etc.

Define the characteristics of a good solution For example, one of our outputs is a probability, which means that the output should be checked to be between 0 and 1. This can result in assertion code in the end program, or tests, or both. We do not yet support full *properties* [6], but that is indeed the logical next step.

Record basic examples For the purposes of testing, it is always good to have very simple examples, especially ones where the correct answer is known a priori, even though the simple examples are considered "toy problems". They provide a useful extra check that the narrative is coherent with our expectations.

Specialization of theories In general, the theories involved will be much more general than what is needed in any given example. For example, Newton's Laws are encoded in their vector form in n dimensions, and thus specialization is necessary.

In the GlassBR example, the thickness parameter is not free to vary, but must take one of a specific set of values. The rationale for this specialization is that manufacturers have standardized the glass thickness they will provide. (This rationale is also something we capture.)

Most research software examples involve significant specialization, such as converting partial differential equations to ordinary, elimination of variables, use of closed forms instead of implicit equations, and so on. Often specialization, driven by underlying assumptions, enable further specializations, so that this step is really one of *iterative refinement*.

Create a coherent narrative Given the outputs we wish to produce, such as the probability that a glass slab will withstand an explosion, we need to ensure that we can go from all the given inputs to the desired output, by stringing together various definitions given in a computational manner. In other words, we need to ensure that there exists a deterministic path from the inputs we are given, through the equations we have, to all of the outputs we've declared we are interested in.

For this example, the computations are all quite simple, but in general this might involve solving ordinary differential equations, computing a solution to an optimization problem, etc.

Make code level choices From a *deterministic model of the solution*, it should be possible to output code in a programming language. To get there, we still need to make a series of choices.

Drasil lets you choose output programming language(s) (see Figure 2), but also how "modular" the generated code is, whether we want programs or libraries, the level of logging and comments, etc. Here we show the actual code we use for this, as it is reasonably direct.

```
code :: CodeSpec
code = codeSpec fullSI choices allMods

choices :: Choices
choices = defaultChoices {
 lang = [Python, Cpp, CSharp, Java, Swift],
 modularity = Modular Separated,
 impType = Program, logFile = "log.txt",
 logging = [LogVar, LogFunc],
 comments = [CommentFunc, CommentClass, CommentMod],
 doxVerbosity = Quiet,
 dates = Hide,
 onSfwrConstraint = Exception, onPhysConstraint = Exception,
 inputStructure = Bundled,
 constStructure = Inline, constRepr = Const
}
```

Figure 2 Code level choices for GlassBR (compare bold box in Figure 1).

Create recipes to generate artifacts The information collected in the previous steps form the core of the various software artifacts normally written. To perform this assembly, we write programs that we dub *recipes*. We have DSLs for creating specifications, code [14], dependency diagrams, Makefiles, READMEs and a log of all choices used.

While we currently generate `Makefiles`, a good example of a DSL for specifying build scripts, our ideas are more properly compared with the kind of thinking behind *Rattle* [20].

Finally, we can put the contents created via the previous steps together and **generate everything**:

```
main :: IO()
main = do
  setLocaleEncoding utf8
  gen (DocSpec (docChoices SRS [HTML, TeX]) "GlassBR_SRS") srs printSetting
  genCode choices code
  genDot fullSI
  genLog fullSI printSetting
```

3 An Idealized Process

It is useful to summarize the *idealized process* we used above.

1. Have a task to achieve where *software* can play a central part in the solution.
2. Verify that the underlying problem domain is *well understood*.
3. Describe the problem:
   a. Find the base knowledge (theories) in the pre-existing library or, failing that, write it if it does not yet exist, for instance the naturally occurring known quantities and associated constraints.
   b. Describe the characteristics of a good solution.
   c. Come up with basic examples (to test correctness, intuitions, etc).
4. Describe, by successive refinement transformations, how the problem description can be turned into a deterministic[2] input-output process.
   a. Some refinements will involve *specialization* (e.g. from n-dimensional to 2-dimensional, assuming no friction, etc). These *choices* and their *rationale* need to be documented, as a crucial part of the solution. Whether these choices are (un)likely to change in the future should also be recorded.
   b. Choices tend to be dependent, and thus (partially) ordered. *Decisions* frequently enable or reveal downstream choices.
5. Assemble the ingredients into a coherent narrative.
6. Describe how the process from step 5 can be turned into code. Many choices can occur here as well.
7. Turn the steps (i.e., from items 4 and 6) into a *recipe*, aka program, that weaves together all the information into a variety of artifacts (documentation, code, build scripts, test cases, etc). These can be read, or executed, or ... as appropriate.

The fundamental reason for focusing on *well-understood* software is to make the various steps in this process feasible. Another enabler is a *suitable* knowledge encoding. Rather than define this a priori, we have used a *bottom up* process to capture "good enough" ontologies to get the job done.

What is missing in the above description is an explicit *information architecture* of each of the necessary artifact. In other words, what information is necessary to enable the mechanized generation of each artifact? It turns out that many of them are quite straightforward.

Note that in many software projects, steps 1 and 3 are skipped; this is part of the **tacit knowledge** of a lot of software. Our process requires that this knowledge be made explicit, a fundamental step in *Knowledge Management* [7].

[2] A current meta-design choice.

4 Concluding Remarks

Others [5] have already remarked that for most software, the *domain knowledge* is the slowest moving part of the requirements. We add to this the ideas that for certain kinds of software, there is a lot of *knowledge duplication* amongst the artifacts, much of which is traceable to domain knowledge. Thus, in well-understood domains, it should be feasible to record the domain knowledge "once", and then write recipes to generate instances based on refinements. For long-lived software, this kind of up-front investment should be worthwhile.

In other words, if we capture the fundamental domain knowledge of specific domains (such as mechanics of rigid body motion, dynamics of soil, trains, etc), most later development time can be spent on the *specifics* of the requirements of a specialized application that may well contain novel ideas at the refinement or recipe level.

In Drasil, as a side effect of organizing things as we have, we obtain traceability and consistency, by construction. Tracking where we use each concept, i.e. traceability, is illustrated in Figure 1. We obtain consistency in our documentation by generating items such as the table of symbols from those symbols actually used in the document, and whose definition is automatically extracted from the base knowledge.

There are further ideas that co-exist smoothly with our framework, most notably software families and correctness, particularly correctness of documentation. As we generate the documentation in tandem with, and using the same information as, the source code, these will necessarily be synchronized. Errors will co-occur in both. This is a feature, as they are more likely to be caught that way.

We've also noticed that we can more easily experiment with "what if" scenarios, which make it easy to understand the ramifications of proposed changes.

We expect to use formal ontologies as we implement more coherence checks on the domain knowledge itself. For example, it should not be possible to associate a *weight* attribute to the concept *program name*, but only to concepts that are somehow "physical". Perhaps a large ontology in the style of Cyc [12] would help.

That work of this scale is possible has strongly been influenced by Eelco Visser's grand projects. His work on the Spoofax language workbench [11] made it clear that many different artifacts can be generated. He was braver than us: his WebDSL [22], at the heart of researchr [21], is indeed expected to be long lived, but certainly was not created for a well-understood domain! The domain analysis of web applications is a non-trivial contribution of WebDSL. Whether it is product lines or program families, weaving them from DSLs is also something he preached [23]. We never got a chance to present Drasil to Eelco, but we hope that it would have resonated with him.

References

1. Christopher Alexander. *A pattern language: towns, buildings, construction.* Oxford university press, 1977.
2. ASTM. Standard practice for determining load resistance of glass in buildings, 2009.
3. ASTM. Standard practice for specifying an equivalent 3-second duration design loading for blast resistant glazing fabricated with laminated glass, 2015.
4. W. Lynn Beason, Terry L. Kohutek, and Joseph M. Bracci. Basis for ASTME E 1300 annealed glass thickness selection charts. *Journal of Structural Engineering*, 124(2):215–221, February 1998.
5. Dines Bjørner. *Domain Science and Engineering.* Monographs in Theoretical Computer Science. An EATCS Series. Springer International Publishing, New York, 2021. doi:10.1007/978-3-030-73484-8.

6 Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 268–279, 2000.

7 Kimiz Dalkir. *Knowledge Management in Theory and Practice*. The MIT Press, Cambridge, Massachusetts, USA, 2nd edition, 2011.

8 William M. Farmer. Biform theories in chiron. In Manuel Kauers, Manfred Kerber, Robert Miner, and Wolfgang Windsteiger, editors, *Towards Mechanized Mathematical Assistants*, pages 66–79, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

9 Madec Gurvan, Romain Bourdallé-Badie, Jérôme Chanut, Emanuela Clementi, Andrew Coward, Christian Ethé, Doroteaciro Iovino, Dan Lea, Claire Lévy, Tomas Lovato, Nicolas Martin, Sébastien Masson, Silvia Mocavero, Clément Rousset, Dave Storkey, Simon Müeller, George Nurser, Mike Bell, Guillaume Samson, Pierre Mathiot, Francesca Mele, and Aimie Moulin. Nemo ocean engine, March 2022. doi:10.5281/zenodo.6334656.

10 IEEE. Recommended practice for software requirements specifications. *IEEE Std 830-1998*, 830:1–40, October 1998. doi:10.1109/IEEESTD.1998.88286.

11 Lennart C. L. Kats and Eelco Visser. The Spoofax language workbench: Rules for declarative specification of languages and IDEs. *ACM SIGPLAN Notices*, 45(10):444–463, October 2010. OOPSLA 2010.

12 Douglas B Lenat. Cyc: A large-scale investment in knowledge infrastructure. *Communications of the ACM*, 38(11):33–38, 1995.

13 Robyn Lutz, David Weiss, Sandeep Krishnan, and Jingwei Yang. Software product line engineering for long-lived, sustainable systems. In Jan Bosch and Jaejoon Lee, editors, *Software Product Lines: Going Beyond*, pages 430–434, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

14 Brooks MacLachlan, Jacques Carette, and Spencer S. Smith. Gool: Generic object-oriented language. In *Proceedings of 2020 SIGPLAM Workskop on Partial Evaluation and Program Manipulation (PEPM 2020)*. ACM, 2020. doi:https://doi.org/10.1145/3372884.3373159.

15 G. Madec, P. Delecluse, M. Imbard, and C. Levy. Opa 8 ocean general circulation model - reference manual. Technical report, LODYC/IPSL Note 11, 1998.

16 David Lorge Parnas. Precise documentation: The key to better software. In *The Future of Software Engineering*, pages 125–148. Springer, New York, 2011.

17 David Lorge Parnas and Paul C Clements. A rational design process: How and why to fake it. *IEEE transactions on software engineering*, (2):251–257, 1986.

18 Suzanne Robertson and James Robertson. *Mastering the Requirements Process*, chapter Volere Requirements Specification Template, pages 353–391. ACM Press/Addison-Wesley Publishing Co, New York, NY, USA, 1999.

19 W. Spencer Smith, Lei Lai, and Ridha Khedri. Requirements analysis for engineering computation: A systematic approach for improving software reliability. *Reliable Computing, Special Issue on Reliable Engineering Computation*, 13(1):83–107, February 2007. doi:10.1007/s11155-006-9020-7.

20 Sarah Spall, Neil Mitchell, and Sam Tobin-Hochstadt. Build scripts with perfect dependencies. *Proc. ACM Program. Lang.*, 4(OOPSLA), November 2020. doi:10.1145/3428237.

21 Elmer van Chastelet, Eelco Visser, and Craig Anslow. Conf.researchr.org: Towards a domain-specific content management system for managing large conference websites. In Jonathan Aldrich and Patrick Eugster, editors, *Companion Proceedings of the 2015 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications (SPLASH): Software for Humanity*, pages 50–51. ACM, 2015.

22 Eelco Visser. WebDSL: A case study in domain-specific language engineering. In Ralf Lämmel, Joost Visser, and João Saraiva, editors, *Generative and Transformational Techniques in Software Engineering II*, LNCS 5235. Springer, July 2008. GTTSE 2007.

23 Markus Voelter and Eelco Visser. Product line engineering using domain-specific languages. In *2011 15th International Software Product Line Conference*, pages 70–79. IEEE, 2011.