GOOL: A Generic Object-Oriented Language

(Short Paper)

Brooks MacLachlan
Department of Computing and Software
McMaster University
Hamilton, Ontario, Canada
maclachb@mcmaster.ca

Jacques Carette
Department of Computing and Software
McMaster University
Hamilton, Ontario, Canada
carette@mcmaster.ca

Spencer Smith
Department of Computing and Software
McMaster University
Hamilton, Ontario, Canada
smiths@mcmaster.ca

Abstract

We present GOOL, a Generic Object-Oriented Language. GOOL demonstrates that with the right abstractions, a language can capture the essence of object-oriented programs. GOOL generates human-readable, documented and idiomatic source code in multiple languages. In it, we can express common programming idioms and patterns. GOOL is an embedded DSL in Haskell that generates code in Python, Java, C#, and C++.

Keywords Code Generation, Domain Specific Language, Haskell, Documentation

1 Introduction

Java or C#? As languages, this is close to a non-question: the two are so similar that only ecosystem issues would be the deciding factor. Unlike the question "C or Prolog?", which is almost non-sensical, as the kinds of applications where each is well-suited are vastly different. But, given a single paradigm, such as Object-Oriented (OO), would it be possible to write a meta-language that captures the essence of writing OO programs? They generally all contain (mutable) variables, statements, conditionals, loops, methods, classes, etc.

OO programs written in different languages appear, superficially, quite dissimilar. But this is mostly due to syntactic differences. Are they so different in the utterances that one can make? Are OO programs akin to sentences in Romance languages (French, Spanish, Portugese, etc.) which, although syntactically different, are structurally very similar?

This is what we set out to explore. One non-solution is to pick a language and implement a translator to the others. It is feasible — say by engineering a multi-language compiler (such as gcc) to de-compile its Intermediate Representation (IR) into most of its input languages. The end-results would be wildly unidiomatic; roughly the equivalent of a novice in a (spoken) language "translating" word-by-word.

So we set out to design a meta-language that embodies the common semantic concepts of OO languages, encoded so that the necessary information for translation is present. This language is agnostic of the eventual target language – and free of the idiosyncratic details of any given language. In fact, with proper case, on can go even further and teach the translator about idiomatic patterns of each target language.

Trying to capture all the subtleties of each language is hopeless — akin to capturing the rhythm, puns, metaphors, similes, and cultural allusions of a sublime poem in translation. But programming languages are most often used for much more prosaic tasks: writing programs for getting things done. This is closer to translating technical textbooks, making sure that all of the meaningful material is preserved.

We want to capture the conceptual meaning of OO programs, so as to fully automate the translation from the "conceptual" to human-readable, idiomatic code, in mainstream languages. At some level, this is not new. Domain-Specific Languages (DSL), are high-level languages with syntax and semantics tailored to a specific domain [17]. A DSL abstracts over the details of "code", providing notation to specify domain-specific knowledge in a natural manner. DSL implementations often work via translation to a GPL for execution. Some generate human-readable code [5, 12, 18, 24]. This is what we do, for the domain of OO programs.

While designing a generic OO language is a worthwhile endeavour, we had a second motive: we needed a means to do exactly that as part of our Drasil project [22, 23]. The idea of Drasil is to generate all the requirement documentation and code from expert-provided domain knowledge. The generated code needs to be human readable so that experts can certify that it matches their requirements. We largely rewrote SAGA [5] to create GOOL[1]. It is implemented as a DSL embedded in Haskell that can currently generate code in Python, Java, C#, and C++. Others can be added, with the implementation effort being commensurate to the (semantic) distance to the languages already supported.

In Section 2, we outline the high-level requirements for GOOL, followed by an outline of the syntax in Section 3, to enable concrete examples. The interesting design details are given in Section 4. How we capture idioms and patterns is in Section 5. We close with a discussion of related work, plans for future improvements and conclusions.

PL'20, January 01–03, 2018, New York, NY, USA
2020.

[1] Available at https://github.com/JacquesCarette/Drasil as a sub-package.

2 Requirements

Our requirements are as follows:

mainstream Generate code in mainstream OO languages.
readable The generated code should be human-readable,
idiomatic The generated code should be idiomatic,
documented The generated code should be documented,
patterns Common OO patterns should be expressible.
expressivity The language should be rich enough to express a set of existing OO programs, which act as test cases for the language.
common Language commonalities should be abstracted.

Targeting OO languages (**mainstream**) reflects their popularity, thus the most potential users — one reasons that the makers of Scala and Kotlin chose to target the JVM to leverage the Java ecosystem, and Typescript for Javascript.

The **readable** requirement is not as obvious. We aim to write high-level OO code once and have it be available in many GPLs. One use case is to generate libraries of functions for a narrow domain. As needs evolve and language popularity changes, it is useful to have it immediately available in a number of languages. We use it to get *extremely well documented* code that would be unrealistic to do by hand, as part of Drasil [22, 23]. **readable** is also a proxy for *understandable*, which is helpful for debugging.

The same underlying reasons for **readable** also drive **idiomatic** and **documented**, as they contribute to the human-understandability of the generated code. **idiomatic** is important as readers would otherwise find the code "foreign". Documentation spans informal one-liners meant for humans to formal, structured comments for generating API documentation with tools like Doxygen, or static analysis. Readability (and thus understandability) are improved when code is pretty-printed [7]. Thus layout, redundant parentheses, well-chosen variable names, using a common style with lines that are not too long, are just as useful for generated code as for human-written code. GOOL does not prevent bad code, it just simplifies creating **readable**, **idiomatic** and **documented** code in multiple languages.

The **patterns** requirement is typical of DSLs: common idioms can be reified into a linguistic form instead of being informal. Even some of the *design patterns* of [10] can become part of the language. This makes writing some OO code even easier in GOOL than in GPLs, but it also helps keep GOOL language-agnostic and facilitates generating idiomatic code.

expressivity is about GOOL capturing the ideas contained in OO programs. We test GOOL against real-world examples from the *Drasil* project, such as software for determining whether glass withstands a nearby explosion and software for simulating projectile motion.

The last requirement (**common**) that language commonalities be abstracted, is internal: we noticed too much code duplication in our initial backends.

3 Creating GOOL

To create a "generic" object-oriented language, we chose an incremental abstraction approach: start from OO programs written in two different languages, and unify them *conceptually*. We abstract from concrete programs, not just to meet our **expressivity** requirement, but also because that is our "domain". Although what can be said in any given OO language is quite broad, what we *actually want to say* is often much more restricted. And what we *need to say* is often even more concise. For example, Java offers introspection features, but C++doesn't, so abstracting from portable OO will not feature introspection (although it may be the case that generating idiomatic Java may do so); thus GOOL does not encode introspection. C++templates are different: while other languages do not necessarily have comparable meta-programming features, it is not only feasible but in fact easy to provide template-like features in GOOL, as well as some partial evaluation . Thus we do not need to generate templates. In other words, we are trying to abstract over the fundamental ideas expressed via OO programs, rather than abstracting over the languages — and we believe the end result better captures the essence of OO programs. Some features, such as types, which don't exist per se in Python but are required in Java, C#and C++, will be present as doing full type inference is unrealistic.

Some features of OO programs are not operational: comments and formatting decisions amongst them. To us, programs are a bidirectional means of communication; they must be valid and executable, but also need to be readable and understandable by humans. Generating code for consumption by machines is well understood, but generating code for human consumption has been given less attention. We tried to pay close attention to program features — such as programmers writing longer methods as blocks separated by blank lines, often with comments — which make programs more accessible to human readers. In GOOL, bodies are not just a sequence of statements, but instead a list of blocks. This additional level is operationally meaningless, but represents the actual structure of OO programs. This is because programmers (hopefully!) write code to be read by other programmers, and blocks increase human-readability.

Basic GOOL syntax is shown in Table 1. Note that we distinguish a variable from its value[2]. This distinction is motivated by semantic considerations; it is beneficial for stricter typing and enables convenient syntax for **patterns** that translate to more idiomatic code.

GOOL is currently an Embedded Domain Specific Language (EDSL) in Haskell. Haskell is very well-suited for this, offering a variety of features (Generalized Algebraic Data Types (GADTs), type classes, parametric polymorphism, kind polymorphism, etc.) that are quite useful for building languages. Its syntax is also fairly liberal, so that with *smart*

[2]as befits the use-mention distinction from analytic philosophy

Table 1. Basic GOOL Syntax - brackets indicate shortcuts for common cases

Types	`bool`, `int`, `float`, `char`, `string`, `infile` (read mode), `outfile` (write mode), `listType`, `obj`
Variables	`var`, `extVar`, `classVar`, `objVar`, `$->` (infix operator for `objVar`), `self`, [`listVar`]
Values	`valueOf` (value from variable), `litTrue`, `litFalse`, `litInt`, `litFloat`, `litChar`, `litString`, `?!`, `?&&`, `?<`, `?<=`, `?>`, `?>=`, `?==`, `?!=`, `#~`, `#/^`, `#|`, `#+`, `#-`, `#*`, `#/`, `#^`, `inlineIf`, `funcApp`, `extFuncApp`, `newObj`, `objMethodCall`, [`selfFuncApp`, `objMethodCallNoParams`]
Statements	`varDec`, `varDecDef`, `assign`, `&=`, `&+=`, `&-=`, `&++`, `&~-`, `break`, `continue`, `returnState`, `throw`, `free`, `comment`, `ifCond`, `ifNoElse`, `switch`, `for`, `forRange`, `forEach`, `while`, `tryCatch`, `block`, `body` [`bodyStatements` (single-block body), `oneLiner` (single-statement body)]
List API	`listAccess`, `at` (same as `listAccess`), `listSet`, `listAppend`, `listIndexExists`, `indexOf`, `listSlice`
Scope	`public`, `private`
Binding	`static_`, `dynamic_`
Functions	`function`, `method`, `param`, `pointerParam`, `mainFunction`, `docFunc`, [`pubMethod`, `privMethod`]
State Variables	`stateVar`, `constVar`, [`privMVar`, `pubMVar` (dynamic), `pubGVar` (static)]
Classes	`buildClass`, `docClass`, [`pubClass`, `privClass`]
Packages	`buildModule`, `fileDoc`, `docMod`, `prog`, `package`, `doxConfig`, `makefile`

constructors, one can somewhat mimic the usual syntax of OO languages. Unfortunately, the embedded nature of GOOL means its syntax is constrained by Haskell's, so a programmer familiar with OO languages may find it difficult to write in GOOL. However, in our anticipated use cases, where the program may have to be written in multiple different languages, it is easier to write it once in GOOL. Also, if GOOL is used as the target language for other DSLs, the ease of writing GOOL programs matters less, since the programs will be written by a computer rather than a human.

4 GOOL Implementation

There are two "obvious" means of dealing with large embedded DSLs in Haskell: either as a set of GADTs, or using a set of classes, in the "finally tagless" style [8] (we will refer to it as simply *tagless* from now on). GOOL uses a "sophisticated" version of tagless involving *type families*, which allows for generic routines to be implemented and *patterns* easily encoded.

Tagless encodes a language, through methods from a set of classes, as a generalized *fold* over any *representation* of the language. Thus what look like GOOL "keywords" are actually class methods which we typically instantiate to language renderers, though we're also free to do various static analysis passes. By using *type families*, each instance can choose different underlying data structures for GOOL's types. For example the C++ instance stores destructor statements with state variables, but destructors are not needed for the other languages. Our language is defined by 328 methods across a hierarchy of 43 classes, grouped by functionality — GOOL is not a small language!

We have also defined 300 functions that abstract over **common**alities between target languages. This makes writing new renderers for new languages fairly straightforward. GOOL's Java and C# renderers demonstrate this well. Out of the 328 total methods, the instances of 229 are shared between the Java and C# renderers, in that they are just calls to the same common function. That is 40% more common instances compared to between Python and Java, for example.

5 Encoding Patterns

There are various levels of "patterns" to encode, from simple library-level functions, to simple tasks (command-line arguments, list processing, printing), on to more complex patterns such as methods with a mixture of input, output and in-out parameters, and finally on to design patterns.

5.1 Internalizing library functions

Consider the simple trigonometric sine function, called `sin` in GOOL. It is common enough to warrant its own name, even though in most languages it is part of a library. A GOOL expression `sin foo` can be seamlessly translated to yield `math.sin(foo)` in Python, `Math.sin(foo)` in Java, `Math.Sin(foo)` in C#, and `sin(foo)` in C++. Other functions are handled similarly. This part is easily extensible, but does require adding to GOOL classes.

5.2 Command line arguments

A slightly more complex task is accessing arguments passed on the command line. This tends to differ more significantly across languages. GOOL offers an abstraction of these mechanisms, through an `argsList` function that represents the list of arguments, as well as convenience functions for common tasks such as indexing into `argsList` and checking if an argument at a particular position exists.

5.3 Lists

Variations on lists are frequently used in OO code, but the actual API in each language tends to vary considerably; we

need to provide a single abstraction that provides sufficient functionality to do useful list computations. Rather than abstracting from the functionality provided in the libraries of each language to find common ground, we instead reverse engineer the "useful" API from actual use cases.

One thing we immediately notice is that lists in OO languages are rarely *linked lists* (unlike in Haskell), but rather more like a dynamically sized vector. In particular, indexing a list by position, which is a horrifying idea for linked lists, is extremely common.

This narrows things down to a small set of functions and statements, shown in Table 1 on the line labelled *List API*. For example, `listAccess (valueOf ages) (litInt 1)` will generate `ages[1]` in Python and C#, `ages.get(1)` in Java, and `ages.at(1)` in C++. List slicing (`listSlice` *statement*) gets a variable to assign to, a list to slice, and three values representing the starting and ending indices and the step size. These last three values are all optional and default to the start of the list, end of the list and 1 respectively. To take elements from index 1 to 2 of ages and assign the result to `someAges`:

```
listSlice someAges (valueOf ages) (Just $ litInt 1)
  (Just $ litInt 3) Nothing
```

List slicing is interesting as the generated Python is particularly simple, unlike in other languages; the Python:

```
someAges = ages[1:3:]
```

while in Java it is

```
ArrayList<Double> temp = new ArrayList<Double>(0);
for (int i_temp = 1; i_temp < 3; i_temp++) {
    temp.add(ages.get(i_temp));
}
someAges = temp;
```

Such idiomatic code generation is enabled by having appropriate high-level information driving the generation.

5.4 Printing

Printing is another feature that generates quite different code depending on the target language. Here again Python is more "expressive" so that printing a list (via `printLn ages`) generates `print(ages)`, but in other languages we must generate a loop; for example, in C++:

```
std::cout << "[";
for (int list_i1 = 0; list_i1 < \
  (int)(myName.size()) - 1; list_i1 ++) {
  std::cout << myName.at(list_i1);
  std::cout << ", ";
}
if ((int)(myName.size()) > 0) {
  std::cout << myName.at((int)(myName.size()) - 1);
}
std::cout << "]" << std::endl;
```

In addition to printing, there is also functionality for reading input.

5.5 Procedures with input, output and input-output parameters

Moving to larger-scale patterns, we noticed that our codes had methods that used their parameters differently: some were used as inputs, some as outputs and some for both purposes. This was a *semantic* pattern that was not necessarily obvious in any of the implementations. However, once we noticed it, we could use that information to generate better, more idiomatic code in each language, while still capturing the higher-level semantics of the functionality we were trying to implement. More concretely, consider a function `applyDiscount` that takes a price and a discount, subtracts the discount from the price, and returns both the new price and a Boolean for whether the price is below 20. In GOOL, using `inOutFunc`, assuming all variables mentioned have been defined:

```
inOutFunc "applyDiscount" public static_
  [discount] [isAffordable] [price]
  (bodyStatements [
    price &-= valueOf discount,
    isAffordable &= valueOf price ?< litFloat 20.0])
```

`inOutFunc` takes three lists of parameters, the input, output and input-output, respectively. This function has two outputs —`price` and `isAffordable`— and multiple outputs are not directly supported in all target languages. Thus we need to use different features to represent these. For example, in Python, a return statement with multiple values is used:

```
def applyDiscount(price, discount):
    price = price - discount
    isAffordable = price < 20

    return price, isAffordable
```

In Java, the outputs are returned in an array of `Objects`:

```
public static Object[] applyDiscount( \
  int price, int discount) throws Exception {
    Boolean isAffordable;

    price = price - discount;
    isAffordable = price < 20;

    Object[] outputs = new Object[2];
    outputs[0] = price;
    outputs[1] = isAffordable;
    return outputs;
  }
}
```

In C#, the outputs are passed as parameters, using the `out` keyword, if it is only an output, or the `ref` keyword, if it is both an input and an output:

```
public static void applyDiscount(ref int price, \
  int discount, out Boolean isAffordable) {
    price = price - discount;
    isAffordable = price < 20;
}
```

And in C++, the outputs are passed as pointer parameters:

```
void applyDiscount(int &price, \
  int discount, bool &isAffordable) {
    price = price - discount;
    isAffordable = price < 20;
}
```

Here again we see how a natural task-level "feature", namely the desire to have different kinds of parameters, ends up being rendered differently, but hopefully idiomatically, in each target language. GOOL manages the tedious aspects of generating any needed variable declarations and return statements. To call an `inOutFunc` function, one must use `inOutCall` so that GOOL can "line up" all the pieces properly.

5.6 Getters and setters

Getters and setters are a mainstay of OO programming. Whether these achieve encapsulation or not, it is certainly the case that saying to an OO programmer "variable `foo` from class `FooClass` should have getters and setters" is enough information for them to write the code. And so it is in GOOL as well, with `getMethod "FooClass" foo` and `setMethod "FooClass" foo`. The generated setters in Python, Java, C# and C++ are:

```
def setFoo(self, foo):
    self.foo = foo
```

```
public void setFoo(int foo) throws Exception {
    this.foo = foo;
  }
}
```

```
public void setFoo(int foo) {
    this.foo = foo;
}
```

```
void FooClass::setFoo(int foo) {
    this->foo = foo;
}
```

The point is that the conceptually simple "set method" contains a number of idiosyncracies in each target language. These details are irrelevant for the task at hand, and this tedium can be automated. As before, there are specific means of calling these functions, `get` and `set`.

5.7 Design Patterns

Finally we get to the design patterns of [10]. GOOL currently handles three design patterns: Observer, State, and Strategy.

For Strategy, we draw from partial evaluation, and ensure that the set of strategies that will effectively be used are statically known at generation time. This way we can ensure to only generate code for those that will actually be used. `runStrategy` is the user-facing function; it needs the name of the strategy to use, a list of pairs of strategy names and bodies, and an optional variable and value to assign to upon termination of the strategy.

For Observer, `initObserverList` generates an observer for a list. More specifically, given a list of (initial values), it generates a declaration of an observer list variable, initially containing the given values. `addObserver` can be used to add a value to the observer list, and `notifyObservers` will call a method on each of the observers. Currently, the name of the observer list variable is fixed, so there can only be one observer list in a given scope.

The State pattern is here specialized to implement *Finite State Machines* with fairly general transition functions. Transitions happen on checking, not on changing the state. `initState` takes a name and a state label and generate a declaration of a variable with the given name and initial state. `changeState` changes the state of the variable to a new state. `checkState` is more complex. It takes the name of the state variable, a list of value-body pairs, and a fallback body; and it generates a conditional (usually a switch statement) that checks the state and runs the corresponding body, or the fallback body, if none of the states match.

Of course the design patterns could already have been coded in GOOL, but having these as language features is useful for two reasons: 1) the GOOL-level code is clearer in its intent (and more concise), and 2) the resulting code can be more idiomatic.

6 Related Work

6.1 General-purpose code generation

Haxe [3] is a general-purpose multi-paradigm language and cross-platform compiler. It compiles to all of the languages GOOL does, and many others. However, it is designed as a more traditional programming language, and thus does not offer the high-level abstractions that GOOL provides. Furthermore Haxe strips comments and generates source code around a custom framework; the effort of learning this framework and the lack of comments makes the generated code not particularly readable. The internal organization of Haxe does not seem to be well documented.

Protokit [14] is a DSL and code generator for Java and C++, where the generator is designed to produce general-purpose imperative or object-oriented code. The Protokit generator is model-driven and uses a final "output model" from which actual code can be generated. Since the "output model" is quite similar to the generated code, it presented challenges with regards to semantic, conventional, and library-related differences between the target languages [14]. GOOL's finally-tagless approach and syntax for high-level tasks, on the other hand, help overcome differences between target languages.

ThingML [11] is a DSL for model-driven engineering targeting C, C++, Java, and JavaScript. It is specialized to deal with distributed reactive systems (a nevertheless broad range of application domains). This means that this is not quite a general-purpose DSL, unlike GOOL. ThingML's modelling-related syntax and abstractions stand in contrast to GOOL's object-oriented syntax and abstractions. The generated code lacks some of the pretty-printing provided by GOOL, specifically indentation, which detracts from readability.

6.2 Object-oriented generators

There are a number of code generators with multiple target OO languages, though all are for more restricted domains than GOOL, and thus do not meet all of our requirements.

Google protocol buffers [2] is a DSL for serializing structured data, which can be compiled into Java, Python, Objective C, and C++. **Thrift** [20] is a Facebook-developed tool for generating code in multiple languages and even multiple paradigms based on language-neutral descriptions of data types and interfaces. **Clearwater** [21] is an approach for implementing DSLs with multiple target languages for components of distributed systems. The **Time Weaver** tool [9] uses a multi-language code generator to generate "glue" code for real-time embedded systems. The domain of mobile applications is host to a bevy of DSLs with multiple target languages, of which **MobDSL** [15] and **XIS-Mobile** [19] are two examples. **Conjure** [1] is a DSL for generating APIs. It reads YML descriptions of APIs and can generate code in Java, TypeScript, Python, and Rust.

6.3 Design Patterns

A number of languages for modeling design patterns have been developed. The **Design Pattern Modeling Language** (DPML) [16] is similar to the Unified Modeling Language (UML) but designed specifically to overcome UML's shortcomings so as to be able to model all design patterns. DPML consists of both specification diagrams and instance diagrams for instantiations of design patterns, but does not attempt to generate actual source code from the models. The **Role-Based Metamodeling Language** [13] is also based on UML but with changes to allow for better models of design patterns, with specifications for the structure, interactions, and state-based behaviour in patterns. Again, source code generation is not attempted. Another metamodel for design patterns includes generation of Java code [4]. IBM developed a DSL in the form of a visual user interface for generation of OO code based on design patterns [6]. The languages that generate code do so only for design patterns, not for any general-purpose code, as GOOL does.

7 Future Work

Currently GOOL code is typed based on what it represents: variable, value, type, or method, for example. The type system does not go "deeper", so that variables are untyped, and values (such as booleans and strings) are simply "values". This is sufficient to allow us to generate well-formed code, but not to ensure that it is well-typed. We have started to statically type GOOL by making the representations for GOOL's `Variables` and `Values` GADTs.

We also want to improve the generated import statements, via tracking actual dependencies on features used. In general, we can do various kinds of static analyses to enhance the code generation quality, such as being more precise about `throws Exception` in Java.

We also want to interface with external libraries, such as a variety of ODE solvers. The API to available solvers varies considerably, so we will need to change the "shape" of generated code depending on the users' choice.

Some implementation decisions, such as the use of `ArrayList` to represent lists in Java, are hard-coded. But we could have used `Vector` instead. We would like such a choice to be user-controlled. Another such decision point is to allow users to choose which specific external library to use.

8 Conclusion

We currently successfully use GOOL to simultaneously generate code in all of our target languages for the glass and projectile programs described in Section 2.

Conceptually, mainstream object-oriented languages are similar enough that it is indeed feasible to create a single "generic" object-oriented language that can be "compiled" to them. Of course, these languages are syntactically quite different in places, and each contains some unique ideas as well. In other words, there exists a "conceptual" object-oriented language that is more than just "pseudocode": it is a full-fledged executable language (through generation) that captures the common essence of mainstream OO languages.

GOOL is an unusual DSL, as its "domain" is actually that of object-oriented languages. Or, to be more precise, of conceptual programs that can be easily written in languages containing a procedural code with an object-oriented layer on top — which is what Java, Python, C++ and C# are.

Since we are capturing *conceptual programs*, we can achieve several things that we believe are *together* new:

- generation of idiomatic code for each target language,
- turning coding patterns into language idioms,

- generation of human-readable, well-documented code.

We must also re-emphasize this last point: that for GOOL, the generated code is meant for human consumption as well as for computer consumption. This is why semantically meaningless concepts such as "blocks" exist: to be able to chunk code into pieces meaningful for the human reader, and provide documentation at that level as well.

References

[1] [n. d.]. Conjure: a code-generator for multi-language HTTP/JSON clients and servers. https://palantir.github.io/conjure/#/ Accessed 2019-09-16.

[2] [n. d.]. Google Protocol Buffers. https://developers.google.com/protocol-buffers/ Accessed 2019-09-16.

[3] [n. d.]. Haxe - The cross-platform toolkit. https://haxe.org Accessed 2019-09-13.

[4] Hervé Albin-Amiot and Yann-Gaël Guéhéneuc. 2001. Meta-modeling design patterns: Application to pattern detection and code synthesis. In *Proceedings of ECOOP Workshop on Automating Object-Oriented Software Development Methods*.

[5] Lucas Beyak and Jacques Carette. 2011. SAGA: A DSL for story management. *arXiv preprint arXiv:1109.0776* (2011).

[6] Frank J. Budinsky, Marilyn A. Finnie, John M. Vlissides, and Patsy S. Yu. 1996. Automatic code generation from design patterns. *IBM systems Journal* 35, 2 (1996), 151–171.

[7] Raymond PL Buse and Westley R Weimer. 2009. Learning a metric for code readability. *IEEE Transactions on Software Engineering* 36, 4 (2009), 546–558.

[8] Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. 2009. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *Journal of Functional Programming* 19, 5 (2009), 509–543.

[9] Dionisio de Niz and Raj Rajkumar. 2004. Glue code generation: Closing the loophole in model-based development. In *10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2004). Workshop on Model-Driven Embedded Systems*. Citeseer.

[10] Erich Gamma. 1995. *Design patterns: elements of reusable object-oriented software*. Pearson Education India.

[11] Nicolas Harrand, Franck Fleurey, Brice Morin, and Knut Eilif Husa. 2016. Thingml: a language and code generation framework for heterogeneous targets. In *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*. ACM, 125–135.

[12] Sungpack Hong, Hassan Chafi, Edic Sedlar, and Kunle Olukotun. 2012. Green-Marl: a DSL for easy and efficient graph analysis. *ACM SIGARCH Computer Architecture News* 40, 1 (2012), 349–362.

[13] Dae-Kyoo Kim, Robert France, Sudipto Ghosh, and Eunjee Song. 2003. A uml-based metamodeling language to specify design patterns. In *Proceedings of Workshop on Software Model Engineering (WiSME), at UML 2003*. Citeseer.

[14] Gábor Kövesdán and László Lengyel. 2017. Multi-Platform Code Generation Supported by Domain-Specific Modeling. *International Journal of Information Technology and Computer Science* 9, 12 (2017), 11–18.

[15] Dean Kramer, Tony Clark, and Samia Oussena. 2010. MobDSL: A Domain Specific Language for multiple mobile platform deployment. In *2010 IEEE International Conference on Networked Embedded Systems for Enterprise Applications*. IEEE, 1–7.

[16] David Mapelsden, John Hosking, and John Grundy. 2002. Design pattern modelling and instantiation using DPML. In *Proceedings of the Fortieth International Conference on Tools Pacific: Objects for internet, mobile and embedded applications*. Australian Computer Society, Inc., 3–11.

[17] Marjan Mernik, Jan Heering, and Anthony M Sloane. 2005. When and how to develop domain-specific languages. *ACM computing surveys (CSUR)* 37, 4 (2005), 316–344.

[18] Arjan J Mooij, Jozef Hooman, and Rob Albers. 2013. Gaining industrial confidence for the introduction of domain-specific languages. In *2013 IEEE 37th Annual Computer Software and Applications Conference Workshops*. IEEE, 662–667.

[19] André Ribeiro and Alberto Rodrigues da Silva. 2014. Xis-mobile: A dsl for mobile applications. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*. ACM, 1316–1323.

[20] Mark Slee, Aditya Agarwal, and Marc Kwiatkowski. 2007. Thrift: Scalable cross-language services implementation. *Facebook White Paper* 5, 8 (2007).

[21] Galen S Swint, Calton Pu, Gueyoung Jung, Wenchang Yan, Younggyun Koh, Qinyi Wu, Charles Consel, Akhil Sahai, and Koichi Moriyama. 2005. Clearwater: extensible, flexible, modular code generation. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*. ACM, 144–153.

[22] Daniel Szymczak, W. Spencer Smith, and Jacques Carette. 2016. Position Paper: A Knowledge-Based Approach to Scientific Software Development. In *Proceedings of SE4Science'16 in conjunction with the International Conference on Software Engineering (ICSE)*. In conjunction with ICSE 2016, Austin, Texas, United States. 4 pp.

[23] Drasil Team. 2019. Drasil Software: Generate All The Things (Focusing on Scientific Software). https://github.com/JacquesCarette/Drasil.

[24] Daniel C Wang, Andrew W Appel, Jeffrey L Korn, and Christopher S Serra. 1997. The Zephyr Abstract Syntax Description Language.. In *DSL*, Vol. 97. 17–17.