



Association for
Computing Machinery

Advancing Computing as a Science & Profession

October 22–23, 2023
Cascais, Portugal



GPCE '23

Proceedings of the 22nd ACM SIGPLAN International Conference on

Generative Programming: Concepts and Experiences

Edited by:

Coen De Roover, Bernhard Rumpe, and Amir Shaikhha

Sponsored by:

ACM SIGPLAN

Co-located with:

SPLASH '23

Association for Computing Machinery, Inc.
1601 Broadway, 10th Floor
New York, NY 10019-7434
USA

Copyright © 2023 by the Association for Computing Machinery, Inc (ACM). Permission to make digital or hard copies of portions of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted.

To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permission to republish from: Publications Dept. ACM, Inc.
Fax +1-212-869-0481 or E-mail permissions@acm.org.

For other copying of articles that carry a code at the bottom of the first or last page, copying is permitted provided that the per-copy fee indicated in the code is paid through the Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923, USA.

ACM ISBN: 979-8-4007-0406-2

Cover photo:

Title: “Cascais resort in Portugal”

Photographer: RossHelen

Licensed to SPLASH 2023

Cropped from original:

<https://elements.envato.com/cascais-resort-in-portugal-KCGY987>

Production: Conference Publishing Consulting
D-94034 Passau, Germany, info@conference-publishing.com

Welcome from the Chairs

Welcome to the 22nd ACM SIGPLAN International Conference on Generative Programming: Concepts & Experiences (GPCE'23). GPCE is the premiere venue for researchers and practitioners interested in techniques that use program generation to increase programmer productivity, improve software quality, and shorten the time-to-market of software products. In addition to exploring cutting-edge techniques of generative software, GPCE seeks to foster cross-fertilization between the programming languages research communities.

Following the past several years, GPCE 2023 is co-located with SPLASH (the ACM SIGPLAN Conference on Systems, Programming, Languages, and Applications: Software for Humanity), which includes OOPSLA 2023, SLE 2023, as well as several other events in Cascais, Portugal. GPCE participants are invited to visit other sessions on the same day and vice versa, which provides the attendees of all events with an overview of current research at the intersection of programming languages and software engineering.

The call for papers attracted 27 abstract submissions among which 24 papers were submitted in three categories: full papers, short papers, and tool demonstrations. Each submission was double-blind and except for one paper with three reviews, all submissions were reviewed by four PC members with the help of external reviewers. The authors had 72 hours to provide a response to preliminary reviews. During a 10-day long period of electronic meeting, the PC members intensively discussed all the submissions, and selected 10 full papers and one short paper for presentation at the conference, covering all the topic areas of the call for papers. The conference program also includes a keynote presentation shared with SLE, by Julia Lawall and a tutorial by Jeremy Yallop. We are also delighted to have two presentations from the Most Influential Papers of GPCE 2012 and GPCE 2013.

Putting together GPCE'23 was a collaborative effort. We would like to thank the authors for providing the content of the program, and the program committee and external reviewers for their hard work in reviewing the papers and contributing to the online PC discussions. We would also like to thank the GPCE steering committee and the Chair Sebastian Erdweg in particular, as well as the previous chairs and various colleagues for their invaluable advice. Special thanks to Youyou Cong for helping with publicity and announcements. We are also grateful to SPLASH for the general organization, and ACM SIGPLAN for their continued support of this conference. We hope that the program we have crafted for GPCE 2023 serves as a source of inspiration, fostering novel ideas within the domain of generative programming for the future.

Amir Shaikhha
GPCE 2023 Program Chair

Coen De Roover, Bernhard Rumpe
GPCE 2023 General Chairs

GPCE 2023 Organization

General Chairs

Coen De Roover (Vrije Universiteit Brussel, Belgium)
Bernhard Rumpe (RWTH Aachen University, Germany)

PC Chair

Amir Shaikhha (University of Edinburgh, United Kingdom)

Program Committee

Aleksandar Dimovski (Mother Teresa University, Skopje, Macedonia)
Coen De Roover (Vrije Universiteit Brussel, Belgium)
Daniel Strüber (Chalmers and University of Gothenburg, Sweden)
Elena Zucca (University of Genova, Italy)
Eli Tilevich (Virginia Tech, United States of America)
Geoffrey Mainland (Drexel University, United States of America)
Jeremy Gibbons (Oxford University, United Kingdom)
Jeremy Yallop (University of Cambridge, United Kingdom)
Julia Lawall (Inria, France)
Lionel Parreaux (The Hong Kong University of Science and Technology, Hong Kong)
Márcio Ribeiro (Federal University of Alagoas (UFAL), Brazil)
Martin Erwig (Oregon State University, United States of America)
Michael O'Boyle (University of Edinburgh, United Kingdom)
Philip Wadler (University of Edinburgh, United Kingdom)
Raffi Khatchadourian (City University of New York, Hunter College, United States of America)
Ruby Tahboub (University of Illinois Urbana-Champaign, United States of America)
Sandro Stucki (Amazon Prime Video, Sweden)
Sebastian Erdweg (JGU Mainz, Germany)
Sheng Chen (UL Lafayette, United States of America)
Shigeru Chiba (University of Tokyo, Japan)
Shoaib Kamil (Adobe, United States of America)
Sibylle Schupp (Hamburg University of Technology, Germany)
Simon Fowler (University of Glasgow, United Kingdom)
Vojin Jovanovic (Oracle Labs, Switzerland)
Walter Binder (Università della Svizzera italiana (USI), Switzerland)
Youyou Cong (Tokyo Institute of Technology, Japan)
Yukiyoshi Kameyama (University of Tsukuba, Japan)

Publicity Chair

Youyou Cong (Tokyo Institute of Technology, Japan)

Steering Committee

Sebastian Erdweg (JGU Mainz, Germany) Steering Committee Chair

Christoph Reichenbach (Lund University, Sweden)

Coen De Roover (Vrije Universiteit Brussel, Belgium)

Eric Van Wyk (University of Minnesota, United States of America)

Jeff Gray (University of Alabama, United States of America)

Yukiyoshi Kameyama (University of Tsukuba, Japan)

Tiark Rompf (Purdue University, United States of America)

External Reviewers

Constantin Buschhaus

Hendrik Kausch

Lucas Wollenhaupt

Prashant Kumar

Tatiana Castro Velez

Contents

Frontmatter

Welcome from the Chairs	iii
-----------------------------------	-----

Papers

Automatically Generated Supernodes for AST Interpreters Improve Virtual-Machine Performance Matteo Basso, Daniele Bonetta, and Walter Binder — <i>USI, Switzerland; Oracle Labs, Netherlands</i>	1
A Monadic Framework for Name Resolution in Multi-phased Type Checkers Casper Bach Poulsen, Aron Zwaan, and Paul Hübner — <i>Delft University of Technology, Netherlands</i>	14
A pred-LL(*) Parsable Typed Higher-Order Macro System for Architecture Description Languages Christoph Hochrainer and Andreas Krall — <i>TU Wien, Austria</i>	29
C2TACO: Lifting Tensor Code to TACO José Wesley de Souza Magalhães, Jackson Woodruff, Elizabeth Polgreen, and Michael F. P. O’Boyle — <i>University of Edinburgh, UK</i>	42
Partial Evaluation of Automatic Differentiation for Differential-Algebraic Equations Solvers Oscar Eriksson, Viktor Palmkvist, and David Broman — <i>KTH Royal Institute of Technology, Sweden; Stanford University, USA</i>	57
Crossover: Towards Compiler-Enabled COBOL-C Interoperability Mart van Assen, Manzi Aimé Ntagengerwa, Ömer Faruk Sayilir, and Vadim Zaytsev — <i>University of Twente, Netherlands</i>	72
Generating Conforming Programs with Xsmith William Gallard Hatch, Pierce Darragh, Sorawee Porncharoenwase, Guy Watson, and Eric Eide — <i>University of Utah, USA; University of Washington, USA</i>	86
Multi-Stage Vertex-Centric Programming for Agent-Based Simulations Zilu Tian — <i>EPFL, Switzerland</i>	100
Unleashing the Power of Implicit Feedback in Software Product Lines: Benefits Ahead Raul Medeiros, Oscar Díaz, and David Benavides — <i>University of the Basque Country, San Sebastián, Spain; University of Seville, Seville, Spain</i>	113
Virtual Domain Specific Languages via Embedded Projectional Editing Niklas Korz and Artur Andrzejak — <i>Alugha, Germany; Heidelberg University, Germany</i>	122
Generating Constraint Programs for Variability Model Reasoning: A DSL and Solver-Agnostic Approach Camilo Correa Restrepo, Jacques Robin, and Raul Mazo — <i>University of Paris 1 Pantheon-Sorbonne, Paris, France; ESIEA, Paris, France; ENSTA Bretagne, Brest, France</i>	138
Author Index	153

Automatically Generated Supernodes for AST Interpreters Improve Virtual-Machine Performance

Matteo Basso

matteo.basso@usi.ch

Università della Svizzera italiana (USI)
Switzerland

Daniele Bonetta

daniele.bonetta@oracle.com

Oracle Labs
Netherlands

Walter Binder

walter.binder@usi.ch

Università della Svizzera italiana (USI)
Switzerland

Abstract

Abstract syntax tree (AST) interpreters allow implementing programming languages in a straight-forward way. However, AST interpreters implemented in object-oriented languages, such as e.g. in Java, often suffer from two serious performance issues. First, these interpreters commonly implement AST nodes by leveraging class inheritance and polymorphism, leading to many polymorphic call sites in the interpreter implementation and hence lowering interpreter performance. Second, widely used implementations of these interpreters throw costly runtime exceptions to model the control flow. Even though Just-in-Time (JIT) compilation mitigates these issues, performance in the first stages of the program execution remains poor.

In this paper, we propose a novel technique to improve both interpreter performance and steady-state performance, lowering also the pressure on the JIT compiler. Our technique automatically generates AST *supernodes* ahead-of-time, i.e., we automatically generate compound AST-node classes that encode the behavior of several other primitive AST nodes before the execution of the application. Our technique extracts common control-flow structures from an arbitrary, given set of ASTs, such as e.g. the functions of popular packages. It is based on matchmaking of AST structures, instantiation of matching supernodes, and replacement of the corresponding AST subtrees with the instantiated supernodes at load-time. We implement our technique in the GraalVM JavaScript engine, showing that our supernodes lead to an average interpreter speedup of 1.24×, an average steady-state speedup of 1.14×, and an average just-in-time compilation speedup of 1.33× on the web-tooling benchmark suite.

CCS Concepts: • Software and its engineering → Interpreters; Source code generation; Virtual machines; Software performance; Just-in-time compilers.

Keywords: AST, Interpreter, Code Generation, Performance Optimization, JavaScript, GraalVM, Truffle

ACM Reference Format:

Matteo Basso, Daniele Bonetta, and Walter Binder. 2023. Automatically Generated Supernodes for AST Interpreters Improve Virtual-Machine Performance. In *Proceedings of the 22nd ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '23)*, October 22–23, 2023, Cascais, Portugal. ACM, New York, NY, USA, 13 pages. <https://doi.org/10.1145/3624007.3624050>

1 Introduction

Abstract syntax tree (AST) interpreters allow implementing programming languages in an elegant and straight-forward way. However, AST interpreters frequently suffer from serious performance issues. AST interpreters are often implemented in object-oriented languages, such as e.g. in Java or C++, and exploit features such as class inheritance and polymorphism. Even though these features improve code maintainability, polymorphic call sites in the interpreter implementation lower interpreter performance. Moreover, widely used implementations of AST interpreters rely on costly runtime exceptions to model the control flow of the interpreted language. For instance, exceptions are used to break the execution of a loop iteration or to return from a function.

To mitigate these performance issues, research has mostly focused on the development and improvement of just-in-time (JIT) compilers. The JIT compiler—executed at runtime and usually concurrently with the application—transforms the ASTs into optimized machine code that the system can execute without the need for interpretation. After the JIT compiler has compiled all relevant ASTs, the system can reach a steady state, i.e., a state where the system internals have stabilized and the system executes predominantly JIT-compiled code. JIT compilation aims at high steady-state performance, but does not solve the problem of poor startup performance of AST interpreters.

In this paper, we make a first step towards the investigation of a new technique to improve both interpreter performance and steady-state performance, lowering also the pressure on the JIT compiler (i.e., feeding the JIT compiler

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org. GPCE '23, October 22–23, 2023, Cascais, Portugal

© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 979-8-4007-0406-2/23/10...\$15.00

<https://doi.org/10.1145/3624007.3624050>

with input code that can be more easily optimized). We base our work on the concept of AST supernode [7] (also known as superoperator [16] in the context of bytecode interpreters), i.e., compound AST-node classes that encode the behavior of several other primitive AST nodes. In particular, we aim at answering the following research questions:

- RQ1. Can supernodes speed up interpreter performance?
- RQ2. Can supernodes help the JIT compiler produce better optimized machine code?
- RQ3. Can supernodes reduce the pressure on the JIT compiler?

Contributions. To answer RQ1, RQ2, and RQ3, we propose a new technique to improve virtual-machine performance by exploiting AST supernodes. Our technique leverages automatic ahead-of-time generation of supernodes and runtime installation of matching supernodes, exploring a new point in the design space between ahead-of-time and just-in-time compilation (Section 4). Our technique automatically generates supernodes from a set of ASTs that exercise common control-flow patterns before building the Virtual Machine (VM). At runtime, our technique performs an efficient matchmaking of AST structures, instantiation of matching supernodes, and replacement of the corresponding AST subtrees with the instantiated supernodes. Despite the ahead-of-time generation of supernodes, our technique does not prevent the JIT compiler from exploiting profiling data to fully optimize the emitted machine code [4].

We implement our technique in the GraalVM JavaScript engine (also known as Graal.js [11]) and we evaluate our implementation on the web-tooling benchmark suite [17], showing that supernodes improve both interpreter and steady-state performance. Moreover, we show that supernodes reduce JIT compilation time, at the cost of moderate extra memory, increased VM building time, and some startup costs for loading the supernode classes (Section 5).

We complement the paper by presenting background information (Section 2), a motivating example (Section 3), a discussion of related work and our technique (Sections 6 and 7, respectively), and some concluding remarks (Section 8).

2 Background

In this section we present the required background on AST interpreters (Section 2.1) and Graal.js (Section 2.2).

2.1 AST Interpreters

AST allows representing a program using a simple tree structure where each AST node represents a language operation. For example, AST nodes may represent control-flow constructs, such as *if* and *while*, or primitive operations, such as arithmetic expressions, memory accesses, function calls, etc. We call AST nodes that represent control-flow constructs *control-flow nodes* and AST nodes that represent primitive operations *non-control-flow nodes*.

In an object-oriented implementation, AST nodes are often subclasses of a common abstract class, and ASTs are created exploiting composition—each AST node instance stores references to its children (if any). Each AST node implements the code to perform the behavior of the operation it encodes, returning the produced result. Each AST node is responsible for invoking the execution of its children.

The control-flow of the language to be interpreted is implemented by exploiting the control-flow structures of the language used to implement the interpreter. To implement *break*, *continue*, and *return* statements that span multiple AST nodes, AST interpreters often employ runtime exceptions [9, 27]. An AST node throws a runtime exception of a specific type and an ancestor AST node catches and handles that runtime exception. We note that control-flow nodes are usually found in the lower tree levels of an AST (i.e., closer to the root node), while non-control-flow nodes are usually found in the higher levels (i.e., closer to the leaf nodes).

2.2 GraalVM JavaScript (Graal.js)

We implement our technique in Graal.js [11], an open-source, high-performance implementation of the JavaScript programming language built on top of GraalVM [26]. GraalVM is a managed language runtime system based on the Java Virtual Machine (JVM), capable of executing several different programming languages such as Ruby, R, Python, and JavaScript. GraalVM supports ahead-of-time compilation thanks to native images [24] and yields high performance thanks to the Graal compiler [4].

Graal.js is implemented using Truffle [25], a language implementation framework that allows implementing self-optimizing AST interpreters running on GraalVM [27], i.e., implementing AST interpreters using custom APIs that allow the Graal compiler to partially evaluate [5] and efficiently JIT compile ASTs. In particular, the goal of the partial evaluator is to remove the AST-interpretation overhead by following the execution path in the program ASTs, before other JIT-compiler optimizations take place.

We define as *VM startup* the initial VM setup before program interpretation begins, as *warmup* the initial stages of program execution taking place after the VM startup that include program interpretation, partial evaluation, and JIT compilation, and as *steady-state* the stages of program execution where the system has stabilized and all the performance-relevant ASTs have been compiled.

In Graal.js, AST nodes are subclasses of the abstract class *Node* provided by the Truffle API. Each node class implements an *execute* method that defines the behaviour of the node and declares the children the node accepts. To model the control-flow, Graal.js exploits runtime exceptions, as described in the previous subsection.

```

1 function fibonacci(n) {
2   if (n < 2) {
3     return n;
4   }
5   return fibonacci(n - 1) + fibonacci(n - 2);
6 }

```

Figure 1. JavaScript implementation of function fibonacci.

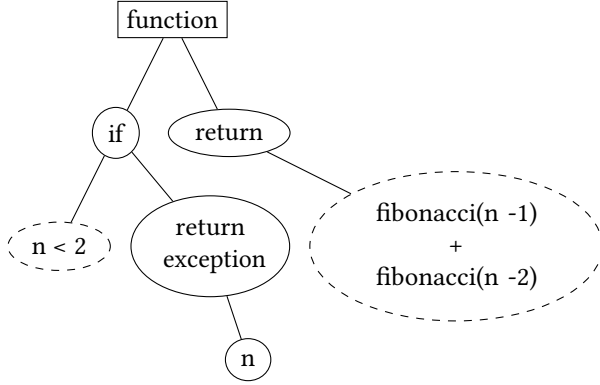


Figure 2. Simplified AST of the JavaScript function fibonacci (Figure 1). Collapsed nodes are represented using dashed borders.

3 Motivating Example

AST supernodes are compound AST-node classes that encode the behavior of several other primitive AST nodes. While supernodes can help improving AST interpreter performance, defining what nodes to aggregate in a supernode is non trivial. In this section, we illustrate our technique, showing how an example AST (consisting of primitive nodes only) is transformed into an AST that contains an (automatically generated) supernode.

Figure 1 shows a JavaScript implementation of the function fibonacci, i.e., a function that, given a number n as parameter, returns the n -th element of the fibonacci sequence. A simplified AST corresponding to the fibonacci function is shown in Figure 2. For a more compact presentation, we do not show all the nodes of the AST; we collapse some subtrees into single nodes (illustrated with dashed borders). We show simplified implementations of the execute methods of the function, if, return, and return exception AST nodes in Figure 3.

In the example, the root function node has two children: an if node and a return node. Unless a control-flow exception is thrown, these children are executed one after the other, as reported in the execute method of Figure 3 at lines 2–17. The if node first evaluates the condition expression $n < 2$ (line 21) and only if the condition evaluates to true, the if node executes its body (line 23), i.e., the subtree with the return exception node as root. The return exception node in the body of the if statement throws an exception

```

1 // execute method of the `function` node
2 public Object execute() {
3   try {
4     int childrenLength = children.length;
5     if (childrenLength == 0) {
6       return null;
7     }
8
9     for (int i = 0; i < childrenLength - 1; i++) {
10      children[i].execute();
11    }
12
13    return children[childrenLength - 1].execute();
14  } catch (ReturnException re) {
15    return re.getValue();
16  }
17 }
18
19 // execute method of the `if` node
20 public Object execute() {
21   if (condition.executeBoolean()) {
22     if (thenPart != null) {
23       return thenPart.execute();
24     } else {
25       return null;
26     }
27   } else {
28     if (elsePart != null) {
29       return elsePart.execute();
30     } else {
31       return null;
32     }
33   }
34 }
35
36 // execute method of the `return` node
37 public Object execute() {
38   return child.execute();
39 }
40
41 // execute method of the
42 // `return exception` node
43 public Object execute() {
44   throw new ReturnException(child.execute());
45 }

```

Figure 3. Simplified implementation of the function, if, return, and return exception nodes of Figure 2.

that encapsulates the value produced by the evaluation of its child node (line 44). This is because this return exception node must break the interpretation loop of the function node at lines 9–11. The function node catches the exception and returns the value the exception encapsulates (lines 14–16). Finally, the return node executes its child, returning the produced result (line 38). In this case, since the return node is the last child of the function node, no exception to model the control flow is required—the function node simply returns the result produced by the last child (line 13).

To avoid the use of an exception (that can cause runtime performance degradation), a supernode can be generated that replaces all the control-flow nodes of the AST, as shown in Figure 4. In particular, the control-flow supernode replaces the function, if, return, and return exception nodes. For the sake of exemplification, a simplified implementation of the execute method of this control-flow supernode is reported in Figure 5 (the details of the generated code will be shown later in Section 4.2) and consists of a ternary operator

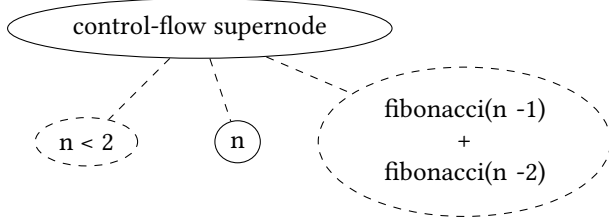


Figure 4. Simplified AST of the JavaScript function `fibonacci` (Figure 1) with a supernode that encodes the control-flow. Collapsed nodes are represented using dashed borders.

```

1 public Object execute() {
2   return condition.executeBoolean()
3     ? thenPart.execute()
4     : fallThrough.execute();
5 }

```

Figure 5. Simplified execute method of the control-flow supernode of Figure 4.

where each expression calls the execute method of a child node (lines 2–4). Child nodes are stored in the instance fields `condition` (line 2), `thenPart` (line 3), and `fallThrough` (line 4). To reduce the number of polymorphic call sites, the declared type of these fields is not `Node`, i.e., the common ancestor class of all the AST nodes. Instead, the declared types of the fields `condition`, `thenPart`, and `fallThrough` are the leaf classes `LessThanNode`, `ParameterNode`, and `AddNode`, respectively. In the figure, dashed edges represent monomorphic calls that can generally be better optimized, hence improving performance.

4 Supernode Generation and Installation

In this section, we first give an overview of our technique to automatically generate and install AST supernodes (Section 4.1). Then, we detail the three steps of our technique, namely *Supernode Generation* (Section 4.2), *Lookup-Tree Generation* (Section 4.3), and *Supernode Installation* (Section 4.4).

4.1 Overview

Figure 6 shows how our technique is integrated into the VM building process and workload execution. The three steps introduced by our technique are represented using gray nodes with dashed borders.

The first step of our technique is *Supernode Generation*. This step takes place at build time, before the building of the production VM, and aims at generating a supernode for the lower tree levels of each AST in a collection of functions, i.e., the part of the AST that contains the control flow of the function the AST encodes.¹ We call this collection of functions

generation set, which will consist of the functions of popular packages. Supernode generation is eager—it creates the maximal supernode that encodes the whole control-flow of the provided AST. We do not create supernodes that encapsulate only part of an AST’s control-flow nodes. Moreover, we do not create supernodes for non-control-flow AST nodes (such as arithmetic operations, function calls, or memory accesses). As detailed in Section 3, to avoid polymorphic call sites within supernodes, we generate supernodes that store their children in instance fields whose declared types are the dynamic types of the children nodes. We assume that the interpreter is implemented in a statically typed language, i.e., Java in the case of Graal.js.

After the supernodes have been generated, the supernodes need to be organized for efficient matchmaking. This is done in the second step of our technique, *Lookup-Tree Generation*. In our technique, each supernode is associated with a sequence of unique IDs (henceforth called *supernode structure* or simply *structure*) that encodes the structure of the supernode; the structure contains the types IDs of the AST control-flow nodes that the supernode encapsulates (in a fixed traversal order) and the types IDs of the children nodes that the supernode accepts, e.g., the type IDs of the corresponding non-control-flow nodes. At execution time, before parsing the source code, we load the supernode classes and use the supernode structures to generate a lookup-tree for the subsequent supernode installation step.

Finally, after supernodes have been generated and organized for efficient matchmaking, suitable supernodes are installed at runtime. The parser needs to be aware of their existence and should instantiate them when appropriate. This is achieved in the third step of our technique, *Supernode Installation*. After the creation of each AST, we match AST structures, instantiate matching supernodes, and replace the corresponding AST subtrees with the instantiated supernodes. To do so, we traverse the lookup-tree generated in the second step of our technique together with each parsed AST, collecting children nodes that will be used to instantiate a supernode, if a matching one is found.

After the supernode installation step, the VM executes the user code by interpreting the optimized ASTs that contain our supernodes (eventually JIT compiling the optimized ASTs to machine code).

4.2 Supernode Generation

Before the VM building phase, we generate the supernodes that will be included in a production build of the VM, e.g., by using the ASTs of the functions of popular packages.

Algorithm 1 depicts the pseudocode for generating supernodes. As input, the algorithm takes a set of ASTs for which supernodes shall be created and produces the supernodes

¹For simplicity, in this paper, we refer to the ASTs of functions written in the interpreted language, whereas our approach is applied to the root of

any generated AST (i.e., for functions, procedures, methods, constructors, etc.).

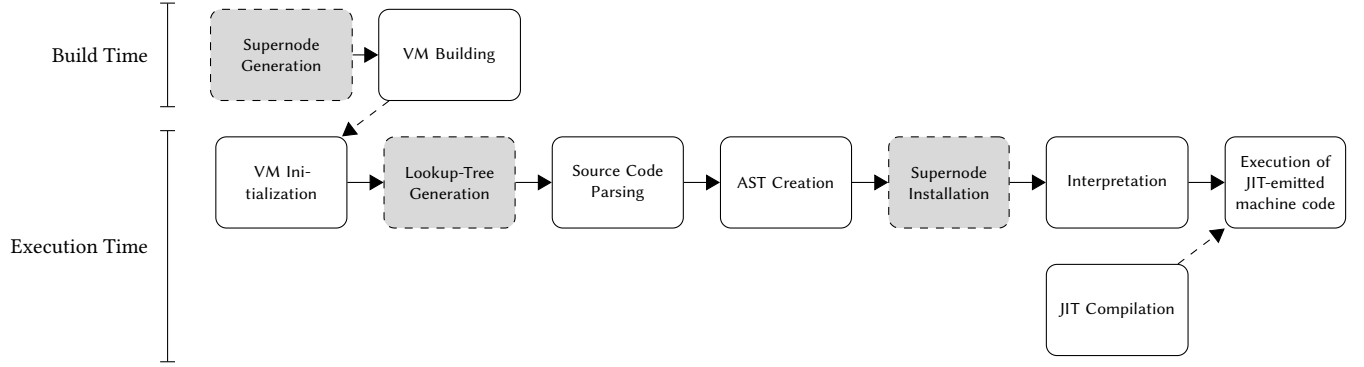


Figure 6. Integration of the proposed technique into the VM building process and workload execution. White nodes with solid borders represent baseline steps. Gray nodes with dashed borders represent the steps introduced by our technique.

Algorithm 1: Supernodes generation

generateSupernodes(A): generate supernodes for the provided ASTs

Input: A , the set of ASTs to be used to generate supernodes

Output: Supernode classes dumped as Java files

```

1  $S \leftarrow \text{new Set}()$ 
2 foreach  $a \in A$  do
3    $root \leftarrow a.root()$ 
4    $structure, code \leftarrow generateSupernode(root)$ 
5   if  $exceedsThreshold(structure)$  and
      $!S.contains(structure)$  then
6      $code.addStructureField(structure)$ 
7      $code.dump()$ 
8      $S.add(structure)$ 
```

as output in the form of Java files. In particular, the algorithm iterates over the provided ASTs (line 2), extracts the root of each AST (line 3), and calls a subroutine that recursively generates the supernode, returning the supernode structure and the corresponding code. Then, our algorithm checks whether the generated supernode should be saved or discarded (line 5). We save supernodes whose structure encodes a minimum number of control-flow nodes (in our experiments, we set this threshold to 3) and whose structure has not yet been encountered before, i.e., we save supernodes that are sufficiently complex and we avoid duplicates. If a supernode is saved, we store its structure in the supernode class in a static final field (line 6), we dump the supernode class as a Java file (line 7), and we update the set S (declared at line 8) that contains the already saved supernodes (line 1).

Algorithm 2 shows the recursive subroutine to generate supernodes. As input, the algorithm takes an AST node for which a supernode is to be created and returns a pair as output. The first element of the pair is the supernode structure

Algorithm 2: Supernode generation

generateSupernode(n): recursively generate a supernode for the lower levels of the provided AST that contain the control-flow nodes

Input: n , root of an AST subtree for which a supernode may be created

Output: A pair that consists of:

- (1) The supernode structure as a list
- (2) The code of a supernode class encoding the lower tree levels of the input AST

```

1  $code \leftarrow \text{new ClassBuilder}()$ 
2  $structure \leftarrow \text{new List}()$ 
3  $id \leftarrow nodeTypeId(n)$ 
4  $structure.append(id)$ 
5 if  $isControlFlowNode(n)$  then
6    $C \leftarrow children(n)$ 
7    $code += emitSourceCodeBeforeChildren(n, C)$ 
8   for  $k \leftarrow 1$  to  $length(C)$  do
9      $c \leftarrow C_k$ 
10     $code += emitSourceCodeBeforeChild(n, c)$ 
11     $cs, ccode \leftarrow generateSupernode(c)$ 
12     $structure.appendAll(cs)$ 
13     $code += ccode$ 
14     $code += emitSourceCodeAfterChild(n, c)$ 
15   $code += emitSourceCodeAfterChildren(n, C)$ 
16 else
17    $code += emitChildInvocation(n)$ 
18 return  $structure, code$ 
```

represented as a list, while the second element is the code of a supernode that corresponds to the returned structure. First, the algorithm creates a new code builder (line 1), a new

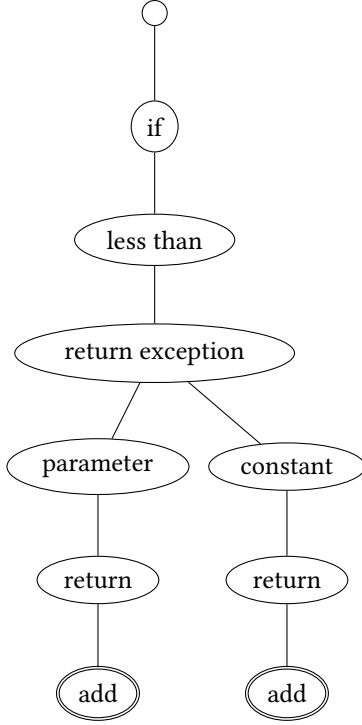


Figure 7. Lookup-tree encoding two supernodes. Nodes containing a reference to a supernode class are represented using double borders. For the sake of exemplification, each illustrated node contains the AST node type name instead of the unique ID of the corresponding node type.

structure (line 2), and appends the id of the input node to the newly created structure (lines 3–4). Then, the algorithm checks whether the input node is a control-flow node (line 5).

If the input node is not a control-flow node (lines 16–17), the algorithm generates a field in the supernode class with the same declared type as the dynamic type of the child node. The dynamic node type is uniquely identified by *id* and the field will be used to store the child node. The algorithm then generates code that invokes the `execute` method of the child node (line 17).

If the input node is a control-flow node, the algorithm emits Java source code that corresponds to the primitive operations encoded by the input node (lines 7, 10, 14, and 15). We provide more detail on the generated Java source code in the next paragraph. The algorithm recursively traverses the child nodes of this input node (line 11), generating the corresponding structures and code. The algorithm merges the generated structure and code into the current structure and code (lines 12–13), allowing a recursive supernode generation. Finally, we return the structure and code as a pair (line 18).

On the one hand, extracting supernode structures from the generation set, creating the lookup-tree of supernodes, and installing supernodes at runtime are interpreter-independent

```

1 public Object execute() {
2     boolean tmp1 = child1.executeBoolean();
3     if (tmp1) {
4         Object tmp2 = child2.execute();
5         return tmp2;
6     }
7     Object tmp3 = child3.execute();
8     return tmp3;
9 }

```

Figure 8. Execute method of the control-flow supernode of Figure 4 without simplifications.

algorithms and will work for any Truffle AST interpreter out-of-the-box. On the other hand, the generation of the Java source code of supernodes requires knowledge of the interpreter implementation and currently is a manual effort by the developers of an AST interpreter. In our implementation, we have dedicated source-code generation functions for 45 different control-flow node types (Gaal.js contains more than 300 AST node types). Figure 8 shows the code emitted by our generation functions that corresponds to the control-flow supernode of Figure 4, without the simplifications reported in Figure 5. Consider the `if` AST node encapsulated by the supernode (Figure 2), the `if` generation function emits a Java `if` statement (line 3) that accepts (as its condition) the result produced by the execution of the condition child node (line 2) and (as its body) the code produced by the traversal of the `if`-body subtree (lines 4 and 5). In supernodes, AST return nodes that throw exceptions to model the control flow are simply replaced by Java `return` statements (line 5). Even though not present in this example, when generating Java source code, we generate also Java labels that can be later referenced by `break` and `continue` statements. In this way, we can avoid the usage of exceptions to model the control flow. We are investigating techniques to automate code generation and further reduce the burden on the interpreter developer.

4.3 Lookup-Tree Generation

After the VM Initialization, we iterate over the structures of all the generated supernodes and build a lookup-tree that efficiently maps AST structures to supernodes. The lookup-tree is a trie (i.e., a prefix tree), for which the alphabet is the set of node IDs that occur in the supernodes' structures. Apart from the root, each node in the lookup-tree stores the type ID of an AST node.

Algorithm 3 reports our pseudocode for generating the lookup-tree. As input, the algorithm takes the supernodes generated by the *Supernode Generation* step and returns the root of the generated lookup-tree as output. In particular, the lookup-tree has an empty root, where later the match-making of AST structures will start (line 1). We iterate over the input supernodes (line 2) and we extract the structure of each supernode (line 4). Then, we iterate over the ids that compose the structure of each supernode (line 5) and

Algorithm 3: Lookup-tree generation using supernode structures.

generateLookupTree(*supernodes*): generates a lookup-tree for the provided supernodes

Input: *supernodes* to register in the lookup-tree

Output: A lookup-tree that contains the provided supernodes

```

1 root ← new Node()
2 foreach m in supernodes do
3   c ← root
4   ids ← structure(m)
5   for k ← 1 to length(ids) do
6     id ← idsk
7     t ← c.get(id)
8     if t == null then
9       t ← new Node()
10      c.addChild(t)
11    c ← t
12  c.setSupernode(m)
13 return root

```

for each supernode we create a path in the tree. This path starts from the root (line 3) and has an edge for each id in the supernode structure (lines 6–7). When generating the lookup-tree, the algorithm traverses the edges that already exist and creates new edges only if one cannot be found (lines 8–10). After the creation of the path for each supernode, the algorithm associates the supernode class to the last node of the path. This supernode class represents the supernode whose structure is equal to the path that led to that node. We note that each node, including non-leaf nodes, potentially may contain a reference to a supernode class, because a supernode may have a structure that is a prefix of another supernode’s structure. Finally, the algorithm returns the (empty) root (line 13).

Figure 7 shows an example lookup-tree that stores two supernodes. The path that starts from the root and ends with the add node on the left side of the Figure encodes the example supernode reported in Figure 4. The path that starts from the root and ends with the add node on the right side of the Figure encodes a supernode that returns a constant (instead of a function parameter) in the body of the if-statement. In practice, the only difference between the two supernodes is the declared type of the field storing a reference to a child node.

4.4 Supernode Installation

After the source code parsing and the creation of each AST, we match AST structures, we instantiate supernodes, and

we replace the matched subtrees with the instantiated supernodes. To do so, we traverse each AST and the lookup-tree at the same time, accumulating children nodes and potentially finding a supernode that replaces part of the AST.

Algorithm 4 reports the pseudocode for replacing AST subtrees with supernodes. The algorithm takes (as input) the root node of an AST subtree that may be modified with supernodes and the root node of the lookup-tree that contains the supernode classes. The algorithm returns (as output) the root node of an AST subtree that contains the installed supernodes. If no supernode is installed, the algorithm returns the root of the unmodified AST subtree provided as input.

The algorithm starts by searching for the matching AST structure, using the recursive subroutine `lookupTreeSearch` (line 1). This subroutine returns a pair that contains (1) a node of the lookup-tree potentially containing the supernode class matching the provided AST or an empty result if the lookup-tree cannot be traversed due to missing edges, and, (2) a list that contains the roots of the subtrees of the input AST needed for supernode instantiation. We will describe this subroutine in the next paragraph. After performing the search, the algorithm checks whether the subroutine found a lookup-tree node and whether this node is associated with a supernode class (line 2). If not, the algorithm returns the root of the unmodified AST subtree (line 6). Otherwise, the algorithm extracts the supernode class associated with the lookup-tree node (line 3) and returns a modified AST, i.e., an instance of the matched supernode class that takes as children nodes the nodes contained in the list returned by the recursive subroutine (line 4).

Algorithm 5 reports the pseudocode of the recursive lookup-tree search subroutine. The algorithm takes (as input) an AST node and a lookup-tree node to start the search, and returns (as output) the aforementioned pair. The algorithm first initializes a list that will contain the children nodes to be used for (potential) supernode instantiation (line 1). Then, the algorithm extracts the unique id of the AST node provided as a parameter (line 2) and traverses the lookup-tree using this id, updating the current lookup node (line 3). If an edge for this id cannot be found in the lookup-tree (line 4), the algorithm returns a pair of null values (line 5). Otherwise, the algorithm checks whether the input node is a control-flow node (line 6). If the input node is not a control-flow node, the algorithm appends the input node to the children list (line 15).

If the input node is a control-flow node, the algorithm iterates over the children of the input node (line 8). In particular, the algorithm performs a recursive search for each child, providing the current child and the current lookup node as parameters (line 9). If the lookup node returned by the recursive call is null (line 11), meaning that the lookup-tree cannot be traversed, the algorithm terminates early by returning a pair of null values (line 12). Otherwise, the algorithm updates the current lookup node, assigning the lookup

Algorithm 4: Supernode installation

installSupernode(n, l): tries to install a supernode in the provided AST

Input:

n , root of an AST subtree for which a lookup-tree node may be found

l , root node of a lookup-tree

Output: Root node of an AST subtree containing installed supernodes. Otherwise, the input root.

```

1  $c, vs \leftarrow \text{lookupTreeSearch}(n, l)$ 
2 if  $c \neq \text{null}$  and  $c.\text{hasSupernode}()$  then
3    $m \leftarrow c.\text{getSupernode}()$ 
4   return  $m.\text{instantiate}(vs)$ 
5 else
6   return  $n$ 
```

node returned by the recursive call (line 10), and appends the children list returned by the recursive call to the local children list (line 13). After processing the input node and potentially all its children nodes, excluding the case of early termination, the algorithm returns the latest lookup-tree node and the updated children list (line 16).

We note that we stop traversing the AST and we do not install any supernode as soon as matching fails. This is because our supernode-generation algorithm is eager and each supernode encodes a specific structure and accepts a specific number of children nodes of exact types—a single failure indicates a mismatch in the AST structure. Since the installation step is deterministic, there is only one match for one supernode, the same AST cannot match two different supernodes. Moreover, if the algorithm does not stop because of missing edges, the algorithm may return a non-leaf node of the lookup-tree. This node may contain a supernode class. We do not need to traverse the lookup-tree up to the leaves. The complexity of the supernode installation algorithm is $O(n)$ where n is the number of nodes of the input AST.

5 Evaluation

In this section, we first present our experimental setup (Section 5.1). Then, we present the interpreter speedups (Section 5.2), steady-state speedups (Section 5.3), and compilation-time speedups (Section 5.4) achieved by our technique. Finally, we discuss the memory overhead (Section 5.5) and the VM build-time overhead (Section 5.6) introduced by our supernodes, as well as the overhead of lookup-tree generation (Section 5.7).

5.1 Evaluation Settings

We run our experiments on a machine equipped with an 18-core Intel i9-10980XE (3.00 GHz) and 256 GB of RAM running

Algorithm 5: Lookup-tree search

lookupTreeSearch(n, l): tries to find a node in the lookup-tree for the provided AST

Input:

n , root of an AST subtree for which a lookup-tree node may be found

l , current lookup node in the lookup-tree

Output: A pair that consists of:

- (1) A node of the lookup-tree potentially containing the supernode class matching the provided AST or *null* if the lookup-tree cannot be traversed further due to missing edges
- (2) A list that contains the roots of the subtrees of the input AST needed for supernode instantiation

```

1  $vs \leftarrow \text{new List}()$ 
2  $i \leftarrow \text{nodeTypeId}(n)$ 
3  $l \leftarrow l.\text{get}(i)$ 
4 if  $l == \text{null}$  then
5   return  $\text{null}, \text{null}$ 
6 if  $\text{isControlFlowNode}(n)$  then
7    $C \leftarrow \text{children}(n)$ 
8   for  $k \leftarrow 1$  to  $\text{length}(C)$  do
9      $lc, vz \leftarrow \text{lookupTreeSearch}(C_k, l)$ 
10     $l \leftarrow lc$ 
11    if  $l == \text{null}$  then
12      return  $\text{null}, \text{null}$ 
13     $vs.\text{appendAll}(vz)$ 
14 else
15    $vs.\text{append}(n)$ 
16 return  $l, vs$ 
```

Linux Ubuntu (kernel v. 5.4.0-58-generic). Frequency scaling, turbo boost, and hyper-threading are disabled, CPU governor is set to “performance”. We conduct our experiments on Graal.js 22.3.0 community edition, based on OpenJDK 11 that uses the Graal compiler. In particular, we modify both Graal.js and Graal to implement our technique. We perform our experiments on the web-tooling benchmark suite [17], consisting of 18 benchmarks.

To generate our supernodes, as generation set, we use a collection of popular JavaScript packages for web development. We set the supernode threshold to 3, i.e., we create supernodes that encode at least 3 control-flow nodes. In this setting, our supernode generation creates ~6500 supernodes.

5.2 Interpreter Speedup

In this section, we answer RQ1 by evaluating the impact of our technique on interpreter performance. To do so, we force interpretation by disabling JIT compilation, we run 10

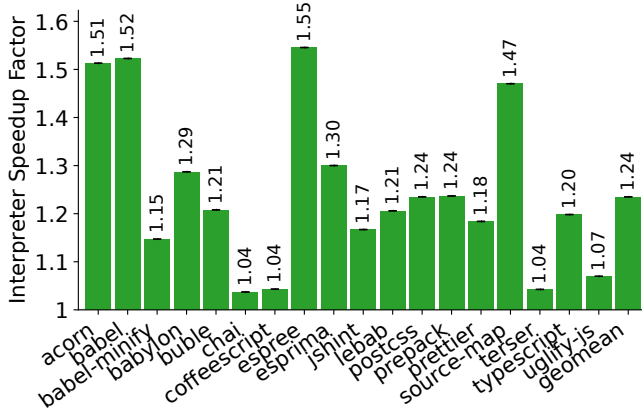


Figure 9. Interpreter speedup factors achieved by the proposed technique.

iterations of each benchmark, and we take the average of the last five time measurements. In this way, we let cache behaviors stabilize and we avoid the potential measurement perturbations of the first iterations. To mitigate measurement noise, we repeat this process five times.

Figure 9 reports the interpreter speedup as $T_{\text{baseline}}/T_{\text{supernodes}}$, where T_{baseline} refers to the average execution time obtained without using our technique and $T_{\text{supernodes}}$ refers to the average execution time obtained using our technique. The benchmarks are reported on the x-axis of the plot, while the speedup factor is reported on the y-axis. Above each bar, we report the exact speedup factor. The black error bars represent the 95% confidence intervals (CI) of the measurements. We note that the error bars are narrow for most of the experiments thanks to the stable measurements obtained, both with and without supernodes. The last bar on the right represents the geometric mean of all the other speedup factors.

We notice that our technique does not introduce any slowdown for any benchmark. Interpreter speedups range from $1.04\times$ (*chai*, *coffeescript*, and *terser*) to $1.55\times$ (*espre*), $1.24\times$ on average.² This is because the VM needs to traverse less AST nodes upon interpretation—our supernodes reduce the size of the ASTs with possible cache improvements. Moreover, our supernodes help reduce the overhead of using expensive exceptions to model the control flow as well as the number of polymorphic call sites. Hence, we positively answer [RQ1](#).

5.3 Steady-state Speedup

Here, we answer **RQ2** by evaluating the impact of our technique on steady-state performance, i.e., we investigate the impact of supernodes on JIT compilation, to understand whether supernodes lead to better optimized JIT-compiled code and consequently to speedups in steady state. We run

²Average speedup factors across multiple benchmarks are computed using the geometric mean.

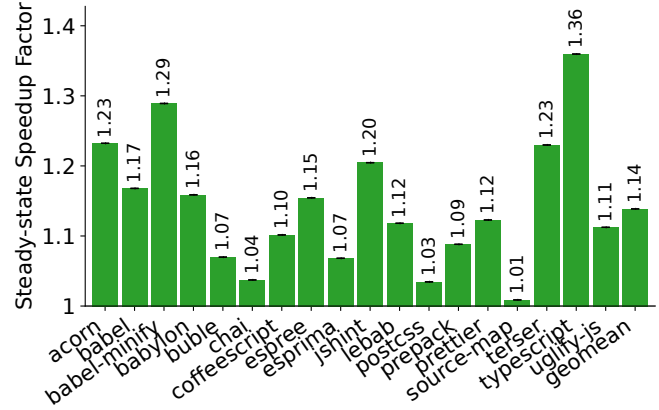


Figure 10. Steady-state speedup achieved by the proposed technique.

1 000 iterations of each benchmark (without disabling JIT compilation) and take the average of the last 10 iteration time measurements. In this way, we let JIT compilation stabilize and we take our measurements only after all the relevant ASTs have been compiled. We repeat this process five times.

Similarly to Figure 9, Figure 10 reports the steady-state speedup as $T_{baseline}/T_{supernodes}$, where $T_{baseline}$ refers to the average execution time obtained without using our technique and $T_{supernodes}$ refers to the average execution time with our technique.

Steady-state speedups range from $1.01\times$ (*source-map*) to $1.36\times$ (*typescript*), $1.14\times$ on average. Our experimental results positively answer **RQ2**, confirming that supernodes improve JIT compilation. As part of our future work, we plan to conduct an in-depth study on the effect of supernodes on JIT-compiler budget-driven optimization heuristics, which may lead to better optimized JIT-emitted code.

5.4 Compilation-time Speedup

We evaluate now the impact of our technique on compilation time, answering **RQ3**. We run 1 000 iterations of each benchmark (without disabling JIT compilation) and take the overall compilation time.

Figure 11 reports the compilation-time speedup as $T_{baseline}/T_{supernodes}$, where $T_{baseline}$ refers to the compilation time obtained without using our technique and $T_{supernodes}$ refers to the compilation time obtained using our technique.

Compilation-time speedups range from $0.62\times$ (*babylon*) to $2.86\times$ (*lebab*), $1.33\times$ on average. The benchmarks *lebab* ($2.86\times$) and *typescript* ($2.76\times$) benefit the most from our supernodes, because these benchmarks contain functions with complex control flow that is captured by supernodes. Our technique yields a compilation-time slowdown on the

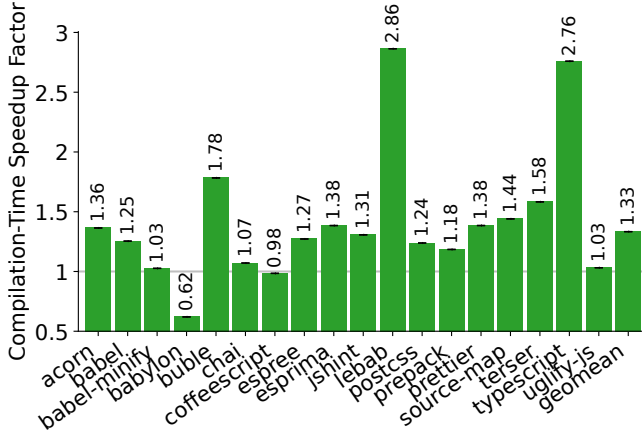


Figure 11. Compilation-time speedup achieved by the proposed technique.

benchmarks *babylon* (0.62 \times) and *coffeescript* (0.98 \times). By investigating the root cause of the compilation-time slowdowns, we find that in *babylon* supernodes lead to a significant increase of compilations w.r.t. the baseline. When using supernodes, the JIT compiler compiles the same functions multiple times with different specializations, increasing the number of compilations. We plan to conduct a thorough investigation of this phenomenon as part of our future work.

Overall, we can positively answer [RQ3](#), stating that supernodes reduce the pressure on the partial evaluator and the JIT compiler. Since the ASTs contain fewer nodes, the partial evaluator needs to traverse and partially evaluate fewer nodes. Moreover, the JIT compiler does not need to optimize runtime exceptions and polymorphic call sites in the supernodes, saving compilation time.

5.5 Memory Overhead

We discuss now the memory overhead introduced by our supernodes. For all our experiments, we install supernodes generated by using a collection of the most commonly used JavaScript packages. For this reason, we do not report the overheads for each benchmark. Instead, we report a single memory overhead factor computed as $M_{supernodes}/M_{baseline}$, where $M_{supernodes}$ is the size of a VM build that contains our supernodes, and $M_{baseline}$ is the size of a VM build without supernodes, respectively.

Using a supernode threshold of 3 (i.e., the minimum number of control-flow nodes subsumed by a supernode, as described in Section 4.2), our technique yields a memory overhead of 1.09 \times (the sizes are \sim 671MB and \sim 615MB for the VM build that contains our supernodes and the VM build without supernodes, respectively).

On modern machines where memory consumption is often not a major issue, we consider such a memory overhead acceptable. Nonetheless, we note that memory consumption may be reduced by performing a study to identify and keep

only the most commonly used supernodes (of the ~6500 automatically generated supernodes from our generation set).

5.6 VM Build-time Overhead

Here, we discuss now the VM build-time overheads introduced by our supernodes. Similarly to Section 5.5, we do not report the overheads for each benchmark but a single VM build-time overhead factor computed as $T_{supernodes}/T_{baseline}$, where $T_{supernodes}$ is the time required to build a VM that contains our supernodes, and $T_{baseline}$ is the time required to build a VM that does not contain our supernodes, respectively.

Using ~ 6500 automatically generated supernodes, our technique yields a VM build-time overhead of $1.73\times$ (121 seconds with supernodes, versus 70 seconds without supernodes). We note that a high VM build-time overhead factor is a minor drawback of our technique, considering the achieved runtime speedups. Indeed, once supernodes have been generated, the VM needs to be built only once before the production release, without any impact on the final user.

5.7 Lookup-Tree Generation Overhead

In this section, we evaluate the overhead of generating the lookup-tree, i.e., the overhead that our technique introduces upon VM startup. In our experiment, we measure the time required to build the lookup-tree and to load and link the Java class files of the supernodes in a fully sequential setting without any optimization—we parallelize neither lookup-tree generation nor supernode class loading, and we do not run the lookup-tree generation concurrently with other VM initialization steps.

Our implementation requires ~ 3 seconds to generate the lookup-tree and to load the code of ~ 6500 supernodes. We note that the startup is rather expensive in Graal.js and the lookup-tree generation is not critical for performance. For this reason, we have not optimized this phase yet. To lower this overhead in the future, we note that the lookup-tree can be built asynchronously and in a parallelized manner (including the loading and linking of the supernode classes), or serialized at VM building time and then deserialized upon VM startup. Moreover, the supernode classes may also be lazily loaded upon the first match in the process of supernode installation. With such optimizations, one can hide (part of) the costs of lookup-tree creation and supernode class loading.

6 Related Work

Different techniques try to improve warmup performance of managed language runtime systems by optimizing and implementing efficient interpreters.

The concept of supernode was initially proposed in the context of bytecode interpreters. In particular, Proebsting

[16] reduce the cost of instruction dispatching in bytecode interpreters by creating superoperators, i.e., compound operations composed of many smaller primitive operations that avoid costly per-operation overheads. Larose et al. [7] apply a similar technique to AST interpreters, manually generating 20 supernodes. Differently from their method, we automatically generate and install supernodes, reducing the burden on interpreter developers. Our technique requires only the implementation of the interpreter-specific generation functions. Moreover, we implement our technique in a more complex Truffle implementation used in industry (Gaal.js), in contrast to the toy language TruffleSOM targeted in [7].

Sun’s JVM implementation [22] dynamically replaces occurrences of certain bytecode instructions—after their first execution—with more efficient *_quick* pseudo-instructions. The pseudo-instructions speed up the execution by taking advantage of the work done the first time the associated normal instructions are executed. In contrast to this method, we perform supernode instantiation right after AST creation, before an AST is executed for the first time.

Static compilation, also known as ahead-of-time (AOT) compilation, improves warmup performance by compiling applications before execution and hence removing interpretation costs. This feature is available for widely used languages, such as Java [6, 14] and JavaScript [19, 20]. The main issue of AOT compilation is the degradation of steady-state performance—specifically in the case of dynamic languages—since aggressive, speculative optimizations may not be performed due to the lack of profiling data. While our supernodes are generated ahead-of-time, our technique does not compile the source code of the application ahead-of-time. For this reason, in contrast to static compilation, our technique allows for steady-state performance improvements.

Other techniques try to improve the performance of either bytecode or AST interpreters. Threaded code [2] solves the branch prediction problem in bytecode interpreters. Savrun-Yeniçeri et al. [18] speed up hosted interpreters on the JVM by providing annotations that enable the generation of efficient threaded code and avoid the insertion of unnecessary runtime checks produced by the JIT compiler. Brunthaler [3] illustrate inline-caching optimizations, a technique to unfold code, a new reduced instruction format, a technique to eliminate reference counting operations in interpreters, and a technique to cache local variables of the host language in the stack frame of the executing language. Sullivan et al. [21] partially evaluate sequences of native instructions with respect to the in-memory representation of the program being interpreted by using instrumentation and a dynamic optimizer. Truffle [25] applies AST specialization [23, 27] during interpretation, enabling partial evaluation [5] and hence the execution of highly optimized code.

Finally, related work proposes strategies to reduce the runtime overhead of JIT compilation and hence improve warmup performance. ShareJIT [28] is a technique to cache

and share JIT-compiled code across processes. Even though this technique improves warmup performance, differently from our technique, it leads to steady-state performance degradation since the compiler cannot emit shared JIT-compiled code that uses absolute addresses. To overcome this limitation, instead of sharing JIT-compiled code, other techniques [1, 8, 15] share profiling data that is used to JIT compile the application either before or during the execution of the application itself. The main limitation of these approaches is that the code for which no profiling data is available is still interpreted.

7 Discussion

In this section, we first discuss use cases of our technique. In particular, we detail how long-running programs can benefit from our technique and how our technique can be employed to speed up specific workloads. Then, we detail ongoing work on native images and how our technique can be used to analyze JIT compilers. Finally, we discuss the generation set used in our experiments and the portability of our technique.

Long-running Programs. For long-running programs where steady-state performance is more relevant than interpreter performance, our technique can be slightly modified to dynamically generate supernodes at runtime. In particular, we compile the generated Java source code and we link the corresponding bytecode at runtime. In this way, we do not separate the supernode-generation, lookup-tree-generation, and supernode-installation steps as in Figure 6, but we perform them altogether after AST creation.

Workload-specific Supernodes. On server machines that frequently execute the same workload, our technique can be employed to generate workload-specific supernodes and so create a dedicated and optimized VM. For instance, we can create workload-specific supernodes for user-provided cloud lambda functions that typically have a short lifetime, which impairs the ability of the system to collect profiling data and JIT-compile the lambda function.

Moreover, we can create workload-specific supernodes for embedded systems with limited hardware resources and power supply. These embedded systems usually cannot JIT compile code due to significant runtime compilation overhead. In both cases, our supernodes may help improve interpreter performance.

Native Images. We conducted our experiments on a Graal.js VM based on OpenJDK, as discussed in Section 5.1. In addition, we are currently investigating the use of our technique for native images [24], i.e., our technique can be employed also when compiling Graal.js to a standalone executable. A Graal.js native image contains the VM internals compiled to machine code ahead-of-time, including the interpreter, the partial evaluator, and the JIT compiler (implemented in Java). At runtime, the partial evaluator and the

JIT compiler compile the input program to machine code, but not the VM internals.

In this setting, our supernodes can still help in reducing the number of polymorphic call sites and thrown control-flow exceptions. When using native images, the lookup-tree generation can take place at build time, the lookup-tree may be serialized and stored in the executable, reducing startup time. The supernode installation process remains unaltered when using native images, i.e., the native image parses the input program, creates the ASTs, and installs the supernodes. Supernodes can be generated before the native image building from the same generation set we used in our experiments.

We plan to conduct an in-depth evaluation of our implementation on native images as part of our future work. Preliminary results show performance improvements similar to those reported in Section 5. The main advantage of native image w.r.t. our technique is that the extra startup overhead associated with supernodes, i.e., class loading, linking, and possibly JIT compilation of supernode classes, becomes a cost of native-image building, but is avoided when the native image is executed. The only remaining sources of overhead are the loading of the lookup-tree (negligible) and supernode installation (which already is a very efficient $O(n)$ algorithm, where n is the number of nodes in an AST).

Analyzing Compiler Optimizations. As shown in Section 5.3, our technique yields speedups also in steady-state performance. One could employ our technique to compare the JIT-emitted machine code executed in the steady-state between the original VM and a VM that uses our supernodes. The JIT-emitted machine code can reveal inefficient code patterns executed in the original VM, and hence shortcomings in the JIT-compiler optimizations or in the heuristics the JIT compiler employs.

Generation Set. In our experiments, we considered a single generation set consisting of the functions of popular web-development packages to generate our supernodes. We note that our supernodes could also be generated using popular Node.js packages or characteristic workloads.

Portability. Our technique could be easily implemented to speed up other Truffle languages such as Ruby, Python, and R. Moreover, the algorithms depicted in Section 4 are interpreter-independent and do not leverage any internal AST-interpreter implementation details, increasing the portability of our technique.

As mentioned in Section 4.2, the Java code generation of the supernodes is the only interpreter-specific part of our technique. Interpreter developers implementing our technique may need to manually write and maintain generation functions for the most common control-flow nodes. We consider the implementation of the code-generation functions an acceptable effort considering the performance gains thanks to supernodes.

8 Concluding Remarks

To conclude, we summarize our contributions, discuss the limitations of our technique, and outline our plans for future research.

Contributions. In this paper we propose a novel technique to generate AST supernodes. Our technique automatically improves the performance of AST interpreters, as it helps reducing typical interpretation overheads related to polymorphic call sites and control-flow-related exception handling. Our technique employs ahead-of-time code generation to automatically create executable supernodes, and runtime installation of matching supernodes.

We implement our technique in the GraalVM JavaScript language runtime (also known as Graal.js), and evaluate our implementation using the well-known web-tooling benchmark suite. Our evaluation shows that supernodes help reducing compilation-time and improve both interpreter and steady-state performance up to a factor of 1.33×, 1.24×, and 1.14×, respectively. Hence, the answers to our research questions RQ1, RQ2, and RQ3 are affirmative.

Our technique is specific to AST interpreters, and is implemented targeting the Truffle language implementation framework of GraalVM. Our technique could be easily ported to other existing Truffle AST interpreters such as, e.g., TruffleRuby [13], FastR [10], or GraalPy [12].

Limitations. The main limitation of our technique is that runtime lookup-tree generation and supernode installation increase VM startup time. We are investigating techniques to serialize the lookup-tree and further reduce the overhead of our technique during VM startup.

Even though our technique reduces the size of the code emitted by the JIT compiler by removing exceptions that model the control flow and several checks in polymorphic call sites, our technique increases the size of the interpreter source code and the VM build time (as shown in Section 5.5 and Section 5.6). For these reasons, when using our technique to build a production VM, it is necessary to find a proper trade-off between code size and performance improvement.

Finally, to generate effective supernodes, it is crucial to select a generation set that contains functions that exercise common control-flow patterns.

Future Work. As part of our future work, in addition to providing an in-depth explanation of the sources of steady-state speedups, we plan to expand our technique to create supernodes that encode non-control-flow nodes as well as supernodes that encode both control-flow and non-control-flow nodes. Moreover, we plan to conduct a large-scale analysis to identify the most frequently used supernodes that may be included in a production build of Graal.js. Finally, we plan to conduct an in-depth evaluation of our implementation on native images.

Acknowledgments

This work has been supported by Oracle (ERO project 1332) and by the Swiss National Science Foundation (project 200020_188688). We thank the VM Research Group at Oracle Labs for their support. Oracle, Java, and HotSpot are trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

References

- [1] Matthew Arnold, Adam Welc, and V. T. Rajan. 2005. Improving Virtual Machine Performance Using a Cross-Run Profile Repository. In *OOPSLA*. 297–311.
- [2] James R. Bell. 1973. Threaded Code. *Commun. ACM* 16, 6 (Jun 1973), 370–372.
- [3] Stefan Brunthaler. 2010. Efficient Interpretation Using Quickening. In *DLS*. 1–14.
- [4] Gilles Duboscq, Thomas Würthinger, Lukas Stadler, Christian Wimmer, Doug Simon, and Hanspeter Mössenböck. 2013. An Intermediate Representation for Speculative Optimizations in a Dynamic Compiler. In *VMIL*. 1–10.
- [5] Neil D. Jones. 1996. An Introduction to Partial Evaluation. *ACM Comput. Surv.* 28, 3 (Sep 1996), 480–503.
- [6] Vladimir Kozlov. 2018. JEP 295: Ahead-of-Time Compilation. <https://openjdk.java.net/jeps/295>
- [7] Octave Larose, Sophie Kaleba, and Stefan Marr. 2022. Less Is More: Merging AST Nodes To Optimize Interpreters. (February 2022). <https://kar.kent.ac.uk/93936/>
- [8] Zoltan Majo, Tobias Hartmann, Marcel Mohler, and Thomas R. Gross. 2017. Integrating Profile Caching into the HotSpot Multi-Tier Compilation System. In *ManLang*. 105–118.
- [9] Robert Nystrom. 2023. Crafting Interpreters. <https://craftinginterpreters.com/>
- [10] Oracle. 2023. fastr. <https://github.com/oracle/fastr>
- [11] Oracle. 2023. graaljs. <https://github.com/oracle/graaljs>
- [12] Oracle. 2023. graalpython. <https://github.com/oracle/graalpython>
- [13] Oracle. 2023. truffleruby. <https://github.com/oracle/truffleruby>
- [14] Oracle and/or its affiliates. 2021. GraalVM: Native Image. <https://www.graalvm.org/22.0/reference-manual/native-image/>
- [15] Guilherme Ottoni and Bin Liu. 2021. HHVM Jump-Start: Boosting Both Warmup and Steady-State Performance at Scale. In *CGO*. 340–350.
- [16] Todd A. Proebsting. 1995. Optimizing an ANSI C Interpreter with Superoperators. In *POPL*. 322–332.
- [17] V8 project authors. 2023. Web Tooling Benchmark. <https://github.com/v8/web-tooling-benchmark>
- [18] Gülfem Savrun-Yeniçeri, Wei Zhang, Huahan Zhang, Eric Seckler, Chen Li, Stefan Brunthaler, Per Larsen, and Michael Franz. 2014. Efficient Hosted Interpreters on the JVM. *ACM Trans. Archit. Code Optim.* 11, 1, Article 9 (Feb 2014), 24 pages.
- [19] Manuel Serrano. 2018. JavaScript AOT Compilation. In *DLS*. 50–63.
- [20] Manuel Serrano. 2021. Of JavaScript AOT Compilation Performance. *Proc. ACM Program. Lang.* 5, ICFP, Article 70 (aug 2021), 30 pages.
- [21] Gregory T. Sullivan, Derek L. Bruening, Iris Baron, Timothy Garnett, and Saman Amarasinghe. 2003. Dynamic Native Optimization of Interpreters. In *IVME*. 50–57.
- [22] Sun Microsystems, Inc. 1996. The Java Virtual Machine Specification. <https://book.huihoo.com/the-java-virtual-machine-specification/first-edition/Quick.doc.html>
- [23] Eugen N. Volanschi, Charles Consel, Gilles Muller, and Crispin Cowan. 1997. Declarative Specialization of Object-Oriented Programs. In *OOPSLA*. 286–300.
- [24] Christian Wimmer, Codrut Stancu, Peter Hofer, Vojin Jovanovic, Paul Wögerer, Peter Bernard Kessler, Oleg Pliss, and Thomas Würthinger. 2019. Initialize Once, Start Fast: Application Initialization at Build Time. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 184:1–184:29.
- [25] Christian Wimmer and Thomas Würthinger. 2012. Truffle: A Self-Optimizing Runtime System. In *SPLASH*. 13–14.
- [26] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. 2013. One VM to Rule Them All. In *Onward!* 187–204.
- [27] Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. 2012. Self-Optimizing AST Interpreters. *SIGPLAN Not.* 48, 2 (Oct 2012), 73–82.
- [28] Xiaoran Xu, Keith Cooper, Jacob Brock, Yan Zhang, and Handong Ye. 2018. ShareJIT: JIT Code Cache Sharing across Processes and Its Practical Implementation. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 124 (Oct 2018), 23 pages.

Received 2023-07-14; accepted 2023-09-03

A Monadic Framework for Name Resolution in Multi-phased Type Checkers

Casper Bach Poulsen

c.b.poulsen@tudelft.nl
Delft University of Technology
Netherlands

Aron Zwaan

a.s.zwaan@tudelft.nl
Delft University of Technology
Netherlands

Paul Hübner*

paul.huebner@googlemail.com
Delft University of Technology
Netherlands

Abstract

An important aspect of type checking is name resolution—i.e., determining the types of names by resolving them to a matching declaration. For most languages, we can give typing rules that define name resolution in a way that abstracts from what order different units of code should be checked in. However, implementations of type checkers in practice typically use multiple phases to ensure that declarations of resolvable names are available before names are resolved. This gives rise to a gap between typing rules that abstract from order of type checking and multi-phased type checkers that rely on explicit ordering.

This paper introduces techniques that reduce this gap. First, we introduce a monadic interface for phased name resolution which detects and rejects type checking runs with name resolution phasing errors where names were wrongly resolved because some declarations were not available when they were supposed to be. Second, building on recent work by Gibbons et al., we use applicative functors to compositionally map abstract syntax trees onto (phased) monadic computations that represent typing constraints. These techniques reduce the gap between type checker implementations and typing rules in the sense that (1) both are given by compositional mappings over abstract syntax trees, and (2) type checker cases consist of computations that roughly correspond to typing rule premises, except these are composed using monadic combinators. We demonstrate our approach by implementing type checkers for Mini-ML with Damas-Hindley-Milner type inference, and LM, a toy module language with a challenging import resolution policy.

CCS Concepts: • **Theory of computation** → **Program analysis; Type structures; Algebraic semantics;** • **Software and its engineering** → **Compilers; Semantics.**

* Author's current affiliation is KTH Royal Institute of Technology, Sweden



This work is licensed under a Creative Commons Attribution 4.0 International License.

GPCE '23, October 22–23, 2023, Cascais, Portugal

© 2023 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-0406-2/23/10.

<https://doi.org/10.1145/3624007.3624051>

Keywords: stable name resolution, scope graph, phasing, applicative functor composition, type checker

ACM Reference Format:

Casper Bach Poulsen, Aron Zwaan, and Paul Hübner. 2023. A Monadic Framework for Name Resolution in Multi-phased Type Checkers. In *Proceedings of the 22nd ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '23), October 22–23, 2023, Cascais, Portugal*. ACM, New York, NY, USA, 15 pages. <https://doi.org/10.1145/3624007.3624051>

1 Introduction

Most modern programming languages have a mutual dependency between typing and name resolution. For example, consider the following program in a language with modules:

```
1 module A {
2   import B
3   def f: Int = 1
4   def g: Int = h
5 }
6 module B {
7   def h: Int = A.f + 2
8 }
```

The named reference `h` (line 4) must resolve to the declaration in `B`, and `A.f` (line 7) must resolve to the declaration in `A`. This raises the question: in what order should we check the modules `A` and `B` such that we can determine that all named references indeed resolve to declarations of the right type?

For most type checkers in practice, the answer is to use multiple phases. For the module language above we can first analyze the overall module structure, associate types with each declared name, and about which names are reachable via declared imports. In a subsequent phase, this information is used to verify that named references on the right hand side of `defs` resolve to declarations of the right type.

In contrast, it is common for typing rules to abstract from phasing concerns. For example, the typing rules for Featherweight Java [15] and related calculi [3, 10, 20, 29] use class tables and abstract from how and when class table entries are constructed.

However, for type checker implementations, it is important to construct and query name binding information (e.g., in a *symbol table* [1] or a *scope graph* [28]) in the correct order. Attempting to resolve a name in a wrongly phased manner can lead to subtle bugs. For example, consider the following program with a nested module, where the reference `x` can

be resolved to an imported definition and a definition in the enclosing module:

```

1 module C {
2   def x: Int = 3
3   module D {
4     import E
5     def y: Int = x
6   }
7 }
8 module E {
9   def x: Int = 4
10 }
```

Since the nested module `D` imports `E`, the reference to `x` on line 5 could resolve to either the declaration in `C` (line 2) or `E` (line 9). Before we can resolve the right hand sides of **defs** and the `x` on line 5, we should *first* resolve the `E` import reference (line 4). This way, we know that the declarations in `E` are reachable from `D`. However, a wrongly phased type checker could fail to resolve imports before type checking the right hand side of **defs**. In this case, `x` on line 5 would resolve to the declaration on line 3. Since the intended semantics of our language is that declarations from imports shadow declarations from the lexical context, this *silently resolves names to the wrong declarations*.

This paper presents abstractions for multi-phased type checking that prevent such subtle errors. We contribute: (1) a new interface of effectful operations for creating and querying name binding information, using scope graphs [28, 33, 37, 38]; and (2) techniques that use *applicative functors* [9, 17, 24] to map abstract syntax trees (ASTs) to compact, explicitly phased, and effectful operations for type checking.

These contributions build on and extend previous work. Our use of applicative functors builds on the work of Gibbons et al. [9], and our scope graph operations are inspired by Rouvoet et al. [33]. A key feature of our operations is that they detect phasing errors during type checking and rule out subtle phasing errors such as the one above. The Statix language [33] provides this guarantee in a different way, via a static ownership type discipline and a sound (but incomplete) query scheduling algorithm. As we show in §5, our approach supports language features which Statix does not.

Most programming language implementations resolve names in multiple phases. For example, Haskell has a relatively simple module system that uses two phases [11, §2.3.2]. Scala combines a range of sophisticated name binding features such as inheritance, import statements, traits, type members, *dependent object types* [2], and *multi-staging* [22] in the MetaML tradition [35]. Languages such as Java, C#, Kotlin, and Rust also have multi phase type checking.

Our operations also require computations to run in a phased order. A naive approach to implementing this ordering is to traverse ASTs in multiple passes. However, such passes add syntactic overhead compared to typing rules that abstract from such phasing, as is common for typing rules

that use scope graphs [33, 38, 39]. We reduce the syntactic overhead of type checker implementations by compositionally mapping AST nodes onto monadic, multi-phased computations, using generic combinators for implementing the required phase ordering. This makes our type checker implementations more compact than explicitly phased implementations, akin to how monadic parser combinators [14] make parsers more compact than recursive descent parsers.

Our focus is on detecting phasing errors and on compactness. We believe our approach is not fundamentally at odds with efficiency but exploring this is left to future work. For now, our type checker implementations are likely have a sub-par performance compared with direct style type checkers.

We make the following technical contributions:

- We present (in §3) a monadic interface of operations for designing phased type checkers, using scope graphs. The operations dynamically detect and report name resolution phasing errors during type checking.
- Building on techniques for multi-phasing from Gibbons et al. [9] and Kidney and Wu [17], we present (in §4) generic combinators for multi-phased computation where later phases may depend on values from prior ones. In §4.5, we discuss how these techniques make type checker implementations more compact and more closely related to typing rules.
- We validate and evaluate our approach (in §5) by considering two case studies: a type checker for Mini-ML that uses Damas-Hindley-Milner type inference, and a type checker for a subset of the LM language due to Neron et al. [28].

The paper is structured as follows. §2 gives an overview of the problem and our solution. Then, §3 and §4 describe the implementation of our scope graph operations and techniques for phased computation using applicative functors. §5 describes case studies, §6 related work, and §7 concludes.

The framework and case studies are available in an artifact [32]. The abstractions in this paper are implemented in Haskell, and familiarity with Haskell is assumed.

2 The Multi-phased Name Resolution Problem and its Solution

Type checking generally requires producing name binding information in multiple phases. How do we represent such name binding information in typing rules and in compilers?

Typing rules most often use *type environments* that map names to types. However, few such specifications model the semantics of, e.g., modules or classes. In practice, compilers traditionally use *symbol tables* [1]. The details of symbol table implementations differ from language to language but a symbol table generally represents a “scope”. It stores the declared names of a scope, and (typically) the types of each name. By linking symbol tables to other symbol tables [1, §2.7], we can represent which scopes are reachable from the

```

1 module F {
2   import G
3   def i: Int = j
4 }
5 module G {
6   import H
7 }
8 module H {
9   def j: Int = 5
10 }

```

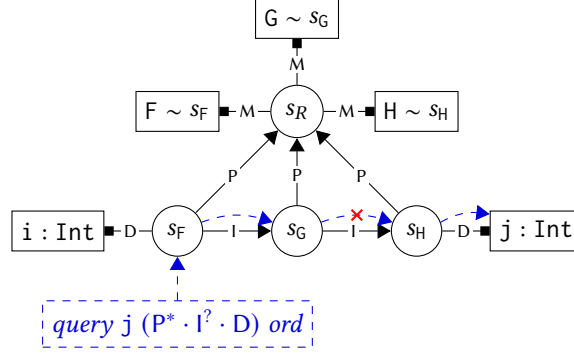


Figure 1. Name reachability example

```

1 module C {
2   def x: Int = 3
3   module D {
4     import E
5     def y: Int = x
6   }
7 }
8 module E {
9   def x: Int = 4
10 }

```

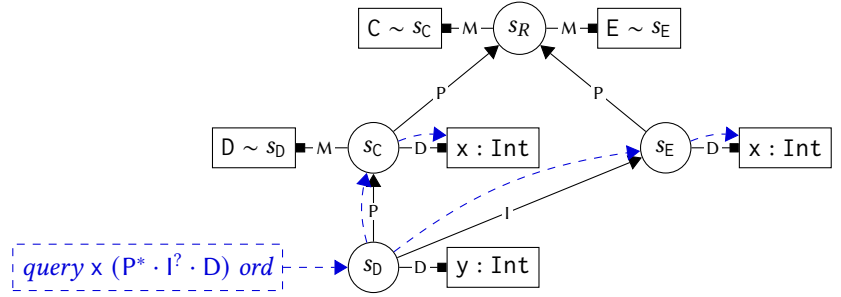


Figure 2. Name shadowing example

current scope; for example, names in the lexical context or names in imported modules. Compilers resolve names by traversing reachable symbol table entries and links.

Scope graphs are a mathematical model of name binding and name resolution that can be used as a stand-in replacement for both symbol tables and type environments. A type environment typically represents the set of *visible* names, whereas symbol tables represent the set of *reachable* names and model visibility as a search through these. We can think of a type environment as a “flattened” symbol table resulting from applying the visibility search procedure. The scope graph analogue to searching a symbol table is resolving a *name resolution query*. Thus scope graphs can replace both symbol tables and type environments.

Following Visser and co-authors [28, 33, 37–39], we can use scope graphs to define both typing rules and type checker implementations. In this section we give an introduction to scope graphs, the problem with phased name resolution, and how we solve the problem using a new set of monadic operations for scope graph construction.

2.1 Scope Graphs as a Model of Name Resolution

A scope graph is a data structure that represents the scopes and declarations of a program. Scopes (nodes in the scope graph) are conceptually similar to symbol tables, in that each scope is associated with declarations, and each scope may be connected to other scopes via *directed, labeled edges*.

Names are resolved by traversing edges in the scope graph and inspecting declarations. With symbol tables, the name resolution policy is given by a language specific algorithm that traverses tables. *Scope graph queries* succinctly define such traversals and name resolution policies. We illustrate how scope graphs and queries provide a declarative model of *reachability* and *visibility* (i.e., *shadowing*).

Reachability. A declaration is *reachable* if we can follow directed edges through the graph to reach it. For example, consider the program and scope graph in fig. 1. The program (left) has three modules that transitively import each other: **F** imports **G** and **G** imports **H**. On the right is its scope graph. There are four scopes, denoted by circles. S_R represents the “root scope” of the program, which contains declarations (labeled $\text{—} \blacksquare$ arrows from scopes) for each of the three modules. These declarations associate module names with their scopes. For example, $C \sim S_C$ associates **C** with scope S_C . Module scopes have declarations for each module member. Member declarations associate names with types; e.g., $i : \text{Int}$ in S_F . Labels on declaration edges indicates the kind of declaration: **D** for module members (**defs**); **M** for modules. Labels on edges between scopes indicates the scoping relation: **P** for lexical parent relations; **I** for import relations.

Named references are resolved by *querying* the scope graph. For example, the dashed blue box connected to S_F is a query for the named reference **j** (line 3). This name is

passed to the first argument of the query, which ensures only declarations with name j are matched. As the dashed blue edges show, it is possible to follow labeled edges to reach a j declaration. However, this path does not reflect the intended import semantics. The regex $P^* \cdot l^? \cdot D$ of the *query* says that a *valid* path has zero or more lexical parent edges, and *at most one import edge*. The shown path has two import steps so it does not match the query. The path would match if the query allowed transitive imports; e.g., $P^* \cdot l^* \cdot D$. The third argument of the query (*ord*) is an *ordering relation on paths*, which defines the *visibility semantics* of queries.

Visibility. The example from the introduction is repeated in fig. 2 (left). Its scope graph is on the right. The reference to x on line 5 can resolve to either x on line 2 or line 9. Which we prefer depends on the visibility semantics, given by an ordering relation on paths. This ordering decides which of the two blue paths (both valid according to the query reachability regex) shadows the other. Any type of ordering is possible, but a partial order on labels ($- < - \subseteq \text{Label} \times \text{Label}$) is sufficient for many languages.¹ For the example in fig. 2:

1. If $P < l$ then we prefer declarations reachable via the lexical context over via imports. A step-wise comparison of the paths in the figure gives precedence to the path through (s_0) , and the declaration on line 2 shadows the one on line 9.
2. If $l < P$ then we prefer declarations reachable via imports over via the lexical context. A step-wise comparison gives precedence to the path through scope (s_1) , and the declaration on line 9 shadows line 2.
3. If neither $P < l$ nor $l < P$, then neither declaration is preferred, and the x reference on line 5 is ambiguous.

2.2 The Multi-phased Name Resolution Problem

Scope graphs (like symbol tables) are data structures containing name binding information. The question is: how do type checkers build this data structure in a way that guarantees all relevant information is available before querying? A key challenge of guaranteeing this is that, to build some parts of the data structure, we need to query it (i.e., resolve names). For example, to construct the import edge between (s_0) and (s_1) in fig. 2, we must first resolve E (line 4). As discussed in the introduction, failing to construct this import edge before resolving x on line 6 causes our type checker to subtly fail. The next section summarizes how we address this challenge.

2.3 A Monadic Solution to the Multi-phased Name Resolution Problem

We introduce monadic operations for scope graph construction and querying that implicitly check that queries are *stable*; i.e., new edges and declarations do not change the results

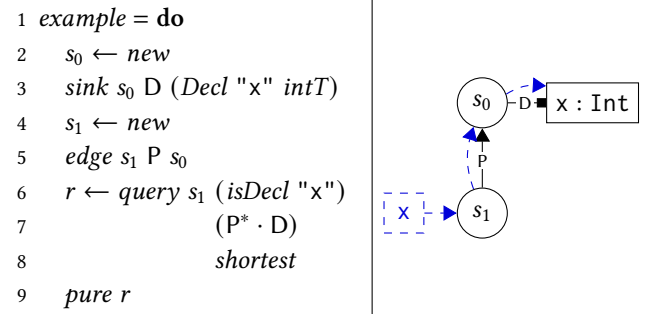
¹Some languages need a more general path ordering. For example, the MiniStatix specification of Scala compares full paths: <https://github.com/MetaBorgCube/scala.mstx#scala-precedence-as-a-path-order>

of previously executed queries. We illustrate query stability by example shortly. First, using M as the type of our monad for scope graph construction, our operations are:

```
new  :: M Scope
edge :: Scope → Label → Scope → M ()
sink :: Scope → Label → Decl → M ()
query :: Scope → (Decl → Bool) → Regex Label
      → (Path Label Decl → Path Label Decl → Bool)
      → M [Path Label Decl]
```

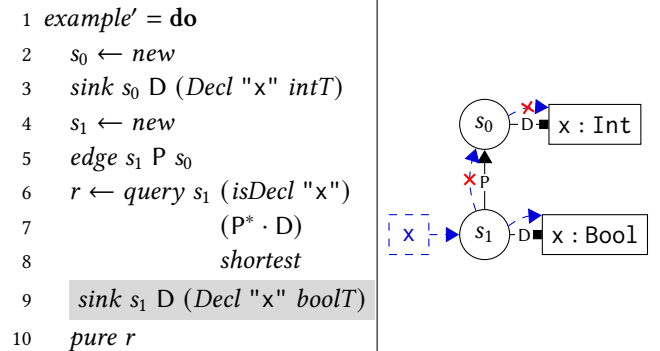
Here *new* creates a new scope, *edge* $s \ l \ s'$ creates an l -labeled edge between scopes, *sink* $s \ l \ d$ creates an l -labeled edge to a declaration (i.e., a node with no direct outgoing edges), and *query* $s \ dm \ re \ ord$ resolves declarations matching the predicate dm starting in scope s , using the reachability regex re , and ordering paths according to ord . The operations are parameterized by the types of *Label* and *Decl* while a *Path* is a sequence of labeled steps between scopes ending in a *Decl*.

To illustrate what it means for a query to be stable, consider the following example and its scope graph:



Here *isDecl* "x" matches a *Decl* named "x", and *shortest* prefers shorter paths, ignoring labels. The function *pure* $:: \forall a. a \rightarrow M a$ used on line 9 is a "pure" computation which returns a value as result without any side effects.

Extending the program with the declaration on line 9 below, the query on line 6 gives the same result. However, this result is *not a valid resolution in the final scope graph*!



In multi-phased type checkers, a query whose result changes depending on when it is run may give rise to subtle name resolution errors. Our operations avoid this by checking that queries are stable and raising an error if they are not.

The following law says that a query is stable if its order of execution is independent from any (possibly graph constructing) computation m :

$$\left(\text{do } m \quad \text{query } s \text{ } dm \text{ } re \text{ } ord \right) \equiv \left(\text{do } xs \leftarrow \text{query } s \text{ } dm \text{ } re \text{ } ord \quad m \quad \text{pure } xs \right)$$

As *example* and *example'* show, this law does not hold in general. However, our operations do satisfy the following law where $- \equiv_{\perp} -$ holds if both sides agree or if either side raises a *stability error*, indicating that a query result may have been violated:

$$\left(\text{do } m \quad \text{query } s \text{ } dm \text{ } re \text{ } ord \right) \equiv_{\perp} \left(\text{do } ys \leftarrow \text{query } s \text{ } dm \text{ } re \text{ } ord \quad m \quad \text{pure } ys \right)$$

§3 describes how our monad guarantees this property.

In summary, our monadic operations detect and reject type checker runs with phasing errors but do not statically guarantee their absence. Type checker engineers must therefore phase type checking in a way that avoids such errors. One solution is to implement multiple phases using multiple AST traversals. A more compact solution which we describe in §4 is to use a single pass to map AST nodes onto phased computations.

3 Monadic Scope Graph Construction

We consider how the operations discussed in the previous section construct scope graphs, and how they detect and reject programs with stability errors.

3.1 An Interface for Scope Graph Construction

Below is a type class that captures this monadic interface discussed in §2.3:²

```
class Monad m => SG l d m | m -> l d where
  new  :: m Scope
  edge :: Scope -> l -> Scope -> m ()
  sink :: Scope -> l -> d -> m ()
  query :: Scope -> (d -> Bool) -> RegEx l
         -> (Path l d -> Path l d -> Bool) -> m [Path l d]
```

One of our core contributions is that we provide an instance of this interface. The instance we provide in the code accompanying this paper [32] is defined using a Haskell embedding of *algebraic effects and handlers* [30]. The benefit of defining our instance in this way is that it is easy to compose the effects summarized by the SG interface above with other effects. For example, the case studies in §§5.1 and 5.2 make use of auxiliary effects for unification (used for type inference), emitting errors and backtracking. However, the details of embedding algebraic effects and handlers in Haskell are

²The $| m \rightarrow l d$ part indicates a functional dependency. That means that l and d should be determined by m . I.e., for any given m , there may be at most one pair of l and d such that a type class instance of $SG \ l \ d \ m$ exists.

beyond the scope of this paper. We summarize at a high level how our code implements the operations and invite readers to consult the code for further details.

Representing Scope Graphs. Our operations construct new nodes, edges, and sinks in scope graphs given by the following record type:

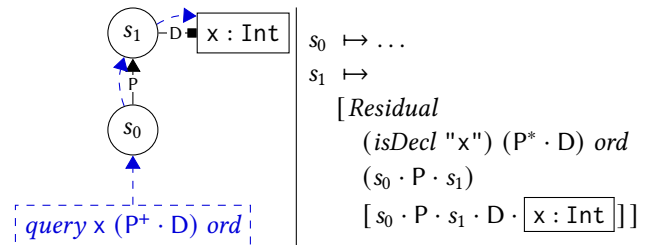
```
type Scope = Int
data Graph l d
  = Graph { scopes :: Scope
           , edges  :: Scope -> l -> [Scope]
           , sinks  :: Scope -> l -> [d] }
```

We use integers to represent scopes such that we have an infinite supply of fresh scopes. Edges in the graph are given by a (curried) mapping from scope-label pairs to a (possibly empty) list of target scopes. Declarations (or sinks) are similarly defined. The implementation of the operations in SG threads *Graphs* through in a stateful manner.

A naive implementation of *new*, *edge*, and *sink* would simply update the graph, and *query* would simply traverse the current graph to find matching paths. However, this naive implementation would suffer from the query stability problem discussed in the introduction and §2.2.

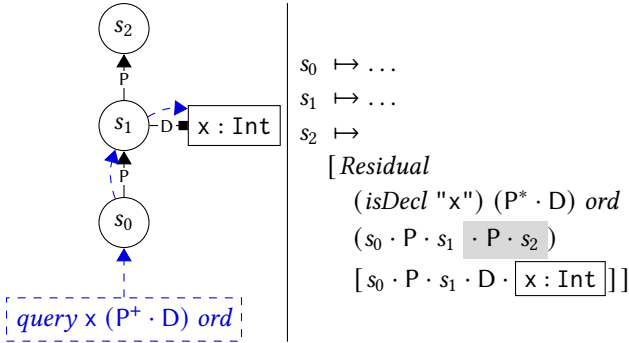
Detecting Stability Errors. Our implementation detects and reports stability violations; i.e., additions to the scope graph that cause an earlier query to give a different result. A simple but expensive way (in terms of runtime) to implement this check is to cache *every* query made during type checking, and then re-run every query when adding sinks or edges. Our operations implement a less expensive approach.

We associate each scope with residual queries which precisely define the traversals that have started from that scope in the past. When adding a new edge or declaration we execute that traversal over the new edge, as though the past query was run on the newly extended graph, and check that the query result remains unchanged. To illustrate, consider the scope graph on the left and the (truncated) residual queries for s_0 and s_1 on the right:



The first three arguments of *Residual* represents the state of the query that traversed scope s_1 to resolve the blue path. Since the query already traversed a *P* edge, the second argument is the *derivative* [4] of the original regex with respect to *P*. The fourth argument ($s_0 \cdot P \cdot s_1$) is the path leading from the source scope (s_0) to the current (s_1). The last argument is the final result of the original query at the time it was run.

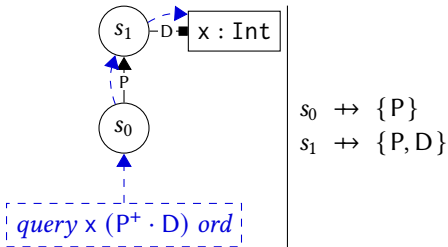
Say we extend scope ① with a new P-labeled edge to a new scope ②, as shown below:



In doing so, we enact the residual queries of s_1 which in turn associates a new residual query with ②. The residual for ② is the same as for ① except that the path leading from the source scope to the current scope now has an additional step, highlighted in gray.

Now say we extend ② with a new x declaration. The *sink* operation first adds this declaration to the scope graph and then enacts the residual queries of ②. This enactment yields a valid path since $s_0 \cdot P \cdot s_1 \cdot P \cdot s_2 \cdot D \cdot [x : \text{Int}]$ matches the (original) regex. This path is compared with the original set of paths using the ordering *ord*. If the path is not shadowed by any of the original paths in the residual, a changed query result is detected, and we raise a stability error.

Over-Approximating Stability Errors. The stability error detection described above is precise. However, that can make it hard to write tests that expose phasing errors. An approach that over-approximates stability but detects common phasing anti-patterns is sometimes desirable. The code artifact accompanying this paper implements both the precise stability error detection discussed above and the over-approximation we describe next. Let us revisit the example from before.



The *head set* (i.e., the set of characters that words accepted by the regex start with) of the query regex $P^+ \cdot D$ is $\{P\}$, so all outgoing P edges of ① are traversed by the query. If we add a new P edge to ① later, new declarations may become reachable which may give rise to stability errors. Our over-approximation of stability errors prevents this possibility by “closing” scope ① under P . ① is closed under both P and D because the head set of the query upon reaching ① is $\{P, D\}$. Attempting to add an edge or sink of a label that a scope is closed under raises a stability error.

Discussion. The code accompanying this paper contains implementations of both of the stability error strategies described above. While the over-approximating approach is more coarse grained, it enforces a certain programming discipline: we should only resolve names via a scope once all of the edges that the query may traverse have been added. Failing to follow this principle causes our interface to raise (potentially over-approximate) stability errors. In our experience, this helpfully pinpoints dubious phasing patterns in multi-phased type checkers.

Our implementation of both strategies satisfy the query stability law discussed in §2.3; i.e., for any m, s, d, r , and o :

$$\text{do } m; \text{query } s \text{ d r o} \equiv_{\perp} \text{do } xs \leftarrow \text{query } s \text{ d r o}; m; \text{pure } xs (*)$$

3.2 Explicitly Phased Type Checking

The previous section discussed how our monadic operations detect and reject stability errors. Let us consider how we can use these operations to define multi-phased type checkers for a simple module language whose abstract syntax is:

```
data MDec = Import String | Def String Ty Expr
           | Module String [MDec]
data Expr = Ident String | Lit Int | Tru | Fals
data Ty   = IntT | BoolT
```

MDec defines module member declarations (**imports**, **defs**, and **modules**), *Expr* expressions, and *Ty* types. For simplicity, expressions can only be identifiers, integer literals, or Boolean literals.

We can define an explicitly phased type checker for this language that uses three phases:

```
top :: SG Label Decl m => MDec -> m ()
top m = do s <- new
         (q1, q2) <- modules m s
         imports q1
         members q2

modules :: SG Label Decl m => MDec -> Scope
         -> m ([ (Scope, String)], [(Scope, (Ty, Expr))])
imports :: SG Label Decl m => [(Scope, String)] -> m ()
members :: SG Label Decl m => [(Scope, (Ty, Expr))]
         -> m ()
```

Here *top* orchestrates three phases which do the following. *modules* takes as input an *MDec* and its scope, and elaborates it into two working lists. As part of this elaboration, **def** declarations are added to their corresponding scopes, scopes are created for nested modules, and lexical parent (P) edges connect these module scopes to their lexical parents. The first working list represents import to be added to that scope, which is generally only possible once we know all module names. The second represents expressions to type check in that scope, which is generally only possible once all names

are imported. These working lists are processed in separate phases, using the *imports* and *members* functions.

The explicit phasing in *top* above uses three traversals: one over the abstract syntax to turn it into two working lists which we traverse next. A more compact alternative that does not use artificial intermediate working lists, is to map abstract syntax nodes onto phased computations that check well typedness, in a single traversal. We show how next.

4 Applicative Phasing and Its Application to Scope Graph Construction

As recently demonstrated by Gibbons et al. [9] and Kidney and Wu [17], applicative functors provides a useful abstraction for phased computation. In §4.1 we recall the concept of applicative functors. Next (§§4.2 and 4.3), we describe how and why we build on and adapt the techniques of Gibbons et al. [9]. Then §4.4 shows how to implement type checkers using applicative functors. Finally (in §4.5) we compare a case from a type checker written in this style with its corresponding typing rule.

4.1 Applicative Functors

Applicative functors are a standard feature in many Haskell libraries. These libraries usually use the *Applicative* type class [24].³ We use the following alternative but equivalent category theory inspired interface [24]:⁴

```
class Functor f ⇒ Monoidal f where
  unit :: f ()
  (★) :: f a → f b → f (a, b)
```

As we will see, the *Monoidal* interface is well-suited for defining phased computations, and we use that instead of the *Applicative* interface. If we think of *f* as a computation, the *unit* operation represents a pure computation returning a unit value whereas *★* combines two computations. The operations are subject to the following laws where $(f \times g) (x, y) = (f x, g y)$ and $\text{assoc } (a, (b, c)) = ((a, b), c)$:

$$\text{fmap } (f \times g) (m_1 \star m_2) \equiv (\text{fmap } f m_1) \star (\text{fmap } g m_2)$$

$$\text{fmap } \text{snd } (\text{unit} \star m) \equiv m$$

$$\text{fmap } \text{fst } (m \star \text{unit}) \equiv m$$

$$\text{fmap } \text{assoc } (m_1 \star (m_2 \star m_3)) \equiv (m_1 \star m_2) \star m_3$$

Next, we consider how to use *★* to compose multi-phased computations.

4.2 Functor Composition and Phasing

In the module language in §3.2, nested modules pose a phasing challenge. The challenge is that, before we can resolve imports, we need to know the names of all modules. Thus, phase 1 first creates the scopes of all modules; and only then

³<https://hackage.haskell.org/package/base-4.18.0.0/docs/Control-Applicative.html>

⁴In categorical terms, an applicative functor is a *strong lax monoidal functor*.

do we, in phase 2, resolve named imports. §3.2 used multiple traversals to implement this phasing. With applicative functors we can use a single traversal that returns a *phased computation* directly. But what is a phased computation?

The answer that Gibbons et al. [9] give to this question is *Day convolutions*. Briefly summarized, a Day convolution $\text{Day } f g a$ consists of a pair of two applicative functors, *f* and *g*, which represent two distinct phases. The idea is to construct phased computations using two functions:

$$\text{phase1} :: (\text{Monoidal } f, \text{Monoidal } g) \Rightarrow f a \rightarrow \text{Day } f g a$$

$$\text{phase2} :: (\text{Monoidal } f, \text{Monoidal } g) \Rightarrow g a \rightarrow \text{Day } f g a$$

Because Day convolutions are applicative functors themselves, computations can then be freely combined, in any order, using the *★* operation. In particular, the following holds for any $m_1 :: f a$ and $m_2 :: g b$ where *f*, *g* are applicative functors:

$$\text{phase1 } m_1 \star \text{phase2 } m_2 \equiv \text{fmap } \text{twist } (\text{phase2 } m_2 \star \text{phase1 } m_1)$$

$$\text{where } \text{twist } (x, y) = (y, x)$$

An attractive property of Day convolutions is that, in order to run a phased computation $\text{Day } f g a$, we only need to assume that *f* and *g* are themselves applicative functors. This means that Day convolutions can be used to phase a general class of computations, including monads since (in Haskell) all monads are applicative functors.

However, Day convolutions generally do not allow using of results from prior computations in subsequent ones, which is a common pattern in multi-phased type checkers. For example, if we infer the module type (associating member names to types, which we represent as a *Scope*) in a prior computation, we want a subsequent computation to use this module type to check that expressions in module members are well typed. That is, for two applicative functors *f* and *g*, we want to phase $m :: f \text{ Scope}$ and $k :: \text{Scope} \rightarrow g a$ where the *Scope* to pass to *k* is the one that *m* computes.

Consider how we might try to write this using only *phase1*, *phase2*, and the applicative functor product *★*:

$$\text{canWeDoThis} = \text{phase1 } m \star \text{phase2 } (k \text{ ??})$$

This will not work: *★* combines *m* and *k* in a way that their computations are independent, so we cannot fill in *??* with the result of *m* in this way. In fact, applicative functors generally do not allow the use of results from prior computations in the definition of subsequent ones, so we cannot in general write this program using only *phase1*, *phase2*, and *★*.

Instead, we could use the operations of *m* and *k* to pass information along from a prior computation to a subsequent one. For example, if *m* has operations for outputting values and *k* has operations that read such values as input, we can wire the outputs from *m* to inputs of *k*. For some applications, such wiring is natural; for example, the phased solution to the *repm* problem considered by Gibbons et al. [9]. We

might use a similar scheme for our type checkers, but then we need to label the scopes produced in a prior phase so that we can retrieve the intended module type by dereferencing the correct label in subsequent phases.

Instead of relying on such a labeling scheme, we use functor composition and monads. This lets us use Haskell functions—specifically, the two combinators we introduce in §4.3—to wire outputs to inputs such that (1) we do not have to invent a labeling scheme for labeling outputs produced in prior phases and unlabeled them in subsequent ones, and (2) the underlying monads do not need to support operations for output and input of labeled data. The downside is that we rely on binding combinators other than monadic bind. On the other hand, our combinators rely on standard machinery (functor composition and monadic bind), and provide a typed interface that helps enforce phase consistency.

The combinators we will introduce in the next section are based on *functor composition*; i.e.:

```
newtype (f ∘ g) a = Comp { getComp :: f (g a) }
```

We use composed functors $f \circ g$ to represent phased computations where f computations run in the first phase, and g in the second. We will exploit that g is nested inside f since this makes it possible for the g computation to *directly* depend on the values produced by the outer f computation. We show how in §4.3. First, we define some auxiliary and standard⁵ functor composition helper functions. First, composed functors are themselves functorial:

```
instance (Functor f, Functor g) => Functor (f ∘ g) where
  fmap f (Comp x) = Comp (fmap (fmap f) x)
```

Second, composed applicative functors are also applicative:

```
instance (Monoidal f, Monoidal g)
  => Monoidal (f ∘ g) where
  unit = Comp (fmap (const unit) unit)
  Comp x ★ Comp y = Comp (fmap (uncurry (★)) (x ★ y))
```

This *Monoidal* instance lets us compose phased computations in any order, similarly to Day convolutions. In particular, the functions below are analogous to *phase1* and *phase2*:

```
here :: (Functor f, Monoidal g) => f a -> (f ∘ g) a
here m = Comp (fmap (λx -> fmap (const x) unit) m)

there :: Monoidal f => g a -> (f ∘ g) a
there m = Comp (fmap (const m) unit)
```

The following holds for any $m_1 :: f\ a$ and $m_2 :: g\ a$ where f and g are applicative functors:

```
here m1 ★ there m2 ≡ fmap twist (there m2 ★ here m1) (†)
```

⁵<https://hackage.haskell.org/package/base-4.18.0.0/docs/Data-Functor-Compose.html>

4.3 Chaining Phases

While *here* and *there* combine phased computations, they do not allow us to define phases where later phases explicitly depend on values produced by earlier phases. The following functions do:

```
(⊞) :: Functor f => f a -> (a -> g b) -> (f ∘ g) b
m ⊞ k = Comp (fmap k m)

(⊡) :: Monad f => f a -> (a -> (f ∘ g) b) -> (f ∘ g) b
m ⊡ k = Comp (m >= (getComp ∘ k))
```

Both functions define a two-phased computation where the second phase may depend on the first. Both $m \oplus k$ and $m \oplus k$ assume that m is a phase 1 computation. The difference is that $m \oplus k$ assumes that the effects in k are phase 2 computations, whereas $m \oplus k$ allows k to have both phase 1 and phase 2 computations. The latter uses the monadic bind of f to sequence m with the phase 1 computations in k . In the next section we illustrate how the combinators above can be used to phase computations in type checkers.

4.4 Implicitly Phased Scope Graph Construction

Using the machinery from sections 4.1 to 4.3 we can use a single traversal over ASTs to construct a phased computation representing the type checking constraints of a program. Figure 3 (left) shows the cases of a type checker for the *MDec* type of the module language from §3.2. On the right in the figure are the corresponding typing rules which we will discuss in §4.5. In addition to the *SG* monad type class from §3.1, it uses the following error monad type class:

```
class Monad m => Err m where err :: String -> m a
```

The type signature on line 2 shows that the *mdec* function produces a computation in three phases where each phase has the same set of effects (m). The *Module* case of *mdec* function on line 3-7 uses \oplus to create a module scope and declaration in phase 1, and passes the created scope to the computation which uses the *traverse* function to recursively call *mdec* to check module member declarations where:⁶

```
traverse :: Monoidal f => (a -> f b) -> [a] -> f [b]
```

Here *traverse* uses *Monoidal* to compose the phased computations resulting from recursively calling *mdec* in a manner that respects eq. (†) from §4.2. The use of \oplus composes the phase 1 computation on line 4-6 with the phase 1 computations recursively created by *mdec* calls in line 7.

Lines 8-13 of fig. 3 use *there* (*here* ...) on line 8 to insert the computation on line 9-13 in phase 2. The computation resolves a named import and creates an import edge between the current scope and the resolved module scope. For simplicity, the module language and query on line 9 only

⁶<https://hackage.haskell.org/package/base-4.16.0.0/docs/Data-Traversable.html>

```

1 mdec :: (SG Label Decl m, Err m)
2   ⇒ MDec → Scope → (m ◦ m ◦ m) ()
3 mdec (Module x mds) s ① = void
4 ((do sm ← new ②
5   sink s M (ModDecl x sm) ③
6   pure sm) ⋈ λsm →
7   (traverse (λm → mdec m sm) mds) ④)
8 mdec (Import x) s ⑤ = there (here (do
9   ps ← query s (isModDecl x) (P* · M) pCompare ⑥
10  case map scOf ps ⑦ of
11    [Just si] → edge s l si ⑦
12    _ → err "bad import"))
13 mdec (Def x t e) s ⑧ = void
14   ( here (sink s D (DefDecl x t)) ⑨
15   ★ there (there (expr e s t) ⑩))

```

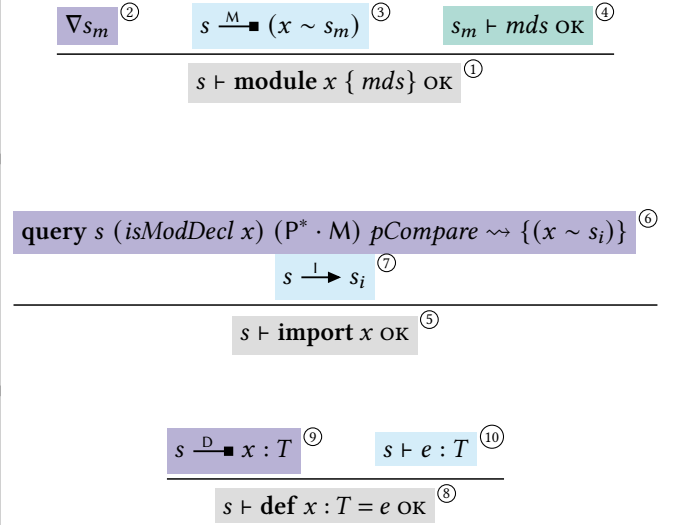


Figure 3. Representative cases of a type checker for the module language from §3.2.

allows modules to be resolved via lexical scoping; i.e., by following P edges until an M-labeled edge leading to a module declaration is found. Also for simplicity, the module language does not support qualified (module) names. Thus, module import resolution in this language is *relative* (i.e., imports are resolved starting in the scope where they occur), *unordered* (imports can occur anywhere and will be in scope for the entire module), *glob* (also known as wildcard—i.e., they import all definitions from a module), and *import insensitive* (i.e., modules cannot be resolved via import statements; only via the lexical context). In §5.2 we show how to support *import sensitive module resolution* (i.e., modules can be resolved via import statements) as well as type inference.

The final case in fig. 3 uses ★ to compose a phase 1 and a phase 3 computation which, respectively, creates a declaration for the *Def* in the current scope, and then checks the expression of the *Def*.

4.5 Correspondence to Typing Rules

The code on the left in fig. 3 defines a type checker for the rules shown on the right in the same figure. The typing rules are written in a similar style as in the work of van Antwerpen et al. [38]. The rules themselves are transcribed from the MiniStatix specification of LM due to Rouvoet et al. [33], except that the rule for imports only allows enclosing modules to be imported. In contrast, the MiniStatix specification of LM due to Rouvoet et al. [33] has support for *import sensitive module resolution* which we will show how

to type check in §5.2 The colors and numbers indicate how each premise and conclusion of the typing rules is checked in our type checker. Each case of *mdec* corresponds with one of the rules, and all premises correspond to an expression within the case. Besides the standard **do** keyword, the uncolored parts of the code either (1) phase the type checker using *here*, *there*, ★, and ⋈, (2) perform error handling, or (3) use *void :: Functor f ⇒ f a → f ()* to discard return values to match the type signature. In contrast, in the type checker discussed in §3.2, the premises would be scattered across different functions: declarations are created by the *modules* function, whereas the right hand sides of **defs** are type checked by the *members* function.

5 Case Studies

We present two case studies (also included in the artifact [32]) that explore the expressiveness of our approach. First, §5.1 shows how Damas-Hindley-Milner type inference (i.e. Algorithm W [7]) can be implemented using our approach. Next, we extend the language from §§3.2 and 4.4 with *import sensitive* module resolution. Both case studies could not be operationalized in previous scope graph based frameworks [33, 39].

5.1 Mini-ML with Damas-Hindley-Milner Inference

According to Zwaan and van Antwerpen [39], one of the primary limitations of Statix is its lack of support for Damas-Hindley-Milner-style type inference [6]. Here we present

the first scope graph based type checker for MiniML [16], a language with let polymorphism [26]. The language has the usual λ calculus constructs, as well as let bindings and number literals/addition. Its (truncated) syntax is:

```
data MiniML
  = Ident String
  | Let String MiniML MiniML
  | ...
```

To infer types for MiniML we will make use of operations for generating new meta-variables and unify first-order terms given by the following syntax:

```
data Term = Var Int | Term String [Term]
```

Here we use integer indices to distinguish different variables. We use terms to represent types, and use the following smart constructors for number and function types:

```
type Ty = Term
numT = Term "Num" []
funT s t = Term "->" [s, t]
```

The operations of the type class below generate new meta-variables (*Vars*) and unify terms:

```
class Monad m => Unif m where
  exists :: m Term
  equals :: Term -> Term -> m ()
  inspect :: Term -> m Term
```

Here *exists* creates a new meta-variable (e.g., *Var x* where *x* is a fresh integer); *equals* $t_1 \ t_2$ either unifies t_1 and t_2 or raises an error if they cannot be unified; and *inspect* t inspects a term, applying all substitutions resulting from previously performed unifications.

Polymorphic types (type *schemes*) in MiniML are given by the *Scheme* data type.

```
data Scheme = Scheme { sbinds :: [Int], stype :: Ty }
```

Using schemes as the type of declarations, our type checker uses a single phase and is given by the *mml* function whose type is shown below:

```
data Label = P | D
data Decl = Decl String Scheme
mml :: (SG Label Decl m, Err m, Unif m)
      => MiniML -> Scope -> Ty -> m ()
```

The function takes as input a *MiniML* expression, the current *Scope*, and the *Type* that the input expression should be checked to have. We use unification to infer the type. If the type of the input expression is not known beforehand, the *Ty* argument of *mml* is a unification variable.

The implementation of *mml* follows Algorithm W [7]. We consider two of the most interesting cases, starting with the case for variables.

```
1 mml (Ident x) s t = do
2   ps <- query s (isDecl x) (P* · D) pShortest
3   case map schemeOf ps of
4     [sc] -> do dt <- inst sc; equals t dt
5     _ -> err ("bad identifier: " + x)
6   inst :: Unif m => Scheme -> m Term
```

The query on line 2 resolves the identifier. If the query succeeds, we call *inst* to instantiate the type scheme, and then unify the resulting term with the input type t (line 4). The (elided) implementation of *inst* substitutes the variables bound by the type scheme by fresh variables.

The other interesting case is for let bindings:

```
1 mml (Let x e body) s t = do
2   t' <- exists
3   mml e s t'
4   t'' <- inspect t'
5   st <- gen s t''
6   s' <- new
7   edge s' P s
8   sink s' D (Decl x st)
9   mml body s' t
```

Lines 2-3 introduce a fresh unification variable t' and use it to infer the type of the let bound expression e . Next, on lines 4-5, we first inspect the inferred type, and then call *gen* to generalize the type relative to the current scope s and create a new type scheme st . This scheme becomes the type of x declared in the sub-scope s' used to check the body of the let expression (line 6-9). Here *gen* is defined as follows:

```
gen :: SG Label Decl m => Scope -> Term -> m Scheme
gen s t = do
  ps <- query s (const True) (P* · D) noOrd
  let fvs = concatMap (fv ∘ stype ∘ schemeOf) ps
  pure (Scheme (fv t \ fvs) t)
```

It first collects all declarations in scope, using *(const True)* to match any and all declarations and *noOrd* to prevent shadowing. Then, it projects all free variables of declaration types, using the utility function $fv :: Ty \rightarrow [Int]$. This is analogous to how Algorithm W [7] inspects all free variables in a type environment. Finally, it creates a scheme that quantifies over the truly free variables in t ; i.e., variables that do not occur in types reachable from the current scope (fv_s).

Discussion. This case study shows that the lack of support for Damas-Hindley-Milner type systems in Statix is not a limitation of scope graphs. In fact, the *generalize* operation, as traditionally defined on environments, maps rather naturally to scope graphs, when given control over the order of operations and the ability to inspect terms. While the definition of *MiniML* only uses lexical scoping, our *mml*, *gen*, and *inst* functions can be extended to support non-lexical scoping (e.g., modules and imports) without significant changes.

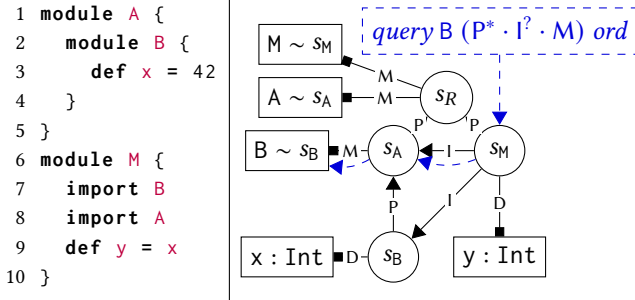


Figure 4. A program and scope graph with import sensitive module resolution.

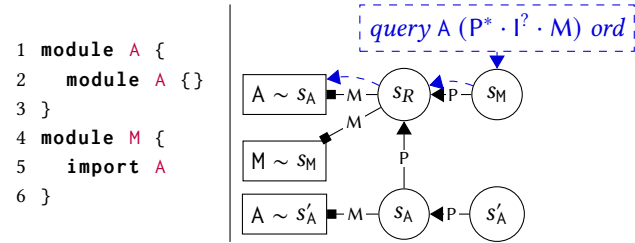


Figure 5. An ambiguous program and its partial scope graph. The program has no model, as adding an import edge from s_M to s_A contradicts its own resolution via the P edge to s_R .

5.2 Language with Modules (LM)

Our second case study is LM, a proof-of-concept language by Neron et al. [28]. LM is similar to the simple module language that we gave a type checker for in §3.2, with two important differences: (1) LM has *optional* type annotations and relies on (monomorphic) type inference; and (2) LM uses *import sensitive module resolution* (in contrast to the example in §4.4 and fig. 3).

Import Sensitive Module Resolution. Consider the example in fig. 4. The imports on line 7 and 8 are unordered and *import sensitive*, in the sense that module names can be resolved through other imports. The **A** import on line 8 resolves via the lexical context, to the declaration on line 1. Because of this, s_M has an import edge to s_A in the scope graph. The **B** import is resolved using the query shown in the figure. The regex of this query allows modules to be resolved via an import edge (that is, it is *import sensitive*). Therefore, the import edge to s_A can be used to resolve module **B**, resulting in an edge to s_B .

The combination of unordered imports and import sensitive module resolution has a subtle semantics in some cases. For example, consider fig. 5 (borrowed from Hübner [13]). It is possible to resolve the **A** import on line 5 to the **A** declaration on line 1 along the shown path. Because of this resolution, we should add an import edge between s_M and s_A in the scope graph. But, if we add this edge, then the query in

the figure becomes *unstable*. In LM, imports have precedence over the lexical context, so the added edge would cause the query in the figure to resolve to the (inner) **A** declared in s_A instead of s_R . Because no graph exists where all names can be stably resolved, the program has no model.

This subtlety illustrates a key challenge of type checking LM programs. Because module resolution is unordered and import sensitive, each import may depend on an arbitrary sequence of other imports. So how do we decide which the order imports should be resolved in?

Implementation. Figure 6 shows how we implement import resolution. The highlights on the left summarize differences from §4.4, fig. 3. These differences stem from how we deal with import resolution. Because of import sensitivity, we use an *Aggr* functor to aggregate the list of all imports to be resolved in phase 2. As shown in the type of *lm* on line 3, this functor is inserted between phase 1 and phase 2. Its definition and relevant *Monoidal* instance is:

```

data Aggr r a = Aggr r (r → a) deriving Functor
instance Monoid r ⇒ Monoidal (Aggr r) where
  unit = Aggr mempty (const ())
  Aggr xs m ★ Aggr ys n =
    Aggr (xs <⊔ ys) (λr → (m r, n r))

```

This *Monoidal* instance assumes that *r* is a monoid, and uses the monoidal product ($<⊔$) for aggregation. In line 11 in fig. 6 we use *Aggr* to aggregate all imports, indexed by a scope. In line 9, we bind the final aggregation to variable *is* and call *imps* to perform import sensitive module resolution.

The *imps* function on the right in fig. 6 implements import sensitive module resolution. The most challenging part of this is that we do not know the correct order in which the imports must be resolved. To compute that, we must be able to speculatively add edges and do queries, backtracking if an import fails. To implement this, we use the *anyOrder* operation:

```

class Monad m ⇒ AnyOrder m where
  anyOrder :: [m (Maybe ())] → m () → m ()

```

This operation implements the following kind of error handling behavior. The first parameter is a list of computations that may fail (i.e., return *Nothing*). The operation tries to find an order to execute these computations in which (1) all computations succeed, and (2) there are no stability errors. If no such order exists, the second argument is invoked to handle the failure.

Lines 14-16 uses the *anyOrder* operation to search for a stable import resolution order. The first argument is a list of computations that resolve each import, given by *is* *s*, of a module. If no stable order is found, the second argument is run to raise an error.

The type checker in fig. 6 was validated using (1) 15/19 of the test cases of Rouvoet et al. [33] (we excluded four because

```

1   $lm :: (SG\ Label\ Decl\ m, Err\ m, Unif\ m, AnyOrder\ m)$ 
2     $\Rightarrow LMDec \rightarrow Scope$ 
3     $\rightarrow (m \circ Aggr\ (Scope \rightarrow [String])) \circ m \circ m\ ()$ 
4   $lm\ (Module\ x\ mds)\ s = void$ 
5     $(\text{do } s' \leftarrow new$ 
6       $sink\ s\ M\ (ModDecl\ x\ s')$ 
7       $pure\ s') \multimap \lambda s' \rightarrow$ 
8       $traverse\ (\lambda m \rightarrow lm\ m\ s')\ mds$ 
9     $\star there\ (Aggr\ (const\ [])\ id \multimap \lambda is \rightarrow here\ (imps\ is\ s')))$ 

```

```

10  $lm\ (Import\ x)\ s =$ 
11    $there\ (here\ (Aggr\ (\lambda s' \rightarrow [x\ |\ s \equiv s'])\ (const\ ())))$ 
12  $imps :: (SG\ Label\ Decl\ m, Err\ m, Unif\ m, AnyOrder\ m)$ 
13    $\Rightarrow (Scope \rightarrow [String]) \rightarrow Scope \rightarrow m\ ()$ 
14  $imps\ is = anyOrder$ 
15    $(map\ (\lambda i \rightarrow resolveMod\ i\ s)\ (is\ s))$ 
16    $(err\ "Could\ not\ resolve\ imports")$ 

```

Figure 6. Representative cases of a type checker for LM.

they used qualified names, which are not included in our language), (2) seven additional test cases from Hübner [13], and (3) two new test cases for query stability corner cases. Unlike Statix, all cases pass.

Discussion. Certain optimisations over this scheme are conceivable. For example, failures can help to prune the remaining permutations. When an import query is invalidated by another import, all permutations containing imports in the same order can be filtered. Similarly, when an import does not resolve, we only need to retain permutations that have at least one unresolved import before the failing one.

While backtracking over all permutations seems expensive, it is an improvement over other approaches found in the literature. NaBL2 [37] re-resolves imports at *every reference*, even variables inside a scope, while we perform import resolution once per module. Apart from this, we are not aware of any implementation for such a module system.

6 Related Work

We discuss related work on scope graphs and phasing.

6.1 Scope Graphs

Scope graphs were originally introduced by Neron et al. [28]. In their model, imports are first-class, whereas we model them using queries and edges. Van Antwerpen et al. [37] introduce *NaBL2*, a type system specification meta-language using scope graph for name resolution. To cover more type systems, van Antwerpen et al. [38] refined the scope graph model, and embedded it in the logic language Statix [38]. We use this model in this paper. In contrast to Neron et al., this model allows interleaving of scope graph construction and querying. Statix is a declarative language, in which a model satisfying all constraints is found by constraint solving. This gives rise to a scheduling problem: when can queries be executed without later edge additions invalidating the result. The answer given by Rouvoet et al. [33] is, whenever the query does not traverse scopes that have *critical edges*; i.e., edges that give rise to new paths for the query. Since

finding critical edges is as difficult as solving the constraint program, Rouvoet et al. use *weakly critical edges*, an over-approximation of critical edges which can be inferred by a static analysis of Statix rules. In our approach, phasing is done manually, as opposed to automatically by Statix. As our case studies show, this lets us type check languages beyond what Statix supports. On the other hand, Statix supports dynamically scheduled scope graph construction which may be difficult to support in our explicitly phased approach.

Our over-approximating stability error detection from §3.1 implements the dual of weakly critical edges: instead of scheduling queries after no edges are added anymore, we prevent adding edges after a query traversed that edge. Our precise stability error detection from §3.1 detects real critical edges, which is not possible in Statix. A more in-depth overview of the evolution and application of scope graphs is given by Zwaan and van Antwerpen [39].

Casamento [5] uses scope graphs to write correct-by-construction type checkers in Agda that yield intrinsically typed syntax for languages where scope graph construction does not depend on querying a partially constructed graph.

6.2 Phasing in Other Type Checkers

Rouvoet et al. [33] observe that the module system of Rust has features comparable to our LM case study. A key difference is that enclosing modules are not reachable by default, but must be brought in scope using (e.g.) a `use super::*` statement. Hence, modules that are in (lexical) scope cannot be shadowed by modules that are imported later. Thus, newly resolved imports can only make other module references *ambiguous*. As such, Rust's module resolution does not require backtracking, but is implemented with a fix-point computation instead. In addition, it has a 'finalize' phase that checks whether the import resolution is stable.⁷ The *anyOrder* operation from §5.2 performs similar checks.

The Scala 3 compiler also generalizes the notion of symbol tables, but in a different way than scope graphs do [21].

⁷https://github.com/rust-lang/rust/blob/55e8df2b0e3c4494b77f2431b912c51e6fe733ba/compiler/rustc_resolve/src/imports.rs#L466

Internally, different phases have different contexts. Each context carries meanings (denotations) for symbols. In this way, information (such as types) can be changed between phases, which is used to keep typing information accurate under transformations. However, *within* the single type checking phase, names are resolved in a way that looks like traditional lookups in symbol tables. Thus, there appears to be no explicit stability checking, neither within the type checking phase nor in the updates to a context that a transformation can introduce. In future research, we could investigate whether our monadic scope graph framework could be extended to track stability under (controlled) transformations.

Another common framework to write type checkers in is *attribute grammars* [19]. In this system, attributes can be associated with productions in a grammar. Attributes are evaluated using small ‘functions’ that can refer to attributes of other nodes. Ironically, this paradigm has made the opposite development regarding stratification as the scope graph framework has. While canonical attribute grammar execution followed a statically determined ordering of multiple traversals, later extensions (aiming to improve expressiveness) introduced dynamic scheduling to the paradigm. Some of these extensions include *reference attributes* [8, 12], which allow attributes to evaluate to references to other AST nodes; *parameterized attributes* [8] which allow passing parameters to attributes; and *collection attributes* [23] allow attributes to be a “combination of contributions from distant nodes in the abstract syntax tree”. Evaluation of each of these kinds of attributes is usually done dynamically; i.e., on-demand. This gives more flexibility than our approach, at the cost of declarative appeal and (sometimes) termination guarantees. Rouvoet et al. [34, §E.1] claim that Statix’ scheduling is more precise than JastAdd’s collection attributes. This would imply that our framework could be able to express phasing that cannot be derived from attribute grammars using collection attributes, although examples are still to be found.

Finally, an earlier version of our library has been used to explore how to type check a Java subset [36], a Scala subset [25], type classes [27], and substructural types [18]. Our LM case study is based Hübner’s work [13], with two main differences. First, we use applicative functors for phasing; second, we use back-tracking to implement import sensitive module resolution whereas Hübner uses a dedicated import resolution algorithm. We conjecture that our implementation is sound and complete w.r.t. the typing rules of LM whereas Hübner’s algorithm is known to be incomplete.

6.3 (Higher-Order) Algebraic Effects and Handlers

In the code artifact accompanying the paper [32], we used a Haskell embedding of *effects and handlers* [30] to provide an implementation of the monad which the code examples in this paper leaves abstract. This allowed us to separately define and subsequently compose effects. For example, our

implementation of LM composes separately defined handlers for unification operations (the *Unif* interface from §5.1) and scope graph construction operations (the *SG* interface from §3.1). To define *higher-order* effects (i.e., effects where operations can have computations as arguments, such as the *anyOrder* operation) in §5.2, we used *hefty algebras* [31] to elaborate higher-order effects into algebraic effects.

7 Conclusion

Implementing type systems for languages with complex (non-lexical) name binding features is challenging. A key challenge is that all relevant name binding information must be aggregated before a name is resolved. We showed how to address this challenge using scope graph constructing operations which dynamically detect and reject programs that fail to aggregate relevant name binding information before name resolution. Scheduling queries correctly typically requires multi-phased type checking. However, it is often possible to define typing rules that abstract from such operational phasing concerns. To make type checker implementations more compact, we used recently developed techniques from previous work on applicative functors to compositionally map program ASTs onto computations representing phased typing constraints. This yields an expressive framework for sound name resolution of complex binding structures, and reduces the gap between type checker implementations and typing rules since type checker cases consist of computations that roughly correspond to typing rule premises, except these are composed using monadic combinators.

Future Work. While our case studies show that our approach supports type systems that cannot be operationalized using Statix, it is an open question whether the reverse is true. We enforce a static number of phases, while (e.g.) constraint-based approaches support more dynamic scheduling. To the best of our knowledge, a precise characterization of static vs. dynamic phasing, and a comparison of their expressiveness, is yet to be made. For example, it is not yet clear whether a type system with simple record inference (e.g. as presented by Van Antwerpen et al. [38, §2.3]) can be ported to our framework. In addition, scheduling schemes designed using this framework might inform refinements of algorithms used in automatically scheduling systems, such as Statix. While our approach is expressive, our type checker implementations are currently not very efficient. In future work, we would like to explore a more efficient implementation of scope graph construction and querying, and explore *fusing* phases similarly to Gibbons et al. [9].

Acknowledgments

Thanks to the anonymous reviewers and Tom Schrijvers for helpful comments that improved the paper. The first author was supported by the Programming and Validating Restructurings project (17933, NWO-TTW, MasCot).

References

- [1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. 1986. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley. <https://www.worldcat.org/oclc/12285707>
- [2] Nada Amin, Samuel Grütter, Martin Odersky, Tiark Rompf, and Sandro Stucki. 2016. The Essence of Dependent Object Types. In *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday (Lecture Notes in Computer Science, Vol. 9600)*, Sam Lindley, Conor McBride, Philip W. Trinder, and Donald Sannella (Eds.). Springer, 249–272. https://doi.org/10.1007/978-3-319-30936-1_14
- [3] G. M. Bierman, M. J. Parkinson, and A. M. Pitts. 2003. *MJ: An imperative core calculus for Java and Java with effects*. Technical Report UCAM-CL-TR-563. University of Cambridge.
- [4] Janusz A. Brzozowski. 1964. Derivatives of Regular Expressions. *J. ACM* 11, 4 (1964), 481–494. <https://doi.org/10.1145/321239.321249>
- [5] Katherine Imhoff Casamento. 2019. *Correct-by-Construction Type-checking with Scope Graphs*. Master's thesis. Portland State University. Department of Computer Science.
- [6] Luís Damas. 1984. *Type assignment in programming languages*. Ph. D. Dissertation. University of Edinburgh, UK. <https://hdl.handle.net/1842/13555>
- [7] Luís Damas and Robin Milner. 1982. Principal Type-Schemes for Functional Programs. In *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages, Albuquerque, New Mexico, USA, January 1982*, Richard A. DeMillo (Ed.). ACM Press, 207–212. <https://doi.org/10.1145/582153.582176>
- [8] Torbjörn Ekman and Görel Hedin. 2007. The JastAdd system - modular extensible compiler construction. *Science of Computer Programming* 69, 1-3 (2007), 14–26. <https://doi.org/10.1016/j.scico.2007.02.003>
- [9] Jeremy Gibbons, Donnacha Oisín Kidney, Tom Schrijvers, and Nicolas Wu. 2022. Breadth-First Traversal via Staging. In *Mathematics of Program Construction - 14th International Conference, MPC 2022, Tbilisi, Georgia, September 26–28, 2022, Proceedings (Lecture Notes in Computer Science, Vol. 13544)*, Ekaterina Komendantskaya (Ed.). Springer, 1–33. https://doi.org/10.1007/978-3-031-16912-0_1
- [10] Robert Griesemer, Raymond Hu, Wen Kokke, Julien Lange, Ian Lance Taylor, Bernardo Toninho, Philip Wadler, and Nobuko Yoshida. 2020. Featherweight go. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 149:1–149:29. <https://doi.org/10.1145/3428217>
- [11] Cordelia V. Hall, Kevin Hammond, Will Partain, Simon L. Peyton Jones, and Philip Wadler. 1992. The Glasgow Haskell Compiler: A Retrospective. In *Functional Programming, Glasgow 1992, Proceedings of the 1992 Glasgow Workshop on Functional Programming, Ayr, Scotland, UK, 6–8 July 1992 (Workshops in Computing)*, John Launchbury and Patrick M. Sansom (Eds.). Springer, 62–71. https://doi.org/10.1007/978-1-4471-3215-8_6
- [12] Görel Hedin. 2000. Reference Attributed Grammars. *Informatica (Slovenia)* 24, 3 (2000).
- [13] Paul Hübner. 2023. *Type-Checking Modules and Imports using Scope Graphs*. Bachelor's Thesis. Delft University of Technology. <http://resolver.tudelft.nl/uuid:e7f16989-9aca-4707-9d4a-74eba2adc5e4>
- [14] Graham Hutton and Erik Meijer. 1996. Monadic parser combinators.
- [15] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. 2001. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.* 23, 3 (2001), 396–450. <https://doi.org/10.1145/503502.503505>
- [16] G. Kahn. 1987. Natural semantics. In *STACS 87*, Franz J. Brandenburg, Guy Vidal-Naquet, and Martin Wirsing (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 22–39.
- [17] Donnacha Oisín Kidney and Nicolas Wu. 2021. Algebras for weighted search. *Proc. ACM Program. Lang.* 5, ICFP (2021), 1–30. <https://doi.org/10.1145/3473577>
- [18] Jan Knapen. 2023. *Type checker for a language with a substructural type system using scope graphs*. Bachelor's Thesis. Delft University of Technology. <http://resolver.tudelft.nl/uuid:0469a179-4228-4d30-a263-8f7e17da7026>
- [19] Donald E. Knuth. 1968. Semantics of context-free languages. *Mathematical Systems Theory* 2, 2 (1968), 127–145.
- [20] Edlira Kuci, Sebastian Erdweg, Oliver Bracevac, Andi Bejleri, and Mira Mezini. 2017. A Co-contextual Type Checker for Featherweight Java. In *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19–23, 2017, Barcelona, Spain (LIPIcs, Vol. 74)*, Peter Müller (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 18:1–18:26. <https://doi.org/10.4230/LIPIcs.ECOOP.2017.18>
- [21] LAMP/EPFL. Accessed: 2023-05-28. Scala 3 High Level Architecture: Symbols. <https://dotty.epfl.ch/docs/contributing/architecture/symbols.html>
- [22] LAMP/EPFL. Accessed: 2023-05-28. Scala 3 Metaprogramming: Runtime Multi-Stage Programming. <https://dotty.epfl.ch/docs/reference/metaprogramming/staging.html>
- [23] Eva Magnusson, Torbjörn Ekman, and Görel Hedin. 2009. Demand-driven evaluation of collection attributes. *Autom. Softw. Eng.* 16, 2 (2009), 291–322. <https://doi.org/10.1007/s10515-009-0046-z>
- [24] Conor McBride and Ross Paterson. 2008. Applicative programming with effects. *J. Funct. Program.* 18, 1 (2008), 1–13. <https://doi.org/10.1017/S0956796807006326>
- [25] Radu Mihăilescu. 2023. *Building Type Checkers Using Scope Graphs: Scope Graph-Based Type Checking for a Scala Subset*. Bachelor's Thesis. Delft University of Technology. <http://resolver.tudelft.nl/uuid:a4050c81-4a1c-4cf0-b26e-7c60aa18503a>
- [26] Robin Milner. 1978. A Theory of Type Polymorphism in Programming. *J. Comput. Syst. Sci.* 17, 3 (1978), 348–375. [https://doi.org/10.1016/0022-0000\(78\)90014-4](https://doi.org/10.1016/0022-0000(78)90014-4)
- [27] Andreea Mocanu. 2023. *Building Type Checker Using Scope Graphs: For a Language with Type Classes*. Bachelor's Thesis. Delft University of Technology. <http://resolver.tudelft.nl/uuid:64aa6cc8-9039-47b1-99f6-decf150dcab8>
- [28] Pierre Neron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. 2015. A Theory of Name Resolution. In *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11–18, 2015. Proceedings (Lecture Notes in Computer Science, Vol. 9032)*, Jan Vitek (Ed.). Springer, 205–231. https://doi.org/10.1007/978-3-662-46669-8_9
- [29] Johan Östlund and Tobias Wrigstad. 2010. Welterweight Java. In *Objects, Models, Components, Patterns, 48th International Conference, TOOLS 2010, Málaga, Spain, June 28 - July 2, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6141)*, Jan Vitek (Ed.). Springer, 97–116. https://doi.org/10.1007/978-3-642-13953-6_6
- [30] Gordon D. Plotkin and Matija Pretnar. 2009. Handlers of Algebraic Effects. In *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22–29, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5502)*, Giuseppe Castagna (Ed.). Springer, 80–94. https://doi.org/10.1007/978-3-642-00590-9_7
- [31] Casper Bach Poulsen and Cas van der Rest. 2023. Hefty Algebras: Modular Elaboration of Higher-Order Algebraic Effects. *Proc. ACM Program. Lang.* 7, POPL (2023), 1801–1831. <https://doi.org/10.1145/3571255>
- [32] Casper Bach Poulsen, Aron Zwaan, and Paul Hübner. 2023. *Mophasco (MONadic framework for PHAsed name resolution using SCOpe graphs)*. <https://doi.org/10.5281/zenodo.8337245>
- [33] Arjen Rouvoet, Hendrik van Antwerpen, Casper Bach Poulsen, Robert Krebbers, and Eelco Visser. 2020. Knowing when to ask: sound

- scheduling of name resolution in type checkers derived from declarative specifications. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 180:1–180:28. <https://doi.org/10.1145/3428248>
- [34] Arjen Rouvoet, Hendrik van Antwerpen, Casper Bach Poulsen, Robert Krebbers, and Eelco Visser. 2020. Knowing When to Ask: Sound scheduling of name resolution in type checkers derived from declarative specifications (Extended Version). Zenodo. <https://doi.org/10.5281/zenodo.4091445>
- [35] Walid Taha and Tim Sheard. 2000. MetaML and multi-stage programming with explicit annotations. *Theor. Comput. Sci.* 248, 1-2 (2000), 211–242. [https://doi.org/10.1016/S0304-3975\(00\)00053-0](https://doi.org/10.1016/S0304-3975(00)00053-0)
- [36] Omar Thabet. 2023. *Phased Type Checker For Java: A Type Checker For a Subset of Java Built On Scope Graph Semantics*. Bachelor's Thesis. Delft University of Technology. <http://resolver.tudelft.nl/uuid:f05ab738-0fde-48bc-b3b3-4a8a1345db4c>
- [37] Hendrik van Antwerpen, Pierre Neron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. 2016. A constraint language for static semantic analysis based on scope graphs. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Martin Erwig and Tiark Ropf (Eds.). ACM, 49–60. <https://doi.org/10.1145/2847538.2847543>
- [38] Hendrik van Antwerpen, Casper Bach Poulsen, Arjen Rouvoet, and Eelco Visser. 2018. Scopes as types. *Proc. ACM Program. Lang.* 2, OOPSLA (2018), 114:1–114:30. <https://doi.org/10.1145/3276484>
- [39] Aron Zwaan and Hendrik van Antwerpen. 2023. Scope Graphs: The Story so Far. In *Eelco Visser Commemorative Symposium, EVCS 2023, April 5, 2023, Delft, The Netherlands (OASlcs, Vol. 109)*, Ralf Lämmel, Peter D. Mosses, and Friedrich Steimann (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 32:1–32:13. <https://doi.org/10.4230/OASlcs.EVCS.2023.32>

Received 2023-07-14; accepted 2023-09-03

A pred-LL(*) Parsable Typed Higher-Order Macro System for Architecture Description Languages

Christoph Hochrainer

TU Wien

Vienna, Austria

christoph.hochrainer@tuwien.ac.at

Andreas Krall

TU Wien

Vienna, Austria

andi@complang.tuwien.ac.at

Abstract

Macro systems are powerful language extension tools for Architecture Description Languages (ADLs). Their generative power in combination with the simplicity of specification languages allows for a substantial reduction of repetitive specification sections. This paper explores how the introduction of function- and record types in a template-based macro system impacts the specification of ADLs. We present design and implementation of a pattern-based syntax macro system for the Vienna Architecture Description Language (VADL). The macro system is directly integrated into the language and is analyzed at parse time using a context-sensitive pred-LL(*) parser. The usefulness of the macro system is illustrated by some typical macro application design patterns. The effectiveness is shown by a detailed evaluation of the Instruction Set Architecture (ISA) specification of five different processor architectures. The observed specification reduction can be up to 90 times, leading to improved maintainability, readability and runtime performance of the specifications.

CCS Concepts: • **Software and its engineering** → *Domain specific languages*; **Macro languages**; • **Hardware** → *Hardware description languages and compilation*.

Keywords: macro system, higher-order macros, pattern-based, generative programming, architecture description language

ACM Reference Format:

Christoph Hochrainer and Andreas Krall. 2023. A pred-LL(*) Parsable Typed Higher-Order Macro System for Architecture Description Languages. In *Proceedings of the 22nd ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '23)*, October 22–23, 2023, Cascais, Portugal. ACM, New York, NY, USA, 13 pages. <https://doi.org/10.1145/3624007.3624052>



This work is licensed under a Creative Commons Attribution 4.0 International License.

GPCE '23, October 22–23, 2023, Cascais, Portugal

© 2023 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-0406-2/23/10.

<https://doi.org/10.1145/3624007.3624052>

1 Introduction

Macros and Domain-Specific Languages (DSLs) are two programming concepts that contribute to faster development of artifacts and code quality. Macros are used to simplify repetitive code patterns by providing a shorter, more concise expression. Domain-Specific Languages offer a higher level of abstraction than conventional General-Purpose Languages (GPLs) for a specific domain. DSLs allow for a wide variety of applications, implementation techniques and design choices [18]. Macros and DSLs are strongly coupled. Many DSLs are implemented as a collection of macro definitions, while on the other hand, macros can contribute to the language extensibility for existing DSLs. A special form of DSLs are Architecture Description Languages (ADLs). In this article we present our experience with the development of a macro system with special focus on ADLs. Through the development of our Vienna Architecture Description Language (VADL), see Section 2, we gathered valuable insights regarding language extensibility for ADLs.

1.1 Architecture Description Languages

ADLs are computer languages used to describe the architecture of hardware and software systems. Particularly interesting for this article is the ADL subgroup of Processor Description Languages (PDLs). PDLs allow hardware designers to describe instruction set, register set, memory hierarchy, and other aspects of a microprocessor. We identified a particular need for macros regarding PDLs, especially when it comes to the specification of an instruction set architecture (ISA). Section 2.2 will provide an overview of the fragment of VADL used to describe ISAs and explain in more depth where the repetitiveness comes from and how it influenced our macro system design. Of course these observations are not limited to us and can also be found in other description languages like LISA [21], ISDL [10] or ArchC [2]. Additionally, we want to clarify some key properties of VADL and PDLs in general. A PDL is not, and should not be, an executable program. It can be thought of as a complex configuration for artifacts like hardware, simulator or compiler. This is an important concept as an error is no longer a programming error, which can be debugged with the specification alone. Debugging a specification requires specially generated tools and techniques like co-simulation. Hence, it is most important to reduce any other sources of errors, e.g. semantic errors, to

a minimum. In Section 1.2, we describe how specific macro designs contribute to this desired property.

1.2 Macro Systems

Macro systems are one of the oldest forms of language extensions. In general, macros are user-defined procedures, transforming one program sequence to another program sequence. This transformation is called *macro expansion*. Based on the technique used the macro system is categorized into a *lexical* or *syntactical*, and *procedural* or *pattern-based* macro system [16]. Lexical macro systems, such as the C preprocessor (CPP) [23] and Unix M4 [12], are language agnostic and work on a lexical level, for example a token stream. In contrast to lexical macro systems, syntax macros are aware of syntactic structures. They are integrated into the core language and usually perform AST (Abstract Syntax Tree) to AST transformations. Representatives for example are LISP [24], Scheme [1] or Racket [9]. If the macro system supports algorithmic computations on their inputs, they are classified as *procedural* macro systems. On the other hand, macro systems that rely on pattern matching and substitution are called *pattern-based*. The presented concepts are not mutually exclusive and may be present in all combinations. The Rust programming language [13] incorporate both, *procedural* and *pattern-based* techniques within its macro system. Another interesting example is the Java Syntactic Extender (JSE) [3], which supports full procedural macros and an extendable pattern-matching engine. Finally, a concept often considered when talking about macro system is *hygienic macros* [4, 8, 14]. The main idea of hygienic macros is to prevent accidental capture of identifiers during expansion. When we started the design of our macro system, we anticipated, that macro hygiene was of secondary importance for us as we either want to capture identifiers or we pass the identifiers as arguments providing us with more control over the used names. We will address hygienic macros again in Section 3.6 together with our *lexical macros*. For now, hygienic macros are part of our future work.

1.3 Macros for DSLs

When we were considering language extensibility for our DSL, *syntax type safety* and *termination* were the top priorities. We use the term *syntax type safety* in the sense that a *syntax type safe* macro system is able to detect *syntax type errors*. Hence, the system prevents the generation of syntactically incorrect code. Furthermore, many macro systems designed for DSLs have a feature-rich host-language or environment they can exploit [5].

VADL on the other hand is a standalone DSL/ADL with no meta- or host-language available. This decision helps us to develop the VADL syntax more freely and explore different design possibilities for PDLs without syntactical restrictions or superfluous features of a host language. The drive of keeping the specification simple led us to investigate

a lightweight and language dependent implementation, i.e. a *syntactical pattern-based* macro system. We also considered a language agnostic approach, but decided against it due to the lack of safety, available debug information and IDE support.

While we were satisfied with the choice of syntactical type safety, the pattern-based templates felt very limiting in expressiveness. Switching to procedural macros is for us (and we believe also for many other DSLs with a non Turing-complete specification language as host language) not beneficial as it compromises the simplicity of the host language. This inspired us to develop the higher-order *models* for our macro system discussed in Section 3.

A final aspect worth considering is computation time for DSL macro systems. Macro expansion becomes a prerequisite for any DSL related analysis and task. Therefore, a main goal should be to make sure that the macro system's execution time is as short as possible. We incorporated our macro system directly into the language grammar without requiring any preprocessor. This helped us to reduce unnecessary pre-computations. Additionally, our LL(k) parsable host language encouraged us to preserve the top-down parsing fashion. We designed the built-in macro system to be pred-LL(*) parsable.

Contribution.

- A simple pred-LL(*) parsable *syntactical pattern-based* macro system for specification languages
- Syntax type safe higher-order macro templates using *models*
- Composable syntax types using *records* and type aliases
- Demonstration of the presented macro system using the Vienna Architecture Description Language

Additionally, we present a variety of smaller macro features supporting a high configurability and usability in the context of specification languages. We found the following implemented features particularly useful for our exploratory language design of VADL.

- Inheritance of macro definitions across language definitions
- Lexical manipulation of identifiers and strings
- Configurable and conditional macro expansions using *match* and command line arguments

2 Overview

In this section we give an overview of the Vienna Architecture Description Language (VADL) with special focus on the instruction set architecture (ISA) section.

2.1 Vienna Architecture Description Language

VADL is a Domain-Specific Language in the domain of computer architecture and compiler construction. It permits the complete formal specification of a processor architecture.

Additionally, it is possible to specify the behavior of generators which produce different artifacts from a processor specification like a compiler or an instruction set simulator. VADL strictly separates the specification of the instruction set architecture (ISA), the micro architecture (MiA) and the application binary interface (ABI). To provide a proof of concept, we only implemented and evaluated our macro language for the instruction set architecture specification section. However, the ideas and techniques presented can be applied to the other sections as well as to any similarly structured DSL.

2.2 ISA Syntax Elements

Presenting the whole syntax and semantics of VADL used to describe ISAs, let alone the VADL language as a whole, is out of the scope of this article. Therefore, we will focus only on the relevant portion of the instruction set architecture definition. First, we have to establish how instructions are defined. VADL separates the abstract concept of an instruction into three parts. The instruction definition, the instruction encoding and the textual representation, i.e. assembly. The instruction definition is the core part of the three definitions, holding information on the name, the used encoding format and the instruction semantics. The instruction encoding specifies the values of the static encoded fields. The instruction assembly definition specifies a pattern on how the assembly string is computed. Figure 1 shows a specification of an *ADD* instruction, which adds two registers together and stores the result in a third. The format fields of the format definition *F*, which are not assigned to a static value inside the encoding definition, become dynamic fields or operands. Inside the instruction semantics, we can observe that *rd*, *rs1* and *rs2* are indeed used as operands. The call expressions to *X(.)* represent indexing of a register bank *X*, defined somewhere else in the ISA.

If we define a new instruction, e.g. *AND*, that differs from *ADD* in a single encoding bit and the binary operator, we would need to create a completely new instruction definition, encoding and assembly. Which brings us to the downside of such element or block based specification languages like VADL. We designed VADL to be descriptive and simple, which led us to a very small core language for the ISA section. While we support functions, our core type system is very simple and does not support these definitions as first class citizens. During our language development phase we also experimented with different language built-in features that could reduce code duplication, but we came to the conclusion that they only introduce a lot of complexity and obfuscate the original code. This led us to the idea of designing a template-based macro system specifically directed towards specification languages.

```

1 format F =
2   { funct7 : Bits<7>
3     , rs2   : Bits<5>
4     , rs1   : Bits<5>
5     , funct3 : Bits<3>
6     , rd    : Bits<5>
7     , opcode : Bits<7>
8   }
9
10 instruction ADD : F = {
11   X(rd) := X(rs1) + X(rs2)
12 }
13
14 encoding ADD =
15   { opcode = 0b011'0011
16     , funct3 = 0b000
17     , funct7 = 0b000'0000
18   }
19
20 assembly ADD =
21   ( "ADD"
22     , register( rd ), " + ",
23     , register( rs1 ), " + ",
24     , register( rs2 )
25   )

```

Figure 1. ISA Example Specification for an ADD instruction

3 VADL's Macro System

In this section we give a detailed description about the syntax and techniques implemented for VADL's macro system.

3.1 Syntax Models

At the core of our macro system are the so-called *syntax models*. A syntax model can be seen as a parameterized and well typed template. Figure 2 shows how such a syntax model can be defined.

```

1 model InstModel (op: BinOp, name : Id)
2   : IsaDefs = {
3     instruction $name : F = {
4       X(rd) := X(rs1) $op X(rs2)
5     }
6   }

```

Figure 2. Syntax Model Definition

Every model has a name, a typed parameter list, a result type and a body. Note how the use of the parameters are indicated by a leading "\$". This design decision has two advantages. First, it simplifies parsing as it explicitly marks the use of a macro element. Second, the "\$" captures the model parameter names, preventing name collisions with

ISA definitions, which strengthens the hygiene of the macros. Similarly to the parameters, we use the "\$" for the instantiation of defined syntax models. Figure 3 shows how the model from Figure 2 can be instantiated. To separate the syntax elements from each other we use ";" as separator inside an instantiation. Recall the dilemma of Section 2.2, the introduction of *models* provides us now with a mechanism to efficiently specify both instructions without code repetition.

```
1 $InstModel( ADD ; + )
2 $InstModel( AND ; & )
```

Figure 3. Syntax Model Instantiation

In the example of Figure 2 we only used the identifier (*Id*), binary operator (*BinOp*) and ISA element (*IsaDefs*) syntax types. However, the syntax model definition supports a variety of types discussed in the following sections.

3.2 Syntax Types

This section introduces all the available core syntax types. We designed our syntax types to have a one-to-one relation to parser rules. This already provides us with a partial order, where the relation is a partially ordered subtype relation. Table 1 gives an overview of all the available base types with a short description and examples. Additionally, it is important to note that the presented base types, function types (Section 3.4), record types and type aliases (Section 3.3) can be arbitrarily nested. The resulting types can be used everywhere a *syntax type* is expected with the only exception being result types of *models* and function types. Figure 4 displays the subtype relation between the presented core types. The macro type system provides an implicit up-casting of the value types. For example, if a model expects a value of type *Val*, any subtype, i.e. *Bool*, *Int* or *Bin* will be accepted as argument.

Table 1. Core Syntax Types

Type	Description	Examples
Ex	Generic VADL Expression	$X(rs1) + X(rs2)$
Lit	Generic VADL Literal	1, "ADD"
Val	Generic VADL Value Literal	1, 0b001
Bool	Boolean Literal	true, false
Int	Integer Literal	1, 2, 3
Bin	Binary or Hexadecimal Literal	0b1, 0xff
Str	String Literal	"ADD"
CallEx	Arbitrary Call Expression	MEM<2>(rs1)
SymEx	Symbol Call Expression	rs1, MEM<2>
Id	Identifier Symbol	rs1, ADD, X
BinOp	Binary Operator	+, -, *
UnOp	Unary Operator	-
Stat	Generic VADL Statement	$X(rd) := X(rs)$
Stats	List of VADL Statements	$X(rd) := X(rs) \dots$
IsaDefs	List of VADL ISA Definition	instruction ADD : F = { ... } ...
Encs	Element(s) of an Encoding Definition	func = 0b000, ...

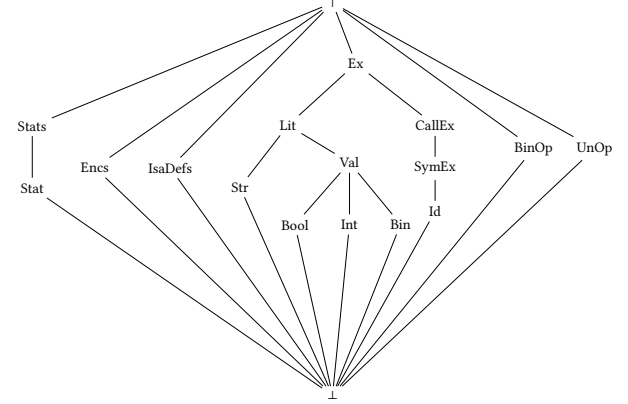


Figure 4. Syntax Type Relation

3.3 Type Alias and Composition

The VADL macro system provides a feature rich type interface. Besides the basic types mentioned in Section 3.2, the macro system also supports type aliasing and a form of type composition to make the typed templates more readable. Figure 5 shows a type alias definition *BinExprType*, which from now on can be used instead of the function type $(Ex, Ex) \rightarrow Ex$. An application of *BinExprType* can be seen in Figure 8.

Figure 6 shows a *record* definition used for type compositions. In this particular case the record definition composes an *Id* and *BinOp* type to the new type *BinInstRec*. The body of a record consists of a parameter list providing typed fields. Figure 7 shows how the record is initialized and the fields *name* and *op* are accessed. Passing a record type argument can be either done by reference or by creating a syntax tuple. A syntax tuple is specified the same way a model argument list is provided, i.e. syntax elements are separated by ";" and enclosed inside brackets. Accessing the passed elements is done using the record's name followed by a "." and the desired field. Accesses of sub-records can be arbitrary chained together. The whole access may be wrapped inside brackets, i.e. "\$(...)", to better indicate what is part of the access and what belongs to the VADL specification. Furthermore, it is important to note that records are treated as type tuples. Their field names do not affect the type and are only used to access the internal elements.

```
1 model-type BinExprType = ( Ex, Ex ) -> Ex
```

Figure 5. Syntax Type Alias Example

```
1 record BinInstRec ( name: Id, op: BinOp )
```

Figure 6. Record Example

```
1 model InstModel (info: BinInstRec)
2   : IsaDefs = {
3     instruction $info.name : F = {
4       X(rd) := X(rs1) $info.op X(rs2)
5     }
6   }
7
8 $InstModel( ( SUB ; - ) )
9 $InstModel( ( ADD ; + ) )
```

Figure 7. Record Application

3.4 Higher-Order Macros

To the best of our knowledge, we have not seen typed higher-order macros in a *pattern-based syntax* macro system as presented in this paper. In this section we will shortly describe how they are used in the context of our macro system. In Section 3.8, we will further discuss why they are important for ADLs and how we use them in VADL. A higher-order macro is a statically evaluated function, mapping a list of syntax types to a result syntax type. We chose the term higher-order, to underline the capability of providing model references as argument. In Figure 5 we have already defined the type signature of a model in form of a function type. We will reuse this type for the higher-order model *BinExStat* in Figure 8. The instantiation of *BinExStat* in the presented figure, produces an assignment statement of *X(rd)* taking the addition of *X(rs1)* and *X(rs2)* as argument.

A valid argument for a parameter with a function type is either a model reference, as seen in the example, or a parameter of function type from an outer model. In both cases the types are evaluated and checked during parse time.

3.5 Conditional Expansion

A minor difference to some pattern-based approaches is our conditional expansion. VADL macros provide an explicitly typed *match*-statement shown in Figure 9. The entries are processed from top-to-bottom and it uses the right-hand-side of the first satisfied left-hand-side for expansion. The *match*-statement has the requirement of providing a default case at the last position, indicated by the *_*. Beside the default case, each entry contains a condition that is either matching equality (*=*) or inequality (*!=*) of a parameter and a syntax element matching the type of the parameter. The comparison is done on a lexical, i.e. token-based, level and performed

```
1 model BinExStat
2   // ( Ex, Ex ) -> Ex
3   ( binEx : BinExType ) : Stat = {
4     X(rd) := $binEx( X(rs1), X(rs2) )
5   }
6
7 model AddExp ( rhs: Ex, lhs : Ex ) : Ex = {
8   $rhs + $lhs
9 }
10
11 $BinExStat( AddExp )
```

Figure 8. Higher-Order Macro Example

on the expanded representation of the parameter. A *match*-statement is only allowed inside a model. Therefore, it is only evaluated if the parent model is instantiated, which is why the argument-parameter pairs are always available for the *match*-statement expansion. Figure 9 shows how it could be used in configuration management. More on configuration can be found in Section 3.7.

```
1 // $BitSize( Arch32 ) -> 32
2 // $BitSize( Arch64 ) -> 64
3 model BitSize ( arch : Id ) : Int = {
4   match : Int
5     ( $arch = Arch32 => 32
6       ; $arch = Arch64 => 64
7       ; _ => 0
8     )
9 }
```

Figure 9. Match Statement Example

3.6 Lexical Macro Functions

While most of the needs are covered by *syntactical* macros, we came to the conclusion that string and identifier manipulation is best done using *lexical macros*. A lexical macro acts on the abstraction level of token streams in contrast to an already parsed AST. Through language exploration, we narrowed the *lexical* macros down to two use-cases. These use-cases are safely implemented using special macro functions. Firstly, templates generating instruction behavior and assembly used to require the instruction name once in form of an identifier (*Id*) and again in form of a string (*Str*). We solved this issue by introducing the *IdToStr* function. This function takes an *Id* typed syntax element and converts it to a *Str* typed syntax element. Secondly, we encountered the problem of not being able to efficiently manipulate our identifiers. This is especially tedious when dealing with different

configurations. To provide a type safe identifier manipulation, we introduced the *ExtendId* function. This function takes an *Id* typed identifier and an arbitrary number of *Str* typed syntax elements, concatenates them together and returns a single *Id* typed syntax element. Figure 10 shows a small example of both functions with their typed result as comment. It is important to note that the context of the *lexical macros* generated identifiers is strictly separated from the context of the *syntactical macros*. Therefore, it is not possible to define or refer to a model name or parameter using a generated identifier.

For VADL, the lexical macro functions are the "alternative" to *hygienic macros*. Consider the ISA elements described in Section 2.2. When defining a new instruction we either want to capture identifiers, e.g. registers or memories, or we want full control over the identifier name. From our experience, VADL's *IdToStr* and *ExtendId* are sufficient for this use-case.

```
1 ExtendId( I, "Am", "An", "Identifier" )
2 // --> IAmAnIdentifier : Id
3 IdToStr( IAmAString )
4 // --> "IAmAString" : Str
```

Figure 10. Lexical Macros Example

3.7 Configuration and Inheritance

VADL provides the possibility of passing configuring information to the macro system using the command line. Currently, this mechanism is kept very simple and is based on *Id* elements. To prepare a configurable macro variable, one has to create a simple model of type *Id* containing a default value. Figure 11 shows such a variable of name *Arch*, with the default setting *Aarch32*. Without any passed configurations the instantiation of *Arch* results in the identifier *Aarch32*. However, if VADL receives the command-line option *-m* or *-model*, followed by the string "*Arch=Aarch64*" the value of *Arch* is overridden. If *Arch* is instantiated given the previous command-line option, it would result in *Aarch64*. In combination with conditional expansion, see Section 3.5 and Figure 9, this simple mechanism already provides powerful configuration capabilities.

```
1 model Arch() : Id = { Aarch32 }
```

Figure 11. Macro Configuration Variable

Additionally, we want to shortly mention VADL's inheritance in this section. Every ISA component in VADL can

inherit from an arbitrary other ISA component using the keyword *extending*. Since we tightly coupled the macro system into our language, the inheritance and visibility does also affect the macro system. Figure 12 shows this mechanism in action and provides additional information in the comments.

```
1 instruction set architecture A = {
2   model ModelA() : IsaDefs = // ...
3 }
4
5 instruction set architecture B
6   extending A = {
7     $ModelA() // OK, defined in ISA A
8   }
9
10 instruction set architecture C
11   extending B = {
12     $ModelA() // OK, defined in ISA A
13   }
14
15 instruction set architecture D = {
16   $ModelA() // ERROR, not defined
17 }
```

Figure 12. ISA Inheritance Example

3.8 Macro Application in Processor Specifications

The following macro application design patterns demonstrate with simplified examples the usage of macros for instruction set architecture specifications. The simplified examples are based on a real specification of the AAarch32 instruction set architecture from ARM. The most common case is a simple argument substitution pattern shown in Figure 13.

AArch32 has a register file called R consisting of 16 registers which are 32 bits wide. Conditions are specified by boolean expressions on flags of the status register APSR, e.g. the zero flag Z. Every instruction can be executed conditionally. There are 15 different conditions which are described by an enumeration in the specification and encoded by the cc field in an instruction word which is 32 bits wide. Arithmetic/logic instructions which have an immediate value as second source operand share a common instruction encoding specified in the ArLoImm instruction format. The ALImmInst instructions themselves differentiate each other only by the unique instruction identifier, the assembly instruction name, the binary operation to be executed and the instruction encoding. Therefore, a model with these four parameters is defined which substitutes these four parts in the instruction specification. Then with a single line macro call an arithmetic/logic immediate instruction can be specified. This leads to concise instruction set architecture specifications,


```

1 register file R: Bits <4> -> Bits <32>
2
3 enumeration cond: Bits <4> =
4 { EQ // equal           Z == 1
5   , NE // not equal      Z == 0
6   // ...
7   , AL // always
8 }
9
10 // arithmetic/logic immediate format
11 format ArLolmm: Bits <32> =
12 { cc [31..28] // condition
13   , op [27..21] // opcode
14   , flags [20] // set status register
15   , rn [19..16] // source register
16   , rd [15..12] // destination register
17   , imm12 [11..0] // 12 bit immediate
18 }
19
20 model ALLmmInstr (id:Id, ass:Str, op:BinOp,
21                  opcode:Bin) : IsaDefs = {
22   instruction $id : ArLolmm = {
23     R(rd) := R(rn) $op imm12
24   }
25   encoding $id =
26     {cc = cond::AL, op = $opcode, flags = 0}
27   assembly $id = ($ass, ' ', register(rd),
28     ', ', register(rn), ' ', decimal(imm12))
29 }
30
31 $ALLmmInstr ( ADD ; "add" ; + ; 0b000'0100 )
32 $ALLmmInstr ( SUB ; "sub" ; - ; 0b000'0010 )
33 $ALLmmInstr ( AND ; "and" ; & ; 0b000'0000 )
34 $ALLmmInstr ( ORR ; "orr" ; | ; 0b000'1100 )

```

Figure 13. Instruction Specification applying simple Macros

improves the maintainability and reduces the probability of errors.

Most instruction set architectures are too complex to get by with the substitution pattern. As in the AArch32 architecture every instruction can be executed conditionally, a basic instruction exists in 15 variants for 15 different conditions. This problem can be solved smartly by an extension macro pattern using higher-order macros as demonstrated in Figure 14.

To reduce the number of macro arguments record types are defined for an instruction and a condition. The Inst record type definition groups the four arguments describing an instruction from Figure 13 together. The Cond record type definition consists of a string representing the extension of the assembly name, the identifier of the enumeration of the condition encoding and a boolean expression for condition evaluation.

In contrast to the previous example in Figure 14 now 15 different instructions with a unique identifier have to be

created. This can be handled with the lexical macro function `ExtendId` by appending the extension string of the condition to the identifier.

The final problem is that there is a set of models which describe different kinds of conditional instructions and all these models should be called 15 times for the 15 different conditions. This can be solved by the higher-order model `CondInstr`, which takes the instruction model as first argument. The instruction model is then called 15 times with an argument list, which has been extended by the conditions. In the above example the 4 macro calls expand to 60 different instructions. The AArch32 architecture has instructions with a lot of additional variants like setting the status register, shifted operands or complex addressing modes. This leads to a specification with multiple higher-order macro arguments.

4 Implementation

In this section we give an overview of our macro system implementation for VADL. VADL manages the macros in two separated phases: parsing and expansion. It is important to note that the parsing phase does only analyze the macros. It guides the parser through the different kind of macro actions while asserting their syntactical and partly semantical correctness. Applications of the macro actions are done in the expansion phase.

4.1 Parsing

Parser. The VADL frontend uses a modified version of the Xtext framework [6]. The Xtext framework is a Java based DSL development tool. It takes a Xtext grammar file as input and generates a variety of useful artifacts, e.g. IDE integration, metamodel classes for the syntax-tree or a parser. We have refrained from using any non LL(k) Xtext grammar functionalities and disabled the backtracking feature of the generated parsers to start our implementation from a true LL(k) parser. A LL(k) parser is a top-down parser processing the language from left to right. The k indicates a constant lookahead, which may be performed by the parser. Additionally, we extended the implementation to allow semantic predicates [20] and code actions. This grammar extension lifts the parser to `pred-LL(*)`. The *pred* prefix indicates the use of predicates in combination with grammar rules and the star (*) lifts the lookahead requirement of a fixed constant k to an arbitrary constant. The Xtext framework targets ANTLR [19], which already supports code actions and semantic predicates. Therefore, extending the grammar was a quite straightforward task for our simple purposes. In Sections 4.1 and 5.3 we will further comment on the context-sensitivity.

Concrete Syntax Tree. The Xtext framework automatically creates classes for each non-terminal grammar rule that has at least one labeled rule field. While parsing a source file, it uses these classes to build a concrete syntax tree (CST). To keep the implementation effort manageable we kept this


```

1 record Instr (id: Id, ass: Str, op: BinOp, opcode: Bin)
2 record Cond (str: Str, code: Id, ex: Ex)
3
4 model ALImmCondInstr (cond: Cond, instr: Instr) : IsaDefs = {
5   instruction ExtendId ($instr.id, $cond.str) : ArLolImm = {
6     if ($cond.ex) then
7       R(rd) := R(rn) $instr.op imm12
8     }
9   encoding ExtendId ($instr.id, $cond.str) =
10     {cc = cond::$cond.code, op = $instr.opcode, flags = 0}
11   assembly ExtendId ($instr.id, $cond.str) =
12     ($instr.ass, $cond.str, ' ', register(rd), ',', register(rn), ',', decimal(imm12))
13   }
14
15 model-type CondInstrModel = (Cond, Instr) -> IsaDefs
16
17 model CondInstr (modelid: CondInstrModel, instr: Instr) : IsaDefs = {
18   $modelid (( "eq" ; EQ ; APSR.Z = 0b1 ) ; $instr)
19   $modelid (( "ne" ; NE ; APSR.Z = 0b0 ) ; $instr)
20   // ...
21 }
22
23 $CondInstr(ALImmCondInstr ; ( ADD ; "add" ; + ; 0b000'0100 ))
24 $CondInstr(ALImmCondInstr ; ( SUB ; "sub" ; - ; 0b000'0010 ))
25 $CondInstr(ALImmCondInstr ; ( AND ; "and" ; & ; 0b000'0000 ))
26 $CondInstr(ALImmCondInstr ; ( ORR ; "orr" ; | ; 0b000'1100 ))

```

Figure 14. Instruction Specification applying Higher Order Macros

default behavior. VADL does not have a separate preprocessor, therefore performs the macro expansion directly on the generated CST.

Wrapper Rules. Since the CST consists of Java classes based on grammar rules, we decided to introduce additional wrapper rules. Each rule, which we would like to use as macro type, is enclosed in an additional rule to create a clear location for replacement later on. In most cases, such a wrapper rule contains two alternatives: a generic macro replacement rule guarded by a semantic predicate and the concrete value rule. The semantic predicate is used to perform a minor lookahead to see if the next tokens are part of a concrete value or a macro action. Unfortunately, this introduces a slight overhead as we have to create an additional wrapper rule for each syntax type. Figure 15 shows a wrapper rule *StringRule* and a concrete rule *ConcreteStringRule* to express strings. Retrieving the current context with reflections or the parser itself was quite tedious, so we decided to simply parameterize our *isMacroAction* predicate with the current syntax type context (*Str*). The *StringRule* rule can now be used anywhere as if it was a normal grammar rule for strings. Inside the *StringRule* the *value* field is used as location for replacement. We implemented all non-terminal rules used as syntax types (see Section 3.2) in the same fashion.

```

1 StringRule :
2   $$ isMacroAction( Str ) $$?=> // sem-pred
3   value=MacroActionRule
4   | value=ConcreteStringRule
5   ;
6
7 ConcreteStringRule: // ...

```

Figure 15. Wrapper Rule Example

Context Sensitivity. By using a context-sensitive parse approach, we guide the parser in such a way that only syntactically and semantically correct macro occurrences are parsable. To manage the context sensitivity, we implemented a parser state specifically for macros. It provides an API used by semantic predicates and code actions to compare and update symbols and macro information. The core of the parse state itself consists of a symbol table containing information on macro related definitions in the current scope. Moreover, it holds a variety of type information on actively parsed macro constructs. In the initial parse state only the core syntax types listed in section 3.2 are registered. During parsing, ISA namespaces, models, their parameters, records and syntax type aliases are added. Figure 16 shows a simplified

version of our grammar rules handling a model definition. The start of the *ModelRule* is straight forward by expecting the model keyword, a name, a typed parameter list and a syntax type. Before the parser enters the model body, the parse state has to be updated. In the example this is done by the code actions, indicated with an opening and closing "\$\$". We feed the parse state the name, the syntax type and the parameters. Note that the passed values are CST nodes and therefore already Java classes, making it possible to access type and name of the parameters. Inside the parse state, the symbol table is extended by the model name and the information on the parameters. The passed type is used to select the correct rule to parse the model body. This is done using syntax predicates, which can be seen in Figure 16 inside the *ModelBodyRule*. They are similar to code actions but with an additional "? =>" after the enclosing "\$\$". The predicates are tested in-order from top to bottom. If a predicate is satisfied, the parser tries to apply the rule(s) on the right-hand side of the current alternative. The *ModelBodyRule* reveals another slight overhead as we have to manually implement the relation between syntax type and desired rule. The presented example was of course just a simplified version of the actual implementation and should help to understand the main idea. A similar approach was applied for all the other context-sensitive tasks, e.g. managing ISA namespaces or checking the correctness of syntax types of passed arguments.

```

1  ModelRule :
2      "model" name=IdentifierRule
3      "(" parameters+=ModelParameterRule* ")"
4      ":" type=SyntaxTypeRule "="
5      "{"
6      $$ state.enterModel
7          ( name, type, parameters ); $$
8      body=ModelBodyRule
9      $$ state.leaveModel
10     ( name, type, parameters ); $$
11     "}"
12 ;
13
14 ModelBodyRule :
15     $$ state.isModelBodyOfType( String ) $$
16     ?=> StringRule
17     $$ state.isModelBodyOfType( Integer ) $$
18     ?=> IntegerRule
19     // ...
20 ;
21
22 ModelParameterRule :
23     name=IdentifierRule
24     ":" type=SyntaxTypeRule
25 ;

```

Figure 16. Model Grammar Rules

4.2 Expansion

The macro expansion is done in a separate pass after parsing and works solely on the CST representation. This pass is responsible to perform all macro related CST manipulations. Note that at this stage the correctness of the macros were already checked by the parser. The examples in Figure 17 and Figure 18 show a textual representation of the CST before and after the expansion pass. The expansion is executed in two steps.

Setup. The first step is to remove all the macro nodes of the CST that do not produce any new nodes. This includes model, syntax type alias and record definitions. During the removal, each definition is stored in a symbol table to preserve its information. Additional to the parsed definitions, the command line configurations are added to the symbol table.

Execution. The second step is the actual execution and expansion of macro code. The expansion is done in a top-down fashion and performs iterative replacements on wrapper nodes. The first top level construct, or root node, the macro expander encounters, is by design a model instantiation node. The instantiation contains information on the model and the passed parameters. Similar to function inlining, the expander creates a copy of the referenced model body and replaces each parameter occurrence with the respective argument. The new body is now passed again to the expander to resolve nested macro actions. Afterwards, the macro node is replaced by the newly created subtree of the updated copy of the model body. Conditional macro expansions are also done during the model instantiation. The implementation is very simple and currently only uses a unified symbolic equality comparison. The *match* statement is symbolically executed and replaced by the right-hand side of the first satisfied left hand side condition. If no condition evaluates to true, the mandatory last wild card statement is used. Similar to the model instantiation, the resulting node is again iteratively expanded. The implementation of the lexical macro actions (*IdToStr*, *ExtendId*) are also straight forward. We simply create a new identifier or string CST node and replace the old macro node occurrence with the newly created node. Recall that the newly created identifier is only part of the expanded program and not available during expansion. This means that it cannot be used to refer to macro models or macro parameters.

Termination. Finally, we want to emphasize again the fact that the termination of our expansion is always guaranteed. Although we allow for multiple nested model instantiations and invocation of higher-order model parameters, the top-down parsing and our conscious opposition to use-before-define, make recursive model calls impossible. Note that a model is only considered to be defined *after* it is fully parsed, which prevents a recursive call to itself or passing itself as

argument inside its body. Therefore, it should be clear that our iterative expansion is bounded by a finite number of steps.

```

1 model Inc( val : Ex ) : Ex = { $val + 1 }
2 model Dec( val : Ex ) : Ex = { $val - 1 }
3 model InstrBody
4   ( func : (Ex) -> Ex ) : Stat = {
5     X(rd) := $func( X(rs) )
6   }
7
8 instruction A : F = { $InstrBody( Inc ) }
9 instruction B : F = { $InstrBody( Dec ) }

```

Figure 17. Model Instantiation Example (before)

```

1 instruction A : F = { X(rd) := (X(rs) + 1) }
2 instruction B : F = { X(rd) := (X(rs) - 1) }

```

Figure 18. Model Instantiation Example (after)

5 Evaluation

For the evaluation of the macro system’s efficiency and expressiveness we used VADL ISA specifications of *AArch64*, *AArch32*, *MIPS IV*, *RISCV* and our RISCV-like toy architecture *TriLen*. The following section is separated into a qualitative evaluation and a runtime evaluation section. The qualitative evaluation investigates a variety of source code properties like amount of *models*, *records* or type-aliases. The runtime section provides an overview of the execution time.

5.1 Qualitative Evaluation

This section should provide an overview on the macro system’s expressiveness with a particular focus on the presented macro concepts, e.g. higher-order *models* or type compositions. For evaluation, we implemented data collection passes before and after the macro expansion pass. Lines of code and lines of comments were collected manually. Lines of code exclude any trailing empty lines in a file. To determine the comments only lines, we used following regex:

$$(^[""]*/./.$) | (^[""]*/\[/\[/s\S\[/n]+? */)$$

Firstly, we present the overall expressiveness by providing a comparison between the original specifications and their expanded, pretty-printed results. Table 2 contains data on the lines of code, lines of comments, CST nodes and instruction definitions before and after expansion for each architecture respectively. The instruction definitions value includes all

instruction language elements (see Section 2.2) no matter if it contains placeholders or if it is located inside a model template. The concept of CST nodes are described in Section 4.1. We believe that instruction definitions and CST nodes are a more accurate metric to demonstrate the actual generative capabilities of our system. Lines of code (including comment lines) is a more tangible metric, which is why we also provided them.

It is important to note that our pretty printer does not insert new lines in-between an expression, which in case of nested if-else-expressions could result in even more output lines. Additionally, all the comments are not preserved during the parse step and are therefore missing in the output.

Table 2. Expansion Statistics

		AArch32	AArch64	MIPS IV	RISCV	TriLen
Original	Lines of Code	1273	2334	1131	635	521
	Lines of Comments	101	177	108	99	113
	CST Nodes	17158	36699	10156	6264	8557
	Instr. Def.	53	52	45	17	20
Expanded	Lines of Code	110369	10227	1432	612	1301
	Lines of Comments	-	-	-	-	-
	CST Nodes	1520796	134601	14123	7698	16066
	Instr. Def.	8865	799	106	37	123

Our prime examples are the ARM specifications *AArch64* and especially *AArch32*. Their specifications heavily use the macro system to model the different instruction variations. This can be seen by their 52 and 53 initial instruction definitions, which expand to 799 and a tremendous 8865 instruction definitions. These big numbers can be explained by the fact that the final specification explicitly models each hardware mode and conditional execution combination as a separate instruction definition. Although the other examples are not that impressive, *MIPS IV*, *RISCV* and *TriLen* still show improvements regarding code size and especially instruction definition abstraction. *RISCV*’s lines of code is a perfect example why we chose to provide additional metrics to compare the expanded specifications to the originals. The lines of codes decrease after the expansion, indicating that the macro system introduced more verbosity. However, when we look at the CST nodes and the instruction definitions, one can see that there actually was an increase in syntax elements after the expansion. Additionally, the instruction definition amount doubled, which is an indication that the system provides a good abstraction. Overall, we are satisfied with the expressiveness of the macro system as the code reduction measured by the CST nodes of our examples are between the factors 1.2 and 90.

Furthermore, we were interested in the absolute frequency of each macro element for each architecture specification respectively. Table 3 displays the overall amount of *model* definitions, their placeholders inside the template, model instantiations (macro invocations), *record* definitions, type-alias definitions and the uses of our macro conditional (*match*).

Table 3. Macro Element Statistics

	AArch32	AArch64	MIPS IV	RISCV	TriLen
Models	90	142	64	4	21
Placeholders	1168	1270	311	30	178
Instantiations	225	322	163	24	56
Records	4	9	0	0	0
Type-Alias	4	8	0	0	0
Match	0	12	2	0	0

An interesting discovery when looking at Table 2 and Table 3 is that more model definitions and model instantiations do not necessary mean a higher increase in CST nodes. This can be seen by comparing the properties for *AArch32* and *AArch64*. In general, complex architectures like *AArch32*, *AArch64* and even *MIPS IV* seem to use the set of features more than the simpler ones like *RISCV* and *TriLen*. Especially, when it comes to the use of type aliases and records, the complex architectures benefit more from it. The reason is that their model templates are more complex. For the given examples, the ratios of placeholders to models seem to give a good indicator on this complexity. Furthermore, it is interesting to see that records, type aliases and *match*-statements have been used sparsely.

Finally, we decided to provide an overview of a variety of parameter related statistics for *model* definitions. Table 4 shows the minimum, maximum, absolute and average amount of arguments for:

- **Normal Parameters:** This includes all model parameters.
- **Flattened Parameters:** This includes all model parameters with a slight manipulation. All parameters, which resolve to the type *record* or tuple are flattened. This means that each element inside a record or tuple is iteratively unpacked and moved to the outer parameter list. The original type container is removed.
- **Higher-Order Parameters:** This includes all parameters that are of higher-order, i.e. are of type, or contain a subtype, of the function type.

Table 4. Model Parameter Statistics

		AArch32	AArch64	MIPS IV	RISCV	TriLen
Normal	Min	0	0	1	4	2
	Max	9	8	5	6	8
	Abs	343	417	187	21	96
	Avg	3.81	2.93	2.92	5.25	4.57
Flattened	Min	0	0	1	4	2
	Max	14	15	5	6	8
	Abs	738	886	187	21	96
	Avg	8.2	6.23	2.92	5.25	4.57
Higher-Order	Min	0	0	0	0	0
	Max	3	2	0	0	0
	Abs	37	36	0	0	0
	Avg	0.41	0.25	0	0	0

Table 4 presents statistics regarding the number of macro arguments. It shows that for the two complex architectures

the usage of record types halves the maximum and average number of arguments. Higher-order and record arguments are only used by the complex architectures.

5.2 Runtime Evaluation

The runtime evaluation was done on an Apple Mac mini M2 Pro with 32 GB memory under macOS Ventura 13.4 using OpenJDK 64-Bit Server VM Temurin-17.0.6 and the newest version of the VADL tool. Figure 5 shows the runtime of the source parsing and macro expansion pass in milliseconds on the original source specification and on the expanded source code which is the result of the expansion of all macros.

Table 5. Runtime of the Macro System in milliseconds

	Original Source		Expanded Source
	Parse	Expand	Parse
AArch32	542	1144	2415
AArch64	579	213	782
MIPS IV	520	28	514
RISCV	449	10	454
TriLen	453	24	462

Each specification was executed 3 times and the minimal time was selected. The variance between the 3 runs was very low. The parse pass scans the text and generates the CST. The expansion pass traverses the CST, does macro expansion and generates an expanded CST. Additionally, we applied the same approach to the already expanded specifications to have a direct comparison between parsing with and without macros. This provides an idea for how much time is actually taken up by the macro system. Table 5 shows that the macro expansion adds almost no additional runtime for specifications with sparse use of macros. More surprisingly is the savings in runtime for *AArch32*. This can be explained by the increased I/O and parsing effort, compared to much faster in-memory expansion operations.

5.3 Reflection

In this section we would like to give a brief overview on interesting experiences we gained while investigating a macro system design for VADL.

To use VADL's existing IDE integration feature, the syntax errors must be detected by the parser. While most definitions, e.g. *models*, did not impose a real problem, we soon realized that *model* instantiations, and parameter uses are not safely parsable in LL(k) without some form of syntax type hints. These hints were essentially the syntax types of parameters or models written before an identifier, e.g. "*\$(Id a)*" or "*\$(Stmt b)*". While this helped in the context of grammar ambiguities, the identifiers may still fail in the latter applied expansion pass as they were never checked. Furthermore, we ran into similar issues when it came to parsing instantiation arguments, as they potentially allow a huge set of syntax

rules. With the small extension to *pred-LL*(*) we were able to not only remove the unwanted syntax, but also guide the parser in a much cleaner manner. Now the parser guarantees syntactic correctness even before the macro expansion, all while preserving the IDE functionalities.

Another interesting realization was that a macro system for PDLs does not necessarily require hygienic macros. Most of the time, capturing identifiers is a desired behavior. For the remaining cases we preferred the control over the names provided by our *lexical* macro functions.

In contrast to GPLs, VADL macros are mainly used for code generation instead of language extension. *Lexical* macros have proven to be prone to errors, especially for larger specifications. *Procedural* macros are very powerful, but we have not yet discovered a use case where the resulting increase in complexity would be profitable. This is why we found the *pattern-based* macro approach extended with our *higher-order macros* a sensible compromise between complexity and implementation effort for PDL specification.

Finally, we would like to make a few comments about the uses of our macro elements. The *record* element improves the readability of our specification greatly as it enables us to group related parameters together. The *models* were mostly used to express variance of instruction semantic. The *match* statement together with the command line configurability were mainly used to model the variance of processors.

6 Related Work

Most of the related work we found that fits our class of macro system, i.e. *syntactical* and *pattern-based*, were focusing on syntactically rich GPLs and not simple DSLs. Our work took inspiration from *<bigwig>* [7] and programmable syntax macros [26]. We tried to capture some core ideas and incorporate them in our higher-order, composable syntax types in form of the presented VADL macro system.

One of the first approaches to bring syntax macros to syntactically rich languages or even languages outside the LISP family are programmable syntax macros for C [26]. The pattern-based macro system by Weise and Crew uses an extended version of the C language as macro language. Similar to our approach, a macro is defined by a meta construct providing the resulting syntax type, typed parameters and a template body with *placeholders*. Weise and Crew's macro headers enable parameter parsing in form of patterns, providing more syntactic freedom than our approach. However, this freedom may lead to unparseable macros if it cannot be deterministically parsed, which is always guaranteed by our approach. Both approaches rely on a context-sensitive parser to manage macros. Weise and Crew's macro system does not support such a rich type system as ours (e.g. higher-order, records, type-alias). Furthermore, they do not support alternatives in their templates.

The *<bigwig>* [7] is an extensible system for interactive web service. Similarities to our approach can be found in the usage of non-terminal grammar rules as syntax types and the way their macros and our *models* are defined. However, our approach differ in fundamental design decisions concerning the *metamorphisms*. Morphing new keywords into the host language's grammar would require costly rebuilds of our parser. As a result, we decided to increase the initial complexity by making our parser context-sensitive, but save ourselves the execution of a preprocessor. The *<bigwig>* macro system has a similar approach to our higher-order *models* by allowing *metamorph* rules as arguments. However, these rules are translated into grammar rules. They are not instantiated with syntax typed arguments, but guide the way on parsing the arguments. This allows for more syntactic freedom, for example arbitrary arity.

Similar to the previous mentioned techniques, the *ExJS* [25], a macro system for JavaScript, also splits its macro into a pattern and template part. Instead of *metamorph* non-terminals, it uses so called *phantom patterns* to establish arbitrary length repetition. Compared to us the template syntax types are rather poor, as it only supports *expressions* or *statements*. The biggest difference to us and the other approaches is the implementation. *ExJS* uses a first stage parser to build a second stage macro-aware parser to retrieve the AST. While similar implementations exist [4, 7], they further convert the AST to S-expression, feed them to a scheme macro expander and convert the resulting S-expression back to macro-free JavaScript.

The *Honu* [22] macro system follows a simpler approach when it comes to macro patterns and templates. Although the patterns are still syntactically more expressive than our simple parameter list, they follow a much simpler more LISP-like convention. The main idea presented is the *enforestation* parsing step, which converts a flat stream of tokens into an S-expression-like tree. It allows for LISP-style extensibility while still providing enough syntactic freedom to supporting macro infix operators.

One of the newer macro systems that is used in various fields ranging from DSLs to symbolic verification language extensions is the one from the Rust programming language [11, 13, 15, 17]. Specifically, the pattern-based *macro_rule!* is of interest compared to our work. It shares the technique of specifying a pattern with typed parameters that is rewritten based on a template. What stands out is that it supports more general and meta types, e.g. token-tree, pattern or item. Furthermore, the macros are invoked using their identifier and a trailing *!*. The defined pattern is then provided inside parenthesis. This is closer related to our LISP-like macro invocation, in contrast to the previous mentioned approaches. As with the previous macro systems, Rust does not support higher-order macros.

7 Conclusion and Future Work

We have designed and implemented a type-safe higher-order macro system, specifically designed for (computationally weak) architecture description languages. Our presented pattern-based syntax macros are pred-LL(*) parsable and require no preprocessor or complex host language. The implementation was based on a context-sensitive top-down parser and an iterative expansion algorithm with termination guarantee. We demonstrated our approach using our work-in-progress specification language VADL. The evaluation of our macro system was conducted with the help of VADL based ISA specifications for the AArch64, AArch32, MIPS IV, RISC-V and TriLen, our RISC-V-like toy architecture.

Our macro system has still shortcomings we want to address in future work. First, the VADL module management and import system is currently put after the macro expansion, which limits the scope of a macro definition to a single file. Second, we would like to increase the syntactical freedom, for example variable arguments or arbitrary patterns, to see if they work well with our type system. Third, we want to extend our work by some form of hygienic macros. Finally, we want to continue our exploration of design possibilities for ADLs using macros.

Acknowledgment

We want to thank the reviewers for their constructive feedback and valuable suggestions. Part of this work was supported by a grant from Huawei.

References

- [1] Norman I Adams IV, David H Bartley, Gary Brooks, R Kent Dybvig, Daniel Paul Friedman, Robert Halstead, Chris Hanson, Christopher Thomas Haynes, Eugene Kohlbecker, Don Oxley, et al. 1998. Revised report on the algorithmic language Scheme. *ACM Sigplan Notices* 33, 9 (1998), 26–76.
- [2] Rodolfo Azevedo, Sandro Rigo, Marcus Bartholomeu, Guido Araujo, Cristiano Araujo, and Edna Barros. 2005. The ArchC architecture description language and tools. *International Journal of Parallel Programming* 33 (2005), 453–484.
- [3] Jonathan Bachrach and Keith Playford. 2001. The Java Syntactic Extender (JSE). In *Proceedings of the 16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Tampa Bay, FL, USA) (OOPSLA '01). Association for Computing Machinery, New York, NY, USA, 31–42. <https://doi.org/10.1145/504282.504285>
- [4] Jonathan Bachrach, Keith Playford, and C Street. 1999. D-expressions: Lisp power, Dylan style. *Style DeKalb IL* (1999).
- [5] Michael Ballantyne, Alexis King, and Matthias Felleisen. 2020. Macros for Domain-Specific Languages. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 229 (nov 2020), 29 pages. <https://doi.org/10.1145/3428297>
- [6] Lorenzo Bettini. 2016. *Implementing domain-specific languages with Xtext and Xtend*. Packt Publishing Ltd.
- [7] Claus Brabrand and Michael I. Schwartzbach. 2002. Growing Languages with Metamorphic Syntax Macros. *SIGPLAN Not.* 37, 3 (jan 2002), 31–40. <https://doi.org/10.1145/509799.503035>
- [8] R Kent Dybvig. 1992. *Writing hygienic macros in Scheme with syntax-case*. Technical Report. CiteSeer.
- [9] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt. 2015. The Racket manifesto. In *1st Summit on Advances in Programming Languages (SNAPL 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.
- [10] George Hadjiyiannis, Silvina Hanono, and Srinivas Devadas. 1997. ISDL: An Instruction Set Description Language for Retargetability. In *Proceedings of the 34th Annual Design Automation Conference* (Anaheim, California, USA) (DAC '97). Association for Computing Machinery, New York, NY, USA, 299–302. <https://doi.org/10.1145/266021.266108>
- [11] Kyle Headley. 2018. A DSL embedded in Rust. In *Proceedings of the 30th Symposium on Implementation and Application of Functional Languages*. 119–126.
- [12] Brian W Kernighan and Dennis M Ritchie. 1977. *The M4 macro processor*. Bell Laboratories Murray Hill, NJ.
- [13] Steve Klabnik and Carol Nichols. 2023. *The Rust programming language*. No Starch Press.
- [14] Eugene Kohlbecker, Daniel P Friedman, Matthias Felleisen, and Bruce Duba. 1986. Hygienic macro expansion. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*. 151–161.
- [15] Benjamin Lamowski, Carsten Weinhold, Adam Lackorzynski, and Hermann Härtig. 2017. Sandcrust: Automatic sandboxing of unsafe components in Rust. In *Proceedings of the 9th Workshop on Programming Languages and Operating Systems*. 51–57.
- [16] Yannis Lilis and Anthony Savidis. 2019. A Survey of Metaprogramming Languages. *ACM Comput. Surv.* 52, 6, Article 113 (oct 2019), 39 pages. <https://doi.org/10.1145/3354584>
- [17] Marcus Lindner, Jorge Aparicius, and Per Lindgren. 2018. No panic! Verification of Rust programs by symbolic execution. In *2018 IEEE 16th International Conference on Industrial Informatics (INDIN)*. IEEE, 108–114.
- [18] 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.
- [19] Terence Parr and Kathleen Fisher. 2011. LL (*) the foundation of the ANTLR parser generator. *ACM Sigplan Notices* 46, 6 (2011), 425–436.
- [20] Terence J Parr and Russell W Quong. 1994. Adding semantic and syntactic predicates to LL (k): pred-LL (k). In *Compiler Construction: 5th International Conference, CC'94 Edinburgh, UK, April 7–9, 1994 Proceedings* 5. Springer, 263–277.
- [21] Stefan Pees, Andreas Hoffmann, Vojin Zivojnovic, and Heinrich Meyr. 1999. LISA — machine description language for cycle-accurate models of programmable DSP architectures. In *Proceedings of the 36th Annual ACM/IEEE Design Automation Conference*. 933–938.
- [22] Jon Raftkind and Matthew Flatt. 2012. Honu: Syntactic Extension for Algebraic Notation through Enforestation. In *Proceedings of the 11th International Conference on Generative Programming and Component Engineering* (Dresden, Germany) (GPCE '12). Association for Computing Machinery, New York, NY, USA, 122–131. <https://doi.org/10.1145/2371401.2371420>
- [23] Richard M Stallman and Zachary Weinberg. 1987. The C preprocessor. *Free Software Foundation* (1987), 16.
- [24] Guy Steele. 1990. *Common LISP: the language*. Elsevier.
- [25] Ken Wakita, Kanako Homizu, and Akira Sasaki. 2014. Hygienic Macro System for JavaScript and Its Light-Weight Implementation Framework. In *Proceedings of ILC 2014 on 8th International Lisp Conference* (Montreal, QC, Canada) (ILC '14). Association for Computing Machinery, New York, NY, USA, 12–21. <https://doi.org/10.1145/2635648.2635653>
- [26] Daniel Weise and Roger Crew. 1993. Programmable Syntax Macros. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation* (Albuquerque, New Mexico, USA) (PLDI '93). Association for Computing Machinery, New York, NY, USA, 156–165. <https://doi.org/10.1145/155090.155105>

Received 2023-07-14; accepted 2023-09-03

C2TACO: Lifting Tensor Code to TACO

José Wesley de Souza Magalhães

jwesley.magalhaes@ed.ac.uk
University of Edinburgh
UK

Elizabeth Polgreen

elizabeth.polgreen@ed.ac.uk
University of Edinburgh
UK

Jackson Woodruff

J.C.Woodruff@sms.ed.ac.uk
University of Edinburgh
UK

Michael F. P. O’Boyle

mob@inf.ed.ac.uk
University of Edinburgh
UK

Abstract

Domain-specific languages (DSLs) promise a significant performance and portability advantage over traditional languages. DSLs are designed to be high-level and platform-independent, allowing an optimizing compiler significant leeway when targeting a particular device. Such languages are particularly popular with emerging tensor algebra workloads. However, DSLs present their own challenge: they require programmers to learn new programming languages and put in significant effort to migrate legacy code.

We present C2TACO, a synthesis tool for synthesizing TACO, a well-known tensor DSL, from C code. We develop a guided enumerative synthesizer that uses automatically generated IO examples and source-code analysis to efficiently generate dense tensor algebra code. C2TACO is able to synthesize 95% benchmarks from a tensor benchmark suite, outperforming an alternative neural machine translation technique, and demonstrates substantially higher levels of accuracy when evaluated against two state-of-the-art existing schemes, TF-Coder and ChatGPT. Our synthesized TACO programs are, by design, portable achieving significant performance improvement when evaluated on a multi-core and GPU platform.

CCS Concepts: • Software and its engineering → Source code generation; Domain specific languages.

Keywords: Program Lifting, Synthesis, TACO, Tensor Algebra

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
GPCE ’23, October 22–23, 2023, Cascais, Portugal

© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 979-8-4007-0406-2/23/10...\$15.00

<https://doi.org/10.1145/3624007.3624053>

ACM Reference Format:

José Wesley de Souza Magalhães, Jackson Woodruff, Elizabeth Polgreen, and Michael F. P. O’Boyle. 2023. C2TACO: Lifting Tensor Code to TACO. In *Proceedings of the 22nd ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE ’23)*, October 22–23, 2023, Cascais, Portugal. ACM, New York, NY, USA, 15 pages. <https://doi.org/10.1145/3624007.3624053>

1 Introduction

In the last decade, we have witnessed a dramatic increase in machine learning (ML) use in applications ranging from cloud computing to edge devices [45]. ML workloads are dominated by tensor code [60], leading to large-scale efforts aimed at improving its performance [67]. Dense tensor algebra is highly parallel, allowing efficient hardware exploitation across platforms. However, extracting effective parallelism from existing languages is difficult with current compiler technology. This language/compiler failure has led to the growth of domain-specific languages (DSLs) aimed at efficient linear algebra e.g., Diesel [27], TACO [37]. These DSLs deliver excellent cross-platform performance outperforming existing approaches [38].

Accessing such performance is straightforward for new applications: just write your program in the appropriate DSL. However, for legacy programs, it is more problematic with the programmer responsible for both rewriting sections in the new DSL and reintegration with the existing application. As DSLs continuously evolve, this rewriting must be repeated several times throughout the lifetime of the application. This is costly and error-prone, presenting a serious barrier to existing applications to harness hardware performance.

1.1 Existing Techniques

Rewriting is a significant issue and there are a number of different approaches aimed at automatically porting programs to access hardware performance without programmer effort.

API Matching: Rather than translate programs into high level-DSLs, some techniques aim to match and replace sections of user code with fast libraries. For example, Ginsbach et al. [30], De Carvalho et al. [24], and Martínez et al. [44] propose schemes to discover specific code patterns, such as matrix multiplication, and replace them with accelerator

calls. However, these matching tools are often brittle and cannot be extended. They require retooling whenever the target API changes, which makes such approaches non-portable.

Program Lifting via Synthesis: There are several lifting approaches based on program synthesis, i.e., algorithms for generating programs from specifications. Synthesis is used directly to lift legacy code in the work by Kamil et al. [34], where the user defines the region of code to lift. However, a compiler from the program source to the internal format and a decompiler to the high-level DSL have to be provided, limiting the applicability of this approach to new DSLs and legacy software. The synthesis used in this lifting is reliant on SMT solvers to guarantee correctness and drive search. This means these techniques cannot be easily applied to the benchmarks we tackle in our paper, which, owing to pointers, and unbounded tensors and loops, are too complex for the state-of-the-art SMT-solver driven software verification tools to reason about. We attempt to verify bounded correctness for some of our synthesized code, but even for simple benchmarks, we cannot verify correctness for tensors of size more than $10 \times 10 \times 10$ within a timeout of 1 hour. This makes this verification impossible to embed into a synthesis loop where we check thousands of candidates. Our synthesis must use alternatives like observational equivalence [22] in order to achieve the necessary scalability.

Neural Machine Translation (NMT): Language models have proved useful in translation/transpilation tasks. In the work by Roziere et al. [55], an unsupervised Java to C# model is learned using a sequence-to-sequence transformer. It is shown to be reasonably accurate, however, like most NMT techniques, it requires a large corpus of source and target code which is not available for emerging DSLs where most source programs do not have a corresponding domain-specific representation.

1.2 Our Approach

This paper presents C2TACO, a synthesis tool for lifting dense tensor code written in C to TACO. We propose a guided enumerative synthesis method to generate TACO programs based on automatically generated IO examples. We use source code analysis to retrieve features from the original programs and use them as search aids during synthesis.

We compared the performance of C2TACO against a neural machine translation approach and two state-of-the-art existing schemes, TF-Coder [59] and ChatGPT [48]. When evaluated on a suite of tensor benchmarks, C2TACO is able to synthesize 95% of the programs, demonstrating considerably higher accuracy than the other techniques (10%, 32% and 24% respectively). Because they are portable, our lifted TACO programs achieve significant performance improvements over the original implementation when evaluated on

```
for (i= 0; i<N; i++){
  for (k = 0; k<N; k++){
    sum = sum + X[i][k]*b[k];
  }
  a[i] = sum + c[i];
}
```

Figure 1. C implementation of matrix vector product and summation.

a multi-core (geo-mean 1.79x) and GPU (geo-mean 24.1x) platform.

This paper makes the following contributions:

- A guided program synthesis technique that discovers and lifts legacy C code to TACO, a domain-specific tensor language based on behavioral equivalence and on the original program structure.
- An extensive evaluation against existing synthesizer and neural machine translation models, showing that our approach has higher coverage and is more accurate than existing approaches.

2 Motivation

In this section we briefly introduce TACO and describe how and why we lift C to TACO.

2.1 TACO

TACO [38] is a high-level programming language for tensor contractions. A tensor is a generalization of a matrix (order 2) to higher orders. It supports tensor expressions of unbounded length and supports tensors of unbounded order. The core TACO language is based on Einstein summation notation (Einsum) allowing concise representation of tensor computation using tensor index notation. It has been used in other frameworks including TVM [19].

Consider the matrix-vector product and summation example: $a_i = \sum_k X_{i,k} b_k + c_i, \forall i$. In C a simple sequential implementation would result in the code shown in Figure 1. While straightforward, targeting this code for different platforms such as multi-cores or GPUs would require significant code restructuring. Writing the example in TACO gives:

$$a(i) = X(i, k) * b(k) + c(i).$$

This is nearer the original formulation and, crucially, does not include any assumptions about whether the platform is sequential or parallel. The TACO compiler takes this program as input and generates platform-specific optimized code.

2.2 Example

We take existing legacy C code, lift it to TACO, and then use TACO’s code generation abilities to target diverse, high-performance platforms. Consider the program in Figure 2. This is a C function from the DSPStone benchmark suite [72]

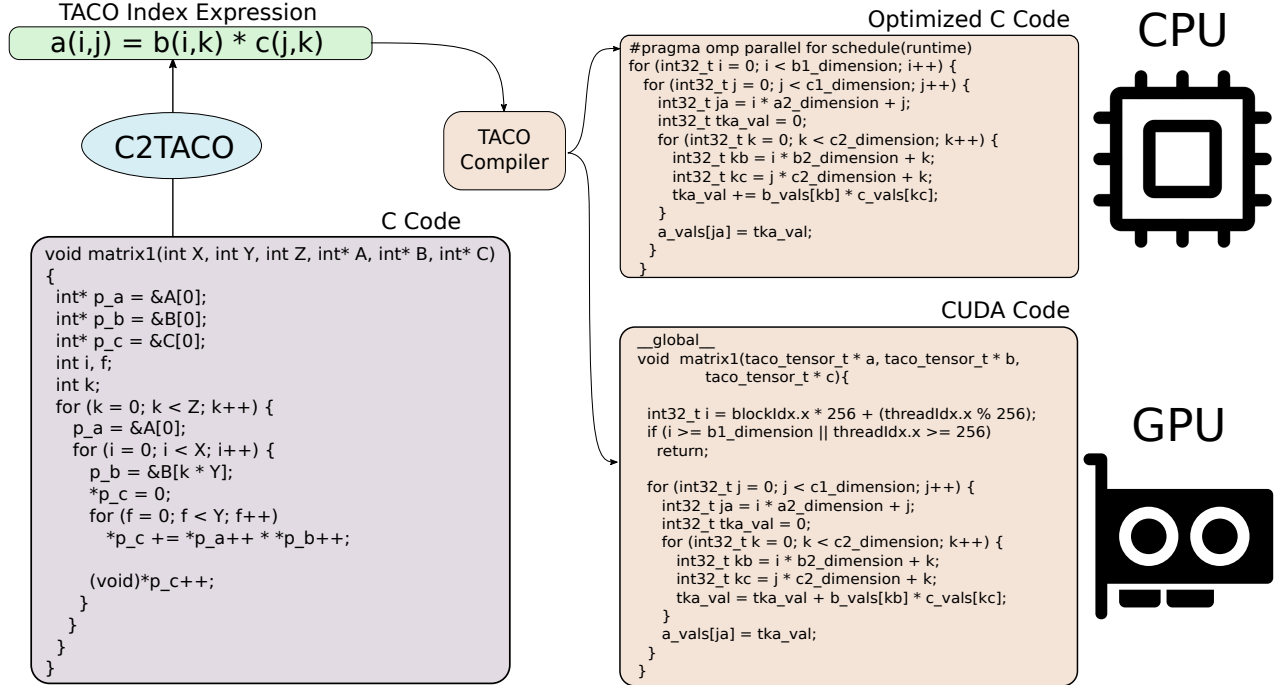


Figure 2. Lifting C code to TACO using C2TACO. Given a program implemented in C, C2TACO generates a equivalent program written in TACO tensor index notation which the TACO compiler can use to produce high-performance code targeting a variety of hardware platforms.

which makes use of post-increment pointer arithmetic to target the addressing modes found in DSP processors. Although the pointers are a hindrance to understanding, this program is in fact matrix multiplication.

C2TACO uses automatically-generated input/output examples as a specification for an enumerative synthesis algorithm. C2TACO uses information about the C program to guide a search through the TACO grammar in a type-directed template-based enumerative fashion and produces the TACO code shown in Figure 2. As well as being higher-level and easier to read than the original C code, the synthesized TACO program can be optimized and targeted at different platforms.

Figure 2 shows the code generated from tensor index notation for a multi-core CPU and an NVIDIA GPU. For the CPU, the TACO compiler generates OpenMP code with a dynamic runtime schedule policy. So in effect, lifting is an automatic parallelization method for certain C programs. For the NVIDIA GPU, the TACO compiler generates CUDA code (also shown in Figure 2). Although, the code is syntactically distinct from the OpenMP version, the TACO compiler again exploits parallelism with implicit concurrence across all of the threads executing the shown kernel.

2.3 Validity

Our synthesized TACO programs are demonstrated to have observational equivalence with the original programs in

C. We also manually inspect the synthesized code. Proving these programs are equivalent is a challenging task due to the unbounded loops and data structures and pointers present in the code. Using CBMC [39], a model checker for C programs, we are able to verify three representative benchmarks using very small loop bounds and tensors (in one case, we can only verify up to a loop bound of 10 within a timeout of one hour). Full verification of synthesized code is an open challenge and out of the scope of this paper.

3 Overview

Figure 3 shows our overall approach. We summarize the pipeline of C2TACO and describe the key components in sections 4 and 5 followed by an extensive evaluation (Section 7).

Given a program P written in C, we first detect the program sections K that are suitable for lifting using neural program classification. Once we have extracted the candidate regions, we generate input-output (IO) examples which are then used as a specification for our synthesis scheme. Our system performs a series of static code analysis to extract relevant features from K . We then search the TACO grammar for equivalent programs that satisfy the IO specification using the features of K to prune the program space. Once we have identified a suitable equivalent TACO program T , we

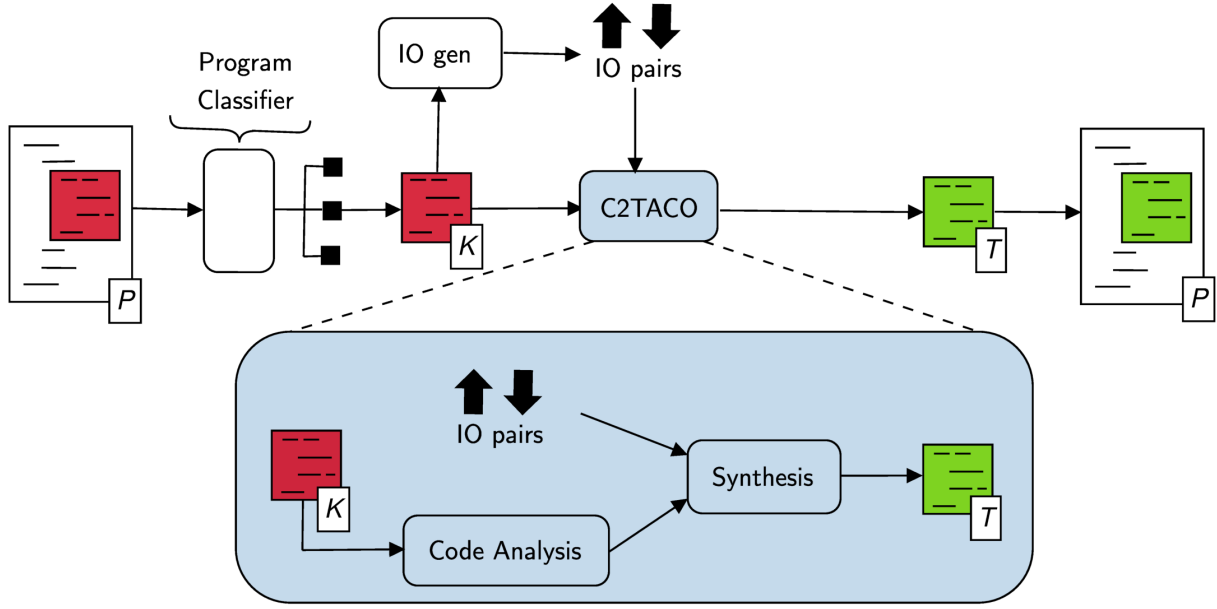


Figure 3. Architecture of C2TACO.

lower it to the target platform and insert it into the original program for execution.

3.1 Classification

We take as input general-purpose programs that perform varied computations and perform lifting to a domain-specific language for tensor contractions. Because we cannot express general computation in TACO, there is a need to identify the code regions that can be lifted and accelerated. We use prior work in neural program classification [69] to determine which parts of the program represent tensor operations.

3.2 IO Generation

Our synthesizer is driven by a specification of observational equivalence (i.e., randomly generated input-output examples). We generate 10 input-output examples. Whilst this means that we cannot *guarantee* absolute equivalence of the synthesized and source code, it allows our synthesis to scale to programs too complex to be reasoned about by the SMT solvers that drive other lifting techniques [17].

3.3 Lifting via Synthesis

Once we have the IO examples of the code to lift, we explore the space of TACO programs using enumeration of templates over TACO's grammar to generate programs that may be equivalent to the original C program. We execute each candidate on the IO samples to see if it is equivalent. The Enumerative Template Synthesis algorithm is described in Section 4. Given the unbounded size of the TACO program space, this can lead to excessive synthesis time. We, therefore, introduce a compiler tool that extracts a set of features

from the original C program and use it to guide search, as described in Section 5.

3.4 Lowering

Once we have a suitable candidate TACO program, we then compile it to the target platform using TACO's platform-specific optimizing compilation. In this paper, we investigate multi-core and GPU targets. The generated code is then patched into the original calling program and evaluated on the target platform.

4 Enumerative Template Synthesis

The task of automatically lifting C to TACO can be defined as a formal program synthesis problem. That is, given a source program $P_C : \vec{x} \rightarrow \vec{y}$, which is written in C, we wish to find an equivalent program $P_T : \vec{x} \rightarrow \vec{y}$, written in TACO, such that the specification $\forall I \in \vec{x}. P_C(I) = P_T(I)$, i.e., the TACO program behaves identically to the C program on all possible inputs.

We use a bottom-up enumerative synthesis algorithm to enumerate *template* TACO programs, i.e., TACO programs that use symbolic variables in place of all tensors and constants. We then check whether there is a valid substitution of inputs and constant literals for these symbolic variables that satisfies the specification. The enumeration of our algorithm is based on classic algorithms in the literature [9, 66], while the use of a sub-procedure to instantiate concrete variable names and constant literals is based on CEGIS(T) [3].

4.1 The Grammar

Our synthesis algorithm enumerates through a grammar G , shown in Figure 4, which defines a search space of possible template TACO programs. The grammar G is defined as a set of nonterminal symbols NT , terminal symbols, and production rules R . For each rule $r \in R$, $|NT|$ indicates the number of non-terminal symbols in the rule. We refer to the nonterminal symbols on the right-hand side of a rule in the order they appear as NT_0, NT_1, \dots . For example, for the production rule $\langle PROGRAM \rangle ::= \langle TENSOR \rangle = \langle EXPR \rangle$, the nonterminals are $NT_0 = \langle TENSOR \rangle$ and $NT_1 = \langle EXPR \rangle$, and $|NT| = 2$.

The grammar includes symbolic constants and symbolic tensor IDs. When we test the program, we substitute these IDs and symbolic constants with input variables and constants from the source program and test all valid substitutions until we find a program that satisfies the specification. We limit our grammar to 4 index variables, which limits the number of tensor dimensions we can reason about to 4.

```

 $\langle PROGRAM \rangle ::= \langle TENSOR \rangle = \langle EXPR \rangle$ 
 $\langle TENSOR \rangle ::= \langle ID \rangle ( \langle INDEX-EXPR \rangle ) | \langle ID \rangle$ 
 $\langle INDEX-EXPR \rangle ::= \langle INDEX-VAR \rangle$ 
    |  $\langle INDEX-VAR \rangle, \langle INDEX-EXPR \rangle$ 
 $\langle INDEX-VAR \rangle ::= i | j | k | l$ 
 $\langle EXPR \rangle ::= \langle EXPR \rangle + \langle EXPR \rangle$ 
    |  $\langle EXPR \rangle - \langle EXPR \rangle$ 
    |  $\langle EXPR \rangle * \langle EXPR \rangle$ 
    |  $\langle EXPR \rangle / \langle EXPR \rangle$ 
    |  $\langle CONSTANT \rangle$ 
    |  $\langle TENSOR \rangle$ 
 $\langle ID \rangle ::= T_0 | T_1 | T_2 | \dots$ 
 $\langle CONSTANT \rangle ::= C_0 | C_1 | C_2 | \dots$ 

```

Figure 4. TACO grammar.

4.2 Specification

Given a source function $P_C : \vec{x} \rightarrow \vec{y}$, we wish to find an equivalent TACO function $P_T : \vec{x} \rightarrow \vec{y}$ such that $\forall I \in \vec{x}. P_C(I) = P_T(I)$. Checking this equivalence is undecidable in general, however, due to the lack of data-dependent control-flow in TACO programs, it is sufficient in almost all cases to check observational equivalence.

We extend the method set out in FACC [69], where inputs are randomly generated according to manually given constraints dictating the length of arrays and favoring smaller values to make evaluation faster. We constrain arrays to be of size 4096, and fix tensor-dimensions to be equal (e.g., a 2-dimensional tensor is of size 64×64).

A single input-output example I, O consists of a set of randomly generated arguments $I = (i_1, \dots, i_m)$, corresponding

to the input parameters $\vec{x} = (x_1, \dots, x_m)$, and an output $O = P_C(i_1, \dots, i_m)$. We generate 10 input-output examples which form a specification: $\phi_{IO} = \{(I, O)_1, \dots, (I, O)_{10}\}$. A program P_T satisfies the specification ϕ_{IO} iff $\forall (I, O) \in \phi_{IO}. P_T(I) = O$. To determine this in practice, we run P_T using the TACO Python API, checking if the behavior matches the corresponding outputs.

4.3 Template Enumeration

We implement bottom-up enumeration i.e., we enumerate templates starting with the shortest first. We define the length of a template as the number of references to tensors or constants in the template, e.g., the template $T_0[i] = T_1[i] + 2$ has length 3 because it refers to T_0, T_1 and 2.

We enumerate templates as shown in Algorithm 2, by iterating through production rules until we have found all possible complete templates of length 1 in the grammar. We then increase the length and repeat the process, using the previously enumerated templates as building blocks, until we have hit the maximum user-given length. Each time the length increases, we add a new tensor ID and a new symbolic constant to the set of candidate templates. This is shown in Algorithm 1.

We discard any invalid candidates during enumeration, i.e., templates that do not type check or are unsupported by TACO. More specifically we discard: any candidate that iterates over two different dimensions with the same index variable (e.g., $T_0(i, i)$); any candidate where the same tensor appears more than once in a program with different orders (e.g.: $T_0(i) = T_1(i) * T_1(i, j)$); and any candidate where the same tensor appears on both sides of an assignment (e.g.: $T_0(i, j) = T_0(i, j) + T_1(j, k)$).

4.4 Instantiating Templates

After we have generated all templates of length L , we check whether any of these templates generate programs that satisfy the specification, ϕ_{IO} (see Section 4.2). To do this, we enumerate through all substitutions that map all symbolic constants in the candidate program to concrete values, and all tensor IDs to inputs in the specification, until we find a substitution that gives us a TACO program that satisfies the specification. This is shown in Algorithm 2. We limit the concrete constant values to constants present in the source program.

We check all possible substitutions until we find a substitution that results in a complete TACO program that satisfies the specification, which is checked by the *check* procedure. Although checking all possible substitutions has $L!$ complexity for a template of length L , L is typically small (< 5). We check the templates of length MAX before any shorter templates, as this is the likely length of the target program.

Algorithm 1: Enumerative Template Synthesis. The subprocedures *instantiate* and *completeRule* are shown in Algorithm 2.

input : source code P_C , grammar G , max length L
output : candidate program, or no solution

Algorithm *synthesize*(P_C, G, L)

```

short  $\leftarrow \emptyset$ ;           // set of short candidates
long  $\leftarrow \emptyset$ ;       // set of long candidates
 $\phi_{IO} \leftarrow \text{generateSpec}(P_C)$ ;
for  $l$  in  $1 \dots L$  do
    short  $\leftarrow$  short  $\cup$  newTensor()  $\cup$  newCons();
    while true do
        nS  $\leftarrow \emptyset$ ;    // new short candidates
        nL  $\leftarrow \emptyset$ ;    // new long candidates
        for Rule  $\in G$  do
            for  $p \in \text{completeRule}(\text{short}, \text{Rule})$  do
                if Length( $p$ ) =  $L \wedge \text{valid}(p)$  then
                    nL  $\leftarrow$  nL  $\cup$   $p$ ;
                else if Length( $p$ ) <  $L \wedge \text{valid}(p)$  then
                    nS  $\leftarrow$  nS  $\cup$   $p$ ;
            if nS  $\subseteq$  short  $\wedge$  nL  $\subseteq$  long then break;
            if  $l = L$  then
                long  $\leftarrow$  long  $\cup$  nL;
            else
                short  $\leftarrow$  short  $\cup$  nS  $\cup$  nL;
        for  $p \in$  long, short do
             $P_T, \text{result} \leftarrow \text{instantiate}(p, \phi_{IO})$ ;
            if result then return  $P_T$ ;
    return no solution

```

5 Synthesis Guided by Code Analysis

The search space of possible TACO templates is large, and so, in C2TACO, we use program analysis to focus the scope of the synthesis search, prioritizing candidates that are more likely to be correct. In particular, we use heuristics to estimate the correct TACO template length (section 5.1), the correct dimensions (section 5.2) and the operators (section 5.3).

5.1 TACO Program Length

The length of a TACO program is related to the number of array/pointer references and constants in the original C code. However, temporary variables to capture common sub-expressions and mutable arrays mean that there is no direct correspondence. Fixing the size of the target TACO program reduces the search space because we only have to enumerate candidates once.

To determine the range of sizes C2TACO explores, we focus on the definition of the *output* array and examine the number of input arrays, or *uses* [23]. At each definition, we

Algorithm 2: Subprocedures. Note $e.\{x \mapsto y\}$ denotes the result of the proper substitution of the expression x by the expression y in the expression e .

Procedure *completeRule*(short, Rule)

```

completions  $\leftarrow \emptyset$ ;
for  $p \in$  short do
    candidate  $\leftarrow$  Rule. $\{NT_0 \mapsto p\}$ ;
    if |NT|  $\in$  Rule = 2 then
        for  $q \in$  short do
            candidate  $\leftarrow$  Rule. $\{NT_1 \mapsto q\}$ ;
    completions  $\leftarrow$  completions  $\cup$  candidate;
return completions

```

Procedure *instantiate*(p, ϕ_{IO}, P_C)

```

X  $\leftarrow$  getInputParams( $P_C$ );
K  $\leftarrow$  getConstants( $P_C$ );
T  $\leftarrow$  getTensors( $p$ );
C  $\leftarrow$  getConstantSymbols( $p$ );
for  $x, t \in \text{cartesianProduct}(X, T)$  do
    for  $k, c \in \text{cartesianProduct}(K, C)$  do
        result  $\leftarrow \text{check}(p.\{t \mapsto x\}.\{c \mapsto k\}, \phi_{IO})$ 
        if result then return
            result,  $p.\{t \mapsto x\}.\{c \mapsto k\}$ ;

```

iteratively build a set of variables used by that definition. We use reaching analysis to disambiguate between different references to the same (mutable) variables. We then reduce the constructed set in the presence of summations or reductions. In C, when writing a reduction or summation, a variable appears on both sides of an assignment but only once in the TACO program. For this reason, we apply simple data dependence analysis to check if there is a recurrence. If there is, we do not count it twice.

For example, in Figure 1, we have the use set sum, X , b , c for the the output array a . This is reduced to X , b , c after detecting the reduction on sum to give 4 (a , X , b , c) as the predicted number of tensors in the TACO program.

5.2 Tensor Dimensions

C programs frequently contain linearized arrays: where a single pointer is used to represent a multi-dimensional tensor. However, in TACO dimensions are explicit, and searching over all possible dimensions is costly. To address this, we apply the dataflow analysis defined in [29] to recover arrays from pointer structures, and then apply delinearization [47] and determine the highest dimension array.

As an example, consider the program in Figure 2. After applying dataflow analysis to $*p_c$, we get $p_c[Z * k + i]$. Let n be the dimensionality of the recovered C array and m be the dimension of the enclosing loop nest J . Here $n = 1, m = 3$. The loop iterators are represented by a column

vector $J = [k, i, f]^T$, and UJ is the affine expression for the array access $[Z * k + i]$:

$$UJ = [Z, 1, 0] \begin{bmatrix} k \\ i \\ f \end{bmatrix} = [Z * k + i]$$

We now delinearize by constructing a transformation S , such that SU gives a matrix of 1's and 0's. For our example $S = [()/Z, ()\%Z]$. For details of this step, we refer the reader to the paper [47]. We apply the transformation to give:

$$S U J = \begin{bmatrix} ()/Z \\ ()\%Z \end{bmatrix} [Z, 1, 0] J = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} J$$

This gives us a 2D delinearized array access $p_c[k, i]$, so we begin our search for TACO programs using 2D tensors.

5.3 Operator Analysis

Finally, we use the source code to predict which operators are likely to be included in the target program. We do this based on a straightforward analysis of the Abstract Syntax Tree of the source code, which counts the number of appearances of each operator type. This effectively reduces the search space of possible TACO programs by eliminating unlikely combinations of operators.

6 Experimental Methodology

To evaluate C2TACO we compared its performance against other techniques. We implemented a simple version of the synthesis process described in Section 4 and an alternative approach based on neural machine translation. In addition, we consider an existing large language model ChatGPT and IO-based synthesizer, TF-Coder.

6.1 Alternative Approaches

ETS. C2TACO uses the synthesis algorithm described in Section 4 combined with the heuristics described in Section 5. To evaluate the contribution of the heuristics in C2TACO, we compare to the most basic enumerative template synthesis algorithm described in Section 4 (without any heuristics), which we refer to as ETS.

Neural Machine Translation. NMT converts text sequences from one language to another by means of a deep neural network and has shown positive results on code tasks. We therefore frame the task of lifting C to TACO as a neural machine translation problem. We train a Transformer [68] that given a C input sequence minimizes the edit distance between the predicted and ground-truth TACO. Once trained, then, given an unseen C program, the model will generate the most likely equivalent TACO program.

The main challenge for any new DSL is the availability of training data. To overcome this, we generate a synthetic dataset based on the TACO grammar shown in Figure 4. We compile the synthetically generated TACO programs to generate the equivalent C programs. We limit our synthetic

dataset to programs that contain a maximum of 5 tensors of no more than 4 dimensions, and where all datatypes are integers.

We enumerate this space in a bottom-up manner, similar to the enumeration performed by our synthesis algorithm, and use testing to eliminate semantically equivalence programs. Since TACO-generated programs contain details that are unlikely to be present in real-world tensor kernels such as memory allocation, we modify the clang compiler to extract only the kernel signature and computation of the program for our equivalent C program.

We generate 800K pairs of C program and TACO expressions of which we separated 5K for validation, 5K for test, and the remaining were used for training. The trained model is a Transformer with 6 encoders and 6 decoders with 16 attention heads and an embedding size fixed at 1024.

6.2 Existing Approaches

TF-Coder. TF-Coder [59] is an open-source publicly available program synthesizer. It takes a single input-output example as source and generates a corresponding TensorFlow program. Although the search space of TF-Coder is not defined by the same grammar we considered in our synthesis methods, we compare C2TACO against TF-Coder because both synthesize programs from IO examples and operate on the domain of tensor computations. We use one of the IO examples automatically generated by our synthesis scheme, but limit it to less than 100 elements as required by TF-Coder.

ChatGPT. ChatGPT [48] is large-scale language model based on GPT 3.5. It has been used for a wide number of tasks including code generation. We used version 3.5 in our experiments. As its accuracy depends on the quality of its prompts, we experimented with various formats and found the following to be the most effective, followed by the original source code:

"Translate the following C code to an expression in the TACO tensor index notation. The expression must be valid as input to the taco compiler. Return the expression and only the expression, no explanations."

6.3 Setup

Benchmarks. To evaluate C2TACO, we designed two different suites of tensor algebra benchmarks. The first contains C programs generated by the TACO compiler a distinct subset of those used to train the NMT model. The second contains programs from existing software libraries. We refer to these suites as *artificial* and *real-world* respectively.

The real-world benchmarks originate from different applications. We selected a subset of the programs used by previous synthesis work [22]:

Table 1. Synthesis coverage of different approaches on the artificial dataset.

TACO Program	TF-Coder	Correct			
		ChatGPT	NMT	ETS	C2TACO
$a(i) = b(i) + c(i) - d(i)$	✓	✗	✗	✓	✓
$a(i,j) = b(i,j) + c(i,j)$	✓	✓	✓	✓	✓
$a(i) = b(i) * c(i)$	✓	✗	✓	✓	✓
$a(i) = b(i) + c(i) + d(i) + e(i)$	✓	✗	✗	✗	✓
$a(i,j) = b(i,j) * c(j)$	✗	✗	✓	✓	✓
$a(i,j) = b(i,k) * c(k,j)$	✗	✓	✓	✓	✓
$a(i,j) = b(i)$	✗	✓	✓	✓	✓
$a(i,j) = b(i) * c(i,j)$	✗	✗	✓	✓	✓
$a(i,j) = b(i,j,k) * c(k)$	✗	✓	✓	✓	✓
$a(i,j) = b(i,k,l) * c(l,j) * d(k,j)$	✗	✓	✗	✗	✓

- **blas**: baseline implementation of functions from the BLAS [18] linear algebra library as synthesized by Collie et al. [20].
- **DSP**: signal processing functions adapted from the TI [2] library.
- **makespear**: programs that manipulate arrays of integers. Originally from Rosin [54].
- **mathfu**: mathematical functions from the Mathfu [1] library.
- **simpl_array**: problems performing different computation on arrays of integers. Originally from the work by So and Oh [62].

In addition to those, we extracted benchmarks from other suites that contain tensor manipulations:

- **darknet**: neural network operations from the Darknet [53] deep learning framework.
- **DSPStone** and **UTDSP**: kernels targeting digital signal architectures from the DSPStone [72] and UTDSP [56] suites.

We gathered 71 benchmarks in total, of which 10 are artificial and 61 come from real-world code.

Software. ETS and C2TACO are implemented in Python version 3.8.10. The NMT Transformer model is implemented using Fairseq [49] 0-12.2 with Google's SentencePiece [40] as the tokenizer. The analyses described on Section 5 are implemented as plugins for the clang compiler version 14.0.0. Operating system is Ubuntu 20.04.6 LTS.

Hardware. We evaluate on a multi-core CPU and GPU platform. The targeted CPU is an 8-core Intel i5-1135G7 at 2.40GHz with 16 GB of RAM (LPDDR4) at 4267 MT/s. The GPU is an NVidia GeForce GTX 1080 Ti using driver version 535.54.03 and CUDA runtime version 12.2.

Metrics. We evaluated the performance of each approach by executing its generated code 10 times and recording the median. In our experiments, we saw little execution time variance. We measure speedup as the ratio of the running

time of lifted programs over the original version. Programs are compiled with gcc -O3 version 9.4. We also recorded the time to produce a lifted TACO program with a timeout of 90 minutes for all approaches in all the experiments conducted.

7 Evaluation

In this section, we evaluate against four criteria: coverage (Section 7.1), error rate (Section 7.2), synthesis time (Section 7.3), and speedup (Section 7.4).

7.1 Synthesis Coverage

Figure 5 shows the lifting coverage of each of the five schemes described in section 6 across the two benchmark suites: artificial and real-world.

Benchmark Suite: Artificial. As described in section 6, these are C kernels generated by the TACO compiler guaranteed to have an equivalent in the TACO language. The coverage of each scheme is shown in Table 1 and Figure 5.

C2TACO is most effective, lifting all benchmarks correctly. ETS lifts 8 out of 10. In two cases it could not find the correct program in time as the space of possible grows too large. C2TACO overcomes this by using the code analysis information to focus the search on parts of the grammar where the programs are most likely to be the solution. TF-Coder is able to synthesize 4 out of 10 benchmarks but is unable to match the coverage of the other synthesis approaches. Like ETS scheme, it times out for the more complex programs.

NMT achieves higher accuracy, translating seven of the benchmarks. The Transformer model was trained using TACO-generated kernels, which have a similar structure to the synthetic programs. Unlike synthesis methods, it always produces a result even though it may be inaccurate and does not timeout. In the three cases where NMT fails, it correctly guesses the number of tensors but misorders them in the resulting programs.

ChatGPT is able to correctly predict 5 of the 10 benchmarks, hallucinating the remainder. In four cases it produces syntactic invalid programs. The syntax errors include wrong

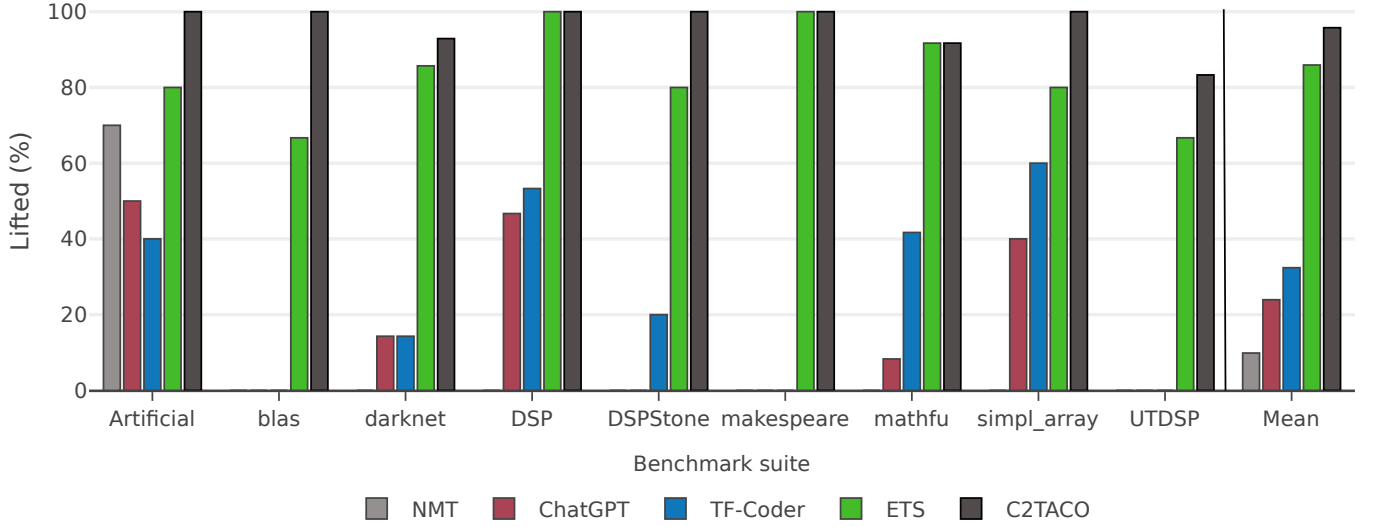


Figure 5. Overall lifting coverage across benchmarks suites.

indices, multiple assignments, and duplication. In one case a tensor was treated as having different orders in the same program. In another, ChatGPT produces a program that is syntactically valid but incorrectly refers to the same tensor twice.

Benchmark Suite: Real-World. Real-world benchmarks are more challenging as shown in Figure 5. Both ETS and C2TACO are able to achieve high coverage of 85% and 95% respectively. ETS times out on 5 out of 61 while the sole instance of failure for C2TACO is the presence of program features not contemplated in our implementation of the grammar 4. TF-Coder manages to correctly synthesize 31% of the benchmarks. Along with timeouts, TF-Coder also produces programs that are semantically incorrect. We further discuss these in Section 7.2.

Real-world programs impose a harder challenge to neural machine translation due to the diversity of their implementation. While artificial programs have a syntactic structure identical to TACO-generated C programs, real-world ones are written in several different fashions, which makes it difficult for sequence-to-sequence methods to recognize patterns. NMT performs particularly poorly compared to the artificial case, generating no correct programs. This reinforces the view that it may be over-specific to a particular style of programming due to its training sample. ChatGPT also has a weak performance, only translating 20% of the benchmarks correctly. As well as in the artificial case, both approaches produce varied hallucinations as we detail below.

7.2 Error Analysis

We identify several different reasons for failure: a large search space causing time out; syntactic and semantically wrong solutions. Figure 6 depicts a summary.

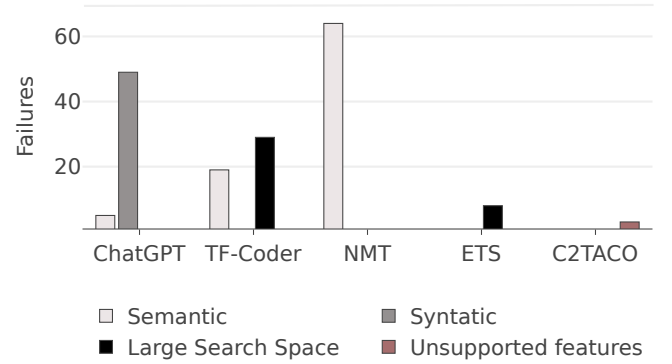


Figure 6. Distribution of failure causes for the different approaches evaluated.

Large Search Space. Enumerative synthesis techniques explore a large search space, which grows as program length increases. This causes 60.42% of TF-Coder’s failures and all failures for ETS. Neural translation approaches, ChatGPT and NMT, always find a solution in time due as they translate a program in a sequence-to-sequence fashion and do not perform an extensive search. Although C2TACO is also based on enumeration, it never times out as program analysis restricts the search space sufficiently.

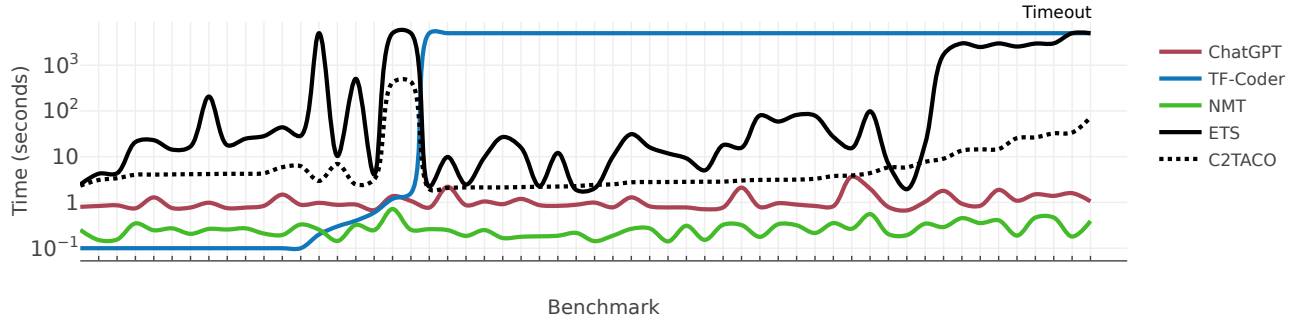


Figure 7. Lifting time on real-world benchmarks. Y-axis is on logarithmic scale.

Syntactic. TF-Coder, ETS, and C2TACO always produce programs that are syntactically correct. On the other hand, neural approaches frequently generate incorrect translations or hallucinations. In addition, 90% of the wrong translations produced by ChatGPT are syntactically incorrect. These hallucinations often include explanations of the ranges of index variables and use braces instead of parenthesis, which is the symbol used for indexation in the TACO tensor index notation language. Example 1 shows a syntactic hallucination produced by ChatGPT.

Example 1. When given as input a program that computes a dot product of two arrays b and c , the expected solution expressed in TACO is

$$a = b(i) * c(i)$$

However, ChatGPT produced the string below which is not a valid TACO program.

$$sum(a[i] * b[i] \text{ for } i \text{ in } 0..< n)$$

Although NMT is also neural-based it always produces well-formed programs. The difference is that NMT is trained on a domain-specific dataset containing only programs generated by the TACO compiler while ChatGPT is trained on more diverse data.

Semantic. These are programs that are syntactically correct, but produce the wrong output when executed. Almost 40% of TF-Coder failures are programs that are semantically wrong. TF-Coder relies on just one IO example and often fails to generalize. The majority of false positives produced by TF-Coder include manipulations on the shape of tensors, which is not present in any of the original benchmarks. Semantic hallucinations also correspond to 9.26% of the incorrect answers produced by ChatGPT. Example 2 shows an example of a hallucination produced by ChatGPT and Example 3 depicts one generated by TF-Coder.

Example 2. For a program that performs general matrix multiplication, the solution can be expressed in TACO as

$$C(i, j) = A(i, k) * B(k, j)$$

ChatGPT generates a program that includes an extra summation and reference to the resulting matrix on the right-hand side. Although that is equivalent according to C semantics, the same is not true in TACO.

$$C(i, j) = C(i, j) + ALPHA * A(i, k) * B(k, j)$$

Example 3. Given a program that computes the product of an array arr with a scalar value v , the correct TACO implementation is:

$$arr(i) = arr(i) * v.$$

TF-Coder synthesizes a solution that, although syntactically valid in TensorFlow, adds arr to itself, which is not semantically equivalent to the original program:

$$tf.add(arr, arr)$$

TACO-generated programs have a particular code structure that does not reflect real-world programming styles, which is why NMT fails to generalize. Semantic hallucinations are the cause of all of NMT's failures.

7.3 Generation Time

Artificial. NMT is by far the fastest approach with a geometric mean of 0.36 seconds. NMT is faster because it does not involve an extensive search and it does not check whether the program is correct using IO examples, which represents the largest part of the synthesis time for the program synthesis approaches. ChatGPT is also fast for the same reasons and translated artificial benchmarks within 1.14 seconds on average.

Despite performing a search, TF-Coder is fast, taking an average of 1.18 seconds to find the solution. Nevertheless, TF-Coder is only able to correctly lift 40% of the artificial benchmarks (Section 7.1). ETS is the slowest method with an average of 238 seconds to find the solution. In contrast, C2TACO takes an average of 21 seconds. That result shows the impacts of the program features obtained by syntactic analyses in guiding the synthesizer to find the correct answer.

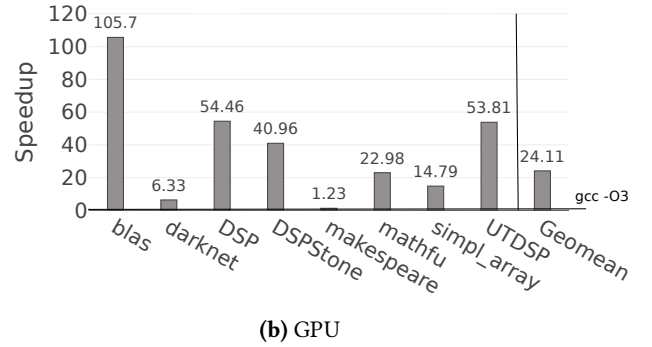
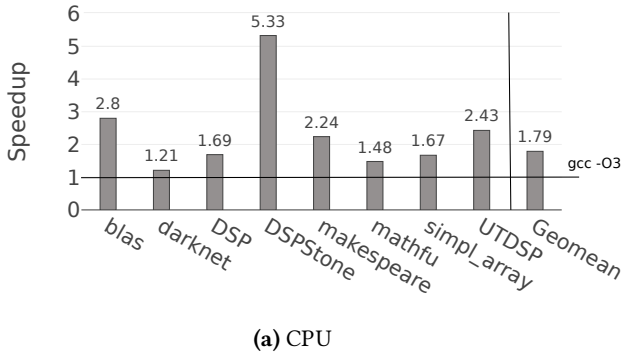


Figure 8. Speedup obtained by the synthesized TACO programs on different hardware platforms. The baseline is the average running time of the original implementations when compiled with gcc -O3.

Real-World. Figure 7 shows the synthesis times for each of the five approaches across the real-world collection. Numbers are on a logarithmic scale. As expected both the neural approaches NMT and ChatGPT are fast and stable across all programs. NMT always returns a program in less than 1 second and ChatGPT takes a maximum of 4 seconds to find a solution. However, as shown in Section 7.1, this speed comes at the expense of frequently generating wrong code.

TF-Coder performs well on the simpler program. It synthesized a solution even faster than neural approaches in 15 cases. Nevertheless, the generation time of TF-Coder rose sharply as the programs became less trivial and it timed out in 42 out of 61 instances. ETS is slower on average however it only times out on 13% of the benchmarks. We observed that ETS particularly struggles with instances of length $N \geq 3$ and programs involving multiple multidimensional tensors, where the number of possible index expressions increases exponentially for each tensor. C2TACO is considerably faster with an average synthesis time of 5.6 seconds and a maximum of 7 minutes. The only cases where C2TACO was slower than ETS involve very simple programs that only perform initialization of arrays with a constant value. For all the other programs C2TACO was able to find a solution faster than ETS and it kept stable across the whole suite.

7.4 Performance of Lifted Code

The main reason we wish to lift code to TACO is to exploit its portable performance. We generated C and CUDA versions of the programs generated by C2TACO and measured their performance on a multi-core CPU and GPU respectively. Figure 8 shows the speedup across the benchmark suite achieved running lifted programs. Baseline is the original implementation compiled with gcc -O3. Only the real-world benchmarks are considered as the artificial ones are directly derived from the TACO compiler and the synthesized version corresponds to the original.

Lifted programs are faster than their original counterparts in both devices. On a multi-core device, the benchmarks are on average 1.79x faster when lifted to TACO. That speedup varies over different benchmark sources. The highest speedup is 5.33x on DSPStone benchmarks and the lowest is 1.21x for the darknet programs. The main reason for the better performance is that the kernels generated by TACO optimize array access by linearizing index expressions and exploit the parallel nature of a multi-core CPU by inserting OpenMP pragmas on loops.

Speedup is even higher on the GPU. The lowest value was 1.23x on the makespere set. However, it is worth emphasizing that makespere contains only 1 program. We noticed high speedups on the digital signal processing benchmarks: DSP, DSPStone, and UTDSP, on which lifted programs are 54.46x, 40.96x and 53.81x faster than the original version. The highest value occurs on the BLAS benchmarks, which run 105.7x faster when lifted. The overall speedup achieved on GPU was 24.11x. Similarly to the multi-core kernels, TACO-generated CUDA kernels are designed to leverage high-level parallelism on GPU accelerators and are optimized aiming to divide the workload uniformly among threads.

Speedup by Program Complexity. We further evaluated the impact of lifting on the performance of programs when such programs become more complex. In our domain, we consider programs more complex as they manipulate tensors with higher orders. We define the concept of dominant order as the highest order among the tensors in a program. For example, the program shown in Figure 1, manipulates tensors of 3 different orders: vectors (order 1), a matrix (order 2) and a scalar variables (order 0). The dominant order for that program is therefore 2.

Table 2 shows the overall speedup obtained on programs with different dominant orders. We observed two categories of dominant orders in the real-world benchmarks, 1 and 2. Programs that handle two-dimensional tensors benefit more

Table 2. Speedup obtained given different tensor dominant orders. We consider the highest order among the tensors in a program as dominant.

Dominant order	Multi-core Speedup	GPU Speedup
1	1.41	20.19
2	3.20	36.97

from being lifted than the ones operating on one-dimensional ones. The speedup goes from 1.41x to 3.20x on the multi-core and from 20.19x to 36.97x on the GPU. These results show that the impact of lifting is even higher for programs that are more complex in the sense that they manipulate multi-dimensional tensors.

7.5 Summary

Overall C2TACO was the most effective method in our evaluation, lifting 95% with an average time of 21 seconds on the artificial suite and 5.6 seconds on the real-world programs. C2TACO was considerably faster than its ETS counterpart, which illustrates that the program analysis used by C2TACO to guide the search shown have a large impact on its generation time. We shown that we obtain performance gains by lifting programs to TACO, achieving an average speedup of 1.79x on a multi-core platform and 24.1x on a GPU.

8 Related Work

In this section we discuss how our work relates to the area of program synthesis and other techniques to automatically construct code.

8.1 Program Synthesis

Program synthesis is a well-studied area where programs are generated based on an external specification. It is the form of specification and the methodology used to generate programs that characterize the different approaches.

Logic. In Syntax-Guided Synthesis(SyGuS) [10] approaches, the program specification is provided in the form of first-order logic. This type of specification allows SMT solvers such as Z3 [25] to be used in a CounterExample Guided Inductive Synthesis(CEGIS) [63] loop to rapidly synthesize candidate programs. Recent work allows extension beyond first-order logic [51], but SyGuS is not well-suited to tensor computations due to the complexity of checking the correctness of a tensor computation using an SMT solver. Due to this limitation, our work uses a testing-based procedure to validate candidates. Our synthesis approach is similar in style to CEGIS(T) [3], in that we enumerate programs with symbolic constants and tensors, and then find the bindings for these constants as part of the correctness check.

IO Examples. IO-based synthesis is part of the programming by example style of synthesis, in which input/output

examples are used as the specification. Early work looked at generating Excel commands from a few examples [31]. The same concept and has been used for other tasks [21, 73], including generating PyTorch or TensorFlow code from tensor inputs [46, 59]. TF-Coder [59] takes as input a single user-provided example to generate equivalent TensorFlow code using type constraints and bottom-up enumerative synthesis. Alternative schemes [16, 46] use deep learning models trained on IO samples to guide the generation of code.

Verified Lifting. Using program synthesis to generate programs from a specification is a long-studied area [28, 61]. Using a low-level program as the specification and a high level-one as the target was tackled by Kamil et al. [34]. Here appropriate stencil-like loops in FORTRAN are lifted to their equivalent in Halide [52]. This has been extended to a more generic LLVM framework [4] based on a common IR. While this has the potential to allow lifting to multiple targets [5–7], it requires the compiler writer to provide a compiler and decompiler from each potential source and target into the IR which is not scalable. Their technique also relies on being able to formally verify the equivalence of the target and source programs in order to give counterexamples to the synthesis algorithm, which we have found is not possible for programs in our benchmark suite. In contrast, whilst it gives weaker guarantees of correctness, our approach is able to synthesize programs based on observational equivalence, and the scalability of our approach is not dependent on the tractability of the equivalence checking problem.

8.2 Other Approaches

Neural Machine Translation. Since the advent of sequence to sequence models [64], neural machine translation has been applied to programming language translation tasks [11, 12, 26], including unsupervised settings [13, 55, 65]. Training data is often extracted from coding websites [42].

Other tasks range from code style detection [50], generating accurate variable names [41], correcting syntax errors and bugs [32, 57] code completion [36] and program synthesis [14] to API recommendation [35], and specification synthesis [43]. While powerful, such approaches are inaccurate and are not mature enough for precise lifting.

API Migration/Matching. Replacing matched code/IR to a fixed API call is a limited form of raising. KernelFaRer [24] works at the program level and restricts its attention to just GEMM API targets, but is more robust than IDL [30] matching significantly more user code. This robustness is extended further by Martínez et al. [44] which uses behavioral equivalence to match code. Such approaches, however, are intrinsically limited as they focus on fixed APIs rather than the open-ended nature of DSLs and their IRs.

Compiling TACO. TACO [37] is a popular DSL for expressing tensor computations. In addition to generating high-performance CPU code [38], it has been extended to compile to GPUs [58], CGRAs [33], high-performance libraries [15] and distributed systems [70]. In addition to these target-specific optimizations, work has been done for sparse tensors [8, 71].

9 Conclusion

This paper presents C2TACO, a synthesis tool for lifting C tensor code to TACO. C2TACO uses equivalence behavior and program analysis to generate code and it is shown to lift more programs in a shorter time with greater accuracy when compared to an alternative NMT and simpler synthesis approaches. C2TACO also outperforms existing techniques, lifting 95% of the benchmarks, against 32% for TF-Coder and 24% for ChatGPT. We demonstrate that the synthesis of equivalent TACO programs is feasible for a range of C programs taken from software libraries and benchmark suites. We also show that we can obtain significant performance improvement over the original source. Using C2TACO we are able to synthesize TACO programs that are 1.79x faster when evaluated on a multi-core CPU and 24.1x when ported to a GPU platform. Future work will explore methods to further improve lifting applicability, by handling sparse tensor algebra, and efficiency using neural-guided synthesis to perform search.

References

- [1] [n. d.]. Mathfu. <https://github.com/google/mathfu>.
- [2] [n. d.]. Texas Instrument Digital Signal Processing (DSP) Library for MSP430 Microcontrollers. <https://www.ti.com/tool/MSP-DSPLIB>.
- [3] Alessandro Abate, Cristina David, Pascal Kesseli, Daniel Kroening, and Elizabeth Polgreen. 2018. Counterexample Guided Inductive Synthesis Modulo Theories. In *CAV (1) (Lecture Notes in Computer Science, Vol. 10981)*. Springer, 270–288.
- [4] Maaz Bin Safeer Ahmad and Alvin Cheung. 2016. Leveraging parallel data processing frameworks with verified lifting. *arXiv preprint arXiv:1611.07623* (2016).
- [5] Maaz Bin Safeer Ahmad and Alvin Cheung. 2017. Optimizing Data-Intensive Applications Automatically By Leveraging Parallel Data Processing Frameworks. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 1675–1678.
- [6] Maaz Bin Safeer Ahmad and Alvin Cheung. 2018. Automatically leveraging mapreduce frameworks for data-intensive applications. In *Proceedings of the 2018 International Conference on Management of Data*. 1205–1220.
- [7] Maaz Bin Safeer Ahmad, Jonathan Ragan-Kelley, Alvin Cheung, and Shoaib Kamil. 2019. Automatically translating image processing libraries to halide. *ACM Transactions on Graphics (TOG)* 38, 6 (2019), 1–13.
- [8] Peter Ahrens, Fredrik Kulstad, and Saman Amarasinghe. 2022. Autoscheduling for Sparse Tensor Algebra with an Asymptotic Cost Model. *PLDI* (2022).
- [9] Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. 2013. Recursive Program Synthesis. In *CAV (Lecture Notes in Computer Science, Vol. 8044)*. Springer, 934–950.
- [10] Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis. In *FMCAD*. IEEE, 1–8.
- [11] Jordi Armengol-Estapé and Michael O’Boyle. 2021. Learning C to x86 Translation: An Experiment in Neural Compilation. In *Advances in Programming Languages and Neurosymbolic Systems Workshop*.
- [12] Jordi Armengol-Estapé, Jackson Woodruff, Chris Cummins, and Michael FP O’Boyle. 2023. SLaDe: A Portable Small Language Model Decompiler for Optimized Assembler. *arXiv preprint arXiv:2305.12520* (2023).
- [13] Mikel Artetxe, Gorka Labaka, and Eneko Agirre. 2019. An Effective Approach to Unsupervised Machine Translation. In *ACL (1)*. Association for Computational Linguistics, 194–203.
- [14] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Terry Michael, Quoc Le, and Charles Sutton. 2021. Program Synthesis with Large Language Models. *CoRR* (2021).
- [15] Manya Bansal, Olivia Hsu, Kunle Olukotun, and Fredrik Kjolstad. 2023. Mosaic: An Interoperable Compiler for Tensor Algebra. *PLDI* (2023).
- [16] Rohan Bavishi, Caroline Lemieux, Roy Fox, Koushik Sen, and Ion Stoica. 2019. AutoPandas: neural-backed generators for program synthesis. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–27.
- [17] Sahil Bhatia, Sumer Kohli, Sanjit A. Seshia, and Alvin Cheung. 2023. Building Code Transpilers for Domain-Specific Languages Using Program Synthesis (Experience Paper). In *ECOOP (LIPIcs, Vol. 263)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 38:1–38:30.
- [18] L Susan Blackford, Antoine Petit, Roldan Pozo, Karin Remington, R Clint Whaley, James Demmel, Jack Dongarra, Iain Duff, Sven Hammarling, Greg Henry, et al. 2002. An updated set of basic linear algebra subprograms (BLAS). *ACM Trans. Math. Software* 28, 2 (2002), 135–151.
- [19] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. TVM: an automated end-to-end optimizing compiler for deep learning. In *Proceedings of the 13th USENIX conference on Operating Systems Design and Implementation*. 579–594.
- [20] Bruce Collie, Philip Ginsbach, and Michael FP O’Boyle. 2019. Type-Directed Program Synthesis and Constraint Generation for Library Portability. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 55–67.
- [21] Bruce Collie and Michael FP O’Boyle. 2021. Program lifting using gray-box behavior. In *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 60–74.
- [22] Bruce Collie, Jackson Woodruff, and Michael FP O’Boyle. 2020. Modeling black-box components with probabilistic synthesis. In *Proceedings of the 19th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*. 1–14.
- [23] Keith D Cooper and Linda Torczon. 2011. *Engineering a compiler*. Elsevier.
- [24] Joao PL De Carvalho, Braedy Kuzma, Ivan Korostelev, José Nelson Amaral, Christopher Barton, José Moreira, and Guido Araujo. 2021. KernelFaRer: replacing native-code idioms with high-performance library calls. *ACM Transactions On Architecture And Code Optimization (TACO)* 18, 3 (2021), 1–22.
- [25] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS (Lecture Notes in Computer Science, Vol. 4963)*. Springer, 337–340.
- [26] Mehdi Drissi, Olivia Watkins, Aditya Khant, Vivaswat Ojha, Pedro San-doval Segura, Rakia Segev, Eric Weiner, and Robert Keller. 2018. Program Language Translation Using a Grammar-Driven Tree-to-Tree Model. *CoRR* abs/1807.01784 (2018).
- [27] Venmugil Elango, Norm Rubin, Mahesh Ravishankar, Hariharan Sandanagobalane, and Vinod Grover. 2018. Diesel: DSL for linear algebra

- and neural net computations on GPUs. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. 42–51.
- [28] Grigory Fedyukovich, Maaz Bin Safeer Ahmad, and Rastislav Bodik. 2017. Gradual synthesis for static parallelization of single-pass array-processing programs. *ACM SIGPLAN Notices* 52, 6 (2017), 572–585.
- [29] Björn Franke and Michael O'Boyle. 2003. Array recovery and high-level transformations for DSP applications. *ACM Transactions on Embedded Computing Systems (TECS)* 2, 2 (2003), 132–162.
- [30] Philip Ginsbach, Toomas Rimmelg, Michel Steuwer, Bruno Bodin, Christophe Dubach, and Michael FP O'Boyle. 2018. Automatic matching of legacy code to heterogeneous APIs: An idiomatic approach. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. 139–153.
- [31] Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. *ACM Sigplan Notices* 46, 1 (2011), 317–330.
- [32] Yining Hong, Kaichun Mo, Li Yi, Leonidas J. Guibas, Antonio Torralba, Joshua B. Tenenbaum, and Chuang Gan. 2022. Fixing Malfunctional Objects With Learned Physical Simulation and Functional Prediction. In *CVPR*. IEEE, 1403–1413.
- [33] Olivia Hsu, Alexander Rucker, Tian Zhao, Kule Olukotun, and Fredrik Kjolstad. 2022. Stardust: Compiling Sparse Tensor Algebra to a Reconfigurable Dataflow Architecture. *CoRR* (2022). Available at <https://arxiv.org/abs/2211.03251>.
- [34] Shoaib Kamil, Alvin Cheung, Shachar Itzhaky, and Armando Solar-Lezama. 2016. Verified lifting of stencil computations. *ACM SIGPLAN Notices* 51, 6 (2016), 711–726.
- [35] Yuning Kang, Zan Wang, Hongyu Zhang, Junjie Chen, and Hanmo You. 2021. APIRecX: Cross-Library API Recommendation via Pre-Trained Language Model. In *EMNLP (1)*. Association for Computational Linguistics, 3425–3436.
- [36] Omer Katz, Yuval Olshaker, Yoav Goldberg, and Eran Yahav. 2019. Towards Neural Decompilation. *CoRR* abs/1905.08325 (2019).
- [37] Fredrik Kjolstad, Stephen Chou, David Lugato, Shoaib Kamil, and Saman Amarasinghe. 2017. taco: A Tool to Generate Tensor Algebra Kernels. *ASE* (2017).
- [38] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The Tensor Algebra Compiler. *OOPSLA* (2017).
- [39] Daniel Kroening and Michael Tautschnig. 2014. CBMC - C Bounded Model Checker - (Competition Contribution). In *TACAS (Lecture Notes in Computer Science, Vol. 8413)*. Springer, 389–391.
- [40] Taku Kudo and John Richardson. 2018. SentencePiece: A simple and language independent subword tokenizer and detokenizer for Neural Text Processing. In *EMNLP (Demonstration)*. Association for Computational Linguistics, 66–71.
- [41] Jeremy Lacomis, Pengcheng Yin, Edward J. Schwartz, Miltiadis Allamanis, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. 2019. DIRE: A Neural Approach to Decompiled Identifier Naming. In *ASE*. IEEE, 628–639.
- [42] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, and Shengyu Fu. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. *CoRR* (2021). Available at <https://arxiv.org/pdf/2102.04664.pdf>.
- [43] Shantanu Mandal, Adhikri Chethan, Vahid Janfaza, S M Farabi Mahmud, Todd A Anderson, Javier Turek, Jesmin Jahan Tihi, and Abdullah Muzahid. 2023. Large Language Models Based Automatic Synthesis of Software Specifications. *CoRR* (2023). Available at <https://arxiv.org/pdf/2304.09181.pdf>.
- [44] Pablo Antonio Martínez, Jackson Woodruff, Jordi Armengol-Estapé, Gregorio Bernabé, José Manuel García, and Michael FP O'Boyle. 2023. Matching linear algebra and tensor code to specialized hardware accelerators. In *Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction*. 85–97.
- [45] MG Sarwar Murshed, Christopher Murphy, Daqing Hou, Nazar Khan, Ganesh Ananthanarayanan, and Faraz Hussain. 2021. Machine learning at the network edge: A survey. *ACM Computing Surveys (CSUR)* 54, 8 (2021), 1–37.
- [46] Daye Nam, Baishakhi Ray, Seohyun Kim, Xianshan Qu, and Satish Chandra. 2022. Predictive synthesis of API-centric code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*. 40–49.
- [47] Michael F. P. O'Boyle and Peter M. W. Knijnenburg. 2002. Integrating Loop and Data Transformations for Global Optimization. *J. Parallel Distributed Comput.* 62, 4 (2002), 563–590.
- [48] OpenAI. [n. d.]. ChatGPT. <https://openai.com/chatgpt>.
- [49] Myle Ott, Sergey Edunov, Alexei Baevski, Angela Fan, Sam Gross, Nathan Ng, David Grangier, and Michael Auli. 2019. fairseq: A fast, extensible toolkit for sequence modeling. *arXiv preprint arXiv:1904.01038* (2019).
- [50] Davide Pizzolotto and Katsuro Inoue. 2021. Identifying Compiler and Optimization Level in Binary Code from Multipler Architectures. *IEEE Access* (2021).
- [51] Elizabeth Polgreen, Andrew Reynolds, and Sanjit A. Seshia. 2022. Satisfiability and Synthesis Modulo Oracles. In *VMCAI (Lecture Notes in Computer Science, Vol. 13182)*. Springer, 263–284.
- [52] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices* 48, 6 (2013), 519–530.
- [53] Joseph Redmon. 2013–2016. Darknet: Open Source Neural Networks in C. <http://pjreddie.com/darknet/>.
- [54] Christopher D Rosin. 2019. Stepping stones to inductive synthesis of low-level looping programs. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 33. 2362–2370.
- [55] Baptiste Roziere, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised Translation of Programming Languages. In *Proceedings of the 34th International Conference on Neural Information Processing Systems (Vancouver, BC, Canada) (NIPS'20)*. Curran Associates Inc., Red Hook, NY, USA, Article 1730, 11 pages.
- [56] Mazen AR Saghir. 1998. *Application-specific instruction-set architectures for embedded DSP applications*. Citeseer.
- [57] Eddie Antonio Santos, Joshua Charles Campbell, Dhvani Patel, Abram Hindle, and José Nelson Amaral. 2018. Syntax and sensibility: Using language models to detect and correct syntax errors. In *SANER*. IEEE Computer Society, 311–322.
- [58] Ryan Senanayake, Changwan Hong, Siheng Wang, Wilson Amalee, Stephen Chou, Shoaib Kamil, Saman Amarasinghe, and Fredrik Kjolstad. 2020. A Sparse Iteration Space Transformation Framework for Sparse Tensor Algebra. *OOPSLA* (2020).
- [59] Kensen Shi, David Bieber, and Rishabh Singh. 2022. TF-Coder: Program synthesis for tensor manipulations. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 44, 2 (2022), 1–36.
- [60] Nicholas D Sidiropoulos, Lieven De Lathauwer, Xiao Fu, Kejun Huang, Evangelos E Papalexakis, and Christos Faloutsos. 2017. Tensor decomposition for signal processing and machine learning. *IEEE Transactions on signal processing* 65, 13 (2017), 3551–3582.
- [61] Rohit Singh, Rishabh Singh, Zhilei Xu, Rebecca Krosnick, and Armando Solar-Lezama. 2014. Modular synthesis of sketches using models. In *Verification, Model Checking, and Abstract Interpretation: 15th International Conference, VMCAI 2014, San Diego, CA, USA, January 19-21, 2014, Proceedings 15*. Springer, 395–414.

- [62] Sunbeom So and Hakjoo Oh. 2017. Synthesizing imperative programs from examples guided by static analysis. In *International Static Analysis Symposium*. Springer, 364–381.
- [63] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. 2006. Combinatorial sketching for finite programs. In *ASPLOS*. ACM, 404–415.
- [64] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. 2014. Sequence to Sequence Learning with Neural Networks. In *NIPS*. 3104–3112.
- [65] Marc Szafraniec, Baptiste Roziere, Hugh Leather Francois Charton, Patrick Labatut, and Gabriel Synnaeve. 2022. Code translation with compiler representations. *arXiv preprint arXiv:2207.03578* (2022).
- [66] Abhishek Udupa, Arun Raghavan, Jyotirmoy V. Deshmukh, Sela Mador-Haim, Milo M. K. Martin, and Rajeev Alur. 2013. TRANSIT: specifying protocols with concolic snippets. In *PLDI*. ACM, 287–296.
- [67] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *arXiv preprint arXiv:1802.04730* (2018).
- [68] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *NIPS*. 5998–6008.
- [69] Jackson Woodruff, Jordi Armengol-Estapé, Sam Ainsworth, and Michael F. P. O’Boyle. 2022. Bind the gap: compiling real software to hardware FFT accelerators. In *PLDI*. ACM, 687–702.
- [70] Rohan Yadav, Alex Aiken, and Fredrik Kjolstad. 2022. DISTAL: The Distributed Tensor Algebra Compiler. *PLDI* (2022).
- [71] Rohan Yadav, Alex Aiken, and Fredrik Kjolstad. 2022. SpDISTAL: Compiling Distributed Sparse Tensor Computations. *International Conference for High Performance Computing, Networking, Storage and Analysis* (2022).
- [72] Vojin Zivojinovic. 1994. DSPstone: A DSP-oriented benchmarking methodology. *Proc. Signal Processing Applications & Technology, Dallas, TX, 1994* (1994), 715–720.
- [73] Amit Zohar and Lior Wolf. 2018. Automatic program synthesis of long programs with a learned garbage collector. *Advances in neural information processing systems* 31 (2018).

Received 2023-07-14; accepted 2023-09-03

Partial Evaluation of Automatic Differentiation for Differential-Algebraic Equations Solvers

Oscar Eriksson

oerikss@kth.se

KTH Royal Institute of Technology
Sweden

Viktor Palmkvist

vipa@kth.se

KTH Royal Institute of Technology
Sweden

David Broman

dbro@kth.se

broman@stanford.edu
KTH Royal Institute of Technology
Sweden
Stanford University
USA

Abstract

Differential-Algebraic Equations (DAEs) are the foundation of high-level equation-based languages for modeling physical dynamical systems. Simulating models in such languages requires a transformation known as index reduction that involves differentiating individual equations before numerical integration. Commercial and open-source implementations typically perform index reduction by symbolic differentiation (SD) and produce a Jacobian callback function with forward-mode automatic differentiation (AD). The former results in efficient runtime code, and the latter is asymptotically efficient in both runtime and code size. However, AD introduces runtime overhead caused by a non-standard representation of real numbers, and SD is not always applicable in models with general recursion. This work proposes a new approach that uses partial evaluation of AD in the context of numerical DAE solving to combine the strengths of the two differentiation methods while mitigating their weaknesses. Moreover, our approach selectively specializes partial derivatives of the Jacobian by exploiting structural knowledge while respecting a user-defined bound on the code size. Our evaluation shows that the new method both enables expressive modeling from AD and retains the efficiency of SD for many practical applications.

CCS Concepts: • **Computing methodologies** → **Symbolic and algebraic manipulation**; *Representation of mathematical objects*; **Simulation languages**; **Simulation tools**; • **Mathematics of computing** → *Differential calculus*; **Solvers**; • **Software and its engineering** → **Compilers**; **Domain specific languages**.

Keywords: Automatic Differentiation, Differential-Algebraic Equations, Partial Evaluation, Jacobian Generation, Compiler

ACM Reference Format:

Oscar Eriksson, Viktor Palmkvist, and David Broman. 2023. Partial Evaluation of Automatic Differentiation for Differential-Algebraic Equations Solvers. In *Proceedings of the 22nd ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '23)*, October 22–23, 2023, Cascais, Portugal. ACM, New York, NY, USA, 15 pages. <https://doi.org/10.1145/3624007.3624054>

1 Introduction

Equation-based Object-Oriented (EOO) modeling [15, 42] is a well-established paradigm where dynamical physical systems—such as aircraft, power plants, and vehicles—can be rapidly modeled using reusable components. In such languages and tools (e.g., Modelica [42], Simscape [40], and VHDL-AMS [6]), the fundamental underlying semantics of the models are described using Differential-Algebraic Equations (DAEs) [38]. A common usage of such models is numerical simulation, which involves a complex combination of compilation and run-time tasks, including static analysis, symbolic computations, and numerical approximation.

A key challenge when developing EOO language tool-chains is achieving high-performance simulations. Specifically, this paper focuses on two parts of the simulation process that are performance critical, both relating to *differentiation*: (i) during *index-reduction* when a subset of the equations needs to be differentiated to obtain a non-singular system, and (ii) during *evaluation of the Jacobian* of the DAE, which is performed in tandem with a numerical approximation library.

In many EOO tools [20, 26], index reduction is commonly done using *symbolic differentiation* (SD). In contrast, Jacobian differentiation is typically performed using *automatic differentiation* (AD) [9]. Symbolic differentiation and automatic differentiation have, however, their pros and cons. Symbolic differentiation admits highly efficient compiled code but can lead to so-called *expression swell* with re-computations. Moreover, symbolic differentiation cannot easily be used when differentiating arbitrary functions that contain loops, recursion, and if-conditions. In contrast, AD works on arbitrary functions and programs, does not result in an expression



This work is licensed under a Creative Commons Attribution 4.0 International License.

GPCE '23, October 22–23, 2023, Cascais, Portugal

© 2023 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-0406-2/23/10.

<https://doi.org/10.1145/3624007.3624054>

swell, but introduces run-time overhead when real numbers are lifted into so-called dual-number representation. A natural question arises: is it possible to find a middle ground between SD and AD that keeps their strengths while mitigating their weaknesses?

State-of-the-art solutions for DAE solving either focus on pure AD for Jacobians [27] or index reduction [30, 45], or alternatively on pure SD for Jacobians [13] or index reduction [26]. There is, however, no existing unifying approach on how to *combine* the strengths of AD and SD to solve DAEs, for *both* Jacobians and index reduction.

This paper proposes a new approach for combining AD and SD by applying *partial evaluation* (PE) [35] to AD, resulting in a residual program that may include both AD and SD terms. Moreover, a key property is that we can *parametrize* the specialization, thus enabling a tradeoff between code size (which increases when the result is closer to SD) and performance (which decreases with the overhead operations in AD). In the general context of AD, surprisingly, only a few attempts [23, 32, 55, 61] have been made to combine AD and partial evaluation, and no previous work has done it in the setting of index reduction and Jacobian calculation.

We call our approach *PEAD*, which combines PE and AD in the context of solving DAEs. Specifically, we make the following key contributions:

- We propose a new program transformation method, called *forward-mode PEAD for DAEs*, that takes a model encoding a high-index DAE as input and outputs a residual function suitable for off-the-shelf numerical DAE solvers. The combination of partial evaluation and forward-mode AD allows models that include general recursion combined with efficient code (Section 3.2).
- We develop a novel approach to forward-mode PEAD for generating system Jacobians for DAEs that allows the end-user to control the tradeoff between code specialization and code size (Section 3.3).
- Besides developing a new theory, we develop an efficient implementation of the proposed approach in the Miking framework [16], which is available as open source¹ (Section 4). We evaluate the implementation on a number of non-trivial equation-based object-oriented models (Section 5).

2 Background and Challenges

This section explains the ideas of DAEs, DAE index, and code generation for numerical DAE solving. Specifically, we discuss key challenges related to differentiation of programming languages containing DAEs as part of the language semantics.

2.1 Differential-Algebraic Equations (DAEs)

DAEs consist of differential equations coupled with algebraic constraints. Figure 1 depicts the classic *hello-world* example for DAEs, a planar pendulum with a DAE model in Cartesian coordinates. Given the DAE in Figure 1b, suppose we have some initial values $x(t_0)$, $y(t_0)$, and $\tau(t_0)$, together with an interval $I = [t_0, t_f]$. A solution (a simulation trace) is then a set of sufficiently smooth functions $x(t)$, $y(t)$, and $\tau(t)$, such that the DAE holds for all $t \in I$.

However, in contrast to Ordinary Differential Equations (ODEs), we cannot always directly solve a DAE. For instance, in the DAE in Figure 1b, the Jacobian of the left-hand sides is singular w.r.t. the dependent variables at the highest differentiation order, \ddot{x} , \ddot{y} , and τ . Note how the third equation does not contain any dependent variable at the highest differentiation order and cannot directly be used to solve the system of equations. Therefore, in general, we need to differentiate some equations of the DAE until the Jacobian w.r.t. the dependent variables at the highest differentiation order is no longer singular.

This transformation by differentiation is called *index reduction*. The DAE's *differential index* (or just index) is the largest number of times we need to differentiate an individual equation to transform the DAE into an implicit ODE [14]. The DAE in Figure 1b has index 3, and Figure 1c shows an index-reduced version, where the third equation is differentiated twice, thus reducing the DAE to index 1. Note the third equation in Figure 1c, which contains \ddot{x} and \ddot{y} , the highest differential orders for x and y , respectively.

Several efficient methods analyze the structure of the DAEs and output which equations need to be differentiated and to what derivative order [49, 53]. These algorithms succeed in most practical applications, and off-the-shelf numerical DAE solvers typically handle DAEs of index 1 or less [33, 51].

It should be noted that simply substituting equations for their differentiated versions can lead to problems with numerical drifting, as the original algebraic constraints are rendered implicit. Several methods exist to solve this problem [5, 41, 47], but we will not consider these algorithms in this work because the drifting problem is orthogonal to the problems discussed in this paper.

2.2 Generating a Low-Index Residual Function

Compilers for DAE-based modeling languages perform index reduction as a part of compiling the simulation code. The index reduction is performed on the so-called *residual function*, a functional representation of the DAE. In languages such as Modelica², compilers typically perform this index reduction on a flattened representation of the model using symbolic differentiation. Such symbolic differentiation can produce

¹<https://github.com/miking-lang/miking-dae>

²<https://modelica.org/>

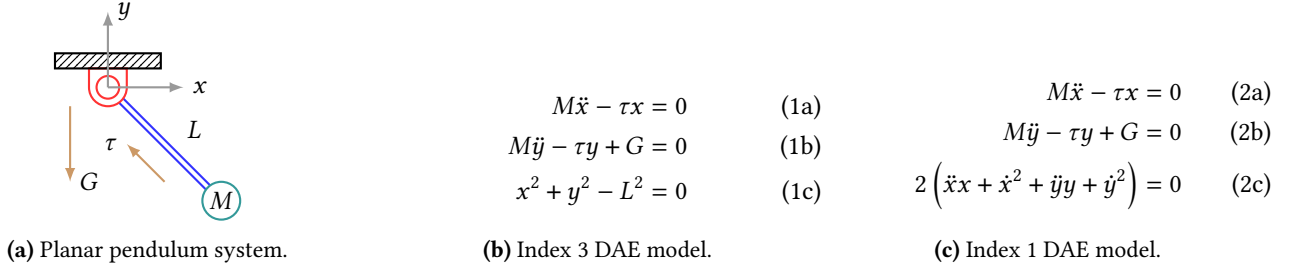


Figure 1. Figure (a) depicts the 2D pendulum example, (b) its corresponding DAE, and (c) the DAE after index reduction. We use lower-case letters for variables (x and y for position and τ for pendulum arm tension per unit length) and upper case for constants (M for mass, G for the acceleration of gravity, and L for pendulum length). The variables depend on the implicit *independent variable* t ; we call these variables *dependent variables* to disambiguate them from other variables. We use dot notation to denote differentiation w.r.t. the independent variable, e.g., $\dot{x} = \frac{dx}{dt}$, $\ddot{x} = \frac{d^2x}{dt^2}$. Equation (2c) is (1c) twice differentiated w.r.t. the independent variable t .

efficient run-time computations of derivatives, given that the expression swell problem is solved [61].

Consider Figure 1b again and suppose that we replace the third equation with

$$x^2 + y^2 - f(t, x, y, \tau), \quad (3)$$

where f is some arbitrary but sufficiently smooth scalar function that may be implemented using recursion and other control-flow language constructs. If only symbolic differentiation is used, function f cannot be an arbitrary program function containing recursion, loops, and if conditions. When equations are extended with general program capabilities, symbolic differentiation leads to an expressiveness problem, where not all functions can be differentiated.

AD is an attractive solution to this expressiveness problem, as it naturally extends to general program constructs, including recursion. However, transforming a program to compute derivatives using forward-mode AD involves transforming elementary functions to operate on dual numbers, usually encoded as pairs, instead of real numbers.

Even though computing derivatives via AD is asymptotically efficient, it introduces a constant run-time overhead compared to a symbolic derivative, as we cannot directly compile elementary functions to cheap floating-point operations. This overhead becomes even more significant when we consider higher-order derivatives. Additionally, AD does not easily allow us to simplify expressions like $0 * e = e * 0 = 0$ as the derivative computation is performed solely at run-time.

In summary, the main challenge is to enable both *efficient* and *expressive* index reduction, thus including the strengths of both AD and SD, and avoiding their weaknesses.

2.3 Generating a System Jacobian

Numerical DAE solvers that use Newton-Raphson iterations need to compute the Jacobian of the residual w.r.t. the dependent variables and their derivatives.

Forward-mode AD allows us to compute one column in the Jacobian matrix in one function evaluation, and the code size is also proportional to the residual function, which makes it the preferred method in DAE-based modeling languages. However, AD in this context suffers from the same drawbacks as discussed when generating the low-index residual function. Moreover, the Jacobian is potentially a very large matrix (the number of equations times the number of unknowns), which may result in a large code size if each partial derivative is computed by symbolic differentiation.

Hence, a key challenge concerns handling the tradeoff between runtime performance (focusing on specialized code using variants of symbolic differentiation) and code size (which does not grow with the matrix size for automatic differentiation).

3 Approach

This section presents our proposed compilation approach for generating efficient residual functions and Jacobian callback functions that are suitable for off-the-shelf DAE solvers. Section 3.1 gives an overview and presents our source language and a running example. Section 3.2 presents our proposed transformation that takes a high-index DAE program as input and outputs a low-index, first-order residual. Finally, Section 3.3 presents our approach for generating a partially specialized callback function that computes the Jacobian of the residual function.

3.1 Overview

Figure 2 gives an overview of our code generation approach. The input p is a (possibly high-index) DAE program. The outputs are a low-index residual function f and a corresponding Jacobian function J . In the figure, *ad* and *peval* denote AD source-code transformations and partial evaluation, respectively. *SA* is a structural analysis that indicates which equations to differentiate and to what order to reduce the

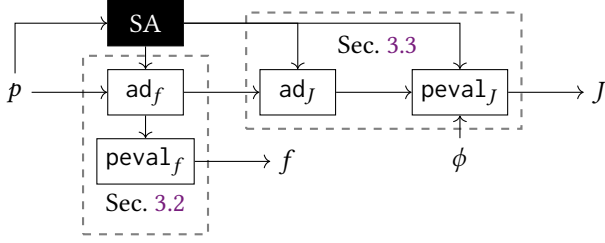


Figure 2. Overview of our code generation approach. The structural analysis in the black box is not part of our contribution. White boxes denote transformations, and the dashed boxes points to the relevant sections in the paper.

DAE index. We use the information produced by this analysis for three purposes: 1) to index reduce the DAE (ad_f in the figure), 2) to avoid unnecessary computations of zero entries in the Jacobian (ad_J), and 3) to select which partial derivatives to specialize (peval_J). The latter is also guided by the user-supplied parameter ϕ . As previously mentioned, we denote the combination of ad and peval as PEAD; Figure 2 thus contains two slightly different instances of PEAD.

We describe a DAE modeling language, encoding high-index DAEs, with syntax as shown in Figure 3a. The construct $x^{(n)}$ with $n \in \mathbb{N}$ is a dependent variable of the DAE, annotated with the differential order n w.r.t. the independent variable. We assume that these dependent variables appear free in p and only allow them to occur in the residual equations (e_0, \dots, e_n) . As syntactic sugar, we use dot notation up to differentiation order 3, e.g., $x^{(1)} = \dot{x}$, $x^{(2)} = \ddot{x}$, $x^{(3)} = \ddot{\ddot{x}}$. We use $x^{(0)} = x$ when it is clear from the context if x is a dependent variable of the DAE.

The rest of an expression e is the standard λ -calculus with some well-explored extensions: tuples, projections ($e.n$ projects the n 'th element from e starting from zero), if-then-else, let, and recursive let³. We represent all constants by c , including mathematical functions such as addition and scalar multiplication. Note that subscripts for e do not denote tuple-projections in the language. However, we will sometimes use indexing by subscripts in other contexts.

A DAE program $p \in \text{Prog}$ consists of top-level lets, top-level recursive bindings⁴, and a set of equations. Note that the set of equations is represented as an n -tuple of residual expressions.

We do not define a formal type-system for p but assume that all dependent variables and residuals are scalars and that the type-system does not allow expressing recursion apart from using the **letrec** construct. However, we will sometimes

use types informally for documentation purposes. Note also that the implementation (discussed in the next section) is written in a typed functional language.

To improve readability, we sometimes write applications of constant functions using infix notation and use tuple de-structuring, e.g., $e_1 + e_2$ and $\lambda(x_0, x_1, \dots, x_n).e$, respectively.

As our running example, we use the input program listed in Figure 3b, which encodes the DAE in Figure 1b, modified to illustrate relevant aspects of our approach. We have added a recursive power function, an equation that computes the kinetic energy of the pendulum, and a run-time parameter θ . For brevity, we also set all physical constants to 1.

3.2 Residual Generation

This sub-section describes how our approach takes a DAE program, p , and generates a residual function

$$f : \mathbb{R}^m \times \mathbb{R}^m \rightarrow \mathbb{R}^m, \quad (4)$$

where $f(y, \dot{y})$ computes the residual of all n equations in a first-order, low-index DAE. We can supply this residual function directly to an index-1 DAE solver, such as the IDA SUNDIALS solver [33].

The transformation is defined in Figure 4a. This transformation uses the output of the structural analysis (SA in Figure 2) in the form of a tuple $\mathbf{d} \in \mathbb{N}^m$, where \mathbf{d}_i is the differentiation order of the i 'th residual expression. We assume that \mathbf{d} is available globally and we use it in the \mathcal{T} transform.

Intuitively, the \mathcal{T} transform produces a sequence of let and relet-bindings where the final expression is a tuple of expressions differentiated according to \mathbf{d} . These differentiated expressions may contain free variables, references to the initial sequence of bindings, thus \mathcal{T} introduces new versions of these, differentiated to different orders as needed. The renaming needed to reference the appropriate version of each binding is handled by the id function, which we introduce shortly.

The \mathcal{T} transformation internally applies the AD source-code transformation \mathcal{D}_n for n 'th derivatives, defined in Figure 4b. In particular, it allows us to straight-forwardly index reduce the DAE while still allowing models with general recursion and other general language constructs.

The AD transform is mostly the standard Taylor-mode AD transform for higher-order derivatives [29, 59]⁵. I.e., it transforms an expression $e : \mathbb{R}$ to an expression $e' : \mathbb{R}^n$, where the n 'th element is the n 'th derivative w.r.t. t , starting from zero. We can recover a particular derivative by projecting the relevant element from e' . We extend this transformation with dependent variables $x^{(n)}$ and free variables.

We define the identifier renaming function we use to associate variables with the correct differentiated expression as

$$\text{id} : \text{Ident} \times \mathbb{N} \rightarrow \text{Ident}, \quad (5)$$

⁵We use this transform instead of nesting dual numbers for the increased runtime performance due to sharing and fewer indirections.

³We include recursive let-bindings and if-then-else because we need to consider recursion explicitly in the partial evaluation to ensure termination rather than encoding it in the core λ -calculus. Moreover, the partial evaluator introduces new let-bindings to enable sharing so it is convenient to also have these in the expression language.

⁴These allows us share code between residual expressions.

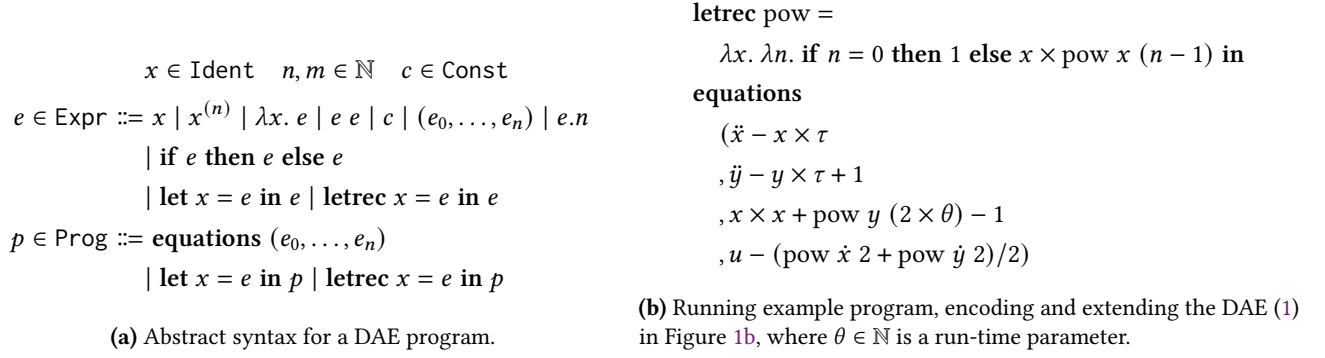


Figure 3. Syntax and running example.

$$\begin{aligned} \mathcal{T}[\text{equations } (e_0, \dots, e_n)] &= ((\mathcal{D}_{d_0}[e_0]).\mathbf{d}_0, \dots, (\mathcal{D}_{d_n}[e_n]).\mathbf{d}_n) \\ \mathcal{T}[\text{let } x = e \text{ in } p] &= \text{let } x = e \text{ in let id}(x, 1) = \mathcal{D}_1[e] \text{ in } \dots \text{let id}(x, N) = \mathcal{D}_N[e] \text{ in } \mathcal{T}[p] \\ \mathcal{T}[\text{letrec } x = e \text{ in } p] &= \text{letrec } x = e \text{ in letrec id}(x, 1) = \mathcal{D}_1[e] \text{ in } \dots \text{letrec id}(x, N) = \mathcal{D}_N[e] \text{ in } \mathcal{T}[p] \end{aligned}$$

(a) The index-reduction transformation, $\mathcal{T} : \text{Prog} \rightarrow \text{Expr}$, for DAE programs p with m equations, where $\mathbf{d} \in \mathbb{N}^m$ denotes the differentiation for each residual expression according to the structural analysis (zero-indexed) and $N = \max(\mathbf{d})$.

$$\begin{aligned} \mathcal{D}_n[x] &= \text{id}(x, n) & \mathcal{D}_n[x^{(m)}] &= (x^{(m)}, x^{(m+1)}, \dots, x^{(m+n)}) & \mathcal{D}_n[\lambda x. e] &= \lambda \text{id}(x, n). \mathcal{D}_n[e] \\ \mathcal{D}_n[e \ e'] &= \mathcal{D}_n[e] \ \mathcal{D}_n[e'] & \mathcal{D}_n[(e_0, \dots, e_m)] &= (\mathcal{D}_n[e_0], \dots, \mathcal{D}_n[e_m]) & \mathcal{D}_n[e.m] &= \mathcal{D}_n[e].m \\ \mathcal{D}_n[\text{if } e \text{ then } e' \text{ else } e''] &= \text{if } \mathcal{D}_n[e] \text{ then } \mathcal{D}_n[e'] \text{ else } \mathcal{D}_n[e''] \\ \mathcal{D}_n[\text{let } x = e \text{ in } e'] &= \text{let id}(x, n) = \mathcal{D}_n[e] \text{ in } \mathcal{D}_n[e'] & \mathcal{D}_n[\text{letrec } x = e \text{ in } e'] &= \text{letrec id}(x, n) = \mathcal{D}_n[e] \text{ in } \mathcal{D}_n[e'] \\ \mathcal{D}_n[r] &= (r, n \text{ zeroes}) \text{ when } r \in \mathbb{R} & \mathcal{D}_2[+] &= \lambda(x_0, x_1, x_2). \lambda(y_0, y_1, y_2). (x_0 + y_0, x_1 + y_1, x_2 + y_2) \\ \mathcal{D}_2[-] &= \lambda(x_0, x_1, x_2). \lambda(y_0, y_1, y_2). (x_0 - y_0, x_1 - y_1, x_2 - y_2) \\ \mathcal{D}_2[\times] &= \lambda(x_0, x_1, x_2). \lambda(y_0, y_1, y_2). (x_0 \times y_0, x_1 \times y_0 + x_0 \times y_1, x_2 \times y_0 + 2 \times x_1 \times y_1 + x_0 \times y_2) \end{aligned}$$

(b) The AD-transform, $\mathcal{D}_n : \text{Expr} \rightarrow \text{Expr}$, for expressions e and selected constants c . The function id is defined in (5).

Figure 4. Residual transformation rules.

which returns a unique identifier for each identifier-integer pair. There is no point in renaming variables associated with undifferentiated expressions, so we define $\text{id}(x, 0) = x$.

As an example, consider the program **let** $f = \lambda x. x + x$ **in equations** $(f \ \dot{x}, f \ \dot{y})$ with $\mathbf{d} = (1, 0)$, where \mathbf{d} says that we need to differentiate the first equation once. Applying \mathcal{T} then gives us

$$\begin{aligned} \text{let } f &= \lambda x. x + x \text{ in} \\ \text{let } f' &= \lambda(x'_0, x'_1). (x'_0 + x'_0, x'_1 + x'_1) \text{ in} & (6) \\ ((f' \ (\dot{x}, \ddot{x})).1, f \ \dot{y}), \end{aligned}$$

where the differentiated expression now refers to f' rather than f .

In Figure 4b, we have only included rules for constants c that we use in our discussion and refer to, e.g., Griewank et al. [29] for transformations of additional elementary functions.

The \mathcal{T} transform with $\mathbf{d} = (0, 0, 2, 0)$ produces a low-index residual function of our running example program in

Figure 3b, even though it contains calls to recursive functions with values known only at runtime. It transforms as

$$\begin{aligned} \text{letrec pow} &= \lambda x. \lambda n. \\ &\text{if } n = 0 \text{ then } 1 \text{ else } x \times \text{pow } x \ (n - 1) \text{ in} \\ \text{letrec pow}'' &= \lambda x''. \lambda n''. \text{if } n'' = 0 \text{ then } 1 \text{ else} \\ &\mathcal{D}_2[\times] \ x'' \ (\text{pow}'' \ x'' \ (n'' - 1)) \text{ in} \\ &(\ddot{x} - x \times \tau \\ &, \ddot{y} - y \times \tau + 1 \\ &, (\mathcal{D}_2[-] \\ &\quad (\mathcal{D}_2[+] \\ &\quad \quad (\mathcal{D}_2[\times] \ (x, \dot{x}, \ddot{x}) \ (x, \dot{x}, \ddot{x})) \\ &\quad \quad (\text{pow}'' \ (y, \dot{y}, \ddot{y}) \ (2 \times \theta))) \\ &\quad (1, 0, 0)).2 \\ &, u - (\text{pow } \dot{x} \ 2 + \text{pow } \dot{y} \ 2) / 2), \end{aligned} \tag{7}$$

where we use $(\mathcal{D}_0[e]).0 = e$, removed dead code, and refrained from expanding the right-hand side of some invocations of \mathcal{D}_n for the sake of brevity. In (7), for a residual expression e , $(\mathcal{D}_n[e]).n$ is its n 'th derivative w.r.t. t , computed by AD at runtime. Additionally, \mathcal{T} produces another definition pow'' of pow , propagating the first and second derivatives because it is used in an expression with derivative order 2.

With some abuse of notation, we can formulate a low-index first-order residual function (4) from (7) as follows. The residual (7) is of the form

$$b \text{ in } (e_0, \dots, e_n), \quad (8)$$

where $b \text{ in}$ collects all top-level lets and recursive lets. We can then form the residual function as

$$\begin{aligned} &\lambda(x, \dot{x}, y, \dot{y}, \tau, u). \lambda(\ddot{x}, \ddot{x}, \dot{y}', \ddot{y}, \cdot, \cdot). \\ &b \text{ in } (e_0, \dots, e_n, \dot{x} - \ddot{x}', \dot{y} - \ddot{y}'), \end{aligned} \quad (9)$$

where \cdot denotes a variable that does not occur in the residual body, the last two residual expressions reduce the DAE order. Figure 5 depicts a fully expanded example (9).

However, (9) exhibits the constant time overhead introduced by AD as elementary functions now operate over triples of reals rather than real numbers. We can improve upon this and, in many cases, recover a more efficient symbolic derivative by partially evaluating the program.

We have a partial evaluator, *peval*. Although *peval* itself is not part of our contribution, we discuss some key features relevant to our code transformation. *peval* takes an expression e and produces specialized code in A-normal form (ANF) that preserves sharing. *peval* is based on the ANF transform [25] but also inspired by techniques for improving the specialization of dynamic let-bindings in offline partial evaluation [11] and the evaluation-based technique of Grégoire and Leroy [31]. Moreover, *peval* aggressively specializes code down to elementary operations. In fact, given a closed residual expression e , *peval* $[(\mathcal{D}_n[e]).n]$ recovers its n 'th symbolic derivative, when we unroll all recursions. Sharing computations stemming from the chain rule is important to avoid expression swell. The sharing preserving property of ANF in the context of differentiation was also observed by Wang et al. [61]. Moreover, *peval* has a user-specified max recursion depth to guarantee termination and avoid expression swell.

The partial evaluator treats dependent variables as free variables, and it allows us to produce code that is a mixture of AD and SD, which we can see by partially evaluating the index reduced residual (7) with a max recursion depth of ≥ 3 .

The result is the following:

$$\begin{aligned} &\text{letrec } \text{pow}'' = \lambda x''. \lambda n''. \\ &\quad \text{if } n'' = 0 \text{ then } 1 \text{ else} \\ &\quad \quad \text{let } t = \text{pow}'' \ x'' \ (n'' - 1) \text{ in} \\ &\quad \quad (x'' . 0 \times t.0 \\ &\quad \quad , x'' . 0 \times t.1 + x'' . 1 \times t.0 \\ &\quad \quad , x'' . 0 \times t.2 + 2 \times x'' . 1 \times t.1 + x'' . 2 \times t.0) \quad (10) \\ &\text{in} \\ &(\ddot{x} - x \times \tau \\ &, \ddot{y} - y \times \tau + 1 \\ &, 2 \times (x \times \ddot{x} + \dot{x} \times \dot{x}) + e.2 \\ &, u - (\dot{x} \times \dot{x} + \dot{y} \times \dot{y})/2), \end{aligned}$$

after standard constant folding and simplification passes, where, in the third residual,

$$e = \text{let } t = \text{pow}'' \ (y, \dot{y}, \ddot{y}) \ (2 \times \theta - 1) \text{ in} \\ (y \times t.0, y \times t.1 + \dot{y} \times t.0, y \times t.2 + 2 \times \dot{y} \times t.1 + \ddot{y} \times t.0).$$

There are three additional points of interest in the third residual expression:

- The expression is simplified because it no longer propagates the values at derivative order 0 and 1, which are unnecessary.
- We have statically computed the terms containing x and its derivatives, while the corresponding terms for y are computed at runtime because θ is a runtime value.
- We do not partially evaluate recursion if it appears inside a branch whose condition is not statically known.

In contrast, the recursive call is completely unrolled for the final residual expression.

If we instead assume a static $\theta = 1$ and a max recursion depth ≥ 3 , we recover the fully symbolic residual

$$\begin{aligned} &(\ddot{x} - x \times \tau \\ &, \ddot{y} - y \times \tau + 1 \\ &, 2 \times (x \times \ddot{x} + \dot{x} \times \dot{x} + \dot{y} \times \dot{y} + y \times \ddot{y}) \\ &, u - (\dot{x} \times \dot{x} + \dot{y} \times \dot{y})/2). \end{aligned} \quad (11)$$

The key benefit of the PEAD approach is that it can produce a mixture of symbolic and AD derivatives, where both the symbolic derivatives and AD benefit from sharing.

3.3 Jacobian Generation

In this section, we discuss generating code for computing the Jacobian

$$J_y = \begin{pmatrix} \frac{\partial f_0(y, \dot{y})}{\partial y_0} & \cdots & \frac{\partial f_0(y, \dot{y})}{\partial y_{m-1}} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_{m-1}(y, \dot{y})}{\partial y_0} & \cdots & \frac{\partial f_{m-1}(y, \dot{y})}{\partial y_{m-1}} \end{pmatrix}, \quad (12)$$

of size m , of the residual function (4). Note that each row corresponds to an equation, and each column corresponds


```

λ(x, ẋ, y, ẏ, τ, u). λ(ẋ', ẍ, ẏ', ÿ, ·, ·).
  letrec pow = λx. λn.
    if n = 0 then 1 else x × pow x (n - 1) in
  letrec pow' = λx'. λn'. if n' = 0 then 1 else
    (λ(x0, x1). λ(y0, y1). (x0 × y0, x1 × y0 + x0 × y1))
    x' (pow' x' (n' - 1))
  in
  letrec pow'' = λx''. λn''. if n'' = 0 then 1 else
    (λ(x0, x1, x2). λ(y0, y1, y2).
      (x0 × y0
       , x1 × y0 + x0 × y1
       , x2 × y0 + 2 × x1 × y1 + x0 × y2))
      x'' (pow'' x'' (n'' - 1))
  in
  ((ẍ - x × τ, ).0
   , (ÿ - y × τ + 1, ).0
   , ((λ(x0, x1, x2). λ(y0, y1, y2). (x0 - y0, x1 - y1, x2 - y2))
     ((λ(x0, x1, x2). λ(y0, y1, y2). (x0 + y0, x1 + y1, x2 + y2))
      ((λ(x0, x1, x2). λ(y0, y1, y2).
        (x0 × y0
         , x1 × y0 + x0 × y1
         , x2 × y0 + 2 × x1 × y1 + x0 × y2))
        (x, ẋ, ẍ) (x, ẋ, ẍ))
      (pow'' (y, ẏ, ÿ) (2 × θ)))
     (1, 0, 0)).2
   , (u - (pow ẋ 2 + pow ẏ 2)/2, ).0
   , ẋ - ẋ'
   , ẏ - ẏ'),

```

Figure 5. A fully expanded example (9) without simplifications and dead code removal. Expressions (e, \cdot) denotes singleton tuples.

to a variable. The DASSL family of Backward Differentiation Formula-based DAE solvers [33, 51] typically expects a callback function to compute the Jacobian of the DAE during the Newton-Raphson integration routine. More precisely, this function should compute

$$J_y + cJ_{\dot{y}}, \quad (13)$$

where $c \in \mathbb{R}$ is a solver-supplied scaling factor. Since generating code for computing $J_{\dot{y}}$ is analogous to J_y , we limit our presentation to J_y for the sake of brevity.

The standard AD approach computes one column in (12), with one evaluation of f transformed to dual numbers. However, this means we also compute zero entries in the Jacobian at runtime.

Fortunately, we can use structural analysis (SA in Figure 2) to do better. For an entry in the Jacobian to be non-zero, the corresponding variable must appear in the corresponding equation; thus, we can trivially produce the row and column pairs of potentially non-zero entries.

For our running example (Equation (9)), we want to compute the Jacobian

$$\begin{pmatrix} \frac{\partial e_0}{\partial x} & \frac{\partial e_0}{\partial \dot{x}} & \frac{\partial e_0}{\partial y} & \frac{\partial e_0}{\partial \dot{y}} & \frac{\partial e_0}{\partial \tau} & \frac{\partial e_0}{\partial u} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \frac{\partial e_5}{\partial x} & \frac{\partial e_5}{\partial \dot{x}} & \frac{\partial e_5}{\partial y} & \frac{\partial e_5}{\partial \dot{y}} & \frac{\partial e_5}{\partial \tau} & \frac{\partial e_5}{\partial u} \end{pmatrix}. \quad (14)$$

We compute the Jacobian column-by-column; thus, we group the non-zero coordinates by column, resulting in the following:

$$\text{idx} = ((0, (0, 2, 4)), (1, (2, 3)), (2, (1, 2, 5)), (3, (2, 3)), (4, (0, 1)), (5, (3))). \quad (15)$$

For example, $(0, (0, 2, 4))$ is in idx because variable 0 (i.e., x) appears in equations 0, 2, and 4 but not in equations 1, 3, and 5. The reader can verify the structure by considering the symbolic residual (11) and the last two residual expressions in (9).

We use this structural information to construct a function that only computes the non-zero elements of (12). Again with a slight abuse of notation, the order-reduced low-index residual function from \mathcal{T} will have the form

$$\lambda y. \lambda \dot{y}. b \text{ in } (e_0, \dots, e_{m-1}), \quad (16)$$

where b in collects top-level bindings (compare with example (9)). We then form the Jacobian callback function as

$$\text{jac} = \lambda y. \lambda \dot{y}.$$

$$\begin{aligned} &\text{let } \dot{y}' = \mathbf{0} \text{ in} \\ &\text{let } fs = \mathcal{D}'_1[b \text{ in } (\lambda y'. e_0, \dots, \lambda y'. e_{m-1})] \\ &\text{in map } (\lambda(j, is). \text{map } (\lambda i. fs.i \delta_j) is), \end{aligned} \quad (17)$$

where $\text{map } f \ e$ maps the function f over the elements in the tuple e , $\mathbf{0}$ is a tuple of zeros of sufficient size, and δ_j produces a tuple of sufficient size with zeros everywhere except at position j . In (17), fs is a tuple of functions where \mathcal{D}'_1 will transform the expressions e_0 to e_{m-1} to not only reference y and \dot{y} , but also their associated tangent tuples $y', \dot{y}' \in \mathbb{R}^m$.

The AD transform \mathcal{D}'_1 is the same as \mathcal{D}_1 in Figure 4b save for two modifications. Firstly,

$$\text{id}(x, n) = x$$

because we do not need to rename free variables to keep track of different derivative orders. Secondly, we special-case projections from the parameter y to use the associated

tangent tuple \mathbf{y}' :

$$\mathcal{D}'_1[\mathbf{y}.j] := (\mathbf{y}.j, \mathbf{y}'.j)$$

The change is analog for $\dot{\mathbf{y}}$. For all other cases, $\mathcal{D}'_1 = \mathcal{D}_1$.

It makes sense to specialize (17) as `peval[jac]` because $\dot{\mathbf{y}}$ is a tuple of static zeroes that will simplify the code. This also motivates producing a specialized code for $J_{\dot{\mathbf{y}}}$, where we instead have $\mathbf{y}' = \mathbf{0}$.

Hence, we can compute all non-zero elements of (12) for (9) as `peval[jac] idx`. Moreover, we can specialize each partial derivative as `peval[jac idx]`, but this will, in general, result in m^2 expressions.

To bound this code growth, we split `idx` into two disjoint sets `idxs` and `idxd`⁶, with the intuition that each column in `idxs` has many zero elements. In contrast, those in `idxd` have few.

We use these to compute the columns with two subtly different expressions: `peval[jac idxs]`, and `peval[jac] idxd`. Note that `idxs` is in the expression being specialized, while `idxd` is not; this has the effect of only specializing the columns with many zeros, which reduces code-swell. Formally we define `idxs` and `idxd` as follows:

$$\begin{aligned} \text{idx}_s &= \{(c, r) \mid (c, r) \in \text{idx}, |r| \leq \phi m\} \\ \text{idx}_d &= \{(c, r) \mid (c, r) \in \text{idx}, |r| > \phi m\}, \end{aligned} \quad (18)$$

where $\phi \in [0, 1]$ is a user-supplied parameter guiding the degree of specialization. This ensures that the number of specialized partial derivative expressions in the Jacobian callback function is bounded by $(\phi m + 1)m$ rather than m^2 . Another useful view of ϕ is that for a DAE with m equations and variables, and $n = \phi m$, columns in the Jacobian with n or fewer non-zero entries will be specialized by PEAD. $\phi = 0$ always results in an AD Jacobian, and $\phi = 1$ always results in fully specialized partial derivatives. However, in the (quite common) case where the DAE is sparse, a $\phi < 1$ may result in fully specialized partial derivatives.

For (9), a static $\theta = 1$, and max recursion depth ≥ 3 , $\phi = \frac{2}{6}$ results in

$$\begin{aligned} \text{peval[jac idx}_s] &= \\ &\lambda((x, \dot{x}, y, \dot{y}, \tau, u)). \lambda((\dot{x}', \ddot{x}, \dot{y}', \ddot{y}, \cdot, \cdot)). \\ &\quad ((1, (2 \times \dot{x}, -\dot{x}/2)), (3, (2 \times \dot{y}, \dot{y}/2))) \\ &\quad , (4, (-x, -y)), (5, (1))) \\ \text{peval[jac] idx}_d &= \\ &\text{peval[jac]} ((0, (0, 2, 4)), (2, (1, 2, 5))), \end{aligned} \quad (19)$$

where the columns of (12) that corresponds to x and y are computed dynamically as `jac idxd` because these have three non-zero elements, and the rest of the elements in (12) are specialized to symbolic partial derivatives. For brevity, we elide writing out the specialized function `peval[jac]`.

⁶Here s is for static and d for dynamic, following partial evaluation conventions.

In summary, our approach allows the user to conveniently and gradually mix AD and symbolic partial derivatives with an upper bound on the size of the generated code.

4 Design & Implementation

This section describes the design and implementation of Section 3 and how it fits into a larger compiler toolchain that takes an EOO model as input and outputs executable simulation code.

Figure 6a gives an overview of the complete compilation toolchain. Section 4.1 discusses the EOO modeling language and the transformation to a DAE program (boxes 1 to 3 in Figure 6a). Section 4.2 discusses the implementation of Section 3 (boxes 3 to 7). Section 4.3 discusses compiling the code of Section 4.2 (boxes 7 to 9).

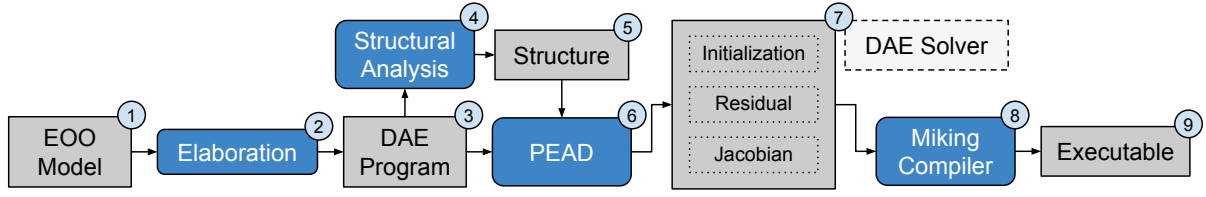
4.1 EOO Modeling and Elaboration

In EOO modeling languages, models consist of sub-models of physical components, e.g., resistors, capacitors, and inductors. These sub-models are connected to form a model of a larger system, similar to how components are connected in a real system. This is also one of the major benefits of the EOO modeling approach, as it offers intuitive models and a large degree of composability.

Figure 6b lists a code snippet of an EOO model in *Modelyze* [17], modeling an analog circuit. The circuit is visualized in Figure 6d. This circuit has the *Cauer* topology with n repeated induction-capacitor sections. In this EOO language, component models are implemented as functions, and one such section is created on line 7 inside a recursive function that builds up all n sections, given an initial model and the common connection point, $n3$, for all capacitors. The function returns the accumulated model and the connection point of the last inductor, i.e., the lower-right in Figure 6d.

The function starting on line 14 constructs the complete model. Line 15 defines the common and initial connection points for the inductor-capacitor sections. Line 16 defines dependent variables for the input and output voltages. Moreover, we model the independent variable as a dependent variable (lines 17 and 22). The voltage source is connected to the start of the circuit and added to the accumulator, which is then passed to the *Sections* function, lines 18 to 21. Finally, we connect the resistor and a voltage sensor and define the input voltage as a smooth step function.

The *elaboration* (item 2 in Figure 6a) takes an EOO program as input and outputs a DAE program. The elaboration has two components. First, the *Modelyze* interpreter symbolically evaluates the model to produce a set of equations from component models and a graph encoding the connections between components. The component equations constitute an underdetermined DAE, and the second step derives the missing equations from the component graph using energy conservation laws. There are several methods for different



(a) Overview of the compiler toolchain. Blue rounded boxes indicate transformations and analyses and gray square boxes indicate artifacts. The dotted squares indicate important functions. The dashed box indicates an external library.

```

1 def Sections(n3: Node, acc: (Model, Node), n: Int)
2   -> (Model, Node) = {
3     if n < 1 then error "n is not a positive number"
4   else
5     def (sections, n1) = acc;
6     def n2: Node;
7     def section = Inductor(L, n1, n2);
8     def section = Capacitor(C, n2, n3);
9     def acc = (section; sections, n2);
10    if n == 1 then acc
11  else Sections(n3, acc, n - 1)
12 }
13
14 def CauerTopology(n: Int) = {
15   def n1, n3: Node;
16   def uIn, uOut: Voltage;
17   def t: Signal;
18   def acc =
19     (VoltageSource(uIn, n1, n3), n1);
20   def (sections, n2) =
21     Sections(n3, acc, n);
22   t' = 1.;
23   sections;
24   Resistor(R, n3, n2);
25   VoltageSensor(uOut, n3, n2);
26   uIn = 1. / (1. + exp(-t))
27 }

```

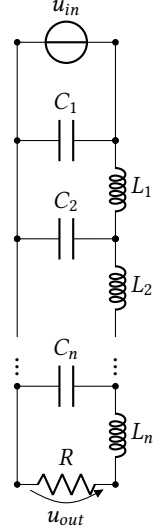
(b) EOO model of the circuit in Figure 6d.

```

1 variables
2   _t_1, _uOut_1, _uIn_1,
3   _u_R_1, i_R_1,
4   _u_L_50, _u_C_50, _i_L_50, _i_C_50, ... : Float
5 init
6 equations
7   _t_1' = 1.;
8   _uIn_1 = 1. / (1. + exp(-._t_1));
9   _u_L_50 = 1. * _i_L_50';
10  _i_C_50 = 1. * _u_C_50';
11  _u_R_1 = 1. * _i_R_1;
12  .
13  _u_R_1 = 1. * _i_R_1;
14  _u_L_50 = (1. * _u_C_49 - .1. * _uOut_1);
15  0. = ((-1. * _i_L_50 - .1. * _i_C_50) - 1. * _i_R_1);
16  _i_C_49 = (-1. * _i_L_50 - .1. * _i_L_49);
17  .
18 output
19 { _uOut_1 }

```

(c) Generated DAE program.



(d) n 'th Cauer topology.

Figure 6. Implementation overview and examples.

physical domains to derive these equations. Our tool implements a graph theoretical method that generalizes Kirchoff's circuit laws [4]. The benefit of this method is that it gives a unified methodology across multiple physical domains, and our tool includes libraries for the electrical, rotational-mechanical, and three-dimensional multi-body mechanics domains. Mixing models from these libraries to model multi-domain systems is also possible.

The EOO modeling language implementation outputs a *DAE program* (item 3 in Figure 6a), which is input to the simulation code generation.

4.2 Simulation Code Generation

Figure 6c shows the DAE program that is the output of elaborating the EOO model in Figure 6b, with $n = 50$. For simplicity, we set all physical constants to 1. The variables block on line 1 defines all dependent variables. Above this block, the user can define let-bindings and recursive functions that can be used in the model. In this case, we have none. The

init block on line 5 is empty, and the default initial values are 0, sufficient to initialize this model. Line 6 starts the equations block, where the equations from lines 7 to 12 stem from component models, and the equations from lines 13 to 17 are derived in the elaboration step. There are 206 dependent variables and equations in this example, but the listing only shows a small fraction of these. The output block on lines 18 to 19 may contain any expression and dependent variables.

We have implemented the analysis and transformations steps corresponding to items 4 and 6 in Figure 6a within the Miking core language [16]. The frontend is implemented using the parser generator of Palmkvist et al. [48]. The Miking core language also serves as the intermediate language of Section 3 and the language of the output program.

We have implemented the PEAD approach in Miking, as described in Section 3 (see item 6 in Figure 6a). Also, we have used and implemented the Sigma-method [53] for the structural analysis (item 4). Similar to, e.g., Shaikhha et al. [56], the function δ_j in (17) is sparsely represented as single

integer j and all projections $y'.i$ are implemented as

if $i = y'$ then 1 else 0.

In (17), let $\dot{y}' = 0$ in is replaced by let $\dot{y}' = -1$ in. The output from the structural analysis contains information on which equation needs differentiation and what dependent variables will appear in the index-reduced DAE.

Finding consistent initial values for DAEs is challenging, typically modeled and solved as a non-convex optimization problem [7]. Our tool supports simple initial value assignments; more general initial value equations are future work.

The output program (item 7 in Figure 6a) contains—in addition to the residual function and Jacobian callback functions discussed in Section 3—an initialization function that sets initial values according to the `init` block.

4.3 Simulation Runtime and Compilation

The simulation code includes a runtime that interfaces to the SUNDIALS IDA numerical DAE solver [33]. The Miking compiler supports different compilation targets where the most complete is OCaml and is currently the only one supported by our tool. The final *executable* (item 9 in Figure 6a) is therefore produced by the OCaml compiler. We interface with SUNDIALS IDA via the SUNDIALS/ML package [12].

5 Evaluation

This section describes the evaluation of PEAD. Section 5.1 provides the experimental setup, and Section 5.2 summarizes and discusses the evaluation results.

5.1 Experimental Setup

We consider DAE models of four physical systems⁷, summarized in Table 1. Except for S_1 , all systems are modeled as EOO models in Modelyze and then elaborated to DAE programs, as described in Section 4.

All experiments are performed on a Xeon Gold 6148 CPU with 64GB of memory on Ubuntu 18.04.06, using the implementation described in Section 4, SUNDIALS IDA 2.7.0, and OCaml 4.14.0. We perform the execution time experiments with hyperfine 1.16.1 [50] and ensure that the executable runs on a single core with `taskset`. The simulations have absolute and relative tolerances of 10^{-8} and 10^{-6} , respectively. The experiment is repeated 100 times with 3 warm-up runs to allow caches to heat up.

To evaluate the residual and Jacobian code generation, we generate simulation code for the DAE models of the systems described in Table 1 in the following two configurations. The first configuration, denoted *ad*, index reduces with the \mathcal{T} transformation and generates a Jacobian function as described in Section 3.3 but where no partial derivatives are specialized. This means it exploits the sparsity information

⁷The circuits with the Cauer topology are the ones depicted in Figure 6d but with the resistor replaced by a resistor and capacitor in parallel to make it index-2.

from the structural analysis to avoid evaluating zero elements of the Jacobian, but all derivatives are computed using pure AD. The second configuration uses the complete PEAD approach of Section 3 for both the residual and Jacobian, with different values on the user-parameter $\phi \in [0, 1]$, with one exception. To make it clearer what optimization affected which part, we refrain from partially evaluating the function that dynamically computes partial derivatives of the Jacobian. I.e., we use `jac idxd` in place of `peval[jac] idxd` (c.f. (19)). The naming scheme for these configurations is *pead-n*, where $n = \phi m$ for a DAE with m equations and dependent variables. We choose ϕ s.t. for each increment in ϕm ; PEAD will specialize more Jacobian columns than the previous value, and the final value of ϕm specializes all partial derivatives in the Jacobian for all models. In particular, *pead-0* specializes no partial derivatives but partially evaluates the residual function, and the residual code is identical for all *pead-n*. Table 2 summarizes the number of specialized partial derivatives for each model and ϕm .

In addition, we execute the simulation code in three different modes to better separate the execution time contribution from the residual function and the Jacobian callback function. In the first mode, denoted *sim*, we simulate the DAE with the IDA solver. In the second mode, *res*, and the third mode, *jac*, we execute the generated residual function and Jacobian function with random input as often as these are evaluated during *sim*-mode. Table 3 summarizes the simulation time and the number of calls for each model. The number of calls is constant for all configurations.

5.2 Experimental Results

Figure 7 summarizes the statistics for execution time and executable size for each model, configuration, and execution mode.

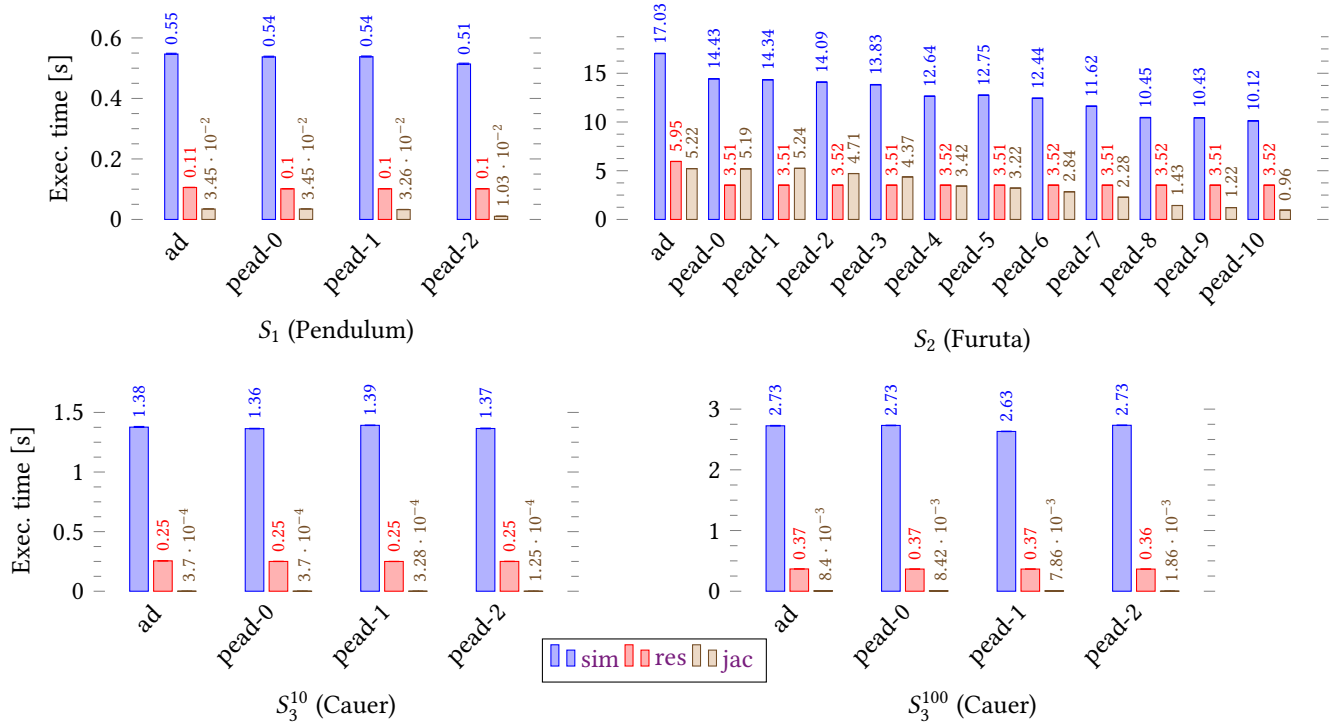
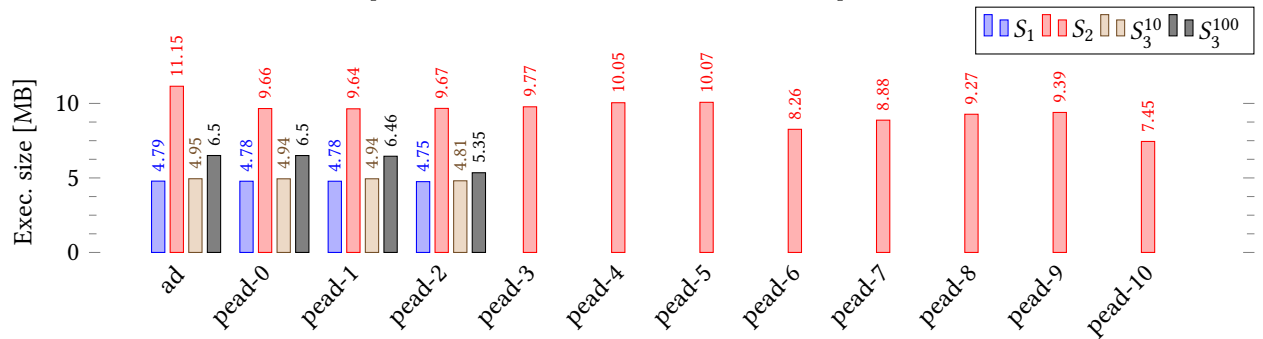
In Figure 7a, we see that the execution time of *jac*-mode decreases as the level of specialization increases for all models, with one exception: For S_2 , *pead-1* does not improve upon *pead-0* (and *ad*). For this model, *pead-1* mostly specializes equations that are either trivial, i.e., $x = c$, where c is a constant, or alias equations of the form $x = y$, which are efficiently computed by AD.

The execution time of *sim*-mode for S_1 improves slightly from *ad* to *pead-n* and with increasing n , while the execution time for *res*-mode remains approximately the same. As shown in Figure 7b, the executable size is approximately the same for all configurations.

For S_2 , we see a significant improvement in execution time for all modes as the level of specialization increases. This model contains large, highly non-linear residual expressions, which seem to benefit from our approach. Moreover, the size of the code increases moderately with increasing ϕm after an initial drop due to simplifications of AD-transformed elementary functions. The drop in size between *pead-9* and

Table 1. Summary of domains, (M)echanical with dimensions (dim) and (E)lectrical; the number of equations solved by the numerical solver (Size); index of the evaluated systems; and the number of equations differentiated (#Diff).

System	M. dim.	E	Size	Index	#Diff	Description
S_1	2	-	5	3	3	A planar pendulum modeled in Cartesian coordinates as a first-order DAE.
S_2	3	-	143	3	55	The Furuta pendulum modeled in absolute Cartesian coordinates.
S_3^{10}	-	✓	48	2	2	A circuit with the 10'th Cauer topology.
S_3^{100}	-	✓	408	2	2	A circuit with the 100'th Cauer topology.

**(a)** Summary of execution time for the different models, execution modes, and configurations. The bars' height and numbers are the median execution time over 25 runs, and the error bars are the upper and lower quartiles. The colors denote the execution mode. Note that the Jacobian code is the same between *ad* and *pead-0*, and the residual code is the same for all *pead-n*.**(b)** Summary of executable size for the different models and configurations. Different colors denote different models. The right-most value for each model is the fully specialized Jacobian.**Figure 7.** Summarizing statistics of execution time and executable size.

pead-10 stems from removing the whole AD Jacobian from the code because all partial derivatives are specialized.

We do not expect res-mode and sim-mode to change considerably between configurations for the models S_3^{10} and S_3^{100} . They consist of simple linear equations, approximately half of the equations involves only two dependent variables (c.f. Figure 6c). Only two of their equations appear differentiated, which means that, in particular, for S_3^{100} with 408 equations, the residual for the ad-configuration is nearly optimal. Indeed, res-mode for all configurations for both S_3^{10} and S_3^{100} are nearly identical. Moreover, because the Jacobian is linear, the IDA solver only needs to evaluate it a small number of times (see Table 3). Consequently, the Jacobian evaluation contribution to the simulation time is small.

As expected, the execution time for sim-mode is approximately the same for S_3^{10} and S_3^{100} . However, for S_3^{100} , there is a slight increase (less than 1%) in sim-mode execution time between pead-1 and pead-2, even though the residual code is identical between the configurations and the Jacobian callback is faster (we have verified this by measuring its wall time during simulation). Using the perf tool, we have profiled the runtime and performed a cache analysis. We found increased cache misses and page-faults for pead-2, which might explain its slight increase in execution time. Moreover, we have observed that small changes in the code generation and minor changes in data structures give small variations in the execution time difference between the different configurations in sim-mode.

In summary, PEAD gives at most a small decrease in performance for the simpler linear model S_3^{100} at $\phi m = 2$. In

Table 2. Summary of the number of specialized partial derivatives in the Jacobian for each model and value ϕm , where - means that all partial derivatives are specialized.

ϕm :	1	2	3	4	5	6	7	8	9	10
S_1	2	11	-	-	-	-	-	-	-	-
S_2	64	113	135	172	189	198	210	229	232	244
S_3^{10}	22	69	-	-	-	-	-	-	-	-
S_3^{100}	202	609	-	-	-	-	-	-	-	-

Table 3. Summary of the simulation time and the number of residual and Jacobian evaluations during simulation. For more stable measurements, we scale the number of evaluations of jac-mode for S_3 with 100 and then scale the result back so that the plots of jac-mode in Figure 7a correspond to the numbers in this table.

	S_1	S_2	S_3^{10}	S_3^{100}
Sim. time (s)	10^4	10^3	10^4	10^3
#Res. evals.	231687	105602	106541	11493
#Jac. evals.	9819	4108	23	29

contrast, it slightly improves S_1 and significantly increases the performance for the more complex non-linear model S_2 , thus demonstrating an overall performance improvements without sacrificing code size.

6 Related Work

This section discusses related work in the topic of partial evaluation and AD, Section 6.1; and AD and DAE solvers, Section 6.2.

6.1 On Partial Evaluation and AD

There is a large body of work on AD for imperative and functional languages. We can broadly categorize AD implementation techniques into source-code transformations, e.g., [1, 32, 37, 55, 57], and operator overloading approaches, e.g., [10, 21, 36, 60]. The operator overloading approach is straightforward to implement and expressive because it is easy to support, e.g., higher-order functions. However, the source-code transformation approach provides more opportunities for optimization. Baydin et al. [8] give a good survey on AD, albeit focusing on machine learning.

ADOL-C combines operator overloading with C++ expression templates to improve performance [34]. Moreover, the operator overloading approach has been combined with multi-stage programming that retains some of the benefits of the source-code transformation approach and the flexibility of operator overloading [61].

The combination of partial evaluation and forward-mode AD has been discussed in the literature. Elsmann et al. [23] identifies the relation between a symbolic derivative and partial evaluation of forward-mode AD. However, their work is focused on providing a unified framework to express both forward and reverse mode AD, over Hilbert spaces, on programs in combinatory form.

Shaikhha et al. [55] propose a source-code transformation that partially evaluates forward-mode AD in a functional higher-order array language. Their empirical results show that their approach can outperform state-of-the-art AD implementations, both reverse and forward mode, in certain machine learning and computer vision applications.

Hascoet and Pascual [32] propose a tool, TAPENADE, and a source-code transformation method that takes a program in a subset of FORTRAN or C as input and outputs a program that computes derivatives with either reverse or forward-mode AD. Although not formally defined, this transformation partially evaluates expressions to produce more efficient code.

Our approach extends the idea of partially evaluated AD to code generation for off-the-shelf DAE solvers.

6.2 On Automatic Differentiation and DAE Solvers

Campbell et al. [18], Campbell and Marszalek [19] discuss and evaluate symbolic differentiation and AD in the context

of DAEs. In their work, they compare code generation using symbolic differentiation via Maple [39] and AD via ADOL-C. Our work's key idea is to combine symbolic differentiation and AD via partial evaluation simultaneously. However, the authors discuss and make contributions to the correctness on index reduction, where we assume that the result of the structural analysis is correct.

Griewank and Walther [30] and Nedialkov and Pryce [43, 44, 45] propose numerical solving of DAEs via Taylor expansion, where Taylor coefficients are computed via Taylor-mode AD. Both solution methods are based on Pryce's method [52]. The former uses ADOL-C, which implements the graph-coloring approach of Gebremedhin et al. [27] to analyze and exploit sparsity patterns when computing full Jacobians. The latter implements a numerical DAE solver, DAETS, which uses the FADBAD++ [58] that implements AD via operator overloading and C++ template expressions. DAETS include optimizations in the form of common sub-expression elimination of the computational graph produced by FADBAD++ [54]. Moreover, DAETS exploits sparsity information from the structural analysis to produce code that incrementally computes the system Jacobian, which is part of the solver routine of Pryce's method.

In contrast to these works, our method targets code generation for off-the-shelf numerical DAE solvers, such as the DASSL family of solvers, rather than solving DAEs via Taylor series. Moreover, it employs AD as a source-code transformation and relates it to symbolic differentiation via partial evaluation.

Pryce et al. [54] give a scheme for incorporating AD for index-reduction and stabilization via the Dummy Derivative method [41]. Although promising, to our knowledge, this scheme has yet to be implemented and evaluated.

Olsson et al. [46] propose using AD in the Modelica languages to extend the expressiveness of the language and the class of models that can be index-reduced. Consequently, current versions of Dymola support differentiation of, e.g., recursive functions if these functions are annotated as sufficiently smooth. AD does not always correspond to an analytic derivative for non-smooth functions. Their work does not, however, discuss the combination of partial evaluation and AD for DAEs which is the main contribution of this work.

Computing the system Jacobian of DAEs in EOO languages is commonly performed using pure AD, e.g., Andersson et al. [2, 3]. These tools implement the method of Gebremedhin et al. [27] to exploit sparsity information in the Jacobian. Another method, as is the case for OpenModelica, uses a symbolic interpretation of forward-mode AD to generate symbolic partial derivatives [13]. Elsheimh [22] gives an efficient method for computing parameter sensitivities of DAE models with AD.

Our approach of partially evaluating AD to produce a code for the system Jacobian lets the user control the trade-off between code size and specialized code.

7 Conclusion

In this paper, we develop a new approach to efficiently solving differential-algebraic equations (DAEs). Generating efficient simulation code for DAE models forms the backbone of Equation-Based Object-Oriented modeling language compilers. Specifically, we design a technique for partially evaluating automatic differentiation that combines the benefits of symbolic and automatic differentiation while mitigating their weaknesses. The approach improves both differentiation during index reduction and evaluation of the Jacobian.

We evaluate the ideas using several different equation-based models within the mechanical and electrical domains. Our experimental results suggest that our approach performs best for large, complex, and non-linear models. There are no clear gains in simulation performance for simpler, near-optimal, linear models.

Our approach to specializing the Jacobian callback function is applicable in other domains, such as minimization of vector-valued functions, where structural information on the Jacobian is available at compile time. Future work includes extending our approach to stabilized index reduction [47], evaluating larger models, including models with recursion, and combining our approach with the graph-coloring approach [28] for the AD-part of the Jacobian.

8 Data-Availability Statement

Software artifact available [24].

Acknowledgments

We thank the reviewers for their valuable feedback. A special thanks to Romy Tsoupidi for proofreading, discussions, and support. This work was financially supported by the Swedish Research Council (Grant No. 2018-04329), and in part by the Swedish Foundation for Strategic Research (Grant No. FFL15-0032), Digital Futures (the DLL project), and the Vinnova Competence Center for Trustworthy Edge Computing Systems and Applications (TECoSA) at the KTH Royal Institute of Technology.

References

- [1] Martin Abadi and Gordon D. Plotkin. 2019. A simple differentiable programming language. *Proceedings of the ACM on Programming Languages* 4, POPL (Dec. 2019), 38:1–38:28. <https://doi.org/10.1145/3371106>
- [2] Joel Andersson, Johan Åkesson, and Moritz Diehl. 2012. CasADi: A symbolic package for automatic differentiation and optimal control. In *Recent advances in algorithmic differentiation*. Springer, 297–307. https://doi.org/10.1007/978-3-642-30023-3_27
- [3] Joel AE Andersson, Joris Gillis, Greg Horn, James B Rawlings, and Moritz Diehl. 2019. CasADi: a software framework for nonlinear

- optimization and optimal control. *Mathematical Programming Computation* 11 (2019), 1–36. <https://doi.org/10.1007/s12532-018-0139-4>
- [4] Gordon C. Andrews. 1977. A General Re-Statement of the Laws of Dynamics Based on Graph Theory. In *Problem Analysis in Science and Engineering*, F.H. Branin and K. Huseyin (Eds.). Academic Press, 1–40. <https://doi.org/10.1016/B978-0-12-125550-3.50006-5>
 - [5] Martin Arnold. 2017. *DAE Aspects of Multibody System Dynamics*. Springer International Publishing, Cham. 41–106 pages. https://doi.org/10.1007/978-3-319-46618-7_2
 - [6] Peter J. Ashenden, Gregory D. Peterson, and Darrell A. Teegarden. 2002. *The System Designer's Guide to VHDL-AMS: Analog, Mixed-Signal, and Mixed-Technology Modeling*. Morgan Kaufmann Publishers, USA. <https://doi.org/10.1016/B978-1-55860-749-1.X5000-2>
 - [7] Bernhard Bachmann, Peter Aronsson, and Peter Fritzson. 2007. Robust Initialization of Differential Algebraic Equations. In *Equation-Based Object-Oriented Languages and Tools (EOOLT)*. 151–163.
 - [8] Atilim Gunes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. 2015. Automatic differentiation in machine learning: a survey. (2015). <https://doi.org/10.48550/ARXIV.1502.05767>
 - [9] Atilim Gunes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. 2018. Automatic differentiation in machine learning: a survey. *Journal of Machine Learning Research* 18 (2018), 1–43.
 - [10] Atılım Güneş Baydin, Barak A. Pearlmutter, and Jeffrey Mark Siskind. 2016. DiffSharp: An AD Library for .NET Languages. <http://arxiv.org/abs/1611.03423> arXiv:1611.03423 [cs].
 - [11] Anders Bondorf. 1992. Improving Binding Times without Explicit CPS-Conversion. In *Proceedings of the 1992 ACM Conference on LISP and Functional Programming* (San Francisco, California, USA) (LFP '92). Association for Computing Machinery, New York, NY, USA, 1–10. <https://doi.org/10.1145/141471.141483>
 - [12] Timothy Bourke, Jun Inoue, and Marc Pouzet. 2018. Sundials/ML: Connecting OCaml to the Sundials Numeric Solvers. In *Proceedings ML Family Workshop / OCaml Users and Developers workshops*, Nara, Japan, September 22–23, 2016 (Electronic Proceedings in Theoretical Computer Science, Vol. 285), Kenichi Asai and Mark Shinwell (Eds.). Open Publishing Association, 101–130. <https://doi.org/10.4204/EPTCS.285.4>
 - [13] Willi Braun, Lennart Ochel, and Bernhard Bachmann. 2011. Symbolically derived Jacobians using automatic differentiation-enhancement of the OpenModelica compiler. In *Proceedings of the 8th International Modelica Conference; March 20th–22nd; Technical University; Dresden; Germany*. Linköping University Electronic Press, 495–501. <https://doi.org/10.3384/ecp11063495>
 - [14] K. E. Brenan, S. L. Campbell, and L. R. Petzold. 1995. *Numerical Solution of Initial-Value Problems in Differential-Algebraic Equations*. Society for Industrial and Applied Mathematics. <https://doi.org/10.1137/1.9781611971224>
 - [15] David Broman. 2010. *Meta-Languages and Semantics for Equation-Based Modeling and Simulation*. Ph.D. Dissertation. Department of Computer and Information Science, Linköping University, Sweden.
 - [16] David Broman. 2019. A Vision of Miking: Interactive Programmatic Modeling, Sound Language Composition, and Self-Learning Compilation. In *Proceedings of the 12th ACM SIGPLAN International Conference on Software Language Engineering (Athens, Greece) (SLE 2019)*. Association for Computing Machinery, New York, NY, USA, 55–60. <https://doi.org/10.1145/3357766.3359531>
 - [17] David Broman and Jeremy G. Siek. 2017. Gradually Typed Symbolic Expressions. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (Los Angeles, CA, USA) (PEPM '18)*. Association for Computing Machinery, New York, NY, USA, 15–29. <https://doi.org/10.1145/3162068>
 - [18] Stephen L. Campbell, R. Hollenbeck, K. Yeomans, and Yangchun Zhong. 1998. Mixed symbolic–numerical computations with general DAEs I: System properties. *Numerical Algorithms* 19 (1998), 73–83. <https://doi.org/10.1023/A:1019154423096>
 - [19] Stephen L. Campbell and Wiesław Marszałek. 1998. Mixed symbolic–numerical computations with general DAEs II: An applications case study. *Numerical Algorithms* 19 (1998), 85–94. <https://doi.org/10.1023/A:1019106507166>
 - [20] Dassault Systems. [n. d.]. DYMOLA Systems Engineering: Multi-Engineering Modeling and Simulation based on Modelica and FMI. <http://www.dymola.com> [Last accessed: September 9, 2020].
 - [21] Conal Elliott. 2009. Beautiful differentiation. In *International Conference on Functional Programming (ICFP)*. <http://conal.net/papers/beautiful-differentiation>
 - [22] Atiyah Elsheikh. 2015. An equation-based algorithmic differentiation technique for differential algebraic equations. *J. Comput. Appl. Math.* 281 (June 2015), 135–151. <https://doi.org/10.1016/j.cam.2014.12.026>
 - [23] Martin Elsmann, Fritz Henglein, Robin Kaarsgaard, Mikkel Kragh Mathiesen, and Robert Schenck. 2022. Combinatory Adjoints and Differentiation. In *Proceedings Ninth Workshop on Mathematically Structured Functional Programming*, Munich, Germany, 2nd April 2022 (Electronic Proceedings in Theoretical Computer Science, Vol. 360), Jeremy Gibbons and Max S. New (Eds.). Open Publishing Association, 1–26. <https://doi.org/10.4204/EPTCS.360.1>
 - [24] Oscar Eriksson, Viktor Palmkvist, and David Broman. 2023. *Partial Evaluation of Automatic Differentiation for Differential-Algebraic Equations Solvers*. <https://doi.org/10.5281/zenodo.8347712>
 - [25] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. 1993. The Essence of Compiling with Continuations. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation* (Albuquerque, New Mexico, USA) (PLDI '93). Association for Computing Machinery, New York, NY, USA, 237–247. <https://doi.org/10.1145/155090.155113>
 - [26] Peter Fritzson, Peter Aronsson, Håkan Lundvall, Kaj Nyström, Adrian Pop, Levon Saldamli, and David Broman. 2005. The OpenModelica Modeling, Simulation, and Software Development Environment. *Simulation News Europe* 15, 44/45 (2005), 8–16.
 - [27] Assefaw Hadish Gebremedhin, Fredrik Manne, and Alex Pothén. 2005. What Color Is Your Jacobian? Graph Coloring for Computing Derivatives. *SIAM Rev.* 47, 4 (2005), 629–705. <https://doi.org/10.1137/S0036144504444711> arXiv:https://doi.org/10.1137/S0036144504444711
 - [28] Narain Gehani and Krithi Ramamritham. 1991. Real-time concurrent C: A language for programming dynamic real-time systems. *Real-Time Systems* 3, 4 (1991), 377–405. <https://doi.org/10.1007/BF00365999>
 - [29] Andreas Griewank, Jean Utke, and Andrea Walther. 2000. Evaluating Higher Derivative Tensors by Forward Propagation of Univariate Taylor Series. *Math. Comp.* 69, 231 (2000), 1117–1130. <http://www.jstor.org/stable/2585017>
 - [30] Andreas Griewank and Andrea Walther. 2004. On the Efficient Generation of Taylor Expansions for DAE Solutions by Automatic Differentiation. In *Proceedings of the 7th International Conference on Applied Parallel Computing: State of the Art in Scientific Computing* (Lyngby, Denmark) (PARA'04). Springer-Verlag, Berlin, Heidelberg, 1089–1098. https://doi.org/10.1007/11558958_131
 - [31] Benjamin Grégoire and Xavier Leroy. 2002. A Compiled Implementation of Strong Reduction. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming* (Pittsburgh, PA, USA) (ICFP '02). Association for Computing Machinery, New York, NY, USA, 235–246. <https://doi.org/10.1145/581478.581501>
 - [32] Laurent Hascoet and Valérie Pascual. 2013. The Tapenade Automatic Differentiation Tool: Principles, Model, and Specification. *ACM Trans. Math. Softw.* 39, 3, Article 20 (may 2013), 43 pages. <https://doi.org/10.1145/2450153.2450158>
 - [33] Alan C. Hindmarsh, Peter N. Brown, Keith E. Grant, Steven L. Lee, Radu Serban, Dan E. Shumaker, and Carol S. Woodward. 2005. SUNDIALS: Suite of Nonlinear and Differential/Algebraic Equation Solvers. *ACM*

- Trans. Math. Softw.* 31, 3 (sep 2005), 363–396. <https://doi.org/10.1145/1089014.1089020>
- [34] Robin J. Hogan. 2014. Fast Reverse-Mode Automatic Differentiation Using Expression Templates in C++. *ACM Trans. Math. Software* 40, 4 (jun 2014), 26:1–26:24. <http://doi.acm.org/10.1145/2560359>
- [35] Neil D Jones, Carsten K Gomard, and Peter Sestoft. 1993. *Partial evaluation and automatic program generation*.
- [36] Jerzy Karczmarczuk. 1998. Functional Differentiation of Computer Programs. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming* (Baltimore, Maryland, USA) (ICFP '98). Association for Computing Machinery, New York, NY, USA, 195–203. <https://doi.org/10.1145/289423.289442>
- [37] Faustyna Krawiec, Simon Peyton Jones, Neel Krishnaswami, Tom Ellis, Richard A. Eisenberg, and Andrew Fitzgibbon. 2022. Provably Correct, Asymptotically Efficient, Higher-Order Reverse-Mode Automatic Differentiation. *Proc. ACM Program. Lang.* 6, POPL, Article 48 (jan 2022), 30 pages. <https://doi.org/10.1145/3498710>
- [38] Peter Kunkel and Volker Mehrmann. 2006. *Differential-Algebraic Equations Analysis and Numerical Solution*. European Mathematical Society.
- [39] Maplesoft. [n.d.]. Maple. <https://www.maplesoft.com/products/Maple/> [Last accessed: September 16, 2023].
- [40] MathWorks. [n.d.]. Simscape - Model and simulate multidomain physical systems. <https://www.mathworks.com/products/simscape> [Last accessed: February 1, 2020].
- [41] Sven Erik Mattsson and Gustaf Söderlind. 1993. Index reduction in differential-algebraic equations using dummy derivatives. *SIAM Journal on Scientific Computing* 14, 3 (1993), 677–692. <https://doi.org/10.1137/0914043>
- [42] Modelica Association. 2017. *Modelica - A Unified Object-Oriented Language for Physical Systems Modeling - Language Specification Version 3.4*. Modelica Association. Available from: <http://www.modelica.org>.
- [43] Nedialko S Nedialkov and John D Pryce. 2005. Solving differential-algebraic equations by Taylor series (I): Computing Taylor coefficients. *BIT-Numerical Mathematics* 45, 3 (2005), 561–592. <https://doi.org/10.1007/s10543-005-0019-y>
- [44] Nedialko S Nedialkov and John D Pryce. 2007. Solving differential-algebraic equations by Taylor series (II): Computing the System Jacobian. *BIT Numerical Mathematics* 47 (2007), 121–135. <https://doi.org/10.1007/s10543-006-0106-8>
- [45] Nedialko S Nedialkov and John D Pryce. 2007. Solving differential-algebraic equations by Taylor series (III): the DAETS code. *Journal of Numerical Analysis, Industrial and Applied Mathematics* 1, 1 (2007), 1–30.
- [46] Hans Olsson, Hubertus Tummescheit, Hilding Elmqvist, AB Dynasim, and AB Modelon. 2005. Using automatic differentiation for partial derivatives of functions in Modelica. In *Proceedings of the 4th International Modelica Conference*.
- [47] Martin Otter and Hilding Elmqvist. 2017. Transformation of differential algebraic array equations to index one form. In *Proceedings of the 12th International Modelica Conference*. Linköping University Electronic Press. <https://doi.org/10.3384/ecp17132565>
- [48] Viktor Palmkvist, Elias Castegren, Philipp Haller, and David Broman. 2023. Statically Resolvable Ambiguity. *Proc. ACM Program. Lang.* 7, POPL, Article 58 (jan 2023), 27 pages. <https://doi.org/10.1145/3571251>
- [49] Constantinos C Pantelides. 1988. The consistent initialization of differential-algebraic systems. *SIAM J. Sci. Statist. Comput.* 9, 2 (1988), 213–231. <https://doi.org/10.1137/0909014>
- [50] David Peter. 2023. hyperfine. <https://github.com/sharkdp/hyperfine>
- [51] L. R. Petzold. 1982. *Description of DASSL: a differential/algebraic system solver*. Technical Report SAND-82-8637; CONF-820810-21. Sandia National Labs., Livermore, CA (USA). <https://www.osti.gov/biblio/5882821>
- [52] J. D. Pryce. 1998. *Numerical Algorithms* 19, 1/4 (1998), 195–211. <https://doi.org/10.1023/a:1019150322187>
- [53] J. D. Pryce. 2001. *Bit Numerical Mathematics* 41, 2 (2001), 364–394. <https://doi.org/10.1023/a:1021998624799>
- [54] John D. Pryce, Nedialko S. Nedialkov, Guangning Tan, and Xiao Li. 2018. How AD can help solve differential-algebraic equations. *Optimization Methods and Software* 33, 4-6 (March 2018), 729–749. <https://doi.org/10.1080/10556788.2018.1428605>
- [55] Amir Shaikhha, Andrew Fitzgibbon, Dimitrios Vytiniotis, and Simon Peyton Jones. 2019. Efficient Differentiable Programming in a Functional Array-Processing Language. *Proc. ACM Program. Lang.* 3, ICFP, Article 97 (jul 2019), 30 pages. <https://doi.org/10.1145/3341701>
- [56] Amir Shaikhha, Mathieu Huot, and Shide Hashemian. 2023. ∇ SD: Differentiable Programming for Sparse Tensors. arXiv:2303.07030 [cs.PL]
- [57] Jeffrey M Siskind and Barak A Pearlmutter. 2008. Using Polyvariant Union-Free Flow Analysis to Compile a Higher-Order Functional-Programming Language with a First-Class Derivative Operator to Efficient Fortran-like Code. (2008), 12.
- [58] Ole Stauning and Claus Bendtsen. [n.d.]. FADBAD++. <http://uning.dk/fadbad.html> [Last accessed: September 16, 2023].
- [59] Matthijs Vákár, Sam Staton, and Mathieu Huot. 2022. Higher order automatic differentiation of higher order functions. *Logical Methods in Computer Science* 18 (2022). [https://doi.org/10.46298/lmcs-18\(1:41\)2022](https://doi.org/10.46298/lmcs-18(1:41)2022)
- [60] A. Walther and A. Griewank. 2012. Getting started with ADOL-C. , 181–202 pages.
- [61] Fei Wang, Daniel Zheng, James Decker, Xilun Wu, Grégory M. Essertel, and Tiark Rompf. 2019. Demystifying differentiable programming: shift/reset the penultimate backpropagator. *Proceedings of the ACM on Programming Languages* 3, ICFP (July 2019), 96:1–96:31. <https://doi.org/10.1145/3341700>

Received 2023-07-14; accepted 2023-09-03

Crossover: Towards Compiler-Enabled COBOL-C Interoperability

Mart van Assen

mart@vanassen.info

Computer Science, University of Twente
Enschede, The Netherlands

Ömer Faruk Sayilir

o.f.sayilir@student.utwente.nl

Computer Science, University of Twente
Enschede, The Netherlands

Manzi Aimé Ntagengerwa

m.a.ntagengerwa@gmail.com

Computer Science, University of Twente
Enschede, The Netherlands

Vadim Zaytsev

vadim@grammarware.net

Formal Methods & Tools, University of Twente
Enschede, The Netherlands

Abstract

Interoperability across software languages is an important practical topic. In this paper, we take a deep dive into investigating and tackling the challenges involved with achieving interoperability between C and BabyCobol. The latter is a domain-specific language condensing challenges found in compiling legacy languages — borrowing directly from COBOL’s data philosophy. CROSSOVER, a compiler designed specifically to showcase the interoperability, exposes details of connecting a COBOL-like language with PICTURE clauses and re-entrant procedures, to C with primitive types and struct composites. CROSSOVER features a C library for overcoming the differences between the data representations native to the respective languages. We illustrate the design process of CROSSOVER and demonstrate its usage to provide a strategy to achieve interoperability between legacy and modern languages. The described process is aimed to be a blueprint for achievable interoperability between full-fledged COBOL and modern C-like programming languages.

CCS Concepts: • Software and its engineering → Interoperability; Maintaining software.

Keywords: legacy languages, integration, compilation

ACM Reference Format:

Mart van Assen, Manzi Aimé Ntagengerwa, Ömer Faruk Sayilir, and Vadim Zaytsev. 2023. Crossover: Towards Compiler-Enabled

COBOL-C Interoperability. In *Proceedings of the 22nd ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE ’23)*, October 22–23, 2023, Cascais, Portugal. ACM, New York, NY, USA, 14 pages. <https://doi.org/10.1145/3624007.3624055>

1 Introduction

Many organisations from the public and private sectors rely on legacy software systems written in the last century for their critical day-to-day operations: 43% of all banking systems are built on COBOL [11], 44 out of top 50 banks [24] and 92 out of top 100 [4] use mainframes. 71% of Fortune 500 companies are relying on legacy systems [4]. The most prominent legacy languages in the 2020s are COBOL (42%), HLASM (37%), PL/I (22%) and various Fourth Generation Languages, or 4GLs (22%–32%) [16]. Among mainframe-using companies, 75% are concerned about having access to the right IT talent to maintain and manage their mainframes [16].

A major part of the challenge of understanding, maintaining and renovating software written in legacy languages, is related to the different data philosophy adapted by them [2, 38, 39]. Mainframe languages normalise mixing the form and the representation, leading developers into writing code that relies on knowing precise byte sizes of data and operating with ad hoc data structures with constraints varying per byte and reuse provided by lexically reimporting data definitions. Modern languages tend towards separation of concerns, leading developers to define reusable types encapsulating and hiding exact implementation details.

In this paper, we introduce CROSSOVER: a compiler that bridges the gap between COBOL’s and C’s data philosophies. Since real legacy programming languages such as COBOL, FORTRAN or PL/I are too large to implement in a similar way as a proof-of-concept [22], CROSSOVER showcases our strategy for language interoperability on BabyCobol [45] (see Section 2.1). This programming language is meant to be quickly implementable and yet still offer many of the challenges that arise from processing legacy languages, the same way Featherweight Java is used for experimental features in language semantics instead of Java.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org. GPCE ’23, October 22–23, 2023, Cascais, Portugal
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 979-8-4007-0406-2/23/10...\$15.00

<https://doi.org/10.1145/3624007.3624055>

CROSSOVER allows for the calling of procedures between BabyCobol and C. Since these languages have vastly different ways of representing data, marshalling of parameters and return values is necessary. To account for this difference in data representation, we designed the “BabyCobol standard library” (BSTD); a shared C library used by the compiler for data representation, marshalling and manipulation during runtime. By also making this library available to the programmer we bring some BabyCobol semantics to C.

To illustrate the impact of the compiler, the following code snippet shows how BabyCobol could use a function call to a C program to calculate the value of the seventh Fibonacci number. First, the variables N and RESULT are defined in the DATA DIVISION (the part of the COBOL program containing data definitions). In the PROCEDURE DIVISION (the code-containing part), the value of N gets set to 7. Then the CALL statement calls the function “fib” of a C program called “fibonacci” and passes N as an argument to indicate the N-th Fibonacci number must be calculated. When reentering BabyCobol, the return value of the C function is parsed into the RESULT variable. The BabyCobol program then displays the result and terminates.

```

1 IDENTIFICATION DIVISION.
2 PROGRAM-ID. FIB.
3 DATA DIVISION.
4 01 WORKING-STORAGE-AREA.
5   02 N PICTURE IS 99.
6   02 RESULT PICTURE IS 9999999999.
7 PROCEDURE DIVISION.
8   MOVE 7 TO N.
9   DISPLAY "Calculating the " N "-th Fibonacci
      number: "
10  CALL fib OF fibonacci
11      USING BY VALUE N AS PRIMITIVE
12      RETURNING RESULT.
13  DISPLAY RESULT.
14  STOP.

```

Listing 1. Calculating the 7th Fibonacci number by calling a C function

CROSSOVER is created using the readily available compiler construction tools ANTLR [28] (a parser generator) and the LLVM Core libraries [21]. At runtime, it also requires the Clang compiler to be installed on the system. Complete engineering details about dependencies and installation requirements can be found in the project repository [33]. CROSSOVER currently implements a subset of the BabyCobol language [45]. Its novel features include:

- The instantiation and modification of BabyCobol data types in C code.
- The invocation of C functions from a BabyCobol codebase, and the invocation of BabyCobol paragraphs from C.
- The automatic generation of C composite structs from BabyCobol data definitions.

Systems similar to CROSSOVER have been created before, as we will see in the next section. However, all of them are tightly integrated parts of proprietary frameworks. CROSSOVER is meant to be open source, it is focused primarily on solving the interoperability problem, and it has a palatable size, suitable for use as an example both in an industrial context of developing similar bridges between languages, as well as for code reading in software evolution education. Furthermore, CROSSOVER is designed to support programmers by abstracting away the dichotomy of data representation between BabyCobol and C.

The paper is structured as follows: Section 2 dives into the background and related work. In Section 3 we formulate the problem statement and pose a number of research questions, to which Section 4 proposes a solution. In Section 5 we describe the evaluation of the proposed solution. Finally, Section 7 discusses the strategy we used, concludes the paper and proposes future work directions.

2 Background

We focus on COBOL as our subject for investigating a legacy data philosophy, since it is one of the largest and by far the most used legacy programming language nowadays [16]. For practical reasons, we substitute COBOL with BabyCobol [45] which is significantly smaller yet following an identical data philosophy.

The choice for modern data philosophy falls on C [17]. The motivation is based on C being an immensely popular language itself, as well as a solid representative for the class of C-like languages which are prevalent in the software engineering practise today.

2.1 BabyCobol

BabyCobol, also known as “the software language engineers’ worst nightmare” [45], is an experimental language specifically designed with an intention to highlight challenges of implementing legacy languages. It has a tiny size (1 data type and 18 statements), palling in comparison to real legacy languages like COBOL (43 statements and 87 functions, most statements being split into up to 8 variants) [14] or IBM’s mainframe assembly language HLASM (3296 directives, instructions and mnemonics) [3, 44], which makes its implementations compact and focused. Yet, it is uncomfortable and perpetually challenging: for example, it combines Fortran-style computable GO TOs (where the name of the target label is a result of a runtime computation) with the COBOL ability to ALTER them (reassign an existing control flow transferring statement to another target at runtime), REXX-like error handling with SIGNAL (comparable to aspect weaving [19]) and non-reserved keywords inspired by PL/I (where variables are allowed to have names identical to keywords).

BabyCobol was meant as a playground for experimenting with various techniques of compilation, software analysis

and transformation, both for educational purposes and in the research context. Thus, instead of investing years [22] into building a parser for some extensive legacy language, and then a compiler covering all variations and subclauses of one of its dialects, we have built a minimal implementation of the language core and extended it with one new CALL statement (for calling external programs written in C or BabyCobol) and one subclause of the PROCEDURE DIVISION (for specifying return values of the program).

BabyCobol’s data philosophy is a strict subset of that of COBOL. The language offers only one primitive type of data: the PICTURE. A field (variable) refers to a single piece of information of a fixed size. Fields are defined by a template, determining how the system interprets the data stored in them. For example, a field can be defined with the template "XX99", which specifies that it is four bytes long and may contain two alpha-numeric characters followed by two decimal digits. The exact semantics will be explored in Section 2.5.

Similarly to COBOL, each BabyCobol program consists of divisions. In BabyCobol there are three divisions: the IDENTIFICATION DIVISION, DATA DIVISION and the PROCEDURE DIVISION.

The IDENTIFICATION DIVISION is mandatory for each BabyCobol program and contains identifying information of the file. This division is made up of name-value pairs and may contain clauses like PROGRAM-ID, AUTHOR, DATE-WRITTEN and DATE-COMPILED, which are typically found in real COBOL programs. In BabyCobol, programmers are not limited by predefined clauses and can extend this division with their own keys. An IDENTIFICATION DIVISION is always the first division in a (Baby)Cobol program.

All explicitly defined variables in a BabyCobol program are declared in the DATA DIVISION. This division is optional: in case of its absence, the program is restricted to the use of constants and implicitly typed variables. If present, the DATA DIVISION is found after the IDENTIFICATION DIVISION and above the PROCEDURE DIVISION. Data structures in BabyCobol are hierarchical, such that fields can be declared at the top-level or as part of a record. A record is a composite data structure which may contain fields and/or other records. Records and fields can also be turned into arrays with the OCCURS clause and a number representing the desired fixed length. In COBOL, DATA DIVISIONs are split into sections, but this is abstracted from in BabyCobol.

The PROCEDURE DIVISION follows the other divisions, and is the last division in a BabyCobol program. It consists of paragraphs, which are comparable to functions in C. However, unlike C, BabyCobol executes statements sequentially from the start of the division, fall through from the end of one paragraph directly to the start of the next paragraph below it until the end of the file or a STOP statement is encountered.

In Table 1 we align the terminology to describe elements and concepts of both BabyCobol and C.

2.2 Foreign Function Interface

Foreign function interfaces (FFIs) are mechanisms that allow one *host* language to invoke procedures written in another *guest* language, using the call semantics of the host language. An FFI bridges the differences in calling conventions and semantics between the two languages. Many modern languages and industrially strong compilers provide FFIs (we will give some examples in Section 6), and there exist libraries like `libffi` [9] which facilitate developers in making their own FFI. Since FFIs make it possible to seamlessly integrate differently typed code, their generalisation can be classified as a form of gradual typing [5, 37]. Practical implementations are often substantially simpler since legacy languages offer only very basic non-parametric forms of polymorphism, if any at all.

2.3 Application Binary Interface

An Application Binary Interface (ABI) is a specification which defines object and executable file formats, as well as calling conventions between applications and the platform on which they run. By specifying how compilation units should be linked together, ABIs play a crucial role in FFIs. The source files for the host and guest language are often processed by different compilers. Linking together these compilation units requires them to adhere to the same ABI. Additionally, at runtime the parameters and return values of procedures must follow the same memory/register layout defined in the ABI. The data types of the passed parameters are not defined in the ABI, and bridging them requires specific steps in the case of BabyCobol and C. We discuss this further in Section 2.5, and expand upon this in Section 4.2.

The CROSSOVER compiler targets the UNIX System V ABI specification [43], which came about in 1998 and went through several minor backward-compatible changes until 2013. It consists of two parts; a generic part (gABI), which is the same for all implementations of System V, and a processor-specific supplement (psABI). The System V ABI comprises the Executable and Linking Format (ELF) [36], which defines the format of object files and how these are linked together to form executable files.

Because ABIs work at a very low level of abstraction, and there exist many different computing platforms [12], writing an application for any specific ABI greatly limits its portability. However, since CROSSOVER is an LLVM-based compiler, it does have a certain degree of portability by construction. CROSSOVER is designed to link ELF objects in its compilation pipeline. Because the ELF format is part of the generic System V ABI (gABI), and LLVM has backends for multiple processor architectures (psABIs), the compiler can target multiple System V-derived platforms (notably modern Linux).

Table 1. Terminology across language domains

Term	BabyCobol	C
variable	field or record	any non-constant of a basic data type or struct
string	a field containing alphabetic or alphanumeric data	a C-style string (char*)
number	a field containing numeric data (with a PICTURE clause containing only {9,S,Z,V})	char, int, long, float, double (any modifications)
procedure	a paragraph	a function

2.4 Standard Libraries and Language Runtimes

A *standard library* [10] can contain language functionality not directly built into the language as a native construct. Functionality from standard libraries can either be expanded into the source code as a macro, or they can be compiled in their entirety and linked with the program binary.

A *language runtime* [42] is a compiler component that needs to be present during the execution of a compiled program written in the language for the executable to function properly. A language runtime could take the form of one or more standard libraries or a virtual machine.

The BabyCobol standard library (Section 4.2) is a central component of our solution and is a hybrid between a standard library and a language runtime.

2.5 BabyCobol vs. C

There are many differences between BabyCobol and C, since they represent different schools of thought and directions of programming language evolution. But despite obvious syntactical differences, there are some similarities to be found. For example, both are compiled imperative languages. They are procedural, thus allowing for the definition of reusable paragraph or function blocks. In BabyCobol, these procedures may have a certain degree of isolation from the main scope of the program, up to being solely dependent on the procedure parameters passed to it by a caller. Similarly, C functions may be solely dependent on their function parameters.

There is no functional purity in either language, but besides minor side effects on the global state, we can identify a class of *isolated* procedures. Such isolated procedures may still have side effects in a system through file or database access, but from a functional perspective they are idempotent.

These characteristics offer a good starting point in developing interoperability between the two languages. All parameters we send over an FFI when invoking a procedure must be available on the other side. If the invoked procedure returns a result, that must be sent back over the FFI and be available on the invoking side.

In BabyCobol and C, we have different ways of representing data. We refer to the representation of data, and the constraints on that representation, as its data type. BabyCobol

only has one inclusive data type constrained by a PICTURE clause. Essentially it gives a pattern to which the possible values have to conform, where each symbol mandates the contents of one letter or digit. Patterns are constructed freely (with some minor variations of what combinations trigger a warning or an error, which varies wildly across COBOL dialects — BabyCobol allows any combination to simplify this issue), but there are some that are more useful and thus more commonly used. For example, S99 means a signed two-digit numeric value, 999V99 defines a fixed point decimal with five digits, two of which define the hundredths and the others the integer part, and XXXXX allows any five-character string.

COBOL has some more complex rules for representing enumerations and redefining the same memory area with alternative representations, but BabyCobol only has two kinds of data entries: a primitive one (a *field* defined with a PICTURE clause as explained above) and a composite one (a *record* that combines several fields or other records). Either of those can be *occurring*, essentially turning them into arrays. In the absence of the REDEFINES and FILLER clauses, all data structures are fairly straightforward and can be represented by a tree structure.

2.5.1 Fields and Basic Data Types. C is similar to (Baby)COBOL in the sense of having byte representation aware types, but it comes with predefined basic data types such as *int* which directly represent data in memory, and has structs which aggregate such basic data types and other structs. Unlike BabyCobol, structs in C are named and reusable by those names. The most significant dissonance in data representation may occur at the level of fields in BabyCobol and basic data types in C. In BabyCobol, fields are position-based, which used to align better with its business-oriented nature and punchcard-oriented implementation. For instance, we may define a field to consist of three decimal digits. This allows for an intuitive description of what data a field holds. The programmer does not need to be concerned with the bit-width of their data type, nor with the intricate concepts of floating-point numbers and rounding errors. BabyCobol's fixed-point numbers are incompatible with C's floating-point data types.

In C, we find that each basic data type represents only a single type of data. In contrast with BabyCobol, C does not let the programmer define a type per character. Instead, its basic data types describe a region in memory with a certain bit-width. Where BabyCobol can distinguish several data types in one Field by specifying the type of each individual character, C's basic data types treat the specified region in memory as one homogeneous data type. There are some obvious "easy" cases such as S99 from the example above that are representable as signed `char` in C (with some additional machinery for decimal overflows), but the general case is complicated and has no universal performant solution.

2.5.2 Records and Structs. Looking at composite data types, BabyCobol records have much in common with C structs. They can define a type by the composition of other types. This composition can, in BabyCobol, always be modelled as a tree, where a field is always a leaf vertex. In C, a struct may contain a reference to any declared type, even its own type. Modelling structs as graphs, we can form trees in C, much like in BabyCobol, but we may also create graphs containing loops. Therefore, we can not model all C structs as BabyCobol records. For interoperability to work, we must find a set of composite data structures that work in both languages. For BabyCobol, the set of possible composite data structures S_b is restricted by $\forall d \in S_b. isTree(d)$. The set of possible composite data structures in C S_c is only "restricted" to graphs: $\forall d \in S_c. isGraph(d)$. It follows that $S_b \subset S_c$. Therefore, we can achieve interoperability for a parameter p over our FFI iff $p \in S_b$:

A data structure is valid under the FFI if and only if it can be modelled as a tree.

All BabyCobol records are valid under this assumption, but not all C structs are.

One thing that should be noted is that BabyCobol records do not define a named type but a variable of a certain anonymous structure. In this sense, they are similar to unnamed structures in C. The type of a record can however be used to define another variable of the same structure through `LIKE` clauses. One crucial role of data types in strongly-typed languages such as BabyCobol and C is for the compiler to determine what operations are allowed on certain data. For example, in COBOL and BabyCobol, it is not allowed to *add* two strings together (see [Section 4.5](#)). A C compiler may provide warnings or fail on specific implicit (automatic) type conversions.

3 Problem Statement

From the related work and background we identify certain challenges which must be overcome to achieve interoperability between BabyCobol and C.

Functions need to be linked across compilation units. The challenge is matching symbols and implementing call semantics at the ABI level.

BabyCobol's `PERFORM` statement does not have the linguistic means to invoke procedures in other compilation units. It lacks the possibility of specifying arguments and return values. Furthermore, the BabyCobol language must be extended in order to support the different parameter forms that are possible in C function definitions.

Data type constraints and data integrity between BabyCobol and C must be aligned and respected across the boundaries of the FFI. This requires the marshalling of variables when crossing the bridge between the two languages.

To achieve interoperability, there must be mechanisms provided for the reliable manipulation and evaluation of data beyond the language boundary, consistent with unified semantics.

We pose the following research questions:

1. *How can a Foreign Function Interface between BabyCobol and C be implemented?*
2. *How can the key differences in data philosophy between BabyCobol and C be addressed?*

4 Design and Implementation

CROSSOVER is a BabyCobol implementation that is designed specifically with interoperability between BabyCobol and C in mind. Alongside BabyCobol source files, users can provide compiled C object files to the compiler, providing the procedures they wish to invoke in BabyCobol.

As mentioned, C and BabyCobol vary greatly in how they handle data. Crossover's BSTD runtime library helps to bridge the gap between the two languages. The library contains a C implementation of the native BabyCobol data types and has the functionality to create, convert to and -from, and modify variables of these data types. The BSTD library plays a central role in this implementation: it is used not only for achieving the interface with C but is also extensively used for compiling pure BabyCobol programs. The compiler uses the interface with C to generate calls to the library whenever data needs to be created or modified, making this library effectively the language runtime. The design of the BSTD is discussed further in [Section 4.2](#).

Besides the BSTD, the compiler also provides the option to automatically generate C headers from BabyCobol identification divisions. These files allow the user to synchronise data structures between the two languages.

The implementation relies heavily on two existing compiler construction tools: ANTLR [28] and LLVM [21]. ANTLR is used for generating a parser and a parse tree visitor from the provided BabyCobol grammar definition, and LLVM is used as a backend for code generation. The compiler is written in C++. This language was chosen due to it being the native language of the LLVM compiler backend, ANTLR allowing for the generation of parsers in this language, and it being interoperable with C code, allowing us to incorporate parts of the BSTD library into the compiler.

4.1 Toolchain Overview

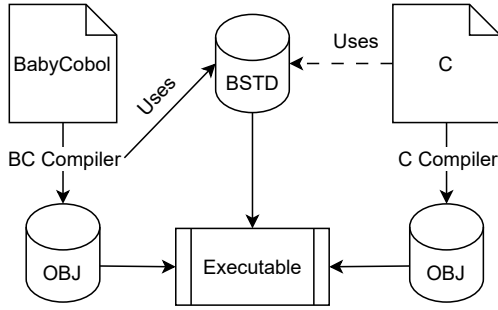


Figure 1. CROSSOVER Compilation Overview

Figure 1 visualises the compilation of a mixed BabyCobol/C codebase using CROSSOVER. During the compilation of a BabyCobol program, a parse tree is generated by ANTLR. When visiting a node of that parse tree, LLVM is used to generate the appropriate instructions for that node. We leverage the fact that calls to external procedures get resolved during linking to create the FFI. This approach is also used to generate calls to the BSTD library for the runtime creation, conversion and modification of BabyCobol variables. This eliminates the need of having to implement this functionality again on a compiler level and guarantees that operations on BSTD data types from C code behave exactly the same as they do in native BabyCobol. When the compilation of the BabyCobol sources is done, the compiler invokes clang to link the newly created object files, the BSTD runtime library, and the C object files into an executable.

4.2 BabyCobol Standard Library: BSTD

The BSTD is the core of our approach to interoperability. Its key responsibility is resolving the differences between fields in BabyCobol and basic data types in C. It can also guarantee data integrity by allowing for the use of the same operation semantics on its data types in both BabyCobol and C (see Section 4.5). BabyCobol has only one data type — the *Field*. The BSTD contains two data types: the *Number*, defined in Table 2, and the *Picture*, shown in Table 3. The reason for this distinction is that certain operations are exclusive to BabyCobol Fields which represent numbers. Specialising a data type for Numbers and *everything else* allows for type checking at compile time whether an operation may be performed on the operands' data types. It also allows for a more specialised representation of data, simplifying its modification.

The fields in Table 2 and 3 are not packed. The C language specification [17] dictates that the size of Numbers and Pictures must both be 24 bytes due to inserted padding. This padding is inserted at the end of the structs and does not affect the offsets shown in Table 2 and Table 3.

The values stored in the BSTD data types create a complete representation of their PICTURE clause specification in BabyCobol.

The BSTD Picture contains an array of bytes, a mask character array and a length. The length property describes the dimensions of the mask- and byte array. For any Picture p with byte array B_p , mask M_p and length l , the lengths of these arrays are equal: $l = |B_p| = |M_p|$.

Masks are character based, and the i^{th} mask character applies to the i^{th} byte. A masking function $mask(byte, mask)$ creates an interpretation of a byte given its corresponding mask character, such that for a Picture p value $v_p^i = mask(B_p^i, M_p^i)$. For example, $mask(0, 9) = 0$ and $mask(65, X) = A$. Truncating may occur for values outside the permitted range for a mask character; $mask(65, 9) = 5$.

The BSTD Picture data type can be converted to a C-style string by iterating over each byte-mask character pair and mapping it to a character using the *mask* function.

The BSTD Number data type stores a base value, a scale and the *isSigned* property. Additionally, it stores the permitted length of the Number and the *positive* property. The data type can represent integer and fixed-point decimal values. Converting these data types to C integers or floating point numbers is done at runtime. The conversion function for any Number n with base value b and scale s , is $v(n) = sign(n) \cdot b \cdot 10^{-s}$, where $sign : \text{Number} \rightarrow \{-1, 1\}$.

Conversely, C basic data types can not trivially be converted to BSTD Numbers because they lack the constraining properties inherent to the Number type. Through this lack of information, we are restricted to the *assignment* of C basic data types to instances of BSTD Numbers which contain the otherwise missing constraints. This forces the programmer to be explicit in defining these constraints.

C integers are assigned to Numbers using the following formula: $b = \left(|k| - \left\lfloor \frac{|k|}{10^{l-s}} \right\rfloor \cdot 10^{l-s} \right) \cdot 10^s$, where an integer k is assigned to a Number with the base value of b , length l , and scale s . Additionally, we set the *positive* property of the Number such that $sign(n) = -1 \iff isSigned(n) \wedge k < 0$. Note that assigning a C data type does not affect any of the other properties of the Number.

BSTD Numbers are representations of integer or fixed-point decimal numbers. They are equivalent to BabyCobol Fields with a PICTURE clause consisting of the characters 9, S, Z and V. With the set of characters $N = \{9, S, Z, V\}$, for any mask M , $isNumberMask(M) \iff \forall c \in M. [c \in N]$.

Because Number data structures are distinct types from Picture data structures, we do not need to specify a mask when we construct one — all mask characters are in N and are represented in the Number data structure and their semantics are fully encapsulated in the struct.

Table 2. The Number Data Structure

Offset	Size	Field	Description
0x00	8 bytes	value	The base value of the Number.
0x08	8 bytes	scale	The scale of the Number.
0x10	1 byte	length	The length of the Number.
0x11	1 byte	isSigned	Set if the Number is signed, unset if not.
0x12	1 byte	isPositive	Set if the Number is positive. -0 and +0 are treated the same.

Table 3. The Picture Data Structure

Offset	Size	Field	Description
0x00	8 bytes	bytes	A pointer to the bytes array belonging to this Picture.
0x08	8 bytes	mask	A pointer to the Picture clause specification string.
0x10	1 byte	length	The length of the Picture.

Similarly, the CROSSOVER compiler will represent any BabyCobol field with a PICTURE clause containing non-number mask characters by a Picture data type. For any field with mask M , $isPictureMask(M) \iff \exists c \in M. [c \notin N] \iff \neg isNumberMask(M)$. Circling back, Numbers are numbers, and Pictures are *everything else*.

4.3 Linking

We compile both C and BabyCobol code to ELF object files. After compilation, we will have lost many details about the original implementation language of the particular object file. Both CROSSOVER and clang do not do name mangling and list functions as symbols by their name. We link these object files without any knowledge of their implementation languages, and thus we can use clang to opaquely create an executable binary including originally both C and BabyCobol functions. This step can be extended to any language which compiles to an ELF object file (see Section 4.6).

During the compilation of BabyCobol source files the compiler checks if any internally unknown CALL target procedure exists in the externally compiled object files. This is done by using the nm tool from the GNU Binutils package [7] to read the symbols in the object files. The nm output is then parsed to construct a symbol table of procedure names found in the external object files.

To link C objects and BabyCobol objects, we must have compiled them both. This creates a problem: How do we get the compiler to understand data structures defined in the other language's source? It seems to be the setup for a

chicken-and-egg scenario. To compile one, the other must have already been compiled. To address this, we offer two tools:

- The BSTD library makes BabyCobol Fields accessible from C. It exposes a struct-representation of Fields and utility functions that allow for converting to and from C basic data types. To a programmer using these utility functions, the BSTD guarantees that BabyCobol semantics apply to all operations performed on such data types. More on this in Section 4.2.
- There is an integrated tool which parses BabyCobol source files and extracts their defined data structures from their DATA DIVISIONs. It then outputs these data structures as C structs in a header file. This header file can be included in C source code and used like any other struct. We can then build and compile against the BabyCobol source code using these data types.

4.4 Language Extensions

We introduce four optional modifiers as extensions to the BabyCobol specification, which allow for more fine-grained control over the way in which parameters are passed over the FFI:

- BY VALUE specifies that the parameter must be copied before being passed over the FFI. Changes made to the copy do not change the original data structure.
- BY REFERENCE specifies that the parameter must be passed over the FFI as a reference to that data structure. The invoked function has direct access to the parameter, and changes on the other side of the FFI are reflected in the data structure.
- AS PRIMITIVE specifies that the parameter or return value must be passed as a C basic data type. We distinguish three basic data types: the integer, the double (floating point), and the character pointer (C-style string). This modifier may only be applied to Fields.
- AS STRUCT specifies that the parameter or return value must be passed as a C struct. This is a predefined BSTD data type for Fields and a generated C struct for Records.

The programmer can choose to specify one of the leading modifiers (either BY VALUE or BY REFERENCE) and/or one of the trailing modifiers (either AS PRIMITIVE or AS STRUCT), as seen on Figure 2. The BY VALUE and BY REFERENCE modifiers are applied to all following parameters until a new modifier is encountered. Conversely, the AS PRIMITIVE and AS STRUCT modifiers are applied to all preceding parameters until a new modifier is encountered. If no optional modifiers are present the implicit behaviour for Fields is BY VALUE and AS PRIMITIVE, respectively. For Records, these defaults are BY VALUE and AS STRUCT, since being a composite data structure, records cannot be mapped to any C basic data type.

This BabyCobol language extension allows a programmer to create calls compatible with almost all possible function definitions in C. BY VALUE results in a non-pointer parameter, whereas BY REFERENCE results in a pointer parameter. Similarly, AS PRIMITIVE results in a C basic data type and AS STRUCT in a BSTD data type or struct. The compiler also ensures the correct marshalling during invocation and re-entry as shown in Figure 3. At the start of a CALL the BabyCobol data is marshalled to the appropriate format for the function based on the parameter modifiers. Upon re-entry, the parameters are marshalled again, as well as the return value. Parameters passed BY REFERENCE can be modified on the C side, and variables defined in the BabyCobol DATA DIVISION must be updated to reflect their new values. In the case of a BY REFERENCE and AS PRIMITIVE clause, the marshalled parameter values need to be inversely marshalled and assigned to their original field. Similarly, return values may need to be marshalled to be assigned to their target Field or Record as defined in the DATA DIVISION.

4.5 Data Type Constraints and Data Integrity

Data defined using CROSSOVER obeys the type constraints that are defined in the BabyCobol specification [45]. This means that the format of each variable has to abide by its description defined in the data division of a BabyCobol file. For instance, if a user tries to MOVE the value 42 to a variable defined as 9V99, the value of the variable will be set to 2.00. Likewise, when a variable is defined with the PICTURE clause AX and the user attempts to MOVE the string "HI!" to the variable, it will be set to "HI" (Note the missing exclamation mark).

While BabyCobol does not specify a limit on the number of characters of a PICTURE clause, our implementation currently limits the length of a Number to nine digits, and the

length of a Picture to 255. The length of Numbers is limited because our implementation uses 64-bit integers to store the properties of the Number struct. The length of a Picture is also limited due to the usage of an 8-bit integer to store the length property.

Because the BSTD is used by CROSSOVER itself, using this library in C guarantees that the implementation of operations on Numbers and Pictures is one and the same in either environment. This provides exact BabyCobol data semantics to C, allowing for safe manipulation of BabyCobol variables within C programs.

The CROSSOVER compiler will adhere to the invariant $\forall z \in F. [isNumber(z) \iff isNumberMask(M_z)]$, where F is the set of all possible Fields in BabyCobol, and M_z is the mask of Field z . This means that any Field with a numeric PICTURE clause will be instantiated as a Number data type.

Outside the CROSSOVER compiler, the BSTD *does* allow for an instance of the Picture data structure to be created with a mask M such that $isNumberMask(M)$. This does not have a negative impact on data integrity, as the type system will prevent Pictures from being used where a Number is expected. Because some operations are exclusive to Numbers, Picture instances with a mask M such that $isNumberMask(M)$ cannot be used as parameters to arithmetic functions because of a mismatch in type.

On re-entry CROSSOVER checks if the passed Picture still has the same length and mask, as this can never be changed. Similarly, the length, scale and isSigned properties of a Number should never be changed and are checked on re-entry.

4.6 C as a Bridge

The C language is a particularly interesting candidate for a language to interface with because many other languages also provide an interface with it. One could use C as an intermediate language to invoke functionality written in a third language using a wrapper and calling that wrapper over the FFI that CROSSOVER provides. This process is illustrated in Figure 4.

Considering the two data types the BSTD provides, the FFI can be extended to any language that can interpret the data structures in Table 2 and 3, and that has a compiler that can create ELF object files with non-mangled function names as linkable symbols.

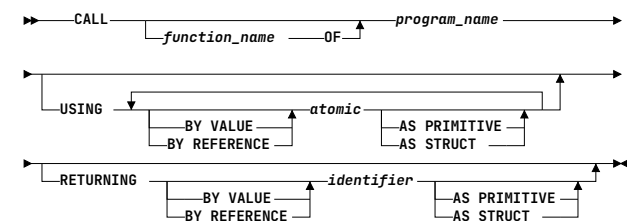


Figure 2. The railroad diagram of the CALL statement, including extensions

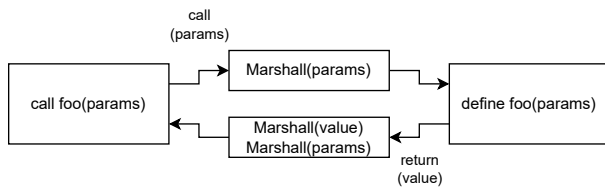
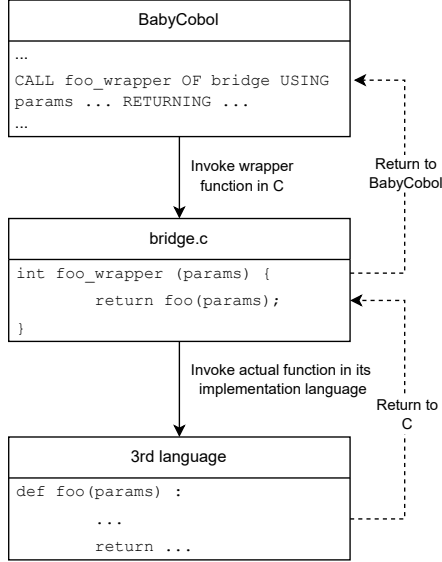


Figure 3. Marshalling When Invoking a Foreign Function

5 Evaluation

In this section, we present the evaluation of our proposed solution, which consists of three parts: unit testing of the BSTD library, performance analysis of the FFI and the demonstration of running examples showcasing how the interoperability of CROSSOVER can be used.

**Figure 4.** Using C as a Bridge

5.1 Testing

To evaluate the stability of our solution, we thoroughly tested the BSTD library, the central piece of our architecture. This was done by writing unit tests using the techniques of equivalence class partitioning and boundary value analysis [15].

Equivalence class partitioning is a method of testing where the inputs of the tested functions are divided into valid and invalid equivalence classes. Under the assumption that each member of an equivalence class would be processed in the same way by the function, one would only need to write one test per equivalence class. Boundary value analysis expands upon the idea of equivalence class based testing by introducing test cases directly on, above or below the boundaries of equivalence classes. The rationale for these additional test cases is that bugs usually exist on and around the boundaries of the equivalence classes.

Table 4 illustrates how equivalence class partitioning was performed for the function `bstd_number_is_integer`. This function was tested by creating cases and creating inputs combining as many valid equivalence classes as possible. After these cases, additional cases are added that test the boundary values. Since behaviour for values outside the valid bounds is undefined, no test cases exist. Only behaviour within valid bounds is tested.

These methodologies helped us to validate the BSTD library and find and fix bugs in our implementation. Some examples of bugs found during our testing include:

- Mismatched types, for example, when a large `int64_t` value is used as an input for a function that takes a 32-bit `int` causing the `int` to overflow.
- Insufficient floating-point value precision; while the default six decimal digits of precision of C is sufficient

Table 4. Equivalence class partitioning for function `bstd_number_is_integer`

Condition	Valid	Invalid
Value of value	$0 \leq \text{value} \leq 999999999$	$\text{value} < 0$, $\text{value} > 999999999$
Value of scale	$0 \leq \text{scale} \leq 9$	$\text{scale} < 0$, $\text{scale} > 9$
Value of length	$1 \leq \text{length} \leq 9$	$\text{length} < 1$, $\text{length} > 9$
Value of isSigned	$\text{isSigned} = \text{true}$, $\text{isSigned} = \text{false}$	
Value of positive	$\text{positive} = \text{true}$, $\text{positive} = \text{false}$	
Relation	$\text{isSigned} = \text{true} \wedge \text{positive} = \text{false}$, $\text{isSigned} = \text{false} \wedge \text{positive} = \text{true}$, $\text{isSigned} = \text{true} \wedge \text{positive} = \text{true}$	
		$\text{isSigned} = \text{false} \wedge \text{positive} = \text{false}$

to pass most of our BabyCobol Number type to C string tests, it proved insufficient once we tested the boundary of the number of decimals a BabyCobol Number could have.

Currently, our test suite consists of 261 tests, covering 87.8% (200/228) of the lines and 91.7% (22/24) of the functions in the BSTD library. The gaps in the coverage are internal auxiliary functions that are not exposed for use outside of the BSTD library.

5.2 Performance Evaluation

In this subsection we evaluate the overhead created by the different permutations of argument modifiers in the `CALL` statement. We created a set of four test programs — one for each possible permutation. Each program runs a loop of 100 million iterations, calling a C function with an empty body (to maximise the overhead) with a single argument under one set of modifiers. We took measurements on the execution time, the results of which are plotted in Figure 5. To obtain a statistical mean, each program was run back-to-back 100 times.

Figure 5 shows that the combination of modifiers {BY REFERENCE, AS PRIMITIVE} takes the longest time to execute. This is in line with our expectations, and can be explained by the fact that when these modifiers are used, the compiler inserts code for marshalling before the reference of the argument is passed to the C function. During the re-entry to BabyCobol the inverse of this happens: the C primitive is marshalled back into the original BabyCobol variable. This ensures that changes to the marshalled argument on the C side are reflected in the original variable. The insertion of marshalling and unmarshalling code has a performance overhead of 40% as compared to the fastest running variant ({BY REFERENCE, AS STRUCT}).

With the combination {BY VALUE, AS PRIMITIVE}, the compiler also inserts code to marshal the argument to a C primitive. However, with the BY VALUE modifier, changes

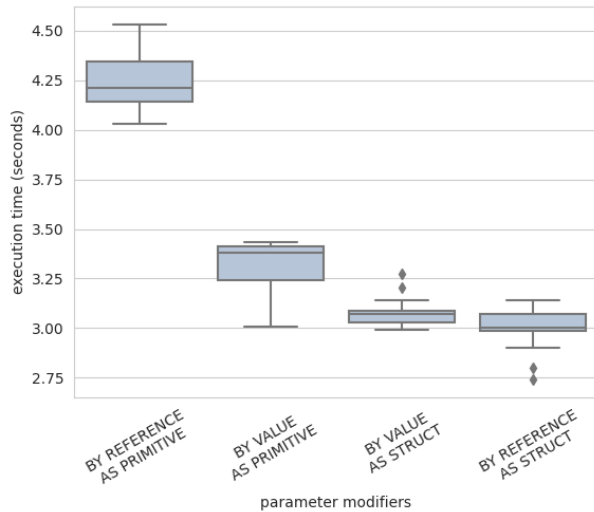


Figure 5. Result of the FFI Performance Analysis

to the argument should (and can) not be reflected upon re-entry. Inserting marshalling code only once implies a lesser performance overhead of **10%**.

The cases {BY VALUE, AS STRUCT} and {BY REFERENCE, AS STRUCT} are both faster than the previous two cases because no marshalling is involved. The parameter is passed to the C function as a copy of or reference to a BSTD structure respectively, which is its internal representation in CROSSOVER. There is a slight performance penalty to passing value copies, it being **2%** percent slower than passing references.

5.3 Demonstration

This subsection features three examples that demonstrate how the new CALL statement can be used for BabyCobol-C interoperability.

5.3.1 Calculating a Square Root. The first demonstration is a simple program that calculates the square root of a fixed-point number. The BabyCobol code is shown in [Listing 2](#). The program defines the variables *X* and *Y* as fixed-point numeric values of two integer digits, and one decimal digit. In the PROCEDURE DIVISION, *X* is first assigned the value 12. Then, the C standard library function *sqrt* is called, passing *X* as the argument. No leading modifier is specified, so the default BY VALUE semantics apply. The AS PRIMITIVE modifier dictates that the argument is to be marshalled into its C basic data type counterpart. The *sqrt* function in C accepts a double-precision floating point parameter. It returns the square root of this parameter. In the CALL statement, the RETURNING clause specifies that the result of the *sqrt* function is a C basic data type which should be marshalled into

the variable *Y* (respecting *Y*'s type constraints). Lastly, the program DISPLAYs the value of *Y*, which shows *03.4*.

```

1 IDENTIFICATION DIVISION.
2   PROGRAM-ID. "radical".
3 DATA DIVISION
4   01 X PICTURE IS 99V9.
5   01 Y PICTURE IS 99V9.
6 PROCEDURE DIVISION.
7   MOVE 12 TO X.
8   CALL sqrt USING X AS PRIMITIVE
9       RETURNING Y AS PRIMITIVE.
10  DISPLAY "The square root of " X " is " Y.

```

Listing 2. Calculating a Square Root

5.3.2 Scanner Example. This example shows how a Baby-Cobol program can use the FFI to accept user input. The code can be seen in [Listing 3](#). First, a four digit numeric variable *READ* is defined in the DATA DIVISION. In the PROCEDURE DIVISION, the program then asks the user to enter a value of up to four digits and calls the *scanf* function from the C standard library. The *scanf* function accepts two parameters. The first is a formatting string, and the second is a character buffer. We supply the following two arguments:

1. A string literal with the modifier AS PRIMITIVE, specifying that it is marshalled and passed to *scanf* as the C equivalent of a string (char*).
2. The *READ* variable, marshalled to a char* is the buffer to which *scanf* writes. It is passed BY REFERENCE, such that changes to the parameter are reflected in the Baby-Cobol variable *READ*.

Upon re-entry, the contents of this buffer are aligned with the *READ* variable's type specified by its PICTURE clause. Finally, the program DISPLAYs this value back to the user.

```

1 IDENTIFICATION DIVISION.
2   PROGRAM-ID. "scanf_demo".
3 DATA DIVISION
4   01 READ PICTURE IS 9999.
5 PROCEDURE DIVISION.
6   DISPLAY "Please enter a number of up to four
7       digits: ".
7   CALL scanf USING "%4d" AS PRIMITIVE
8       BY REFERENCE READ AS PRIMITIVE.
9   DISPLAY "You entered " READ.
10  END.

```

Listing 3. Scanner in BabyCobol

5.3.3 Banking Application. The final example, [Listing 4](#) (in BabyCobol) and [Listing 5](#) (in C) illustrate a toy banking application and API where a certain amount of currency is withdrawn from a balance. The DATA DIVISION specifies the

variables *BALANCE* as a signed fixed point decimal number, *AMOUNT* as an unsigned fixed-point decimal number and *SUCCESS* as a single digit number. In the PROCEDURE DIVISION, the two MOVE statements first assign values to the *BALANCE* and *AMOUNT* variables. The CALL statement invokes the *withdraw* function of the *banking_api* with arguments *BALANCE* and *AMOUNT*, both AS STRUCT. These arguments are thus passed as BSTD number struct pointers. The *BALANCE* argument is passed BY REFERENCE. This specifies that changes to the argument in C code are reflected in the BabyCobol *BALANCE* variable. In Listing 5, a check is first done to ensure the *withdraw* amount is not greater than the *balance*. If it is, false is returned and no funds are withdrawn. If the balance is sufficient, the amount to withdraw is subtracted from the *balance*, and *true* is returned. Upon re-entry, the return value of the *withdraw* function is marshalled and assigned to the variable *SUCCESS*. A check is done to either inform the user that the *BALANCE* was insufficient, or that the withdrawal was successful, in which case the new *BALANCE* is displayed. Note that the BSTD functions used in C allow for data evaluation and manipulation which implement the exact BabyCobol semantics.

```

1 IDENTIFICATION DIVISION.
2   PROGRAM-ID. "withdraw".
3 DATA DIVISION
4   01 BALANCE PICTURE IS S99V99.
5   01 AMOUNT PICTURE IS 99V99.
6   01 SUCCESS PICTURE IS 9.
7 PROCEDURE DIVISION.
8   MOVE 42,50 TO BALANCE.
9   MOVE 3,50 TO AMOUNT.
10
11  CALL withdraw OF banking_api USING
12    BY REFERENCE BALANCE
13    BY VALUE AMOUNT AS STRUCT
14    RETURNING SUCCESS AS PRIMITIVE.
15
16  IF SUCCESS = 0 THEN
17    DISPLAY "Your balance (" BALANCE ") is
18      too low! No funds were withdrawn."
19  ELSE
20    DISPLAY "Withdrawal successful. Your
      balance is now " BALANCE
21  END.

```

Listing 4. BabyCobol banking application

6 Related Work

We have previously mentioned the existence of prior work on FFI. For example, CFFI [31] is a foreign function interface for Python that allows users to call C code. A similar FFI exists for R and allows for the use of C libraries in R code [1], and uFFI for Smalltalk, providing C foreign function interface for Pharo [29]. Some approaches are more formal, such as

```

1 #include <bstd/numutils.h>
2 #include <bool.h>
3
4 bool withdraw(bstd_number* balance,
5               bstd_number amount) {
6
7     if (bstd_greater_than(withdraw, balance))
8         return false;
9
10    bstd_subtract(balance, amount);
11    return true;
12 }

```

Listing 5. Banking API implementation

MiniML⁺ [20], for which Larmuseau and Clarke define secure operational semantics for combining a subset of ML with C. Others are more practice-driven, like Clasp [34, 35], which analyses C++ code and generates a Common Lisp interface with a performant garbage collector to be used with an LLVM backend. Languages like Julia [18] and Rust [32] have built-in foreign function interfaces which commoditise Fortran, C or C++ calls. MATLAB also has an extensive family of external language interfaces [23], allowing bidirectional integration with Fortran, C, C++, Java, Python and .NET languages.

On the legacy side, both traditional vendors as well as their competitors provide comprehensive integration functionality. For instance, IBM provides PL/I InterLanguage Communication (ILC) and Java Native Interface (JNI) [13], foreign function interfaces that allow C and Java code to be called from PL/I. Micro Focus has a C and C++ FFI in their commercial ACUCOBOL-GT modernisation portfolio, and interoperability between COBOL and Java and .NET [25]. Similarly, Raincode offers interoperability with C# in their compiler [30], and Fujitsu with Visual Basic in theirs [8].

The only open source implementation of COBOL, with or without FFI, available to the general public, is GnuCobol, also known as OpenCobol [26]. Essentially it compiles COBOL to C, and that fact can be used for interoperability purposes if one is sufficiently familiar with the internals of the compiler. Its FFI functionality is built on top of another open source library called libffi [9]. GnuCobol is an interesting subject of study for academics, but due to its (L)GPL licensing it often cannot be considered in a commercial setting. Additionally, GnuCobol is also essentially its own dialect of COBOL, so migrating to this compiler implies code changes.

Commercial compilers with a C-COBOL interface are sometimes based on external function prototypes — besides being proprietary, they expose many compiler internals to initialise, stop, manipulate runtime data, and leave data conversion to programmers to code explicitly. Unlike them, CROSSOVER brings COBOL semantics to C where desired,

with its bespoke runtime conversion between data representations across the language bridge without loss of data validity.

Deep considerations for COBOL data semantics are scarce in the academic literature, but occasionally discussed in smaller consortia like IFIP Working Groups. For instance, Andersson [2] provides an excellently professional overview of the data philosophy of COBOL and its implications on reverse engineering endeavours, providing solution sketches for particularly problematic features like implicitly modelled references between records, and internally unspecified fields. Data model discovery, as the process of extracting reusable conceptual data (meta)models from entangled mazes of partially duplicate PICTURE-based definitions, has also sometimes been described as a part of larger renovation packages like COBOL/SRE [6]. Such activities are often going way beyond interoperability and tend to include pragmatic transformations that are provably and obviously wrong in the formal sense, but “good enough” to aid modernisation efforts. For example, Ueda and Ohara from IBM proposed to merge records with similar data layout and somewhat similar field names [40].

In contrast to our FFI/ABI based solution, some take an alternative path with a CORBA-based approach, which relies on a middleware architecture for communication among different languages and platforms. On that path, examples of relevant standards and specifications could be CORBA [27, 1991–2021] or SOAP [41, 1998–2007]. One can also design solutions based on RESTful APIs or Remote Procedure Calls (RPC). A compiler-based solution like the one we proposed, might seem more complex to implement, but it eliminates the need for complex middleware layers, potentially reducing overhead and enhancing performance, while providing greater flexibility and allowing for fine-tuned control over the interaction between COBOL and C components.

7 Concluding Remarks

COBOL/C interoperability has not been satisfactorily solved in decades of language coexistence, despite industrial need, which shows the issue being at least somewhat challenging. In this paper, we have examined how interoperability between the languages BabyCobol and C could be achieved. The resulting CROSSOVER compiler can serve as a guiding blueprint for developers implementing similar bridges in the context of real large legacy languages such as COBOL or PL/I. Its implementation is released as open source for the sake of reproducibility [33].

There were two research questions to be answered: the first one on how a foreign function interface between BabyCobol and C could be realised, and the other one about how the key data differences between BabyCobol and C could be addressed.

We address the first question by proposing a solution using LLVM as the framework for our compiler, allowing us to generate ELF object files. This is the same binary format that C code is usually compiled to on Linux systems. Having the source code of both languages compiled into ELF files allows them to be linked into one executable. Both C and our implementation of BabyCobol (supported by our language extensions) allow for external procedure calls. This allows for calling foreign functions that are defined in other compilation units, including those originating from other languages.

To address the second research question, we proposed a solution where data creation, modification and marshalling within CROSSOVER is handled by a library that functions both as a standard library and a runtime library; the BSTD. The BSTD is used on both sides of the communication, either by developers directly or by the compiler generating appropriate glue code. On the side of C, the library introduces BabyCobol data types and semantics to the C environment and provides the developer with tools to create and modify variables of these data types, as well as functionality to convert data into instances of native C data types. On the BabyCobol side, the BSTD is implemented in such a way that CROSSOVER makes use of the foreign function interface to construct function calls to the runtime library. The functions are then invoked at runtime whenever data is created or modified. This bridges the gap between BabyCobol fields and records, on one side, and C basic data types and structs, on the other.

CROSSOVER is a minimal implementation of BabyCobol, which mainly implements features that directly impact interoperability, such as data definition, manipulation, and function invocation. By limiting the scope of the implementation, we could focus on the challenges of implementing interoperability instead of the rest of the language, accelerating the development of this solution.

The strategy described in this paper, is meant for interoperability between BabyCobol and C. The concepts of BabyCobol which are important for interoperability (such as the data definitions in the DATA DIVISION and data exchange in the CALL statement), are a strict subset and simplification of their counterparts in COBOL. Hence, the strategy proposed in this paper could be applied similarly to achieve interoperability between COBOL and C. However, since COBOL has more features and higher complexity than BabyCobol, there may be issues that have not arisen in our project thus far that could come up when applying the interoperability strategy between COBOL and C. From our prior experience with COBOL, we foresee some minor additional implementation challenges dealing with REDEFINES and FILLER clauses and their interaction with the INITIALIZE statement. These challenges, albeit non-trivial, are purely engineering in nature, and no further research challenges should arise. For instance, when REDEFINES gives two alternative representations of

the same memory fragment, C can handle it with several structs and manipulating pointers to them.

Finally, an interesting research direction would be to apply the approach introduced in this paper to implement interoperability with other languages that compile to ELF object files. This would allow users to incorporate their code written in these languages without using C as a bridge as described in Section 4.6. The major challenge in undertaking this would be to write or generate a BSTD-like library for the target language to allow for marshalling and manipulating data at runtime.

Designing CROSSOVER required much careful consideration, data semantics alignment, and ample experimentation. We hope that opening its design contributes to the field of software maintenance by exposing the general public to the problem instead of keeping it in the ever shrinking legacy developer pool.

Acknowledgements

The authors express their gratitude to the GPCE programme committee members for their efforts and expertise in reviewing this document. We are especially grateful to Julia Lawall for shepherding the last steps. We also appreciate advice, encouragement and interest in this project from Raincode engineers Johan Fabry and Darius Blasband, as well as for the participants of the Formal Methods and Tools Colloquium on 1 June 2023 for their feedback.

References

- [1] Daniel Adler. 2012. Foreign Library Interface. *The R Journal* 4, 1 (2012), 30. <https://doi.org/10.32614/rj-2012-004>
- [2] Martin Andersson. 1998. Searching for Semantics in COBOL Legacy Applications. In *Data Mining and Reverse Engineering: Searching for semantics. IFIP TC2 WG2.6 IFIP Seventh Conference on Database Semantics (DS-7) 7–10 October 1997, Leysin, Switzerland*, Stefano Spaccapietra and Fred Maryanski (Eds.). Springer, Boston, MA, 162–183. https://doi.org/10.1007/978-0-387-35300-5_8
- [3] Volodymyr Blagodarov, Yves Jaradin, and Vadim Zaytsev. 2016. Tool Demo: Raincode Assembler Compiler. In *Proceedings of the Ninth International Conference on Software Language Engineering (SLE)*, Tijs van der Storm, Emilie Balland, and Daniel Varró (Eds.). ACM, Amsterdam, 221–225. <https://doi.org/10.1145/2997364.2997387>
- [4] David Cassel. 2017. COBOL Is Everywhere. Who Will Maintain It? <https://thenewstack.io/cobol-everywhere-will-maintain/>.
- [5] Giuseppe Castagna, Victor Lanvin, Tommaso Petruciani, and Jeremy G. Siek. 2019. Gradual Typing: A New Perspective. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–32.
- [6] A. Engberts, W. Kozaczynski, E. Liongosari, and J.Q. Ning. 1993. COBOL/SRE: A COBOL System Renovation Environment. In *Proceedings of Sixth International Workshop on Computer-Aided Software Engineering*. IEEE, Singapore, 199–210. <https://doi.org/10.1109/CASE.1993.634821>
- [7] Free Software Foundation. 1998. GNU Binutils. <https://sourceware.org/binutils/>
- [8] Fujitsu. 2015. FUJITSU Software NetCOBOL V11.0. Getting Started. Integrating COBOL Programs with Visual Basic. <https://software.fujitsu.com/jp/manual/manualfiles/m150009/b1wd3354/01enz200/b3354-00-03-02-00.html>.
- [9] Anthony Green, Richard Henderson, Josh Triplett, Zachary Waldowski, Landon Fuller, et al. 1996. libffi. <https://github.com/libffi/libffi>.
- [10] Dick Grune, Kees van Reeuwijk, Henri E. Bal, Criel J. H. Jacobs, and Koen G. Langendoen. 2012. *Modern Compiler Design* (second ed.). Addison-Wesley, New York, NY, USA. https://dickgrune.com/Books/MCD_2nd_Edition/
- [11] Travis Hartman. 2017. COBOL blues. Reuters Graphics, <http://fingfx.thomsonreuters.com/gfx/rngs/USA-BANKS-COBOL/010040KH18j/index.html>.
- [12] John L. Hennessy and David A. Patterson. 2019. *Computer Architecture: A Quantitative Approach. Appendix K: Survey of Instruction Set Architectures* (sixth ed.). Morgan Kaufman Publishers, Waltham, MA, USA.
- [13] IBM. 2019. Enterprise PL/I for z/OS Programming Guide Version 5 Release 3.
- [14] IBM Library. 1987. SX26-3721-05: VS COBOL II Application Programming Reference Summary, Release 4. IBM. https://publibz.boulder.ibm.com/cgi-bin/bookmgr_OS390/BOOKS/IGYR1101/CCONTENTS.
- [15] Burnstein Ilene. 2003. *Practical Software Testing: A Process-Oriented Approach*. Springer, New York, NY, USA.
- [16] Feargus Illingworth et al. 2022. 2022 Mainframe Modernization Business Barometer Report. Technical Report. Advanced. <https://modernsystems.oneadvanced.com/en/reports/modernisation2022/>.
- [17] ISO/IEC JTC 1/SC 22. 2018. ISO/IEC 9899:2018: *Information Technology—Programming Languages—C*. International Organization for Standardization. <https://www.iso.org/standard/74528.html>.
- [18] Julia Programming Language. 2023. Calling C and Fortran Code — The Julia Language. <https://docs.julialang.org/en/v1/manual/calling-c-and-fortran-code/>.
- [19] Ralf Lämmel and Kris De Schutter. 2005. What Does Aspect-Oriented Programming Mean to Cobol?. In *Proceedings of the Fourth International Conference on Aspect-Oriented Software Development (AOSD)*. ACM, New York, NY, USA, 99–110. <https://doi.org/10.1145/1052898.1052907>
- [20] Adriaan Larmuseau and Dave Clarke. 2015. Formalizing a Secure Foreign Function Interface. *LNC5 9276* (2015), 215–230. https://doi.org/10.1007/978-3-319-22969-0_16
- [21] Chris Lattner and Vikram S. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the Second IEEE / ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE Computer Society, San Jose, CA, USA, 75–88. <https://doi.org/10.1109/CGO.2004.1281665> <https://llvm.org>.
- [22] Vadim Maslov. 1998. Re: An Odd Grammar Question. <http://compilers.iecc.com/comparch/article/98-05-108>.
- [23] MathWorks. 2023. External Language Interfaces — MATLAB & Simulink — MathWorks. <https://www.mathworks.com/help/matlab/external-language-interfaces.html>.
- [24] Joris Mertens. 2020. IBM zSystems fundamentals: An introductory Q&A. <https://developer.ibm.com/articles/what-is-ibm-z/>.
- [25] Micro Focus. 2014. A Guide to Interoperating with ACUCOBOL-GT Version 9.2.5. <https://www.microfocus.com/documentation/extend-acucobol/925/GUID-0617E68F-C102-4A3B-9797-279F653777D7.html>.
- [26] Keisuke Nishida, Roger While, Edward Hart, Sergey Kashyryn, Ron Norman, Simon Sobisch, et al. 2002. GnuCOBOL: A free/libre COBOL compiler. <https://gnucobol.sourceforge.io>.
- [27] OMG. 2012. *Common Object Request Broker Architecture (CORBA) Specification* (3.3 ed.). Object Management Group. <https://www.omg.org/spec/CORBA/3.4/>
- [28] Terence Parr. 2023. ANTLR—ANother Tool for Language Recognition, release 4.12.0. <http://antlr.org>.
- [29] Guillermo Polito, Stéphane Ducasse, Pablo Tesone, and Ted Brunzje. 2017. *Unified FFI — Calling Foreign Functions from Pharo*. Pharo.org, CC-BY-SA. <https://books.pharo.org/booklet-uffi/>

- [30] Raincode. 2023. How Can I Migrate My COBOL Applications to the Cloud? <https://www.raincode.com/cobol/>.
- [31] Armin Rigo and Maciej Fijalkowski. 2018. CFFI 1.15.1 documentation. <https://cffi.readthedocs.io/en/latest/>.
- [32] Rust Programming Language. 2021. FFI — The Rustonomicon. <https://doc.rust-lang.org/nomicon/ffi.html>.
- [33] Ömer F. Sayilir, M. Aimé Ntagengerwa, and Mart H. van Assen. 2023. Crossover. <https://github.com/Crossover-Compiler/Crossover>.
- [34] Christian A. Schafmeister and Alex Wood. 2018. Clasp: Common Lisp Implementation and Optimization. In *Proceedings of the 11th European Lisp Symposium (ELS)*. European Lisp Scientific Activities Association, Marbella, Spain, Article 8, 6 pages.
- [35] Christian E. Schafmeister. 2015. Clasp — A Common Lisp that Interoperates with C++ and Uses the LLVM Backend. In *Proceedings of the Eighth European Lisp Symposium (ELS)*. ELS, London, UK, 90–91.
- [36] SCO Developer Network. 2014. System V Application Binary Interface. Chapter 4. <https://www.sco.com/developers/gabi/latest/ch4.intro.html>.
- [37] Jeremy G. Siek and Walid Taha. 2007. Gradual Typing for Objects. In *Proceedings of the 21st European Conference on Object-Oriented Programming (ECOOP) (LNCS, Vol. 4609)*. Springer, Berlin, Heidelberg, 2–27. https://doi.org/10.1007/978-3-540-73589-2_2
- [38] Harry M. Sneed. 1992. Migration of Procedurally Oriented COBOL Programs in an Object-Oriented Architecture. In *Proceedings of the Eighth International Conference on Software Maintenance*. IEEE, Orlando, FL, USA, 105–116. <https://doi.org/10.1109/ICSM.1992.242552>
- [39] Andrey A. Terekhov and Chris Verhoef. 2000. The Realities of Language Conversions. *IEEE Software* 17, 6 (2000), 111–124. <https://doi.org/10.1109/52.895180>
- [40] Yohei Ueda and Moriyoshi Ohara. 2014. *Refactoring of COBOL Data Models Based on Similarities of Data Field Name*. Technical Report. IBM.
- [41] W3C. 2007. *Simple Object Access Protocol (SOAP) (1.2 ed.)*. World Wide Web Consortium. <https://www.w3.org/TR/soap/>
- [42] David S. Wile. 2001. Supporting the DSL Spectrum. *Journal of Computing and Information Technology* 9, 4 (2001), 263–287.
- [43] Xinuos Inc. 2013. Developers | SCO Developer Network. <https://www.sco.com/developers/gabi/>
- [44] Vadim Zaytsev. 2020. Modelling of Language Syntax and Semantics: The Case of the Assembler Compiler. *Proceedings of the 16th European Conference on Modelling Foundations and Applications (ECMFA) 19* (July 2020), 22 pages. Issue 2. <https://doi.org/10.5381/jot.2020.19.2.a5>
- [45] Vadim Zaytsev. 2020. Software Language Engineers' Worst Nightmare. In *Proceedings of the 13th International Conference on Software Language Engineering (SLE)*. ACM, New York, NY, USA, 72–85. <https://doi.org/10.1145/3426425.3426933>

Received 2023-07-14; accepted 2023-09-03

Generating Conforming Programs with Xsmith

William Gallard Hatch

u0468220@utah.edu
University of Utah
USA

Pierce Darragh

pierce.darragh@utah.edu
University of Utah
USA

Sorawee Porncharoenwase

sorawee@cs.washington.edu
University of Washington
USA

Guy Watson

guy.watson@utah.edu
University of Utah
USA

Eric Eide

eeide@cs.utah.edu
University of Utah
USA

Abstract

Fuzz testing is an effective tool for finding bugs in software, including programming language compilers and interpreters. Advanced fuzz testers can find deep semantic bugs in language implementations through differential testing. However, input programs used for differential testing must not only be syntactically and semantically valid, but also be free from nondeterminism and undefined behaviors. Developing a fuzzer that produces such programs can require tens of thousands of lines of code and hundreds of person-hours. Despite this significant investment, fuzzers designed for differential testing of different languages include many of the same features and analyses in their implementations. To make the implementation of language fuzz testers for differential testing easier, we introduce Xsmith.

Xsmith is a Racket library and domain-specific language that provides mechanisms for implementing a fuzz tester in only a few hundred lines of code. By sharing infrastructure, allowing declarative language specification, and by allowing procedural extensions, Xsmith allows developers to write correct fuzzers for differential testing with little effort. We have developed fuzzers for several languages, and found bugs in implementations of Racket, Dafny, Standard ML, and WebAssembly.

CCS Concepts: • Software and its engineering → Software testing and debugging; Compilers.

Keywords: automated testing, compiler testing, fuzzing, random program generation, random testing

ACM Reference Format:

William Gallard Hatch, Pierce Darragh, Sorawee Porncharoenwase, Guy Watson, and Eric Eide. 2023. Generating Conforming Programs



This work is licensed under a Creative Commons Attribution 4.0 International License.

GPCE '23, October 22–23, 2023, Cascais, Portugal

© 2023 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-0406-2/23/10.

<https://doi.org/10.1145/3624007.3624056>

with Xsmith. In *Proceedings of the 22nd ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '23)*, October 22–23, 2023, Cascais, Portugal. ACM, New York, NY, USA, 14 pages. <https://doi.org/10.1145/3624007.3624056>

1 Introduction

The effectiveness of random testing, or “fuzzing,” is determined both by the chosen input generation strategy and the method used to detect failing tests, or *test oracle*. The generation or mutation of test cases using random bytes can in theory generate any test and therefore cover any code path, but typically exercises only “shallow” code in parsing and input validation stages of a program. Meanwhile, grammar- and type-aware generators can exercise “deep” code paths that pass validation steps. The test oracle of detecting crashes can be used with any test case generator, but can only find obvious errors such as memory or assertion violations, and not subtle semantic bugs. Property-based testing can find semantic bugs, but requires users to write invariant properties of test results or side effects, which is expensive.

The test oracle of interest for this paper is *differential testing* [17], where the same input is given to multiple implementations of a system—in our case, a programming language. If the multiple implementations are correct, then giving them all the same input program (and executing the returned output if the implementation is a compiler) should produce the same result. When there is a difference in program output, a bug has been found. While differential testing can find subtle semantic bugs without writing extra properties, there is a catch. If the input program relies on any behavior that is not guaranteed to be the same between multiple executions and multiple implementations of the language, differences in output do *not* necessarily indicate a bug. Therefore, differential testing requires programs that conform strictly to the language specification, as well as avoiding undefined, implementation-defined, or nondeterministic behavior. We call such inputs *conforming* inputs.¹

¹We considered other words to convey that a program is suitable for differential testing, such as the word *differentiable*. However, the word *differentiable* has other meanings that would make this confusing.

$$\begin{aligned}
\langle \text{int} \rangle &::= z \mid 0 < z < 100 \\
\langle \text{exp} \rangle &::= \langle \text{exp} \rangle + \langle \text{exp} \rangle \\
&\quad \mid \langle \text{exp} \rangle / \langle \text{exp} \rangle \\
&\quad \mid \langle \text{int} \rangle
\end{aligned}$$

Figure 1. The Grammar of the calc Language

One notable generator of conforming programs is Csmith [24]. Csmith successfully identified hundreds of bugs in mainstream C compilers, including LLVM and GCC. However, the development of Csmith took hundreds of person-hours and resulted in nearly 40,000 lines of code. Although practically any language could benefit from such a program generator, it is impractical for most language developers to write a variant of Csmith targeting their own language.

To ease the development of fuzzers that generate conforming programs, we created *Xsmith*. Xsmith is a domain-specific language (DSL), implemented as a Racket library, that allows for the rapid and concise implementation of conforming program generators for arbitrary programming languages.²

The primary contributions of this paper are:

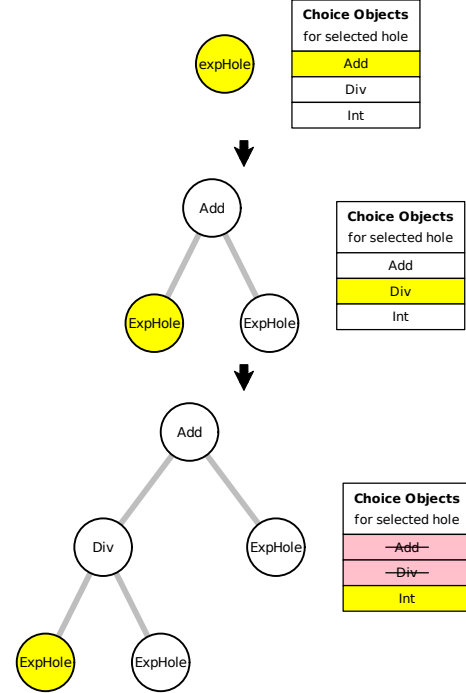
- A domain-specific language for creating conforming program generators.
- A generic framework for declaring type and effect systems for program generation used in the DSL implementation.
- Several example fuzzers implemented using the DSL, some of which have been used to find bugs in language implementations used in production.
- An analysis of the effectiveness and cost of the example fuzzers.

2 Design

In this section we describe the overall design of Xsmith at a high level. Throughout the section, we develop a simple fuzzer for a small toy calculator language (named *calc*) as an example. The grammar of this language is shown in Figure 1.

Xsmith fuzzers generate program trees by starting with a “hole” node for the top-level production of the grammar. Xsmith iteratively fills hole nodes in the tree with nodes corresponding to appropriate grammar productions, which may themselves have holes as children, as shown in Figure 2. A fuzzer author provides Xsmith with a grammar declaration that determines the grammar of program generation.

The grammar used by an Xsmith fuzzer generally matches the logical structure of the language or a subset of the language, but is usually not the same as the grammar used for parsing the language or the grammar of a compiler’s AST representation. This is because parser grammars typically encode complicated rules for turning linear text into an AST,



The tree starts as a single ExpHole node. Choices that alternate for Exp are listed and one is chosen at random. The process repeats with a new hole until no holes are left. At some points choices are filtered. For example, non-atomic choices are filtered when the tree gets too deep.

Figure 2. The Process of Hole Filling

and because full language grammars can be large and difficult to model for producing conforming programs. Xsmith’s simplified AST-focused grammar system allows users to ignore syntax complexity to focus on modeling semantics for finding “deep” bugs. Users can start with a small subset of the grammar, and iteratively grow their subset to include more features.

The declared grammar is compiled to generate an object-oriented attribute grammar specification for the RACR attribute grammar library [3]. RACR facilitates analysis of generated program fragments, allowing data flow both up and down the tree for any analysis. The generated attribute grammar matches the input grammar but adds a hole production as an alternative for each user-provided production.

While Xsmith program generation is grammar-driven at its core, generation of conforming programs requires considerations besides the grammar, such as for types and nondeterminism. To enable filtering and probability weighting based on these other considerations, the grammar is also compiled into a set of *choice classes*, one for each given production. When the generation algorithm selects the next hole to fill,

²Xsmith is open source, with code available at <https://gitlab.flux.utah.edu/xsmith/xsmith/>

a choice object is instantiated for each production that could be used to replace the hole, and the resulting list of choice objects is used to make the decision.

The difference between the attribute grammar and choice classes is often confusing to people learning about Xsmith for the first time. During generation, the AST being generated is represented by attribute grammar nodes, including hole nodes. Choice objects, or instances of choice classes, are not part of the AST, but are constructed when considering a particular hole node to fill. Each generated choice object corresponds to a different AST node choice that could be used to replace the currently chosen hole node, according to the grammar. Choice class methods are used to determine whether the given production choice is appropriate for the hole, given constraints besides the grammar, such as for types or unspecified behavior. To make this determination, choice class methods may query attributes of the AST, as well as other choice class methods on the same choice object. When a particular choice is made, the *fresh* method of the chosen choice object is used to construct a new attribute grammar node to replace the hole node. Then generation either moves on to another hole, which constructs new choice objects, or terminates.

Below is an example Xsmith specification that implements the grammar of the *calc* language. Each arm of the *add-to-grammar* form contains the name of a production, the super-type of that production, and a list of fields for that production. The *Add* and *Div* productions each have two children, and these children must themselves be *Exp* (expression) productions. These *Exp* children will be instantiated as *Exp* holes. The *Int* production specifies an integer literal, whose *val* child may contain arbitrary Racket data. In this example, *val* is initialized to a random integer between 1 and 100. (This random selection occurs each time an *Int* hole is filled.)

```
;; Fuzzers may use multiple add-to-grammar forms to
;; declare a grammar in a modular fashion.
(add-to-grammar calc
  [Exp #f ()
    #:prop may-be-generated #f]
  [Add Exp ([lhs : Exp]
            [rhs : Exp])]
  [Div Exp ([lhs : Exp]
            [rhs : Exp])]
  [Int Exp ([val = (random 1 100)])])
```

In addition to writing a grammar, a fuzzer author can declare various *attributes* and *choice methods* of each grammar production.

During generation, an Xsmith-based fuzzer uses choice methods to determine whether a particular choice of grammar production is suitable for replacing a hole. For example, the *_xsmith_satisfies-type?* choice method is used to filter out choices with invalid types, the *_xsmith_wont-over-deepen* method is used to make choices that will keep the generated program size bounded, and the *_xsmith_fresh*

choice method determines how a chosen node is initialized.³ Choice methods may call other choice methods, as well as attributes of their corresponding hole node.

Attributes are methods of AST nodes, and are used to perform analysis. For example, the *xsmith_type* attribute computes the type of a node, used by choice methods such as *_xsmith_satisfies-type?*, and the *_xsmith_visible-bindings* attribute computes a list of bindings available for reference at a given point in the tree. Attributes may query other attributes on the same AST node or on other nodes during computation, allowing data to flow up and down the tree as needed to make choices.

Besides attributes and choice methods, users may also declare *properties* of each grammar production. Properties are an abstraction on top of attributes and choice methods which simplify the specification of many aspects of a language. Each property is essentially a small custom DSL for describing an aspect of a language. Properties are compiled into attributes and choice methods. While choice methods and attributes may be written directly in a procedural style, properties allow semantic details of a grammar production, such as their binding structure and types, in a declarative style. Properties range from being convenient syntax sugar for defining a single attribute to being complicated DSLs that analyse multiple properties together to generate a family of attributes and choice methods.

Typical Xsmith fuzzers describe most language details with properties rather than defining many attributes or choice methods directly. For example, the *type-info* property described in section 4.2 is used to generate both the *xsmith_type* attribute and the *_xsmith_satisfies-type?* choice method. A collection of properties is included with Xsmith for describing key features of a language. Users may additionally define custom properties with their own compilation transformers to abstract patterns between fuzzers, although Xsmith is designed to support the majority of properties most languages' fuzzers would need out of the box.

Properties may be declared inline with a grammar definition, as with the *may-be-generated* property used in the *calc* grammar shown above, or separately with the *add-property* form. In the following code, *choice-weight* of the *Div* and *Int* nodes is adjusted to change their generation probabilities.

```
(add-property calc choice-weight
  [Div 10]
  [Int 5])
```

When a tree with no holes is finally completed, it must be converted to text for programming language implementations to consume. The *xsmith_render-node* attribute, defined by the *render-node-info* property, is used to convert

³Because attribute and method names are bare symbols without namespacing, we use the *xsmith_* prefix by convention for names defined by Xsmith itself, and we use a leading underscore by convention for private attributes and methods intended for use only in Xsmith's implementation.

the tree into text. Below is an example renderer for our calc language.

```
(define (render-infix operator)
  (lambda (n)
    (format "~a ~a ~a"
      (att-value 'xsmith_render-node
        (ast-child 'lhs n))
      operator
      (att-value 'xsmith_render-node
        (ast-child 'rhs n)))))
(add-property calc-grammar render-node-info
  [Add (render-infix "+")]
  [Div (render-infix "/")]
  [Int (lambda (n)
    (number->string (ast-child 'val n)))]])
```

2.1 Validating the Grammar

Because users define their own subset of a grammar rather than using published parser grammars, and because they must encode many rules about the language semantics, a user may write a generator that is incorrect. While writing a generator, users can cross-validate it by running generated test cases against implementations of the language, or systems under test (SUT). There are four major ways in which an Xsmith-based fuzzer may be incorrect:

- Test cases produced by Xsmith are syntactically or statically semantically invalid (e.g., with static type errors). In this case, the SUT will reject them, often with helpful error messages.
- Test cases are dynamically invalid (e.g., triggering run-time type errors). A user would also observe errors when interacting with the SUT.
- Generated test cases execute invalid or nondeterministic behaviors. Such cases can often be detected through differential testing of multiple implementations or through compile-time sanitizers.
- The grammar and associated generation rules provided to Xsmith describe only a fraction of the language. In this case, Xsmith would generate conforming test cases, but without leading to good coverage of implementations and/or bug-finding power. A user can discover this by inspecting generated test cases and the obtained coverage of the SUT following a fuzzing campaign.

3 Example

To give a sense of what a small but still featureful Xsmith fuzzer looks like, we present a small JavaScript fuzzer. Figure 3 is an abbreviated example of a simple JavaScript fuzzer, with elided code sections marked by "...". A full version of this example is included in the Xsmith source repository. While the full version is longer, it is still only 412 lines as measured by the wc utility.

This example demonstrates a fuzzer that takes advantage of the canned-components library to generate conforming

JavaScript programs that may utilize arrays, first-class functions, objects (encoded in Xsmith as structural record types), if statements, and loops. The largest amount of code elided from the full version is in program rendering. The rendering step tends to be verbose and varies by language, but is not complicated or difficult to code.

This example uses a `safe_divide` function, defined in a header, to avoid issues that arise from dividing by zero. Similarly, the full version defines more safe wrappers for array reference and assignment. While these operations are not undefined or even necessarily troublesome behavior in JavaScript, we use them to avoid having values collapsing to JavaScript’s undefined value, which would otherwise be overwhelmingly common. This demonstrates a common pattern used when creating fuzzers with Xsmith to avoid undefined behavior or the raising of common exceptions.

This example also does not directly use any `add-to-grammar` forms, because the entire abstract grammar used is provided by the canned components library, discussed in section 4.3. A larger fuzzer will generally include canned components as well as `add-to-grammar` forms that add various built-in functions specific to the language.

4 Cost Reduction Features

Xsmith has many features that work together to make the creation of conforming program generators inexpensive in terms of development time and effort. These features include forms for declaring grammar, types, and name scoping and resolution, as well as a “canned components” library to encapsulate language similarities, features for undefined behavior handling, and so on. In this section we give an overview of these features, discussing their usage and design.

4.1 Grammar and Syntax

The first step to generating programs that are conforming is to follow a grammar. Xsmith generates program trees according to a grammar provided by the user.

Usage. A user can define a grammar for their generator by using the `add-to-grammar` form. Each grammar production is declared as a subtype of another grammar production (possibly the abstract base grammar production, referenced by `#f`). Grammar productions may have any number of children, which can be specified as either being grammar productions (of a given type), or storage locations for arbitrary Racket data (used, for example, to hold values for number literals). Children may be annotated with a Kleene star to indicate repetition (zero or more repetitions). Below is an example of a partial grammar definition.

```
(add-to-grammar js-component
  [ArrayLiteral Expression ([elem : Expression *])]
  [IntLiteral Expression ([value]])])
```

```

(require xsmith xsmith/canned-components racr pprint ...)

;; An Xsmith specification starts with a "spec-component"
(define-basic-spec-component js-component)

;; Use canned-components to get common grammar definitions.
(add-basic-expressions js-component
  #:LambdaWithBlock #t
  #:MutableArray #t
  ...)
(add-basic-statements js-component
  #:ProgramWithBlock #t
  #:IfElseStatement #t
  ...)

;; Use canned-component loop generator.
;; It has many options, some elided.
(add-loop-over-container js-component
  #:name ForLoopOverArray
  #:loop-ast-type Statement
  #:body-ast-type Block
  #:collection-type-constructor
  (λ (elem-type)
    (mutable (array-type elem-type)))
  ...)

;; This header defines safe wrapper operations, and is included
;; when rendering the program.
(define header-definitions-block
  "function safe_divide(a,b){return b == 0 ? a : a / b} ...")

(add-property js-component render-node-info
  [VariableReference
   (λ (n) (text (ast-child 'name n)))]
  [SafeDivide
   (λ (n) (h-append
            (text "safe_divide(")
            (render-child 'l n)
            (text ", ")
            (render-child 'r n)
            (text ")")))]
  [IfElseStatement
   (λ (n)
     (h-append
      (text "if (")
      (render-child 'test n)
      (text ")")
      (render-child 'then n)
      (text " else ")
      (render-child 'else n)))]
  ...)

;; This macro defines, among other things, a function
;; to run the command line parser and start
;; generation with the given parameters.
(define-xsmith-interface-functions [js-component]
  #:program-node ProgramWithBlock
  #:format-render (λ (doc) (pretty-format doc 120))
  ...)

```

Figure 3. Sample JavaScript Generator Written with Xsmith

Design and Implementation. Xsmith’s grammar and AST data structures rely on the RACR [3] attribute grammar library. RACR allows Xsmith grammar nodes to have attributes that compute data that can depend dynamically on attributes or data from parent or child nodes. As an AST grows, RACR automatically keeps track of which attributes need to be recomputed.

Xsmith relies on grammars to allow users to define and re-use language components. To transform the AST into an input that is syntactically valid for a compiler or interpreter, Xsmith includes a multi-step render phase. The goal of the render phase is to produce a program as text. However, rather than defining a transformation from each grammar node to a string, it can be easier to have pleasantly formatted output by using an intermediate format. For example, the AST can be rendered into the data structures of Racket’s `pprint` pretty printing library or to s-expressions, which both have pretty-printing functions available.

4.2 Types

Generators of conforming programs need to produce well-typed code to pass the type-checking stage of programming language implementations. The requirement for type-correct code is perhaps less strict for dynamically typed languages than for statically typed languages. However, if code for dynamically typed languages is generated without regard to types, most expressions will raise run-time type errors. Xsmith includes a type system framework that allows fuzzers to generate well-typed code for a variety of languages.

Usage. The type system used by a fuzzer is defined by the `type-info` property. This property takes two specifications per grammar node. The first specification defines the types a grammar node can inhabit. The second specification is a function that receives a tree node of the specified production and its type and returns a dictionary of types for the node’s children. In the code below, the `LiteralString` and `StringAppend` productions are declared to always have type `string`, while the `VariableReference` is declared to use a type variable that can be unified with any type. The `LiteralString` and `VariableReference` productions have no children, but the `StringAppend` production constrains its children to inherit its type.

```

(define no-child-types (lambda (n t) (hash)))
;; The `hash` function constructs a hash table
(add-property
 js-component
 type-info
 [LiteralString [string no-child-types]]
 [StringAppend [string (lambda (n t) (hash 'l t 'r t))]]
 [VariableReference [(fresh-type-variable)
                     no-child-types]])

```

Design and Implementation. Xsmith allows its user to specify type systems that contain base types, function types, product types, generic types (such as lists and arrays),

nominal records, and structural records. Xsmith supports less expressive type systems than some other tools do, such as PLT Redex [6]. Some of these tools, like Redex, support arbitrary type judgments that are compiled to first-order logic and subsequently used for type checking and generating random terms [7]. However, Xsmith has more constraints on generated programs than merely being well-typed. Other analyses, such as those for the effect system described in section 4.4, require structural reasoning on types. Therefore, we have built a more limited type definition framework that allows this cross-analysis. Despite these limitations, Xsmith’s type checking framework is sufficient to support fuzzing many features of popular programming languages.

Type systems specified in Xsmith may also support subtyping. During type checking, Xsmith performs *subtype unification* between the types that a tree node declares that it supports, the types provided by its parent node, and any types declared by relevant definition nodes for references. Subtype unification, like traditional variable unification during type inference, mutates type variables to indicate relationships between type variables and between type variables and concrete types. However, unlike traditional unification, subtype unification reflects the asymmetric relationship of subtyping. Type variables in subtype relationships form a lattice of related types, where (subtype-unify! a b) relates a as a subtype of b, placing a below b in the lattice. Symmetric unification in this model is implemented merely as two subtype unifications:

```
(define (unify! a b)
  (subtype-unify! a b)
  (subtype-unify! b a))
```

When a type variable a is already related as a subtype to type variable b and (subtype-unify! b a) is executed, the relationship lattice is squashed such that a, b, and all variables between them in the lattice are unified into a single type variable.

While it is well known that unification-based type inference is incompatible with subtyping for type checking of existing programs, Xsmith can use unification because it type checks program fragments while generating a fresh program. So, assuming the type system specification is correct, for any needed type, it is always possible to produce a term of that type.

4.3 Language Similarities

Many programming languages have similar language features. To help Xsmith users avoid implementing these features afresh for each language they write an Xsmith fuzzer for, Xsmith provides a library of “canned components” that automatically define the necessary grammar nodes and properties.

Usage. The main forms provided by the library are the add-basic-statements and add-basic-expressions

macros. Each of these has a variety of optional keyword arguments and extends a grammar with forms specified by those arguments. These productions include literals, accessors and mutators for mutable arrays and dictionaries, and function application and definition. The code below demonstrates how many standard productions can be added to a grammar, with appropriate type rules and other properties, with a canned-components macro.

```
(add-basic-statements js-component
  #:Block #t
  #:ReturnStatement #t
  #:IfElseStatement #t
  #:AssignmentStatement #t
  ...
)
```

The canned-components library also provides the add-loop-over-container macro, which has various keyword arguments allowing a user to specify whether the loop form is a statement or an expression, which types of containers it can loop over, and the type of the loop’s result. These productions are added with all relevant properties for the type and effect systems, name analysis, etc. The only non-optional property that the user must add is the render-node-info property. The code below demonstrates how a loop form can be added to a grammar.

```
(add-loop-over-container js-component
  #:name ForLoopOverArray
  #:loop-ast-type Statement
  #:body-ast-type Block
  #:collection-type-constructor
  (λ (elem-type) (mutable (array-type elem-type)))
  ...)
```

Design and Implementation. The canned components are implemented as a library of macros that generate the most common patterns of grammar and property definitions. The canned-components library reduces the amount of code required to write a new fuzzer, and it reduces the duplication of tedious and error-prone type rules and other properties that are easy to get slightly wrong.

4.4 Unspecified and Implementation-Defined Behavior

For practical reasons, some programming languages leave the semantics of certain constructs either up to individual implementations or completely undefined. A generator of conforming programs must avoid every type of unspecified or non-deterministic behavior in the programs that it generates. One common unspecified behavior concerns the order of evaluation of subexpressions, such as multiple arguments in a function call. While the evaluation order is unimportant in the evaluation of purely functional code, effectful code that assigns variables or mutates values requires a consistent ordering to be conforming.

Usage. To avoid generating code with an unspecified effect order, a user simply annotates which nodes include

different effects, such as reading and writing to variables or mutable data structures. This is demonstrated in the following code.

```
(add-property js-component reference-info
  [Reference (read)]
  [Assignment (write)])
```

Design and Implementation. Xsmith includes an effect analysis that enumerates the possible effects of code evaluation and conservatively avoids ordering conflicts. Tracked effects include variable reference and assignment, projection and mutation of values like arrays, and higher-order function application. Whenever a potential conflict arises, such as referencing a variable in one function argument while assigning to the same variable in another argument, Xsmith filters out the choices that would lead to the generation of offending programs. A user may annotate grammar nodes that impose a specified ordering on their children, such as block and sequence constructs, with the `strict-child-order?` property.

Besides effect ordering, programming languages have an inconsistent variety of features that cause undefined, or at least unhelpful, behavior. For example, out-of-bounds array access is an undefined behavior in C, while in many other languages it is defined to raise an exception. Although a raised exception is well defined and potentially an interesting part of the language API to fuzz test, in typical fuzz testing an array access exception is likely not a useful behavior. For example, because the set of possible values of type `int` in a given programming language is likely much larger than the set of usable array sizes, array access with approximately uniformly generated `int` values will raise exceptions much more often than it will yield values. For both defined and undefined semantics of array access, it is usually best to generate code that wraps such accesses to convert the index to a number within bounds or provide a fallback result value.

Since these behaviors are language-specific, each fuzzer needs some amount of unique attention to them. The common pattern used in our example fuzzers is to include program header text that defines safe wrapper functions for potentially problematic functionality, possibly including extra fallback arguments (e.g., for accessing an empty list). The grammar can then target the safe wrappers instead of the raw unsafe operations. Some of these behaviors are common and have been captured in the canned-components library.

4.5 Name Scoping and Resolution

Generators of conforming programs need to produce programs where variables referenced are defined in scope. Xsmith includes a generic analysis to ensure that variables are well scoped. If a reference is generated in a position where no

appropriately typed variable is in scope, Xsmith will automatically add an appropriate definition node into a scope that is visible to the new reference.

Usage. Users can annotate which grammar nodes bind and reference variables using the `binder-info` and `reference-info` properties, such as with the following code.

```
(add-property js-component binder-info
  [Definition ()]
  [FormalParam (:#binder-style parameter)])
```

However, common patterns for binders and references have also been captured in the canned-components library, so most users do not need to interact with these properties directly.

Design and Implementation. Our resolution system is based on scope graphs [18], which is a generic system for representing variable scoping in programming languages. Based on user-provided annotations, Xsmith will generate scope graph models for generated programs, and use them to find which variables that are in scope at any position.

4.6 Language-Specific Analyses

While Xsmith includes several generic analyses, an advanced fuzzer may benefit from a language-specific analysis. Because they are language-specific, such analyses can not reasonably be included in the Xsmith framework. However, Xsmith provides features that aid a user in writing custom analyses.

Users can define custom attributes and choice methods, as well as custom Xsmith properties. Custom properties are essentially mini-DSLs that can compile declarative data into attributes and choice methods. Defining custom properties requires familiarity with Racket macro writing techniques, and we will leave discussion of custom properties to the Xsmith documentation. Finally, users can leverage Xsmith's generic analyses as dependencies of their analyses, such as by querying a node's type during a custom analysis.

Custom properties, attributes, and choice methods provide a way for Xsmith users to extend Xsmith with arbitrary Racket code. This allows Xsmith fuzzers to include features never imagined by Xsmith's authors.

4.7 Making Decisions

Xsmith includes features for both filtering potential decisions and for adjusting the probability of different choices when filling holes in the generated AST.

Usage. A user can add custom choice methods as filters by using the `choice-filters-to-apply` property. The following code applies filter `choice-method` defined above to restrict `VariableReference` generation.

```
(add-property choice-filters-to-apply js-component
  [VariableReference (allow-ref)])
```

A user can adjust the frequency of different grammar node choices with the `choice-weight` property, shown below.

```

(add-property choice-weight js-component
  [IfStatement 50]
  [AssignmentStatement
    (λ (hole) (if (eq? (ast-node-type
                        (ast-parent hole))
                      'IfStatement)
                  20
                  30))])

```

The choice weight may be given a positive integer or a function that returns a positive integer based on an analysis of the program.

Design and Implementation. When filling a hole, Xsmith instantiates one choice object with the appropriate class for each subtype of the required node type. For example, in a hole of type `Expression`, Xsmith will instantiate a choice object for each of `IntegerLiteral`, `VariableReference`, `Addition`, and so on. Each of these choices is filtered based on the specifications given to the `choice-filters-to-apply` property, including default filters such as type satisfaction. After a list of valid choices has been filtered, each remaining choice has its `choice-weight` computed. A choice is then randomly made, with each choice having probability $\text{choiceWeight} / \text{weightSum}$.

4.8 Additional Features

Xsmith includes various other features that we lack room to discuss, such as an integrated automatic test-case reducer, an extensible command-line interface, support for modular fuzzer declaration, iterative refinement, parametric generation in the manner of Zest [20], and more.

5 Evaluation

We evaluate Xsmith by considering a set of fuzzers built with Xsmith as well as bugs found with those fuzzers. We assess the difficulty of creating fuzzers with Xsmith and the number and quality of bugs found. In particular, we examine a particular case study of fuzzing Dafny.

5.1 Fuzzers

Generators of conforming programs typically require a lot of effort to create. Csmith, a predecessor to Xsmith and its major inspiration, required hundreds of person-hours and tens of thousands of lines of code. Xsmith fuzzers require substantially less effort and code. Figure 4 compares sizes of a selection of conforming program generators in terms of code size.

While the implementations of Xsmith-based fuzzers are significantly smaller than similar conforming program generators like Csmith [24], Verismith [10], and SQLSmith [21], they still produce programs that are syntactically and semantically valid as well as free from undefined or nondeterministic behavior. Additionally, Xsmith fuzzers can be featureful, generating correct code for conditionals, rich types, variable references, and so on.

Generator	LOC	Language
Csmith	38,988	C++
Verismith	10,139	Haskell
SQLSmith	3,909	C++
Xsmith Racket Fuzzer	1,265	Racket
Xsmith Dafny Fuzzer	1,666	Racket
Xsmith Standard ML Fuzzer	1,151	Racket
Xsmith WebAssembly Fuzzer	1,433	Racket
Xsmith Python Fuzzer *	1,800	Racket
Xsmith Lua Fuzzer *	450	Racket
Xsmith Javascript Fuzzer *	412	Racket

All line counting was done with Unix `wc`. Fuzzers marked with * have not been exercised in substantial fuzzing campaigns.

Figure 4. Comparison of Conforming Program Generators

Generator	LOC	Language
Xsmith Framework	13,325	Racket
Xsmith Racket Fuzzer	1,265	Racket
Xsmith Dafny Fuzzer	1,666	Racket
Xsmith Standard ML Fuzzer	1,151	Racket
Xsmith WebAssembly Fuzzer	1,433	Racket
Xsmith Python Fuzzer *	1,800	Racket
Xsmith Lua Fuzzer *	450	Racket
Xsmith Javascript Fuzzer *	412	Racket
StarSmith Framework	19,524	Java
StarSmith C Fuzzer	1,702	LaLa
StarSmith Lua Fuzzer	1,578	LaLa
StarSmith SQL Fuzzer	~ 3,500	LaLa
StarSmith SMT Fuzzer	~ 700-900	LaLa
Polyglot Framework		Mostly C++
Polyglot C Fuzzer	1,508	Mix
Polyglot JavaScript Fuzzer	1,618	Mix
Polyglot PHP Fuzzer	2,013	Mix
Polyglot Solidity Fuzzer	2,090	Mix

Counts prefixed with ~ are approximate. Fuzzers marked with * have not been used in substantial fuzzing campaigns.

Figure 5. Comparison of Generic Fuzzing Frameworks

Xsmith is not the only generic framework for creating programming language fuzzers. Other generic frameworks include Polyglot [4] and StarSmith [13]. While Xsmith has the greatest focus on differential testing compared to other generic frameworks, it compares well in terms of implementation effort per fuzzer, as shown in Figure 5.

Some StarSmith line counts are approximate, because for SQL and SMT their repository contains multiple versions of each fuzzer with various modifications. The size of the Polyglot framework is difficult to ascertain, as the bulk of its implementation is a modification to AFL, and the Polyglot authors keep the entire modified copy of AFL in their source tree. Polyglot grammar specifications are given with a mix of JSON specification, Python code, and other formats that are specific to Polyglot.

5.2 Fuzzing Dafny

In the summer of 2021, one of the authors of this paper⁴ began the XDsmith project [12] using Xsmith to fuzz Dafny [14], a verification-aware programming language. He was experienced with Racket but had no previous experience with Xsmith. In about a week, he prototyped his Xsmith-based Dafny generator. During a three-month period, he improved his fuzzer and found 28 Dafny bugs. Since that period, his fuzzer has found an additional 2 bugs in the open source Dafny implementation. His fuzzer implementation has 1,666 lines of code, as well as differential testing and verification testing code totaling 722 more lines.

While Dafny fuzzing primarily used differential testing (comparing different Dafny compiler back ends) and compiler error detection, it also included a verification oracle that found one bug. Additionally, one bug was found as a side-effect of writing the fuzzer. While the author was trying to determine the proper type constraint to write for one feature of the fuzzer, he decided to manually test violations of the constraint, and found a bug.

This experience shows that Xsmith can be utilized to write an effective fuzzer in a small time period with a small amount of code.

5.3 Summary of Bugs Found

We have found bugs using Xsmith fuzzers for various programming languages, listed in Figure 6. All reported bugs are unique.

Aside from one Racket bug that had been fixed before we found it, all bugs listed are new (not publicly reported or fixed before we discovered them through fuzzing). All Racket bugs were confirmed by Racket’s maintainers, and all but one have been fixed. All Dafny bugs and issues were confirmed by Dafny maintainers, and 6 have been fixed. All of the WebAssembly and Standard ML bugs we found have been confirmed, and most of them have been fixed.

We have written fuzzers for various other languages besides those in the bug table, such as Python, JavaScript, and Lua. However, we have not performed extensive fuzzing with them or with the WebAssembly or Standard ML fuzzers.

These less-used fuzzers likely all need at least minor improvements to be effective at finding bugs.

Approximately half of the bugs found by Xsmith-based fuzzers so far have been semantic errors detected by differential testing. These bugs can effectively only be found by program generators that reliably generate conforming test cases. Otherwise, if semantically valid but non-conforming (or semantically invalid) programs are regularly generated, differential testing oracles would be overrun with false positive results, and would be practically useless.

Similarly, approximately half of the bugs found could only effectively be found by generators of semantically correct (though not necessarily conforming) program generators. Bugs characterized by valid programs failing to compile would have too many false positives if tested using a generator that does not reliably generate semantically valid test cases. Some bugs found were characterized by a “successful” compilation that produced ill-formed output. Testing for ill-formed output when the compiler is successful could reasonably be performed with generators of semantically or even syntactically invalid code, but such bugs would likely be difficult for such generators to trigger.

In our experiments, Xsmith program generators have found few crash bugs, the bug class most commonly found by fuzz testing. Differential fuzz testing with Xsmith appears to be very complementary to other fuzzing practices, such as fuzzing with generators of random bytes such as AFL [25]. Xsmith seems well suited to finding a different class of bugs than tools like AFL, and Xsmith does not effectively stress early compiler stages such as parsing.

5.4 Bug Discussion

We present and discuss a small selection of bugs found. The code snippets presented are simplified presentations to illustrate the bugs, not the actual code generated by Xsmith.

5.4.1 Racket and Chez Scheme Float Modulo Bug.

When using a large floating point number, Racket CS (Racket built on Chez Scheme) would give wrong answers to the modulo operator. The bug was found through differential testing.⁵

```
#lang racket/base
;; This number is big enough that despite
;; the .1 it passes `integer?`.
(define num
  ;; The number has been shortened to fit inside
  ;; margins. The real number had 14 more digits.
  93674811510424315205562331463211094477254417232.1)
(println (integer? num)) ;; Prints true
;; But modulo doesn't stay in bounds of the divisor.
(println (modulo num 10)) ;; Prints a number >10
```

⁴The XDsmith author was invited to become a co-author of this paper after writing the fuzzer. At the time of writing the fuzzer, he was completely independent of Xsmith’s original authors.

⁵This bug was submitted as <https://github.com/racket/racket/issues/3469>.

Language	Implementation	Crash Bugs	Semantic Bugs	Static Non-crash Bugs	Total
Racket	BC	0	5	0	5
Racket	CS	0	4	0	4
Racket	Both Back Ends	0	2	0	2
Total Racket Bugs					11
Dafny	Java Back End	0	0	8	8
Dafny	C# Back End	1	3	3	7
Dafny	Go Back End	0	3	0	3
Dafny	JavaScript Back End	0	4	1	5
Dafny	All Back Ends	0	1	7	8
Total Dafny Bugs					30
WebAssembly	Wasmer	1	4	0	5
Standard ML	ML MLKit	0	0	1	1

Crash Bugs: Bugs characterized by a memory error or assertion violation in the compiler or interpreter causing the system under test to exit abnormally. This is not simply failure to compile an input or an interpreter exiting with an exception.

Semantic Bugs: Bugs characterized by a wrong program result. Found primarily by differential testing, but also by testing various properties. For example, a raised exception when the fuzzer has been constrained not to generate code that raises exceptions.

Static Non-crash Bugs: This category includes various kinds of bugs whose finding would have required syntactically and semantically correct programs but not necessarily conforming programs. For example, this category includes failure to compile correct programs, ill-formed compiler output (eg. Dafny compiler outputting ill-formed Java code that the Java compiler can't compile), etc.

Figure 6. Bugs Found with Xsmith-Based Fuzzers

It was determined that it was actually a bug in Chez Scheme itself, and was fixed.⁶

5.4.2 Racket BC GCD Bug. This is an example of a bignum boundary bug. The bug was determined to be at least 20 years old, and included in the oldest repository import to CVS. The bug was found by differential testing.⁷

```
#lang racket/base
(define num -4611686018427387904)
;; The gcd function should always return non-negative
;; numbers, but RacketBC returns a negative number.
(gcd num num)
```

5.4.3 Racket Serialization Bug. Besides differential testing, some bugs are found with various properties. For example, our Racket fuzzer was designed to produce code that does not raise exceptions. Thus, a program that raises an exception is evidence of a bug (in Racket or in our fuzzer). This bug was present in both Racket BC (the old C back end for Racket) and Racket CS (the new Chez Scheme back end), and was found with an interesting property.

The write and read functions are primitive serialization and deserialization functions in Racket and Scheme. In version 7.9, the latest release at the time of fuzzing, the handling in the read function for Unicode character U+FEFF, the byte-order-mark, was changed. However, the write function was not changed. Thus, data could be altered in a round-trip between the read and write functions.

When printing generated Racket programs, our Racket fuzzer pretty prints them using a function that ultimately uses the write function. When compiling our generated programs, the Racket compiler uses the read function. The bug was found because the compiler was rejecting ill-formed programs that were mangled between generation and compilation by the write and read mismatch.⁸ We found multiple write and read mismatch bugs in this manner.

5.4.4 Dafny Bugs Related to Zero Multiplicity. This class of bugs was found with differential testing. Internally, the multiset data structure in Dafny is implemented as a dictionary mapping from elements to the multiplicity. Many multiset operations assumed the invariant that the multiplicities will always be positive. However, this invariant in

⁶The Chez Scheme bug was fixed in <https://github.com/cisco/chezScheme/pull/537>.

⁷Reported as <https://github.com/racket/racket/issues/3484>.

⁸This bug was reported as <https://github.com/racket/racket/issues/3486>.

fact did not hold, as the multiplicity changing operation can break the invariant. The following code would produce `true false` when compiled to C#, `false true` when compiled to Go, `true true` when compiled to JavaScript, and `false false` (which is the correct output) when compiled to Java.⁹

```
method Main ()
{
  var a := multiset{12}[12 := 0];
  var b := multiset{42};
  print 12 in a, " ", a == b, "\n";
}
```

5.4.5 Dafny Bug Found by Verification Testing. In addition to differential testing, generated Dafny programs were also used for verification testing, which aims to find soundness and precision issues in the Dafny verifier. Additionally, it is useful for finding discrepancies of the underlying semantics between the verifier and the compilers. Given a generated Dafny program with `print` statements, verification testing compiles and runs the program, and correlates `print` statements with their outputs. The program is subsequently transformed to turn the `print` statements into assertions. In the most basic form of verification testing, we expect that these assertions should be verified by the verifier. These assertions therefore test the Dafny verifier’s ability to predict the output emitted by the compiled program.

In the following bug, all compiled Dafny programs returned the same incorrect answer, so the bug could not be discovered by differential testing. Yet, it was determined to be incorrect by the verification testing. The problem was that the superset operation was compiled into a subset operation.¹⁰

```
method Main ()
{
  var a := {1};
  var b := {1, 2};
  print a > b;
}
```

6 Discussion

We discuss a qualitative comparison of Xsmith to other systems for creating programming language fuzzers, and discuss Xsmith’s limitations.

6.1 Comparison to Polyglot

Polyglot [4] is a generic language processor that can be configured to produce programs in different languages by providing configuration in a custom format. Polyglot has been very effective at finding crash bugs, finding over 100 bugs

⁹These bugs were reported as <https://github.com/dafny-lang/dafny/issues/1359> and <https://github.com/dafny-lang/dafny/issues/1361>.

¹⁰This bug was reported as <https://github.com/dafny-lang/dafny/issues/1357>. Because the output of the `print` statement is incorrect, when it is turned into an assertion the verifier detects that the assertion is violated, thus revealing the bug.

in implementations of 9 programming languages. Polyglot uses constrained mutation and a semantic validation step that improves its probability of generating semantically valid programs. These features prevent Polyglot from generating programs with problems such as references to undefined variables, and prevents many type errors. However, these steps do not guarantee semantic correctness. In their evaluation of Polyglot, Chen et al. show that none of their generators produces semantically valid test cases more than 60% of the time. This rate of producing semantically invalid test cases renders it ineffective as a generator for oracles that consistently require semantically valid test cases to prevent false positives, including differential testing.

6.2 Comparison to StarSmith

The StarSmith [13] program generator is a generic framework for generating semantically correct programs. It is configured using a DSL called LaLa, in which users specify the grammar and other rules for program generation, similar to Xsmith. While StarSmith has no built-in or default steps to prevent undefined or other behaviors that are unsuitable for differential testing, users may write custom LaLa code to prevent some such behaviors. For example, the StarSmith authors created a Lua fuzzer that includes a filter preventing the generation of any events in positions where the order of operations is not guaranteed.

While LaLa makes it possible to create a fuzzer for differential testing, it does not encapsulate common patterns to make it easy. If StarSmith users want to make a similar fuzzer that prevents unspecified effect ordering, they would need to write a similar filter. Additionally, this filter is stricter than Xsmith’s generic effect analysis, which can still allow non-conflicting effects when evaluation order is unspecified.

Aside from this observation about encapsulating patterns, it is difficult to compare the suitability of Xsmith and StarSmith for differential testing, since StarSmith’s authors expressed a lack of confidence that their programs were actually free of undefined behavior. They reported total instances of test cases uncovering a bug, from fuzzing with conforming configurations, rather than unique bugs found or confirmed as they did for other generators. This makes it difficult to know if they found many bugs or few bugs repeated many times. However, they did report unique bugs for their differential testing of SQL, in which they found 11 semantic bugs and 13 segfault bugs among 3 SQL implementations using a SQL generator of approximately 3500 lines of code in their LaLa DSL. This generator was significantly larger than their other generators. Creating specifications for StarSmith to create conforming program generators is possible, but its focus appears to be on program generators for crash fuzzing.

6.3 Limitations of Xsmith

While Xsmith inhabits a new and useful point in the space of fuzzing tools, it has various limitations.

6.3.1 Type System. Xsmith’s type system is flexible but not comprehensive. For example, Xsmith can not currently generate functions with parameters that can be multiple different types but that are not fully generic enough to allow any type, such as a function that accepts either a string or an integer but no other type. Additionally Xsmith can not generate functions with optional arguments or other forms of variadic functions. Built-in functions with such types can be specified as productions in an Xsmith grammar, allowing Xsmith to generate calls to them. However, for complicated types, multiple grammar nodes may need to be specified to allow x generation of uses of all potential types of a function.

The type system also lacks a notion of classes. While we have implemented structural records with subtyping and nominal records, which can be used to represent some aspects of object-oriented languages, we have not yet designed a generic system to encode the semantics of classes and objects that is well-suited to a variety of languages. Because classes, objects, and methods have widely varying semantics in different languages, it is difficult to determine what essential features such an encoding should have.

Another limitation of the type system currently is a lack of “negative types” to constrain type variables. There are often situations where one or more particular types are disallowed but where any other type is allowed. An example situation is a position where functions are disallowed, but base types like `int` and `string` are allowed, as well as composite types such as `list` composed of other allowed types (perhaps recursively). Rather than specifying that function types are disallowed, users must enumerate allowed types. Since the set of allowed types may be large or infinite, users might only enumerate a subset of allowed types.

6.3.2 Probabilities. Xsmith follows a long history of using weight specifications to determine the probability of generating any given grammar production [22]. Besides weights, the probability of generating any given production is also affected by filters that consider the type system, the effect system, AST depth, and other factors. While Xsmith’s weighting system is flexible and allows for dynamic weight determination, it is difficult to determine how any weighting scheme will ultimately affect the probability of generating a particular production or combination of productions.

Xsmith provides a logging mechanism to view the frequency at which different productions were generated or under consideration for generation. However, logging does not provide a guide to understand how to improve a weighting scheme to achieve a more desired probability distribution, nor does it provide insight into what distributions would be effective at finding bugs.

6.3.3 Effects. The generic effect analysis in Xsmith is both conservative and limited. Because it is conservative, limiting generation to only programs that are guaranteed to be free of unspecified effect ordering, Xsmith filters out many choices

that would lead to generating valid, conforming programs. This limitation particularly affects higher-order values. For example, Xsmith has no analysis to determine whether two container values (such as arrays) are the same, so it conservatively assumes that any mutation to a particular container type may conflict with any other access or mutation of that same container type. Because the effect analysis must be generic enough to analyze programs in many languages with varying semantics (most of which is unspecified within an Xsmith fuzzer), the analysis is very limited. Therefore the set of rejected programs is very large, and has great potential to include many interesting and bug-inducing programs.

Xsmith could potentially include a more precise effect analysis by including a way for users to specify language-specific value- and control-flow analyses. However, that would greatly increase the difficulty of writing a new fuzzer, and it is unclear how much it would improve a fuzzer’s bug-finding capabilities.

7 Related Work

We compare Xsmith to related systems. In particular, we discuss conforming program generators, program generator generators, and grammar-directed fuzzers.

7.1 Conforming Program Generators

Some random testers for programming languages generate conforming programs, or programs that are syntactically and semantically valid as well as being free from nondeterminism, undefined behavior, and unspecified behavior. An early major conforming program generator was Csmith [24]. Similar systems include Verismith [10] and SQLsmith [21]. YARPGen [15] statically generates programs that are free from undefined behavior with no dynamic checks. Csmith and other conforming program generators have been very successful at finding bugs in programming language implementations. These bugs include semantic bugs found with differential testing that can not be found by more common crash oracles. However, conforming program generators like Csmith require much programmer time and tend to be tens of thousands of lines of code to implement.

JFQ [19] is a framework for imperative property-based testing that has been successfully used to fuzz programming languages. JFQ allows users to write correctness properties for generated data. However, JFQ does not include a general framework for program analysis to generate conforming programs, so users must write their own analyses.

Xsmith is a DSL and library for building conforming program generators like Csmith, but with less time and code. Xsmith eases development of conforming program generators by providing a declarative DSL, along with generic analyses, generation strategies, and other tools as a library.

Thus Xsmith allows users to build conforming program generators at a fraction of the cost of stand-alone generators like Csmith.

7.2 Program Generator Generators

Polyglot [4] is a framework that takes a language specification in a custom format and produces random programs. Polyglot includes a semantic validation step that improves the percentage of syntactically and semantically valid test cases generated. However, programs generated by Polyglot are not guaranteed to be syntactically or semantically correct, or to be conforming. Polyglot has been successfully used to find well over 100 bugs in implementations of at least 9 programming languages. However, it is not a suitable system for finding semantic bugs via differential testing because generating non-conforming programs leads to impractically high false positive rates.

StarSmith [13] is a framework that takes a language specification in a DSL called LaLa and produces random programs. StarSmith generates programs that are syntactically valid and well-typed. Some StarSmith generators have been further crafted to generate programs that are free of unspecified behavior for differential testing. StarSmith generators that have filters to generate only conforming programs must individually be constrained to not generate offending code. For example, the StarSmith authors created a Lua generator that included custom code to prevent any effects when the order of evaluation is not guaranteed. Such code would need to be repeated for each generator in StarSmith for languages without guarantees of evaluation order.

PLT Redex [6] is a framework that allows users to specify a semantics for a programming language. It includes a testing feature that generates random well-typed programs [7]. However, it does not generate programs that are free from undefined behavior. This is useful in Redex, since it helps users to find where undefined behavior exists in their semantics definitions, however it limits its use as a generator for differential testing.

LangFuzz [11] is a fuzzer that uses grammar-directed fuzzing. Additionally, LangFuzz can incorporate code patterns learned by parsing program corpuses. LangFuzz has successfully found many bugs. However, LangFuzz does not generate conforming programs that are free from undefined behavior, limiting its practical utility for differential testing.

Xsmith has been designed to generate programs suitable for differential testing. Xsmith fuzzers generate programs that are syntactically and semantically valid, in addition to being free from undefined or nondeterministic behavior. Xsmith includes generic effect analyses for the common case of unspecified order of evaluation, allowing programs to include effects while still preserving a well-defined relationship between effects. While some unspecified behaviors need to be handled on a per-language basis, Xsmith's canned components library helps with some common patterns.

7.3 Grammar-Directed Fuzzers

Some fuzzers, such as Lava[22] and Yagg [5], are grammar-directed[1, 8, 9, 16], meaning they only build program trees that match a given grammar. These fuzzers are useful because they can generate syntactically valid test cases that pass early stages of a compiler or other language processor, allowing fuzzing to exercise deeper code paths. However, grammar-directed fuzzers, without other guidance, do not guarantee that their outputs semantically correct or conforming. Thus they can not reliably be used to find semantic bugs through differential testing.

Some grammar-directed fuzzers such as Grimoire [2] automatically learn a grammar instead of taking a user-supplied grammar as input. While this automatic learning step means that the fuzzer requires less up-front effort to craft a grammar specification, it provides even weaker guarantees about the syntactic and semantic validity of produced outputs.

Xsmith is grammar-directed in addition to being directed by other analyses, such as type and effect analyses. These analyses constrain program generation to produce conforming programs that are useful for finding semantic bugs.

8 Conclusion

Differential fuzz testing can be a powerful tool for finding semantic bugs in programming languages. Xsmith is a library and DSL that provides shared infrastructure, declarative specification, and extension hooks that allow users to easily build featureful fuzz testers for differential testing. Xsmith has been used to create a variety of fuzzers requiring modest effort and code size that have found bugs in different language implementations. Compared to related work, Xsmith is the first tool focused on easily creating fuzzers for differential testing. Xsmith fuzzers can be used synergistically with other fuzzing techniques to find bugs in many language implementations.

Acknowledgments

We thank the anonymous GPCE reviewers for their valuable comments on this work. We performed our experiments in the Utah Emulab [23] testbed. This material is based upon work supported by the National Science Foundation under Grant Number 1527638.

References

- [1] Michael Beyene and James H. Andrews. 2012. Generating String Test Data for Code Coverage. In *Proc. 2012 IEEE 5th International Conference on Software Testing, Verification and Validation (ICST)*. 270–279. <https://doi.org/10.1109/ICST.2012.107>
- [2] Tim Blazytko, Cornelius Aschermann, Moritz Schlögel, Ali Abbasi, Sergej Schumilo, Simon Wörner, and Thorsten Holz. 2019. GRIMOIRE: Synthesizing Structure While Fuzzing. In *Proc. 28th USENIX Security Symposium*. 1985–2002. <https://www.usenix.org/conference/usenixsecurity19/presentation/blazytko>
- [3] Christoff Bürger. 2015. Reference Attribute Grammar Controlled Graph Rewriting: Motivation and Overview. In *Proc. 2015 ACM SIGPLAN*

- International Conference on Software Language Engineering (SLE)*. 89–100. <https://doi.org/10.1145/2814251.2814257>
- [4] Yongheng Chen, Rui Zhong, Hong Hu, Hangfan Zhang, Yupeng Yang, Dinghao Wu, and Wenke Lee. 2021. One Engine to Fuzz 'em All: Generic Language Processor Testing with Semantic Validation. In *Proc. 42nd IEEE Symposium on Security and Privacy (S&P)*. 642–658. <https://doi.org/10.1109/SP40001.2021.00071>
 - [5] David Coppit and Jiexin Lian. 2005. Yagg: An Easy-to-Use Generator for Structured Test Inputs. In *Proc. 20th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 356–359. <https://doi.org/10.1145/1101908.1101969>
 - [6] Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. 2009. *Semantics Engineering with PLT Redex*. MIT Press.
 - [7] Burke Fetscher, Koen Claessen, Michał Pałka, John Hughes, and Robert Bruce Findler. 2015. Making Random Judgments: Automatically Generating Well-Typed Terms from the Definition of a Type-System. In *Programming Languages and Systems*, Jan Vitek (Ed.). Springer Berlin Heidelberg, 383–405. https://doi.org/10.1007/978-3-662-46669-8_16
 - [8] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. 2008. Grammar-Based Whitebox Fuzzing. In *Proc. 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 206–215. <https://doi.org/10.1145/1375581.1375607>
 - [9] K. V. Hanford. 1970. Automatic Generation of Test Cases. *IBM Systems Journal* 9, 4 (Dec. 1970), 242–257. <https://doi.org/10.1147/sj.94.0242>
 - [10] Yann Herklotz and John Wickerson. 2020. Finding and Understanding Bugs in FPGA Synthesis Tools. In *Proc. 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*. 277–287. <https://doi.org/10.1145/3373087.3375310>
 - [11] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with Code Fragments. In *Proc. 21st USENIX Security Symposium*. 445–458. <https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/holler>
 - [12] Ahmed Irfan, Sorawee Porncharoenwase, Zvonimir Rakamarić, Neha Rungta, and Emina Torlak. 2022. Testing Dafny (Experience Paper). In *Proc. 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 556–567. <https://doi.org/10.1145/3533767.3534382>
 - [13] Patrick Kreutzer, Stefan Kraus, and Michael Philippsen. 2020. Language-Agnostic Generation of Compilable Test Programs. In *Proc. IEEE International Conference on Software Testing, Verification and Validation (ICST)*. 39–50. <https://doi.org/10.1109/ICST46399.2020.00015>
 - [14] K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning*, Edmund M. Clarke and Andrei Voronkov (Eds.). Springer Berlin Heidelberg, 348–370. https://doi.org/10.1007/978-3-642-17511-4_20
 - [15] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2020. Random Testing for C and C++ Compilers with YARPGen. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 196 (Nov. 2020), 25 pages. <https://doi.org/10.1145/3428264>
 - [16] Peter M. Maurer. 1990. Generating Test Data with Enhanced Context-Free Grammars. *IEEE Software* 7, 4 (1990), 50–55. <https://doi.org/10.1109/52.56422>
 - [17] William M. McKeeman. 1998. Differential Testing for Software. *Digital Technical Journal* 10, 1 (1998), 100–107. <https://www.hpl.hp.com/hpjournal/dtj/vol10num1/vol10num1art9.pdf>
 - [18] Pierre Neron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. 2015. A Theory of Name Resolution. In *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11–18, 2015. Proceedings (Lecture Notes in Computer Science, Vol. 9032)*, Jan Vitek (Ed.). Springer, 205–231. https://doi.org/10.1007/978-3-662-46669-8_9
 - [19] Rohan Padhye, Caroline Lemieux, and Koushik Sen. 2019. JQF: Coverage-Guided Property-Based Testing in Java. In *Proc. 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 398–401. <https://doi.org/10.1145/3293882.3339002>
 - [20] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. 2019. Semantic Fuzzing with Zest. In *Proc. 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 329–340. <https://doi.org/10.1145/3293882.3330576>
 - [21] Andreas Seltenreich. 2020. SQLsmith software repository. <https://github.com/anse1/sqlsmith>
 - [22] Emin Gün Sirer and Brian N. Bershad. 1999. Using Production Grammars in Software Testing. In *Proc. 2nd Conference on Domain Specific Languages (DSL)*. 1–13. <https://www.usenix.org/conference/dsl-99/using-production-grammars-software-testing>
 - [23] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. 2002. An Integrated Experimental Environment for Distributed Systems and Networks. In *Proc. 5th Symposium on Operating Systems Design and Implementation (OSDI)*. 255–270. <https://www.usenix.org/legacy/event/osdi02/tech/white.html>
 - [24] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *Proc. 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 283–294. <https://doi.org/10.1145/1993498.1993532>
 - [25] Michał Zalewski. 2020. American Fuzzy Lop. <https://lcamtuf.coredump.cx/afll/>

Received 2023-07-14; accepted 2023-09-03

Multi-Stage Vertex-Centric Programming for Agent-Based Simulations

Zilu Tian

zilu.tian@epfl.ch

EPFL

Lausanne, Switzerland

Abstract

In vertex-centric programming, users express a graph algorithm as a vertex program and specify the iterative behavior of a vertex in a compute function, which is executed by all vertices in a graph in parallel, synchronously in a sequence of supersteps. While this programming model is straightforward for simple algorithms where vertices behave the same in each superstep, for complex vertex programs where vertices have different behavior across supersteps, a vertex needs to frequently dispatch on the value of supersteps in compute, which suffers from unnecessary interpretation overhead and complicates the control flow.

We address this using meta-programming: instead of branching on the value of a superstep, users separate instructions that should be executed in different supersteps via a staging-time `wait()` instruction. When a superstep starts, computations in a vertex program resume from the last execution point, and continue executing until the next `wait()`. We implement this in the programming model of an agent-based simulation framework CloudCity and show that avoiding the interpretation overhead caused by dispatching on the value of a superstep can improve the performance by up to 25% and lead to more robust performance.

CCS Concepts: • Software and its engineering → Domain specific languages; • Computing methodologies → Simulation languages; Parallel algorithms.

Keywords: agent-based simulations, vertex-centric programming, distributed systems, staging, metaprogramming

ACM Reference Format:

Zilu Tian. 2023. Multi-Stage Vertex-Centric Programming for Agent-Based Simulations. In *Proceedings of the 22nd ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '23)*, October 22–23, 2023, Cascais, Portugal. ACM, New York, NY, USA, 13 pages. <https://doi.org/10.1145/3624007.3624057>

Publication rights licensed to ACM. ACM acknowledges that this contribution was authored or co-authored by an employee, contractor or affiliate of a national government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.

GPCE '23, October 22–23, 2023, Cascais, Portugal

© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 979-8-4007-0406-2/23/10...\$15.00

<https://doi.org/10.1145/3624007.3624057>

1 Introduction

The vertex-centric programming paradigm is first proposed and popularized by Google's Pregel [21] for the efficient execution of graph algorithms over large, distributed graphs. The core of its computational model is based on the bulk-synchronous parallel (BSP) processing model [37], which is an abstract machine model consisting of parallel processors that communicate by sending messages. Computations proceed synchronously in a sequence of supersteps,¹ separated by global synchronization. Messages between parallel processors are sent at the end of a superstep and are available for processing at the beginning of a superstep. BSP achieves high scalability in distributed systems by performing computations in parallel. Additionally, the BSP model ensures that a parallel program is free of deadlocks and data races, due to its synchronous, shared-nothing architecture, hence making parallel programming easy.

In vertex-centric programming, users specify an input graph and express a graph algorithm as a vertex program [21]. The vertex program contains a vertex's state variables and a compute function that describes the behavior of a vertex, such as processing messages, performing local computations, and sending messages to other vertices. Computations proceed in a sequence of supersteps, in each of which vertices execute compute in parallel for exactly once.

To clarify, let s_0 be the value of the initial state of an arbitrary vertex in a graph and s_k be the value of this vertex's state after executing compute in the k -th superstep of a graph algorithm, then we can describe the change of this vertex's state using the following state transition, where each arrow is labeled with the transition name, that is, executing compute:

$$s_0 \xrightarrow{\text{compute}} s_1 \xrightarrow{\text{compute}} s_2 \xrightarrow{\text{compute}} s_3 \dots$$

For a complex vertex program that executes different instructions across supersteps, interpreting compute is inefficient. We demonstrate it with the following example written in Scala.² Let A, B, and C represent three blocks of instructions. In every superstep, a vertex needs to check whether

¹In the BSP model, a step refers to a primitive operation like reading or writing the processor's local memory, similar to the PRAM model. The term *superstep* emphasizes that a processor can perform arbitrarily many steps before synchronizing with other processors.

²We will use Scala syntax throughout this paper.

<pre> 1 var color: Int = 0 2 def compute(): Unit = { 3 if (superstep == 0) { 4 color = 1 5 // syntactically correct 6 // but semantically wrong 7 if (superstep == 1) 8 color = 2 9 if (superstep == 2) 10 color = 3 11 } 12 } </pre>	<pre> 1 var color: Int = 0 2 @lift 3 def compute(): Unit = { 4 color = 1 5 wait() 6 color = 2 7 wait() 8 color = 3 9 wait() 10 } </pre>	<pre> 1 var color: Int = 0 2 var idx: Int = 0 3 var yield: Boolean = false 4 val sc = Array(5 ()=>{color=1; yield=true; idx=1}, 6 ()=>{color=2; yield=true; idx=2}, 7 ()=>{color=3; yield=true; idx=0}) 8 def compute(): Unit = { 9 yield = false 10 while (!yield) { 11 sc(idx)() 12 } 13 } </pre>
--	--	--

(a) Dispatching on the value of a superstep is error-prone. (b) A staged vertex program expressed with `wait()`. (c) Vertex program generated from the staged vertex program in (b).

Figure 1. In a vertex program, describing different vertex behavior across supersteps is commonly achieved by branching on a superstep value. However, this suffers from interpretation overhead and permits erroneous programs like (a), where instructions that ought to be executed in superstep 1 (line 8) can still be expressed as instructions for superstep 0 (lines 4–10) instead. We eliminate the need to dispatch on the superstep value by introducing a staging-time coroutine-like instruction `wait()` in (b), which transforms vertex behavior into an array of closures in the generated vertex program in (c).

the current superstep value is 0 (line 2) or 1 (line 4), which is unnecessary and inefficient for a long-running program.

```

1  def compute(): Unit = {
2    if (superstep == 0) {
3      A // arbitrary computation, same for B, C
4    } else if (superstep == 1) {
5      B
6    } else {
7      C
8    }
9  }

```

The need for complex vertex programs where vertices have different behavior across supersteps arises for agent-based simulations, where each vertex is regarded as an agent. Per superstep, an agent performs arbitrary local computations, sends messages, and processes received messages.³ In such applications, agents often engage in diverse operations across supersteps, and multiple types of concurrent agents exhibit distinct behaviors within the same superstep. For example, an epidemics simulation can model both hospitals and people as agents, which have different behaviors. People agents also behave differently across supersteps as the disease progresses.

We would like a vertex to only execute instructions that are *useful*. Intuitively, in superstep 1, a vertex only needs to execute line 5, without having to evaluate the conditions on lines 2 and 4 first. We capture the intuition of useful instructions for a given superstep in `compute` using notations

from partial evaluation [14]. Let α be a meta-algorithm that partially evaluates `compute` when the variable `superstep` has value t . We use $\alpha(\text{compute}, t)$ to denote the residual program generated after the partial evaluation of `compute` with respect to the variable `superstep` at value t , which is also referred to as *useful* instructions for superstep t in `compute`. The value of a vertex's state when only executing useful instructions in each superstep changes as below:

$$s_0 \xrightarrow{\alpha(\text{compute}, 0)} s_1 \xrightarrow{\alpha(\text{compute}, 1)} s_2 \xrightarrow{\alpha(\text{compute}, 2)} s_3 \dots$$

In our example, $\alpha(\text{compute}, 0) = A$, $\alpha(\text{compute}, 1) = B$.

Additionally, eliminating the need to branch on the value of supersteps has the benefit of preventing erroneous programs like Figure 1a, where instructions that describe behavior in superstep 1 (line 8) are specified in the scope of instructions to be executed in superstep 0 (lines 3–11).

In this work, we describe how we ensure that vertices only execute useful instructions by making the superstep-dependent structure in a vertex program explicit through a coroutine-like [7] staging-time instruction `wait()`.⁴ Just like coroutines, computations of a vertex function yield when executing `wait()` and later resume from the last saved execution point when the next superstep begins, avoiding the interpretation overhead caused by dispatching on the value of a superstep. The previous pseudocode can be expressed as follows using `wait()`:

³For simplicity, we assume that the semantics of an agent-based simulation is precisely the BSP model, where computations of parallel agents proceed iteratively in a sequence of supersteps. An agent-based simulation can be implemented as a vertex program.

⁴As pointed out by a reviewer, the term `wait` may cause confusions with similarly named primitives in *asynchronous* programming. Although a vertex program is single-threaded and sequential, `wait()` is a blocking instruction that delays the execution of later instructions until the next superstep starts, not unlike asynchronous programming.

```

1 def compute(): Unit = {
2   A // arbitrary computation, same for B, C
3   wait()
4   B
5   wait()
6   while(true){
7     C
8     wait()
9   }
10 }

```

We implement `wait()` in the programming model of a distributed agent-based simulation system *CloudCity* [35] and evaluate the performance benefit of staging compared with unstaged vertex programs. Our results show that for an agent with 5 to 20 branches, staging reduces the total execution time by 10% – 25%. As we increase the number of agents, the overall hardware utilization efficiency decreases as resource contention worsens. For 1000 agents, the speedup of staging is around 10%. Though the speedup is modest in both cases, staging greatly improves the robustness of the performance. We repeat each experiment three times: the standard deviation is 10× lower for staged programs than unstaged ones, for both one and 1000 agents.

The rest of the paper is structured as follows. We start by explaining in more detail what we mean by vertex-centric programming and agent-based simulations in Section 2. Afterward, we describe the limitations of the current approach in Section 3 and possible solutions. In Section 4, we examine the staging-time `wait()` instruction closely. We then discuss how we implement `wait()` in *CloudCity* in Section 5 and evaluate the performance impact in Section 6. Finally, we discuss related work in Section 7 and end this paper with conclusions and future works in Section 8.

2 Background

Vertex-centric programming was first proposed and popularized by Pregel [21]. Later other frameworks like GraphX [39] and Flink [9] have also adopted this paradigm for parallel graph processing, implementing the vertex-centric paradigm via a Pregel-like operator. Though there are differences among such frameworks concerning details such as whether the graph topology is mutable, all these frameworks share the same assumption as Pregel, where vertices interpret the same user-defined function iteratively in every superstep.

In this section, we use Pregel as an example and explain its syntax and semantics to familiarize users with vertex-centric programming. More concretely, we show how to implement a classic graph algorithm, PageRank [3], which represents the target applications that Pregel is designed for. We also explain what agent-based simulations are and how they can be expressed as complex vertex programs.

```

1 abstract class Vertex[VertexValue, EdgeValue,
    MessageValue] {
2   var value: VertexValue
3   val vertexId: String
4   val numVertices: Int
5
6   def superstep: Int = 0
7
8   def compute(msgs: Iterator[MessageValue]):
    Unit
9   // type Edge is built-in
10  def getOutEdgeIterator(): Iterator[Edge]
11  def sendMessageTo(dest: String, message:
    MessageValue): Unit
12  def voteToHalt(): Unit
13 }

```

Figure 2. Core Pregel DSL.

2.1 Pregel syntax

Pregel is a domain-specific language (DSL) for vertex-centric computing embedded purely in the host language C++ [21]. For demonstration, we present Pregel using Scala pseudocode, summarized in Figure 2. Each vertex has a unique id (line 3) and can obtain the current superstep (line 6). Users need to define `compute` (line 8), which specifies the behavior of a vertex that is executed in each superstep, such as updating local states and sending messages to neighbors (lines 2, 10–11). If there is no further work until new messages arrive, a vertex votes to halt (line 12).

Users define a vertex program by creating a subclass that extends the `Vertex` class. In particular, users provide types for `VertexValue`, `EdgeValue`, and `MessageValue` (line 1) and override the `compute` method (line 8) with vertex behavior. A graph algorithm can only define one vertex program.

2.2 Pregel semantics

The semantics of Pregel closely follows the BSP processing model [21], which is an abstract parallel machine that is commonly used by distributed frameworks for parallel computing. This abstract machine contains:

- a set of processors. Each processor can be viewed as a core-memory pair, where the memory is private to the core. A core can perform arbitrary computations over values stored in its memory. Processors communicate by sending messages, which arrive at the beginning of a superstep;
- a synchronization facility to synchronize all processors periodically.

A Pregel program proceeds in a sequence of supersteps separated by global synchronization. Initially, all vertices are *active*. Per superstep, every active vertex executes its `compute` method in parallel. A superstep ends when all active

vertices have completed executing `compute`. An active vertex becomes *inactive* by voting to halt (through the Pregel instruction `voteToHalt`), suggesting that it has no more work to do until new messages arrive. An inactive vertex becomes active if it has received at least one message at the beginning of a superstep, and remains inactive otherwise. Messages are sent at the end of a superstep and are available for processing at the beginning of the next superstep. In Pregel, a program terminates when all vertices have become *inactive* and there is no more message in the system.

Example 2.1 (PageRank). The PageRank algorithm computes the importance of web pages based on the assumption that a page is more important if it receives links from other important pages [3]. Each page has a PageRank value, where a higher value suggests that a page is more important. This algorithm proceeds in the following two phases:

- Initialization: Assign an initial PageRank value to each page in the system. Initially, all pages have equal PageRank values;
 - Iterative Update:
 - For each page, calculate its new PageRank value based on the incoming links from other pages.
 - Distribute a fraction of the current page's PageRank to the pages it links to. The amount distributed is proportional to the current page's PageRank and inversely proportional to the number of its outgoing links.
 - Update the PageRank values for all pages.
- Repeat the above steps until the PageRank values converge (no significant changes).

To account for user behavior, PageRank introduces a damping factor that models the probability of a user continuing to click on links instead of jumping to a random page, typically set to 0.85. The damping factor adjusts the distribution of a page's PageRank during the iterative update.

Implementing PageRank in Pregel is straightforward, shown in Figure 3 (adapted from [21]). Each vertex has a local state that denotes its current PageRank value (type `double`). In superstep 0, the PageRank value of each vertex is the same, given in a separate input graph file (not shown here). Per superstep, vertices send messages to their neighbors (lines 14–16), where each message contains its current PageRank value divided by the number of outgoing edges. Since messages take one superstep to arrive, starting from superstep 1, each vertex also processes all received messages and updates its value of PageRank before sending messages to neighbors (lines 7–11). In this example, the PageRank algorithm runs only for 30 supersteps (line 13); vertices vote to halt afterward (line 18).

2.3 Agent-based simulations

Agent-based simulations are simulations in which a number of agents, each with their own thread, code, and state,

```

1  class PageRankVertex extends Vertex[Double,
    Unit, Double] {
2    // Attributes value, vertexId, numVertices,
3    // getOutEdgeIterator are initialized,
    omitted
4
5    override def compute(msgs: Iterator[Double])
      : Unit = {
6      if (superstep >= 1){
7        var sum: Double = 0
8        while (msgs.hasNext) {
9          sum += msgs.next()
10        }
11        value = 0.15 / numVertices + 0.85 * sum
12      }
13      if (superstep < 30) {
14        val n: Int = getOutEdgeIterator().
          toIterable.size
15        // A syntactic sugar for sending
          messages to neighbors
16        sendMessageToAllNeighbors(value / n)
17      } else {
18        voteToHalt()
19      }
20    }
21  }

```

Figure 3. PageRank implementation.

interact in a virtual environment. These simulations enable users to make changes to the micro behavior of agents and observe the collective impact of such changes at the macro scale. An agent may execute arbitrary code, affecting its own state as well as the state of the virtual world and any other agents living within.

There are two natural flavors of agent-based simulations, synchronous and asynchronous. Here we only consider synchronous agent-based simulations, where computations proceed in a sequence of locksteps, just like supersteps in the BSP model. An agent can be viewed as a vertex, computing locally and communicating with other agents synchronously; an agent-based simulation can be expressed as a graph algorithm using a vertex program.

Example 2.2 (Traffic light simulation). We consider a minimal traffic simulation that models a pedestrian and a traffic light. The traffic light repeatedly iterates over three colors: green, yellow, and red. By default, the traffic light waits for three supersteps for each color. Additionally, the traffic light has a pedestrian crossing button, which causes the current signal to change to green when activated. Initially, the traffic light is in a random color and notifies the pedestrian of the signal. Whenever the traffic signal changes, the traffic light notifies the pedestrian of the new color. The pedestrian wants to cross and waits patiently for the green light. If the light is yellow, the pedestrian presses the crossing button. If red, the

pedestrian waits for up to two supersteps before pressing the crossing button. The traffic light turns green when receiving an activation signal from the pedestrian button.

This example highlights the need to define different vertex behavior within the same superstep in agent-based simulations, which is achieved by dispatching on the vertex id in the current vertex-centric programming, incurring unnecessary interpretation overhead.

3 Limitations of Vertex-Centric Programming

Vertex-centric programming has gained wide popularity due to its simple yet flexible programming model. Still, there are several limitations to this approach, which we describe in this section.

3.1 Interpretation overhead caused by dispatching on the value of a superstep

In Pregel, users dispatch on the value of supersteps to express different vertex behavior across supersteps. Our example in Section 2 also illustrates this problem. The code snippet in Figure 3 (lines 6, 13) clearly demonstrates the overhead of dispatching on the value of supersteps.

One standard approach to eliminate such interpretation overhead is to generate a specialized program that is more efficient to execute by partially evaluating compute with respect to a superstep value t .

Since the value of the current superstep t is only changed by the Pregel runtime system and cannot be reassigned by a vertex program, we can lift compute to pass the superstep value t as an additional input argument:

$$\overline{\text{Compute}}^t(t : \text{Int}, \text{msgs} : \text{Iterator}[\text{Message}]),$$

and replace function applications of superstep in compute in a vertex program with t via β -reduction.

Generating a specialized program with respect to t by partially evaluating $\overline{\text{Compute}}^t$ with respect to t can be described using the following transformation [14]:

$$\overline{\text{Compute}}^t(t, \text{msgs}) = \alpha(\overline{\text{Compute}}^t, t)(\text{msgs}),$$

where α is a partial evaluation algorithm that takes the program $\overline{\text{Compute}}^t$ and a superstep value t as partial evaluation variables and produces a residual program that is specialized for the superstep t . For our analysis, the details of α are not important. We assume that α executes $\overline{\text{Compute}}^t$ symbolically on a given value of t .

Assuming that the test expression of a control flow that contains t can be evaluated at specialization time [17], the partial evaluation approach requires precomputing a specialized program $\alpha(\overline{\text{Compute}}^t, t)$ for each possible value of t from 0 to K for a Pregel program that runs for K supersteps, as shown in Figure 4 for a vertex program that executes for

```

1  var idx: Int = 0
2  val instructions = Array(
3    () => { $\alpha(\overline{\text{Compute}}^t, 0)$ ; idx = 1},
4    () => { $\alpha(\overline{\text{Compute}}^t, 1)$ ; idx = 2},
5    () => { $\alpha(\overline{\text{Compute}}^t, 2)$ ; idx = 3},
6    ...,
7    () => { $\alpha(\overline{\text{Compute}}^t, 29)$ ; idx = 30},
8    () => {idx = 30}
9  )
10 def compute(msgs: Iterator[Message]): Unit = {
11   instructions(idx)()
12 }
```

Figure 4. A Pregel program with partial evaluation.

30 supersteps. The attribute instructions is an array of generated programs $\alpha(\overline{\text{Compute}}^t, t)$ indexed by t . At each superstep, only useful instructions in the generated program $\alpha(\overline{\text{Compute}}^t, t)$ are interpreted, hence more efficient than executing compute.

But precomputing residual programs for t may not always be feasible, since the test expression in a control flow that contains t can be dynamic, where t is compared to a dynamic variable whose value is known only at runtime. In addition, this approach makes it difficult to exploit *locality* across supersteps, where a vertex program repeatedly executes the same instructions in different supersteps. For instance, in the PageRank example,

$$\alpha(\overline{\text{Compute}}^t, 1) = \alpha(\overline{\text{Compute}}^t, t), \text{ for } 1 < t < 30.$$

Instead of repeatedly computing $\alpha(\overline{\text{Compute}}^t, t)$ for $1 \leq t < 30$, we would like to only compute $\alpha(\overline{\text{Compute}}^t, 1)$.

3.2 Lack of binding constructs for instructions executed in different supersteps

In vertex-centric programming, there is no binding construct for instructions executed in different supersteps. In particular, this permits erroneous programs like in Figure 1a, where it is syntactically correct to express instructions that should be executed in different supersteps within the same superstep, which are semantically incorrect.

More concretely, we summarize a simplified grammar for vertex-centric programs in Figure 5, which contains two data types, Booleans (line 1) and Integers (line 2), and selected operations (line 5) over these data types. The superstep (line 3) is a constant expression whose value is that of the current superstep. A program defined in compute is an expression in *Exp* (line 6). This allows for incorrect programs like Figure 1a, where instructions that should be executed in superstep 1 are captured in the scope of instructions that should be executed in superstep 0 instead.

We observe that this problem can be addressed by restricting the usage of superstep in *Exp* (line 6). We introduce an

```

1  Booleans b ∈ B = {true, false}
2  Integers n ∈ N = {..., -1, 0, 1, ...}
3  ConstSymbol s = {superstep}
4  VariableSymbols v ∈ Sym
5  Operations op ∈ {+, ≥, ==}
6  Expr e ∈ Exp ::= n | b | s | v := e | e op e |
    if e then e else e | e; e | while e do e

```

Figure 5. A simplified grammar for vertex-centric programs.

explicit binding construct:

```
sc(step : Int, inst : () => Unit) : Unit,
```

which allows programmers to express instructions that should be executed in different supersteps. The *step* is an integer that denotes the value of a superstep, and *inst* denotes a block of instructions that should be executed in the given superstep.

The compute method contains a sequence of *sc* instructions. In each superstep, compute evaluates the closure specified in the *sc* with the current superstep value, defaulting to no action if no such bindings are defined.

The example in Figure 1a can be expressed as below.

```

1  val sc = Mutable.Map[Int, () => Unit]()
2  var color: Int = 1
3  def compute(): Unit = {
4    sc(0, () => {color = 1})
5    sc(1, () => {color = 2})
6    sc(2, () => {color = 3})
7
8    // a syntactic sugar for
9    // sc.getOrElseDefault(superstep, ()=>Unit)()
10   sc(superstep)()
11 }

```

In this example, lines 4–6 specify a sequence of closures in the compute method. Per superstep, the closure defined for the current superstep is evaluated (line 10).

3.3 Code duplication

While introducing a binding construct *sc* can eliminate incorrect programs like Figure 1a, this can lead to undesirable code duplication when a vertex has shared instructions across supersteps, such as in intricate branching patterns. To demonstrate, we show how the vertex program in Figure 3 can be expressed using the new binding construct below.

```

1  val sc = Mutable.Map[Int, () => Unit]()
2  def compute(msgs: Iterator[Double]): Unit = {
3    sc(0, () => {
4      val n: Int = getOutEdgeIterator().
        toIterable.size
5      sendMessageToAllNeighbors(value / n)
6    })
7    Range(1, 30).foreach(idx => {
8      sc(idx, () => {

```

```

9      var sum: Double = 0
10     while (msgs.hasNext) {
11       sum += msgs.next()
12     }
13     value=0.15/numVertices+0.85*sum
14     val n: Int = getOutEdgeIterator().
        toIterable.size
15     sendMessageToAllNeighbors(value / n)
16   })
17 }
18 sc(superstep)()
19 }

```

Compared with Figure 3, readers may notice that there is no longer a *voteToHalt* instruction in the body of *compute*. We point out that this is because we have assumed a different termination condition than that of naive Pregel, to make it more suitable for agent-based simulations: a user specifies a fixed number of supersteps total that a vertex program should run. This is desirable for agent-based simulations, where users may want to express that an agent waits for some supersteps before performing another action. Without such changes, vertices that are waiting to perform the next action will be considered *inactive* and Pregel simply terminates when all vertices are waiting.⁵

In the code above, we see that lines 4–5 and lines 14–15 are duplicated, which is not the case for the Pregel implementation in Figure 3. To address this, we want to adjust the granularity of the closures and let each superstep execute multiple closures instead of one. In particular, we bind shared instructions into closures and reference them using the De Bruijn index [6], specifying the index of a continuation at the end of each closure. To illustrate, the previous code snippet can be refactored as below, where *idx* (line 1) denotes the De Bruijn index of the continuation and *sc* is an array of closures (lines 2–15). Now there is no more duplicated code in the closures on lines 4–6 and lines 9–14.

```

1  var idx: Int = 0 // De Bruijn index
2  val sc: Array[() => Unit](
3  () => {
4    val n: Int = getOutEdgeIterator().toIterable
        .size
5    sendMessageToAllNeighbors(value / n)
6    idx = 1
7  },
8  () => {
9    var sum: Double = 0
10    while (msgs.hasNext) {
11      sum += msgs.next()
12    }
13    value=0.15/numVertices+0.85*sum
14    idx = 0
15  })

```

⁵Assume that there are no messages in the system.

What is missing from the code above is how to determine which closures to execute in a given superstep. In this example, superstep 0 should execute closure 0, and each of the later supersteps should execute closures 1 and 0. We observe that the computation of a new superstep resumes precisely from the state where computations have yielded in the last superstep, similar to coroutines [7]. Hence we introduce a yield flag that is similar to the yield coroutine instruction. In any superstep, a vertex program continues evaluating the indexed closures until yield becomes true. The full vertex program for the PageRank example can be described below.

```

1  var idx: Int = 0 // De Bruijn index
2  var yield: Boolean = false
3  val sc: Array[() => Unit](
4  () => {
5    val n: Int = getOutEdgeIterator().toIterable
      .size
6    sendMessageToAllNeighbors(value / n)
7    yield = true
8    idx = 1
9  },
10 () => {
11   var sum: Double = 0
12   while (msgs.hasNext) {
13     sum += msgs.next()
14   }
15   value=0.15/numVertices+0.85*sum
16   idx = 0
17 })
18
19 def compute(): Unit = {
20   yield = false
21   while (!yield) {
22     sc(idx)()
23   }
24 }
```

It is easy to verify that in superstep 0, only closure 0 (lines 6–9) is executed; in later supersteps, both closures 1 (lines 12–17) and 0 are executed, just as expected.

3.4 Explicit dependency on the value of vertex ids

In Pregel, users define only one vertex program for a graph algorithm. To express different behavior in the same superstep, as illustrated in the traffic light simulation example, users need to branch on the value of vertex ids. Similar to the problem of branching on the value of supersteps as we have just discussed, conditioning on the value of vertex ids is error-prone and inefficient. This can be addressed by allowing polymorphic vertices, where users define multiple vertex classes that correspond to different types of vertices, which is straightforward.

4 Staging-Time wait() Instruction

We have discussed various limitations of the existing vertex-centric programming and how such limitations can be addressed. In particular, we have shown how users can specify the behavior of a vertex as a sequence of closures using the binding construct `sc` to avoid:

- interpretation overhead caused by dispatching on the value of a superstep;
- incorrect programs where instructions that should be executed in superstep t are specified in the body of the branch for superstep k instead ($t \neq k$); and
- duplicated code that arises from executing a single closure in a superstep.

Still, defining such closures and their continuation directly by users is low-level and error-prone. We want to provide a high-level user-friendly interface and let the system automatically generate such closures.

To this end, we introduce a staging-time instruction `wait()` that separates instructions in a vertex program into different supersteps, making it easy to automatically generate a sequence of closures based on static analysis at staging time.⁶

The PageRank example can be expressed below using `wait()`. Users no longer need to create different closures or specify the index of the continuation that points to the closure that should be executed next.

```

1  @lift
2  def compute(msgs: Iterator[Double]): Unit = {
3    while (true) {
4      val n: Int = getOutEdgeIterator().
        toIterable.size
5      sendMessageToAllNeighbors(value / n)
6      wait()
7      var sum: Double = 0
8      while (msgs.hasNext) {
9        sum += msgs.next()
10     }
11     value=0.15/numVertices+0.85*sum
12   }
13 }
```

This program is a staged program – as suggested by the `@lift` annotation on line 1 – that is lifted into some intermediate representation for preprocessing before generating instructions. In the generated code, an array of closures that specifies the behavior of a vertex is created outside the scope of `compute`, which is executed only once during initialization, like other state variables of the vertex.

⁶For the ease of presentation, we only discuss `wait()` with arity 0. In practice, we implement a 1-arity `wait(n: Int)`, which blocks until n supersteps have passed. This makes it straightforward to specify a vertex with idle behavior across supersteps. For example, a vertex can perform computations only in supersteps 0 and 100. In this case, users can use `wait(99)` to clarify that a vertex is idle and has no behavior for 99 supersteps. Here we focus on explaining how `wait()` separates instructions into different supersteps.

5 CloudCity

We implement the staging-time `wait()` instruction in CloudCity [35], a distributed agent-based simulation system. An *agent* is single-threaded and communicates by sending messages. For this purpose, each agent has a mailbox and is uniquely addressable. Incoming messages are buffered in the mailbox, waiting passively to be processed by the agent. An agent-based simulation consists only of such agents.

The software stack of CloudCity is shown in Figure 6. There are four layers in total: users interact with the core indirectly via the programming model in the *frontend*; *core* refers to the distributed engine for agent-based simulations; *optimizations* contain system optimizations, both at compile-time and runtime; *backend* refers to the low-level distributed library used in the implementation.

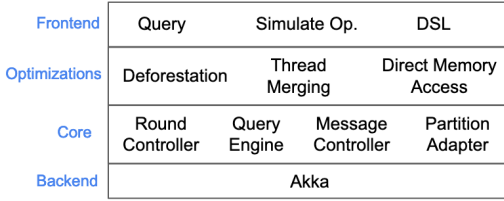


Figure 6. Software stack of CloudCity. The *core* refers to the distributed agent-based simulation engine. Users interact with the core indirectly via the programming model in the *frontend*. There are different runtime and compile-time system optimizations, summarized in the layer *optimizations*. The *backend* refers to the low-level distributed library used in the implementation.

Our discussion here concerns only the DSL in the frontend. For simplicity, we assume that computations proceed in a sequence of supersteps, exactly as described in BSP, and all messages take one superstep to arrive.⁷

The DSL is embedded in the host language Scala, shown in Figure 7: `wait()` is a synchronization instruction since it plays the role of a barrier instruction that synchronizes all agents at the end of a superstep.⁸

The DSL also contains communication instructions that let agents communicate in a distributed environment. To send a message, an agent calls

```
send(rid : Long, m : Message) : Unit,
```

where *rid* is the id of the receiver agent and *m* is a message object. Message class is defined in our library and can be extended. Retrieving a message is done via

```
receive() : Option[Message],
```

⁷Compare with [35], we eliminate *delay* from the signature of RPC instructions `callAndForget` and `asyncCall` for simplicity.

⁸For simplicity, here we consider `wait()`, as we have discussed in the previous section. This is different from `wait(n) : Unit` in [35], which takes an integer parameter. The semantics of `wait()` is the same as `wait(1)`.

which returns `None` if the mailbox is empty.

Together, `send` and `receive` implement the basic message-passing protocol, but this protocol places no restriction on what messages contain and how messages are processed. Specifically, a message can be ill-formed and does not contain all arguments that are required for processing. In practice, it is desirable to ensure that messages are well-formed. Hence, CloudCity also supports remote procedure calls (RPCs), a special type of message-passing protocol that limits senders to only send valid messages that can be processed by receivers. RPCs have two types of messages, requests and replies.

We view public methods in an agent program as *RPC methods*. An RPC request message contains the identifier of an RPC method defined in the receiver. When processing an RPC request, the receiver looks up the corresponding RPC method and invokes it locally. For performance reasons, we differentiate two types of RPC requests in our DSL, depending on whether a receiver sends an RPC reply message: a *two-sided* RPC request requires the receiver to send an RPC reply message back to the sender, which contains the return value of the local call; a *one-sided* RPC request does not have an associated reply.

Agents send a two-sided RPC request message with

```
asyncCall(() => receiver.API(args*) : T) : Future[T],
```

which returns a future object used by the sender to check whether the RPC reply has arrived and to retrieve the return value in the RPC reply. A future object has type `Future[T]`, which is defined in the CloudCity library, but with a similar usage as that in the standard Scala library. *T* is a type variable that denotes the type of return value in an RPC reply. Sending a one-sided RPC request can be achieved using

```
callAndForget(() => receiver.API(args*) : T) : Unit,
```

which does not return a future object.

To process RPC request messages, an agent can repeatedly call `receive`, parse the message, and invoke the corresponding RPC methods. Alternatively, DSL allows a receiver to retrieve and process RPC requests in batch via

```
handleRPC() : Unit,
```

which traverses received messages in an arbitrary order. Per RPC request, the receiver calls the corresponding method and sends a reply when applicable. An RPC reply has the same delay and `sessionId` as the corresponding request.

RPC instructions like `asyncCall` and `callAndForget` are convenient for users by isolating them from low-level implementation details such as how to package the corresponding RPC message. CloudCity automatically derives available RPC methods from agent class definitions and assigns these methods a unique method identifier at specialization time before generating the corresponding message. In other words, such instructions are *code generators* that emit different instructions during staging time. Similarly, `handleRPC` is also a

code generator that can produce different instructions for different agent class definitions.

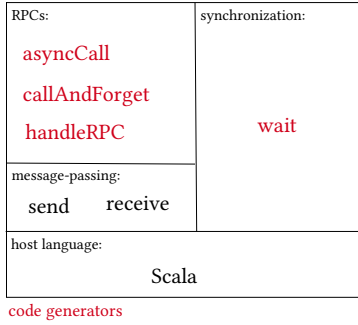


Figure 7. DSL stack. CloudCity DSL is embedded in the host language Scala and contains three components: message-passing, RPCs, and synchronization. While message-passing instructions can be implemented as Scala functions directly, instructions in RPCs and synchronization (shown in red) are code generators that require binding-time analysis to dynamically generate corresponding Scala instructions.

An agent program defined using DSL is a metaprogram since it contains code generators. The metaprogram is lifted using Squid library [25] into A-normal form (ANF) [13] as an intermediate representation (IR), before generating the corresponding instructions and being transformed into a Scala program. The ANF names intermediate results to avoid code duplication, generating one or more closures for each user instruction in the vertex program, which are later combined to reduce the number of closures. The T-diagram of CloudCity is shown in Figure 8.



Figure 8. An agent program in CloudCity written in DSL (left) which contains `wait()` is lifted using Squid into ANF before being transformed into a Scala program (right).

More concretely, we show how to implement PageRank in CloudCity below. Line 1 is the annotation for using the class-lifting macro in Squid. The `neighborRank` (lines 4–6) is an RPC method that can be specified in RPC requests (line 12). The behavior of an agent is defined in the `main` method (lines 7–15). The `outNeighbors` is an attribute defined in the Agent class in CloudCity, which contains a collection of neighboring agents that an agent can send messages to.

```

1 @lift
2 class Vertex(var value: Double) extends Agent{
3   var sum: Double = 0
4   def neighborRank(s: Double): Unit = {

```

```

5     sum += s
6   }
7   def main(): Unit = {
8     while (true) {
9       sum = 0
10      handleRPC()
11      value = 0.15/numVertices+0.85*sum /
12      outNeighbors.size
13      outNeighbors.foreach(i => callAndForget
14      (())=>i.neighborRank(value)))
15      wait()
16    }
17  }

```

After defining a vertex program, users lift it using Squid into an ANF IR and compile the IR into a Scala program:

```

1 // Squid representation of the lifted class
2 val cls = Vertex.reflect(IR)
3 // CloudCity compiler that translates Squid IR
4 // to generate (write) a target Scala program
5 compile(cls)

```

CloudCity compiler rewrites the ANF IR using Squid. For example, the rewrite rule for `callAndForget` using code pattern matching in Squid is:

```

1 cde match {
2   case code"callAndForget[$mt]((())=>${
3     m@MethodApplication(msg)}:mt, $t:Int)"
4     =>
5     val receiverActorVar = msg.args.head.head
6     val argss = msg.args.tail.map(a => a.map(
7       arg => code"$arg")
8     )
9     val methodId = methodIdMap(msg.symbol.
10      asTerm.name)
11     Send[T](receiverActorVar, methodId, t,
12      argss)
13 }

```

On line 1, `cde` is the lifted agent definition in IR. `Send` (line 6) is a CloudCity compiler IR that is later translated to Scala. Similarly for other code generator instructions.

CloudCity compiler generates an array of closures that describe the agent behavior, as we have discussed in Section 4. This is achieved by walking the abstract syntax tree of the agent behavior defined in the `main` method, where each node of the tree is transformed into a closure. Closures that correspond to a sequence of straight-line instructions that do not contain branches or `wait()` are later merged into one closure to reduce the number of closures.

6 Evaluation

In this section, we quantify the performance benefit of eliminating the need to dispatch on the value of supersteps empirically. For hardware, we use a server that has 24 cores (two Intel Xeon E5-2680 v2, 48 hardware threads), 128GB of RAM, and 200GB of SSD.

We evaluate both approaches – with and without eliminating dispatching overhead – using a microbenchmark with nested branching, where if-statements are nested within other if-statements, for up to two levels. In the innermost body of a condition, we model simple computations where an agent performs some calculations and assigns the result. For instance, the pseudocode for a vertex program with three branches is shown below, for one and two levels respectively.

```

1 // one level
2 def compute(): Unit = {
3   if (time % period == 0)
4     color += Random.nextInt() + 1
5   else if (time % period == 1)
6     color += Random.nextInt() + 2
7   else if (time % period == 2)
8     color += Random.nextInt() + 3
9   ...
10 }
11
12 // two levels
13 def compute(): Unit = {
14   if (Random.nextBoolean()) {
15     if (time % period == 0)
16       color += Random.nextInt() + 1
17     else if (time % period == 1)
18       color += Random.nextInt() + 2
19     else if (time % period == 2)
20       color += Random.nextInt() + 3
21     ...
22   }
23   ...
24 }

```

Per experiment, we measure the total execution time when running 1000 supersteps, both with and without staging, and repeat three times after warming up the cache. We first consider a single agent and increase the number of the innermost branches from 5 to 20, for both one and two levels, shown in Figure 9. The x-axis denotes the number of branches and the y-axis denotes the total execution time of each experiment, averaged over three runs. The standard deviation is shown using error bars. The blue bars that are shaded with slashes (on the left) show the results for experiments without staging, and the orange bars filled with circles (on the right) represent the results after applying staging.

Figure 9 shows that for a single agent, staging improves the overall average execution time by 10% to 25% for one level branch, and 10% to 15% for nested branches of two levels. In addition, experiments with staging behave much more robust than those without staging, as evident from significantly lower standard deviations for experiments with staging: 5-14× lower for one level and 3-10× lower when there are two nested levels of branches.

As we increase the number of agents from one to 1000, Figure 10 shows that staging still results in modest speedup, improving the overall average execution time by around

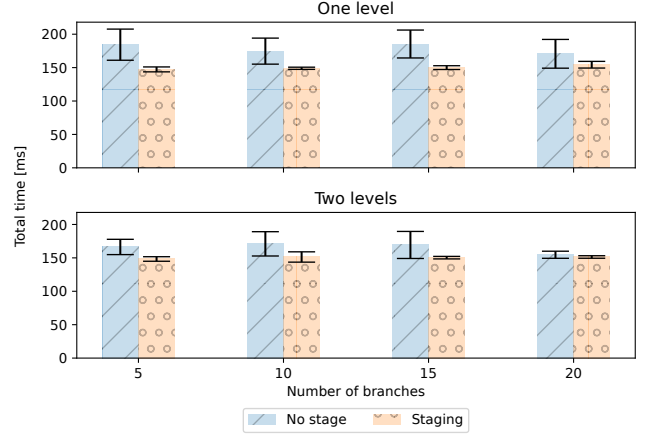


Figure 9. Increase the number of branches (one agent).

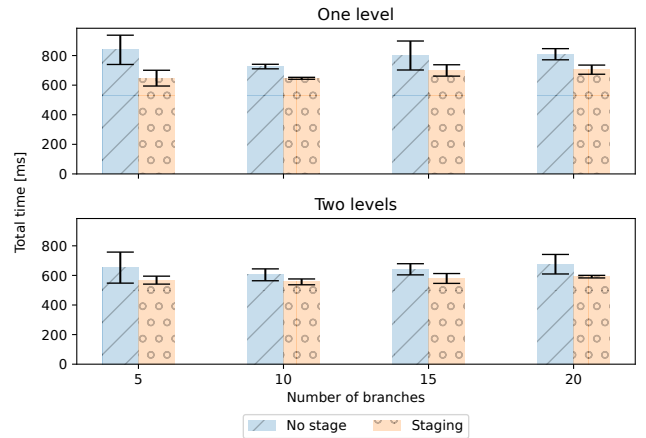


Figure 10. Increase the number of branches (1000 agents).

10%, for both one and two levels of branches. Similar to Figure 9, staging also leads to more robust performance for 1000 agents, having significantly lower standard deviation than branching, up to 10×. We note that for 1000 agents, the performance benefit of applying staging is slightly less compared with that of a single agent. This is due to the presence of other performance factors, such as increased cache contention, as the number of agents increases.

Ideally, the total time should remain unchanged (flat) as the number of branches increases with staging, after we have eliminated the overhead of dispatching. However, we see that the total time for experiments with staging increases slightly as the number of branches increases, in both Figures 9 and Figure 10. This can be caused by the overhead of locating the corresponding instructions in the array. Nevertheless, experiments with staging have overall better performance than those without staging for all experiments.

7 Related Work

Our work was initially motivated by the question of how to scale an agent-based simulation up (increase total agents) and out (increase total machines). Agent-based simulations have a long history [31] and a diverse set of simulation frameworks have been developed by domain scientists [24], but these frameworks are either single-machine [20, 22, 36] or do not scale well. To illustrate, we explain the scalability of one of the best-known distributed agent-based simulation frameworks, Repast HPC [5]. A simulation in Repast HPC is a discrete event simulation at its core: users schedule various events at different time ticks in a global schedule queue [27]. These events can be generated from agents, but can also be global events. At any time tick, a simulation proceeds by executing the scheduled events sequentially. After all such events have been processed for a given time tick, the simulation continues to the next time tick. In discrete event simulations, events that are scheduled for the same time tick are added sequentially with possible ordering constraints, hence cannot be easily parallelized.

In computer science, the bulk-synchronous parallel (BSP) processing model [37] has become standard for parallel programming [1, 8, 21, 40]: computations are performed locally over distributed data and intermediate data are combined to advance computations. This computational model also underpins vertex-centric programming, a programming paradigm for distributed graph processing: per superstep, all vertices execute computations embarrassingly parallel; the intermediate data shuffled across supersteps takes the form of messages, which are sent between vertices.

Vertex-centric programming as a programming paradigm was proposed by Google [21] and has gained wide popularity due to its simplicity and demonstrated performance. Many state-of-the-art distributed frameworks support this paradigm, which we briefly describe below.

Giraph [4] is an open-source implementation of Pregel and has been widely used by industry, including tech companies like Facebook. GraphX [39] is a graph library built on top of Spark [40], a general-purpose data flow processing framework that provides a resilient distributed dataset (RDD) abstraction that makes it easy to perform computations over distributed data. Since graph processing is an important workload in distributed analytics, GraphX provides built-in implementations of commonly used graph algorithms, which have been highly optimized to improve their performance, compared with naive implementations on Spark. One of the graph operators supported by GraphX is Pregel, which lets users program in a vertex-centric way just like Pregel. Flink [9] supports both graph and stream processing; Flink Gelly provides a Pregel operator, similar to that in GraphX, which supports vertex-centric programming for graphs. While differing in their target workloads, all such frameworks support vertex-centric programming as defined in Pregel, where

users define a single compute function that is executed by every active vertex in each superstep.

A natural question is that given all existing graph processing frameworks that already support vertex-centric programming, why is there a need for another framework for agent-based simulations? This question is addressed in [35], where our experiments showed that the performance of these existing frameworks could differ by up to three orders of magnitude when executing a benchmark consisting of representative agent-based simulation workloads selected from population dynamics, economics, and epidemics, due to different design choices of these systems.

While applying vertex-centric programming to agent-based simulations by modeling each agent as a vertex, it became clear to us that the existing programming model in the vertex-centric paradigm is tailored for graph algorithms which often have simple vertex behavior. The compute method becomes tangled with branches that dispatch on the value of supersteps as the complexity of a vertex behavior increases, such as performing different actions across supersteps, which is characteristic of agent-based simulations.

The problem that a vertex program becomes convoluted and error-prone when the control flow of the vertex program gets complex has also been examined in Fregel [12, 16], a functional library for specifying vertex behavior. A Fregel program provides a functional abstraction that hides the complex control flow from users, but is then compiled to generate a vertex program with complex control flows for existing frameworks like Giraph, hence still suffering from sources of inefficiency as we have discussed in this paper.

Our work is the first to introduce a staging-time instruction to make superstep-dependent program structures explicit in an iterative vertex program, which avoids complex control flows that dispatch on the value of supersteps and mitigates sources of inefficiency in vertex programs.

We note that staging is a standard technique used to address sources of program inefficiency like dispatching overhead, and is commonly used in areas such as designing domain-specific languages (DSL) [2, 10, 11, 15, 19, 28, 32, 33] and constructing compilers to generate more efficient code [18, 23, 26, 29, 30]. A classic example that illustrates the use case of staging is the generic power function $\text{power}(x, n)$ [34]:

```
1 def power(x: Int, n: Int): Int = {
2   if (n==0)
3     1
4   else
5     x * power(x, n-1)
6 }
```

This high-level generic function provides a nice abstraction that makes defining other special functions straightforward:

```
1 def square(x: Int): Int = power(x, 2)
2 def cube(x: Int): Int = power(x, 3)
```


However, this abstraction comes at the cost of performance. In particular, a low-level implementation of square that multiplies x with itself will be faster than invoking power, which computes the result recursively. Not all hope is lost though. On line 1, the body of square has a constant argument 2 that is known statically. Therefore, we can apply staging to rewrite the body of square with the partially evaluated program generated from power($x, 2$), making it possible to achieve both the high-level programming abstractions and the high-performance of low-level implementations.

The parallelism between how staging is applied in the classic power example and our work is clear: we similarly exploit the fact that when dispatching on the value of supersteps in vertex programs, the value of supersteps in the conditions are often known statically, which allows us to avoid such overhead by rewriting the original program with more efficient instructions that only compute useful code via staging.

8 Conclusions and Future Work

In this work, we explained the limitations of existing vertex-centric programming and described how to address such limitations by making the superstep-dependent structure in a vertex program explicit using a staging-time instruction `wait()`. We implemented this approach in an agent-based simulation framework CloudCity and demonstrated empirically that staging improves the overall performance of our experiments by 10%-25% while reducing the standard deviation of multiple runs by over 5 \times .

We have only scratched the surface of how staging can be applied in this work. Other use cases of staging include providing a platform-independent programming interface while emitting platform-specific instructions for heterogeneous hardware [19], generating a DSL implementation from a declarative specification [33], enabling a just-in-time compiler to call back into the running program to perform compile-time computations that define custom optimizations [30]. Each of these possible use cases directs to future works that can enable more aggressive compiler optimizations and alternative approaches for designing agent-based simulations. There is also potential for other optimizations in the future, such as applying program slicing [38] to dynamically resize the array of closures in a generated vertex program at runtime and discard closures that are no longer relevant.

More generally, vertex-centric programming is only a special case of the BSP paradigm. It is interesting to see whether the insight of introducing a staging-time barrier instruction that makes static scheduling explicit in vertex programs, as we have done in this work, can be generalized to applications in other BSP systems, especially for latency-critical systems where robust performances are crucial.

References

- [1] Apache Hadoop Developers. 2006. Apache Hadoop. <https://hadoop.apache.org/>
- [2] Emil Axelsson, Koen Claessen, Gergely Dévai, Zoltán Horváth, Karin Keijzer, Bo Lyckegård, Anders Persson, Mary Sheeran, Josef Svenningsson, and András Vajda. 2010. Feldspar: A domain specific language for digital signal processing algorithms. In *8th ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2010)*, Grenoble, France, 26-28 July 2010. IEEE Computer Society, Grenoble, France, 169–178. <https://doi.org/10.1109/MEMCOD.2010.5558637>
- [3] Sergey Brin and Lawrence Page. 1998. The Anatomy of a Large-Scale Hypertextual Web Search Engine. *Comput. Networks* 30, 1-7 (1998), 107–117. [https://doi.org/10.1016/S0169-7552\(98\)00110-X](https://doi.org/10.1016/S0169-7552(98)00110-X)
- [4] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. 2015. One Trillion Edges: Graph Processing at Facebook-Scale. *Proc. VLDB Endow.* 8, 12 (8 2015), 1804–1815. <https://doi.org/10.14778/2824032.2824077>
- [5] Nicholson T. Collier and Michael J. North. 2013. Parallel agent-based simulation with Repast for High Performance Computing. *Simul.* 89, 10 (2013), 1215–1235. <https://doi.org/10.1177/0037549712462620>
- [6] Nicolaas Govert De Bruijn. 1972. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. In *Indagationes Mathematicae (Proceedings)*, Vol. 75. Elsevier, 381–392.
- [7] Ana Lúcia de Moura and Roberto Ierusalimsky. 2009. Revisiting coroutines. *ACM Trans. Program. Lang. Syst.* 31, 2 (2009), 6:1–6:31. <https://doi.org/10.1145/1462166.1462167>
- [8] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *6th Symposium on Operating System Design and Implementation (OSDI 2004)*, San Francisco, California, USA, December 6-8, 2004, Eric A. Brewer and Peter Chen (Eds.). USENIX Association, San Francisco, California, USA, 137–150. <http://www.usenix.org/events/osdi04/tech/dean.html>
- [9] Flink Gelly Developers. 2022. Source code of Pregel operators in Flink Gelly. <https://nightlies.apache.org/flink/flink-docs-master/api/java/org/apache/flink/graph/pregel/>. Accessed: 2023-03-10.
- [10] JetBrain MPS Developers. [n. d.]. JetBrain MPS. <https://www.jetbrains.com/mps/>.
- [11] Zachary DeVito, James Hegarty, Alex Aiken, Pat Hanrahan, and Jan Vitek. 2013. Terra: a multi-stage language for high-performance computing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13*, Seattle, WA, USA, June 16-19, 2013, Hans-Juergen Boehm and Cormac Flanagan (Eds.). ACM, 105–116. <https://doi.org/10.1145/2491956.2462166>
- [12] Kento Emoto, Kiminori Matsuzaki, Zhenjiang Hu, Akimasa Morihata, and Hideya Iwasaki. 2016. Think like a vertex, behave like a function! a functional DSL for vertex-centric big graph processing. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, Jacques Garrigue, Gabriele Keller, and Eijiro Sumii (Eds.). ACM, Nara, Japan, 200–213. <https://doi.org/10.1145/2951913.2951938>
- [13] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. 1993. The essence of compiling with continuations (with retrospective). In *20 Years of the ACM SIGPLAN Conference on Programming Language Design and Implementation 1979-1999, A Selection*, Kathryn S. McKinley (Ed.). ACM, USA, 502–514. <https://doi.org/10.1145/989393.989443>
- [14] Yoshihiko Futamura. 1999. Partial Evaluation of Computation Process - An Approach to a Compiler-Compiler. *High. Order Symb. Comput.* 12, 4 (1999), 381–391. <https://doi.org/10.1023/A:1010095604496>
- [15] Christoph Armin Herrmann and Tobias Langhammer. 2006. Combining partial evaluation and staged interpretation in the implementation of domain-specific languages. *Sci. Comput. Program.* 62, 1 (2006), 47–65. <https://doi.org/10.1016/j.scico.2006.02.002>

- [16] Hideya Iwasaki, Kento Emoto, Akimasa Morihata, Kiminori Matsuzaki, and Zhenjiang Hu. 2022. Fregel: a functional domain-specific language for vertex-centric large-scale graph processing. *J. Funct. Program.* 32 (2022), e4. <https://doi.org/10.1017/S0956796821000277>
- [17] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. 1993. *Partial evaluation and automatic program generation*. Prentice Hall, USA.
- [18] Ulrik Jørring and William L. Scherlis. 1986. Compilers and Staging Transformations. In *Proceedings of the 13th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (St. Petersburg Beach, Florida) (POPL '86). Association for Computing Machinery, New York, NY, USA, 86–96. <https://doi.org/10.1145/512644.512652>
- [19] HyoukJoong Lee, Kevin J. Brown, Arvind K. Sujeeth, Hassan Chafi, Tiark Rompf, Martin Odersky, and Kunle Olukotun. 2011. Implementing Domain-Specific Languages for Heterogeneous Parallel Computing. *IEEE Micro* 31, 5 (2011), 42–53. <https://doi.org/10.1109/MM.2011.68>
- [20] Sean Luke, Claudio Cioffi-Revilla, Liviu Panait, Keith Sullivan, and Gabriel Catalin Balan. 2005. MASON: A Multiagent Simulation Environment. *Simul.* 81, 7 (2005), 517–527. <https://doi.org/10.1177/0037549705058073>
- [21] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6–10, 2010*, Ahmed K. Elmagarmid and Divyakant Agrawal (Eds.). ACM, Indianapolis, Indiana, USA, 135–146. <https://doi.org/10.1145/1807167.1807184>
- [22] Michael J. North, Nicholson T. Collier, Jonathan Ozik, Eric R. Tataru, Charles M. Macal, Mark J. Bragen, and Pam Sydelko. 2013. Complex adaptive systems modeling with Repast Simphony. *Complex Adapt. Syst. Model.* 1 (2013), 3. <https://doi.org/10.1186/2194-3206-1-3>
- [23] Georg Ofenbeck, Tiark Rompf, Alen Stojanov, Martin Odersky, and Markus Püschel. 2013. Spiral in scala: towards the systematic construction of generators for performance libraries. In *Generative Programming: Concepts and Experiences, GPCE '13, Indianapolis, IN, USA - October 27 - 28, 2013*, Jaakko Järvi and Christian Kästner (Eds.). ACM, 125–134. <https://doi.org/10.1145/2517208.2517228>
- [24] Constantin-Valentin Pal, Florin Leon, Marcin Paprzycki, and Maria Ganzha. 2020. A Review of Platforms for the Development of Agent Systems. *CoRR abs/2007.08961* (2020). arXiv:2007.08961 <https://arxiv.org/abs/2007.08961>
- [25] Lionel Parreaux and Amir Shaikhha. 2020. Multi-Stage Programming in the Large with Staged Classes. In *Proceedings of the 19th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences* (Virtual, USA) (GPCE 2020). Association for Computing Machinery, New York, NY, USA, 35–49. <https://doi.org/10.1145/3425898.3426961>
- [26] Markus Püschel, José M. F. Moura, Jeremy R. Johnson, David A. Padua, Manuela M. Veloso, Bryan Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nicholas Rizzolo. 2005. SPIRAL: Code Generation for DSP Transforms. *Proc. IEEE* 93, 2 (2005), 232–275. <https://doi.org/10.1109/JPROC.2004.840306>
- [27] Repast HPC developers. 2023. Repast HPC API. https://repast.github.io/docs/api/hpc/repast_hpc/classrepast_1_1_schedule.html#details Accessed: 2023-03-02.
- [28] Tiark Rompf and Martin Odersky. 2010. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. In *Generative Programming And Component Engineering, Proceedings of the Ninth International Conference on Generative Programming and Component Engineering, GPCE 2010, Eindhoven, The Netherlands, October 10–13, 2010*, Eelco Visser and Jaakko Järvi (Eds.). ACM, 127–136. <https://doi.org/10.1145/1868294.1868314>
- [29] Tiark Rompf, Arvind K. Sujeeth, Nada Amin, Kevin J. Brown, Vojin Jovanovic, HyoukJoong Lee, Manohar Jonnalagadda, Kunle Olukotun, and Martin Odersky. 2013. Optimizing data structures in high-level programs: new directions for extensible compilers based on staging. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, Roberto Giacobazzi and Radhia Cousot (Eds.). ACM, 497–510. <https://doi.org/10.1145/2429069.2429128>
- [30] Tiark Rompf, Arvind K. Sujeeth, Kevin J. Brown, HyoukJoong Lee, Hassan Chafi, and Kunle Olukotun. 2014. Surgical precision JIT compilers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, 41–52. <https://doi.org/10.1145/2594291.2594316>
- [31] Thomas C Schelling. 1969. Models of segregation. *The American economic review* 59, 2 (1969), 488–493.
- [32] Tim Sheard, Zine-El-Abidine Benaissa, and Emir Pasalic. 1999. DSL implementation using staging and monads. In *Proceedings of the Second Conference on Domain-Specific Languages (DSL '99)*, Austin, Texas, USA, October 3–5, 1999, Thomas Ball (Ed.). ACM, 81–94. <https://doi.org/10.1145/331960.331975>
- [33] Arvind K. Sujeeth, Austin Gibbons, Kevin J. Brown, HyoukJoong Lee, Tiark Rompf, Martin Odersky, and Kunle Olukotun. 2013. Forge: Generating a High Performance DSL Implementation from a Declarative Specification. In *Proceedings of the 12th International Conference on Generative Programming: Concepts & Experiences* (Indianapolis, Indiana, USA) (GPCE '13). Association for Computing Machinery, New York, NY, USA, 145–154. <https://doi.org/10.1145/2517208.2517220>
- [34] Walid Taha. 2003. A Gentle Introduction to Multi-stage Programming. In *Domain-Specific Program Generation, International Seminar, Dagstuhl Castle, Germany, March 23–28, 2003, Revised Papers (Lecture Notes in Computer Science, Vol. 3016)*, Christian Lengauer, Don S. Batory, Charles Consel, and Martin Odersky (Eds.). Springer, Germany, 30–50. https://doi.org/10.1007/978-3-540-25935-0_3
- [35] Zilu Tian, Peter Lindner, Markus Nissl, Christoph Koch, and Val Tannen. 2023. Generalizing Bulk-Synchronous Parallel Processing Model: From Data to Threads and Agent-Based Simulations. In *Proc. ACM. Management of Data (PACMOD)*. ACM, USA, 439–480. <https://doi.org/10.1145/3589296>
- [36] Seth Tisue and Uri Wilensky. 2004. Netlogo: A simple environment for modeling complexity. In *International conference on complex systems*, Vol. 21. Citeseer, Boston, MA, 16–21.
- [37] Leslie G. Valiant. 1990. A Bridging Model for Parallel Computation. *Commun. ACM* 33, 8 (1990), 103–111. <https://doi.org/10.1145/79173.79181>
- [38] Mark Weiser. 1984. Program slicing. *IEEE Transactions on software engineering* 4 (1984), 352–357.
- [39] Reynold S. Xin, Joseph E. Gonzalez, Michael J. Franklin, and Ion Stoica. 2013. GraphX: a resilient distributed graph system on Spark. In *First International Workshop on Graph Data Management Experiences and Systems, GRADES 2013, co-located with SIGMOD/PODS 2013, New York, NY, USA, June 24, 2013*, Peter A. Boncz and Thomas Neumann (Eds.). CWI/ACM, USA, 2. <https://doi.org/10.1145/2484425.2484427>
- [40] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25–27, 2012*, Steven D. Gribble and Dina Katabi (Eds.). USENIX Association, San Jose, 15–28. <https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia>

Received 2023-07-14; accepted 2023-09-03

Unleashing the Power of Implicit Feedback in Software Product Lines: Benefits Ahead

Raul Medeiros
raul.medeiros@ehu.eus
University of the Basque Country
San Sebastián, Spain

Oscar Díaz
oscar.diaz@ehu.eus
University of the Basque Country
San Sebastián, Spain

David Benavides
benavides@us.es
University of Seville
Seville, Spain

Abstract

Software Product Lines (SPLs) facilitate the development of a complete range of software products through systematic reuse. Reuse involves not only code but also the transfer of knowledge gained from one product to others within the SPL. This transfer includes bug fixing, which, when encountered in one product, affects the entire SPL portfolio. Similarly, feedback obtained from the usage of a single product can inform beyond that product to impact the entire SPL portfolio. Specifically, *implicit* feedback refers to the automated collection of data on software usage or execution, which allows for the inference of customer preferences and trends. While implicit feedback is commonly used in single-product development, its application in SPLs has not received the same level of attention. This paper promotes the investigation of implicit feedback in SPLs by identifying a set of SPL activities that can benefit the most from it. We validate this usefulness with practitioners using a questionnaire-based approach (n=8). The results provide positive insights into the advantages and practical implications of adopting implicit feedback at the SPL level.

CCS Concepts: • Software and its engineering → Software evolution; Maintaining software; Software design techniques.

Keywords: Implicit feedback, User behavior, Software Product Lines, Code generation

ACM Reference Format:

Raul Medeiros, Oscar Díaz, and David Benavides. 2023. Unleashing the Power of Implicit Feedback in Software Product Lines: Benefits Ahead. In *Proceedings of the 22nd ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
GPCE '23, October 22–23, 2023, Cascais, Portugal
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 979-8-4007-0406-2/23/10...\$15.00
<https://doi.org/10.1145/3624007.3624058>

(GPCE '23), October 22–23, 2023, Cascais, Portugal. ACM, New York, NY, USA, 9 pages. <https://doi.org/10.1145/3624007.3624058>

1 Introduction

Implicit feedback refers to collecting and analyzing user behavior and interactions with the system to enhance the user experience, prioritize features, conduct A/B testing, etc [18]. Unlike explicit feedback, which is directly provided by users through surveys, reviews, or ratings, implicit feedback is derived from user actions such as clicks, navigation patterns, and time spent on specific features. Implicit feedback is derived from the real usage of the applications, whereas explicit feedback requires direct involvement from users who are often unable to specify their needs [11]. Implicit feedback is currently a common practice for single-product development [32].

However, implicit feedback has not received the same attention for Software Product Lines (SPLs) [8]. Unlike single-product development, SPLs focus on producing a reusable set of software components known as the SPL platform. This platform is designed to handle a wide range of software products, which together conform the SPL portfolio. The process of building a product, also known as a *variant*, changes from starting from scratch to deciding which platform's features are to be exhibited by the new variant, i.e., variant configuration. This shift in the development process may also impact how implicit feedback is conducted. Traditionally, implicit feedback is collected by embedding “usage trackers” directly into the software's code (e.g., Google Analytics' hits) [32]. While this approach may work for single-product development, it goes against one of the main principles of SPL: minimizing or even eliminating direct coding into the individual variant's code, and instead deriving the variant's code directly from the platform code. In other words, usage trackers (i.e. the code fragment tracking user interaction) should also be subject to generative programming where the variant needs for usage trackers are also handled using configuration-like mechanisms. We refer to this approach as “platform-centric feedback”. Platform-centric feedback differs from “application-centric feedback” in terms of both what needs to be tracked (the variant vs the platform) and the programming approach (direct vs generative).

The relevance of collecting implicit feedback at the platform level needs to be argued in terms of the benefits it brings to different SPL engineering activities, even if it means

sacrificing agility at the variant level. Ultimately, platform-centric feedback shifts decision-making from the product unit (Application Engineers) to the platform developers (Domain Engineers), removing or at least delaying decisions on which features or variants to track. This shift to the platform level may ultimately slow down time-to-market and undermine the goal of prompt feedback. Therefore, our research question is: “What activities can benefit from incorporating platform-centric feedback into software product lines?” On addressing this question, we contribute by

- listing a set of SPL activities that can benefit implicit feedback (Sections 3, 4, and 5)
- providing first evidence about the perceived usefulness of these benefits through a questionnaire evaluation among practitioners (n=8) (Section 6)

2 A Brief on Implicit Feedback

Implicit feedback is inferred from user behavior, such as mouse clicks or time spent using particular features [17]. A common approach to implicit feedback is the use of analytics tools such as Google Analytics or Mixpanel where usage trackers are directly embedded into the code [13]. Usages are many-fold, including:

- product design. Software developers can use implicit feedback to identify user needs and preferences and design features accordingly. Specifically, implicit feedback is used for: *Requirement Prioritization* [11, 15, 17, 17, 32], *Requirement Refinement* (i.e., improving the accuracy, completeness, and consistency of existing requirements) [3, 17], and *Requirement Elicitation* (e.g., discovering of new requirements by analyzing the different usage paths of the users) [15, 27],
- development. Here, developers can monitor implicit feedback to measure user engagement, improvement of different quality indicators [18, 23], such as the optimization of usability issues [9, 11, 32], or the early identification and fix of bugs by capturing them through crash reports and raised errors in the logs [18, 25, 32],
- post-release monitoring. Here, developers can monitor implicit feedback to measure user engagement, identify areas for improvement, and make data-driven decisions for future software development [18, 26, 32].

These benefits should not be limited to single-product development but expanded to SPLs. Indeed, products derived from an SPL platform are likely to benefit even more, as insights from the usage of one product can inform other products. Currently, SPL practitioners replicate single-product development practices by injecting usage trackers directly into the code once the variant is generated. The question arises as to why practitioners keep injecting usage trackers into the variants’ code rather than embedding them directly into the platform’s code. There are two rationales for this. First, feedback analysis evolves at a different pace (and often

involves distinct decision-makers) than feature analysis. Embedding usage trackers into the features’ code would couple feedback analysis with feature realization, hindering the separate evolution of each concern. Second, implicit feedback is a crosscutting concern related to usage. It often involves how smoothly users progress through a given workflow where distinct features are involved. Thus, implicit feedback might cross-cut distinct features. As a result, decoupling is achieved by moving implicit feedback down to the variant code. Unfortunately, this contravenes SPL principles, as variant derivation should be limited to configuration without further modifying the code of the derived variant [20].

This calls for implicit-feedback to be moved to the SPL-platform level. In previous work, we addressed the feasibility of specifying feedback requirements at the platform level by using features as the unit of tracking: the Feedback Model [10]. At the time the variant is generated from the Configuration Model, the Feedback Model is checked out. If the configuration of the variant meets the *Context* (i.e., a feature-based boolean expression) of the feedback model, then the corresponding usage tracker is introduced. Figure 1 depicts this vision. Feedback is collected from a set of running variants and stored in the feedback log, and subsequently consulted along the distinct SPL cycles: domain engineering, application engineering, and management [7].

3 Implicit Feedback & Domain Engineering

Domain Engineering involves defining and creating a set of reusable assets that can be used to build multiple products within a particular domain. These assets may include the feature model, the architecture, the design patterns, and the SPL-platform codebase.

3.1 Automated Analysis of Feature Models

Automated Analysis of Feature Models (AAFM) refers to the computer-aided extraction of information from feature models to assist in various tasks [12]. For example, dead-feature detection involves identifying features that cannot be selected in any valid product line configuration. This situation may arise due to incorrect constraint definitions (such as exclude or requires). Using implicit feedback, the potential of analysis increases because new relevant information shows up. For instance, we could complement *dead features* with *drag features*, i.e., selected but seldom used features. A drag feature is a feature that is selected in different variants (i.e., it appears in different configurations), yet its usage is scarce or practically non-existent. This information could help decision-makers determine whether to deprecate the drag features to save maintenance effort or rather upgrade them to promote usage. It is easy to imagine the definition of *usage constraints*: “if the usage of a feature *X* falls below a certain threshold within a given time-frame, then the priority for testing feature *X* is decreased”.

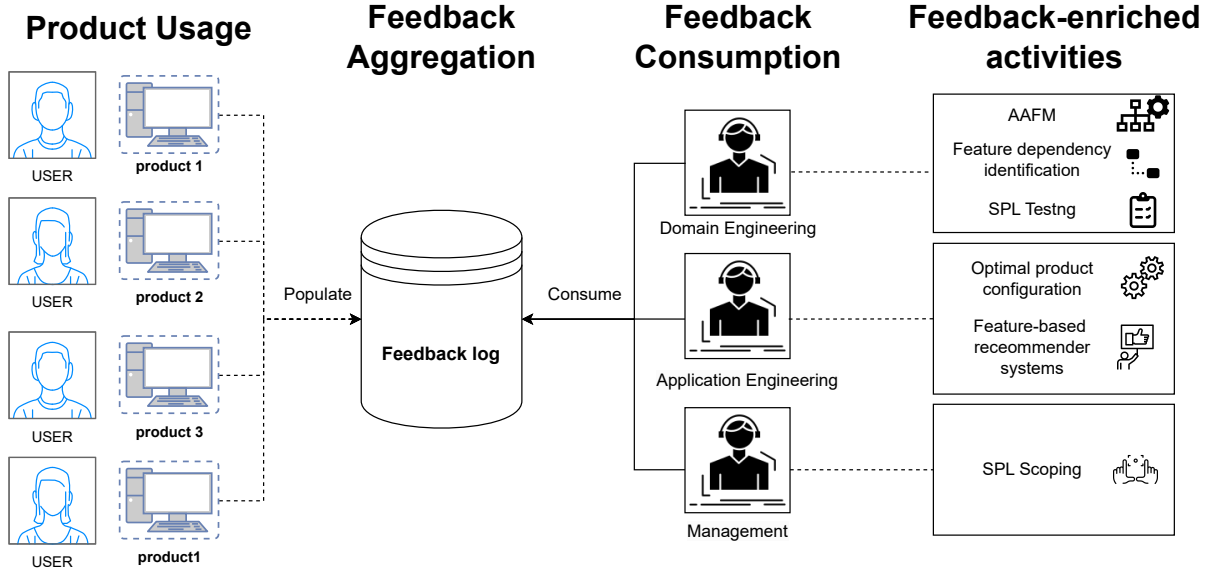


Figure 1. The vision of implicit feedback in SPLs. The Feedback Log collects usage hits from the different installations of variants from a (sub)set of the SPL portfolio. Subsequently, engineers tap into this Feedback Log through dedicated dashboards.

Challenges ahead. Traditionally, AAFM resorts to static models, namely the feature model and the configuration models available at a given point in time. In contrast, implicit feedback provides dynamic data, which is constantly changing as the products are in production use. If feature models are to be supplemented with implicit feedback, more flexible representations are required, moving away from traditional feature diagrams to more dynamic ones that will keep implicit feedback data up-to-date.

3.2 Feature Dependency Identification

Feature dependencies (aka feature constraints) document relationships between two or more features. Failure to identify such dependencies can result in invalid products being derived from the SPL [30]. Traditionally, feature-dependency identification rests on static sources: sample configurations, implementation code, or natural-language assets [24]. Implicit feedback opens the door to introducing dynamic data in the decision process. Specifically, usage data could help validate or infer dependencies based on how users engage with the features. On the validation front, if a *requires* dependency has been modeled between two features, but they are seldom used together, it may indicate that the relationship might need to be looked into more carefully. On the inference front, feature models can have layered dependencies based on actual usage. If two features are frequently used together, an “escort” relationship can be defined, even if the features are not *requires*-dependent. These “escort dependencies” can later be utilized for recommender systems based on past experiences. This new type of relationship can also

provide a fresh visual perspective of the feature model by displaying only implicit feedback-based relationships, which can be analyzed automatically.

Challenges ahead. The number of new relationships based on implicit feedback can be substantial, and it is challenging to determine which ones are more useful for analysis and decision-making. Empirical studies and validation need to be reinforced.

3.3 SPL Testing

To scale up, SPL testing resort to product sampling techniques [33]. Al-Hajjaji et al. propose a similarity-based prioritization approach for sample-based product line testing [1]. Their approach consists of three steps: (1) test the product with the maximum number of features; (2) test the product that is most dissimilar to the previously tested product; and (3) test the product that is most dissimilar to the previously tested products until the testing budget is consumed. SPL testing also resorts to uniform random sampling techniques. However, there is a well-known bottleneck in terms of scalability and uniformity of the sampler used [14]. The main objective of this approach is to increase the fault detection rate, regardless of the impact of the faults. In situations where detecting faults that have a significant impact on users is more critical than identifying multiple faults, prioritization based on implicit feedback may be more appropriate. Furthermore, using implicit feedback information can help in selecting sample configurations by imposing constraints to reduce the configuration space and improve scalability while

still maintaining uniformity in a uniform random sampling approach.

Challenges ahead. While feedback-based approaches have advantages over similarity-based methods, they should not rely solely on feedback for prioritizing products for testing. If prioritization is based solely on feature usage rate, it can lead to redundant testing. This is because products with the same features are tested repeatedly due to their high usage rate. This approach becomes inefficient if the goal is to uncover unique flaws that significantly impact the user experience. Therefore, it is important to balance the redundancy of feedback-based prioritization by incorporating other methods, such as similarity-based approaches. We propose further research that integrates techniques and evaluates outcomes using empirical data to address these challenges.

4 Implicit Feedback & Application Engineering

SPL engineers develop products by selecting from a range of available features. These features might already be in use in other products, so implicit feedback can become an informant during product configuration.

4.1 Optimal Product Configuration

Optimal product configuration involves configuring a product to meet an optimization function. An optimization function determines the best product configuration that satisfies specific requirements [16, 22]. The function may include one or more optimization parameters [28]. For instance, a function with two parameters could aim to maximize *performance* while minimizing *energy and resource consumption*. Typically, this type of solution takes a feature model and one or more optimization parameters as input, and generates an optimal set of product configurations represented on a Pareto front as output.

In this context, implicit feedback can serve as an additional optimization parameter, contributing to two primary optimization objectives: promoting usage (i.e., generating a product with frequently utilized features) and promoting satisfaction (i.e., generating a product with features that align better with the preferences of targeted users). To illustrate this concept, let us consider an e-commerce platform that offers a range of features, including order management, payment gateway integration, product reviews, and a loyalty program. These features can be evaluated based on implicit feedback gathered from previously deployed products. In the initial stages of a business, an e-commerce owner may prioritize a product configuration that minimizes costs while maximizing value for customers by focusing on highly used features. In this scenario, the optimal product configuration could include features such as order management and payment gateway, while excluding less frequently used features

like loyalty programs and product reviews due to their associated costs. However, when the objective shifts towards enhancing user satisfaction, particularly when the target user highly values the opinions of other customers, incorporating features such as product reviews becomes imperative in the optimal product configuration.

Challenges ahead. Usage is highly contextualized. Most non-functional attributes, such as power consumption and price, tend to remain stable regardless of the product or context in which the feature is deployed. However, feature usage is highly influenced by context. Incorporation of other features in the product, along with user demographics or cyber-physical systems where the hosting product is deployed, can significantly impact feature usage. A highly-used feature in one context may be poorly utilized in another due to differences in user demographics or other factors. Therefore, traditional optimal product configuration needs to consider the notion of context so that the results can be qualified by comparing the historical context (i.e., where implicit feedback was obtained) and current context (i.e., where the new product will be deployed). While single-product development is typically done on a project-by-project basis without expectation for reuse or sale to other customers, SPL development aims to create a set of related products that can be sold to multiple customers or markets. In this case, “the market” from which implicit feedback is collected may differ from “the market” for the envisioned product. This highlights the need for careful comparison between the actual and envisioned markets to assess how closely the implicit-feedback collection mimics the market for the new product.

4.2 Feature-based Recommender Systems

A second intervention in product configuration is recommender systems. Here, the input is a valid partial configuration, which is a subset of the possible features selected for a particular product that complies with the constraints defined in the feature model. Using this input, the recommender system utilizes previous configurations to estimate and predict the relevance of potentially selectable features [6, 29]. However, the information derived from the configuration models is static and primarily derived from the configurations of previous products. To ensure that the recommendations align with the users’ present needs, it becomes more reasonable to enrich these recommendations by incorporating data on how users interact with features of products that are similar to the one being configured. Moreover, this approach is in line with how recommender systems are employed outside of SPLs, where users’ implicit feedback is used as a main recommendation factor [34]. For example, Zhou et al. [35] recommend appropriate APIs to developers based on natural language queries and previous user interactions with the recommendation engine.

Challenges ahead. Recommender systems that rely on implicit feedback may become unable to recommend less popular or niche items, also known as long-tail items. This is because implicit feedback data tends to be biased towards popular items, leaving less room for recommendations for items with which only a few users have interacted. This issue can be challenging for recommender systems that aim to accurately suggest long-tail items to users who might be interested in them. One potential solution to this problem is to develop customer profiles that capture feature preferences for different customer segments. By identifying unique preferences among various sectors of customers, we can promote the recommendation of long-tail items that are more likely to benefit specific customer groups.

5 Implicit Feedback & Management

The Management Cycle involves managing the life-cycle of the reusable assets, monitoring and evaluating the effectiveness of the assets, including tracking their usage, performance, and impact on development time, cost, and quality. This may involve reconsidering the scope of the SPL [21].

5.1 SPL Scoping

Clements’ popular definition of an SPL as addressing ‘a particular market segment’ entails that the evolution of ‘this market segment’ should, naturally, go hand in hand with the evolution of the SPL [7]. Hence, SPL scoping (i.e., deciding on the features to be covered by the SPL) is necessarily ‘continuous’ to keep pace with this market segment. In this setting, an essential aspect of prioritization, often done regarding features, is deciding what features should be evolved or included in the next SPL release [2]. Botterweck et al. identify three feature-scoping states: mandatory, optional, and product-specific (see Fig. 2) [4]. Features might transit from one state to another in each scoping release. SPL engineers resort to deciding what features should be made mandatory and which should be optional or product-specific in the next release.

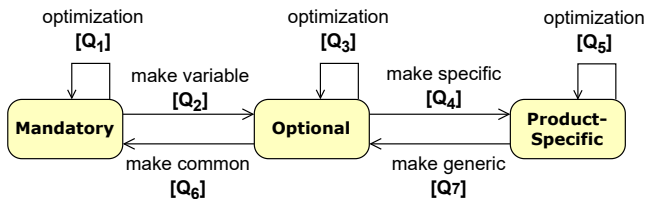


Figure 2. Scope of feature states along three levels: mandatory, optional, and product-specific

So far, scoping decisions mainly rest on explicit feedback, i.e., gathered in planning meetings [19] or inferred from business goals [31]. Implicit feedback can be used as an additional source of information to track changes in the state of

a feature. If the implicit feedback is stored in a feedback log, it can be used to create a query (Q) that captures the state transitions based on this log. These changes are not meant to happen automatically; instead, dashboards can be employed to visualize these scenarios and set up alerts when certain thresholds in Q are exceeded. By using different colors to represent distinct thresholds, these dashboards can help monitor the overall usage of the SPL, taking into account the usage of its entire portfolio. This dashboard could be helpful in identifying the most urgent features and anticipating the need for a new SPL scoping effort.

Challenges ahead. In SPLs, implicit feedback is derived from the installation of different variants. However, not all variants may carry the same weight. The importance of the customer (and their installations) or the relevance of a feature in terms of competitive advantage may hold more significance in the decision-making process. Therefore, the feedback analysis model should be able to accommodate these varying weights. Another challenge lies in identifying temporal usage patterns. Implicit feedback can unveil usage patterns that suggest a need to modify the scoping state of a feature. For instance, if an optional feature is selected in numerous products and heavily used across all of them, it may indicate that the feature’s scoping state should be changed from “optional” to “mandatory”. Nevertheless, this usage pattern should be consistently observed over time rather than being a momentary fluctuation. Thus, including time intervals for observing usage can be a crucial factor in feedback analysis models.

So far, we have provided argumentative narratives for different scenarios where platform-centric feedback can be beneficial. The next section collects insights from practitioners.

6 Platform-centric Feedback: the practitioners’ perspective

This section collects feedback from practitioners about the envisaged SPL activities that can benefit from implicit feedback. The evaluation follows the Goal-Question-Metric approach [5].

Goal. Analyze implicit feedback with the purpose of *characterizing its usefulness for SPL activities* from the point of view of the *SPL practitioners/consultants* in the context of *pure-systems’ employees*.

Design. The evaluation took place during a monthly online seminar hosted by *pure-systems*, the company behind *pure::variants*, a leading variant manager tool. They also provide consulting services to various companies. This context allows *pure-systems* employees to assess the usefulness and integration of implicit feedback in SPL activities. The seminar lasted 90 minutes and covered the significance of implicit feedback in single-off development. It also discussed the

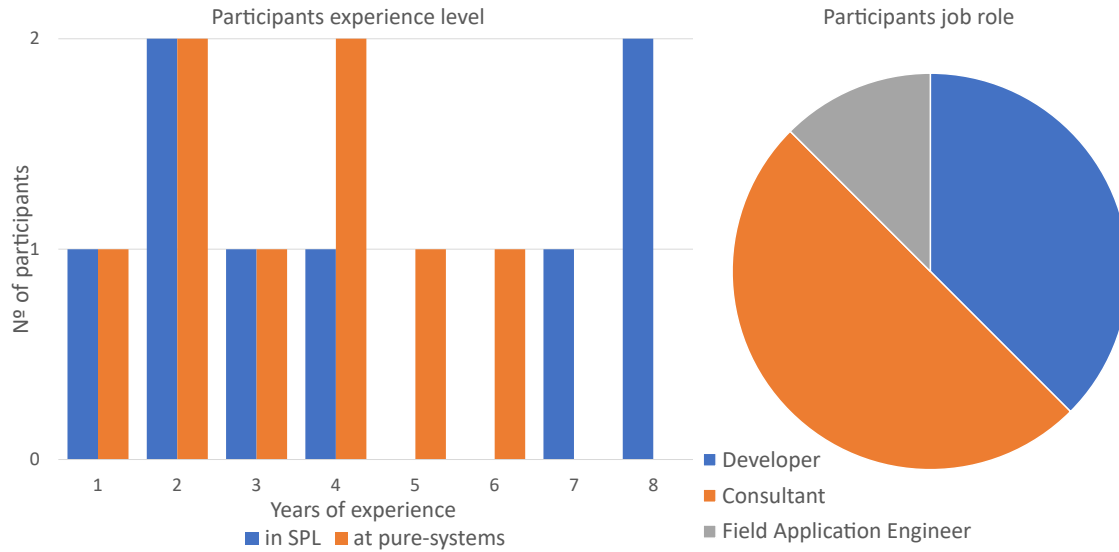


Figure 3. Participants demographics: SPL experience & role played.

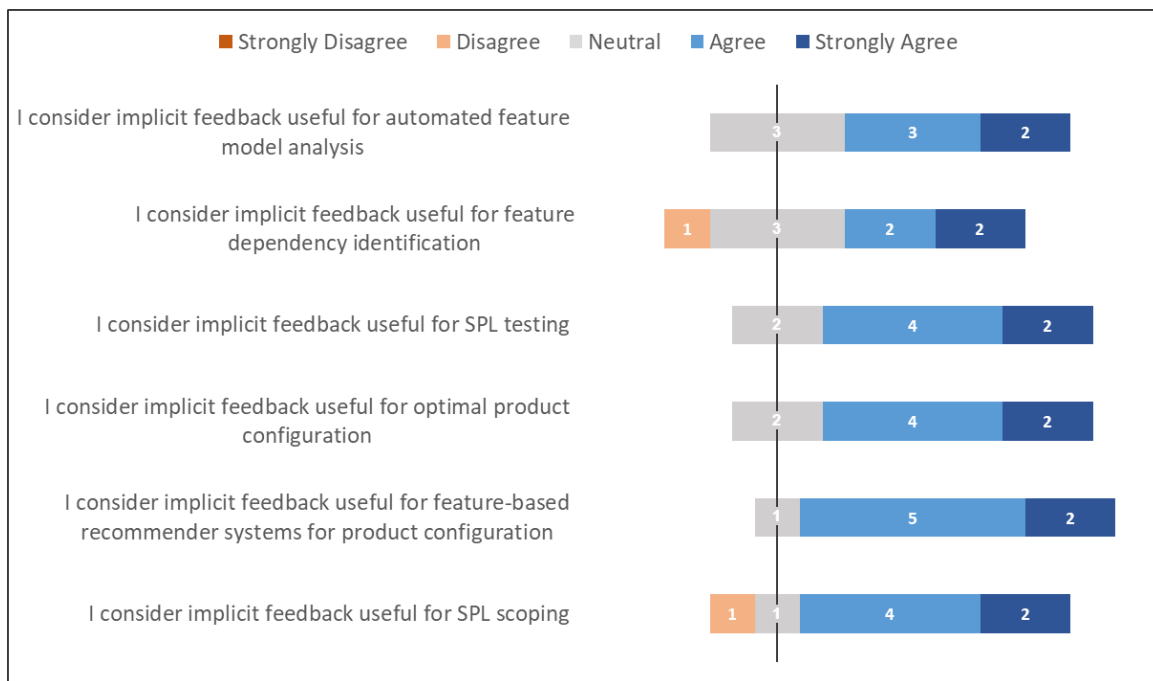


Figure 4. Perceived usefulness of implicit feedback for SPL practices

benefits and challenges of implementing implicit feedback in SPLs. Participants had the opportunity to ask questions throughout the seminar. At the end, they were asked to complete a questionnaire using a Likert scale, indicating their agreement or disagreement with statements about the usefulness of implicit feedback. To ensure clarity, the survey was pilot tested with three developers. The questionnaire also included an open-ended question. Eight out of the fifteen seminar attendees completed the survey. These eight

employees' demographic data is presented in Fig. 3. All participants, except one, had at least two years' experience in SPLs at *pure-systems*.

Results. Fig. 4 shows the evaluation results in a stacked bar chart with the respective frequencies of each Likert option. Specifically:

- **Automated Feature Model Analysis:** The majority agreement (5 out of 8 participants) reflects the positive perception of using implicit feedback for automatic feature model analysis.
- **Feature dependency identification:** The responses indicate a somewhat mixed perception regarding the usefulness of implicit feedback for unveiling dependencies. While there is a slight inclination towards agreement (4 out of 8 participants), the varying responses highlight the need for further exploration and clarification. Participants' differing perspectives may stem from factors such as their specific roles or experiences (consulting vs development), indicating potential challenges in effectively utilizing implicit feedback to uncover dependencies.
- **SPL testing:** The agreement of the majority (5 out of 8 participants) on the usefulness of implicit feedback as an additional parameter for SPL testing.
- **Optimal Product Configuration.** The high agreement (6 out of 8 participants) suggests that incorporating implicit feedback as an additional parameter might lead to more tailored and satisfactory products.
- **Feature-based Recommender Systems.** The majority agreement (6 out of 8 participants) suggests that participants consider implicit feedback as useful content for feature-based recommender systems.
- **SPL scoping:** The mixed responses regarding the usefulness of implicit feedback in SPL scoping (4 out of 8 participants leaning towards agreement) indicate that while participants recognize the potential benefits, there may be some reservations or challenges associated with incorporating implicit feedback into the scoping process.

The open-ended question revealed two major concerns. First, privacy issues have been raised associated with tracking software, particularly when it comes to customers who provide services to other organizations. Since participants believe that their customers are unlikely to be granted permission to collect data from systems they do not own, it is crucial to consider privacy concerns as a variable in research on implicit feedback for SPLs. Second, platform-centric feedback may hinder the agility of product-based feedback. When it comes to single-product development, engineers have the flexibility to add tracking hits as needed. However, with platform-centric feedback, there is an additional step where application engineers must communicate their feedback requirements to domain engineers. This information is then gathered for the different variants and integrated into a new feedback model. This process might delay decisions, which might lead application engineers to inject their own tracking hits in the variant code once it is generated without waiting for domain engineers to update the feedback model.

Threats to validity. Two main threats are anticipated. Firstly, the validity of the conclusions may be compromised due to the small number of respondents. At the outset, we gave priority to expertise rather than quantity, as all participants were experienced in SPLs and belonged to a prominent SPLs consultancy company. Secondly, the external validity is at risk as all participants are from a single company. However, since our focus was on a highly specialized group (i.e., SPL engineers), valid conclusions can be supported by a smaller sample size than would be required for a larger and more diverse population.

7 Conclusions

This research aims to establish the value proposition of incorporating implicit feedback into SPLs by advocating for a platform-centric approach to feedback in SPLs. This approach addresses the variants' needs for usage trackers through generative programming and configuration-like mechanisms. The potential benefits of incorporating implicit feedback into decision-making in various SPL activities were presented in an argumentative manner and supported by evidence from a questionnaire given to eight practitioners. All the activities were deemed as potential candidates for improvement through implicit feedback. However, when it came to Feature dependency identification and SPL scoping, practitioners had mixed perceptions that necessitate further clarification through dedicated studies. Furthermore, practitioners expressed concerns about potential privacy issues that could arise in their customers' organizations if implicit feedback practices were implemented.

While the study suggests that platform-centric feedback may have advantages, it also highlights some envisaged challenges. Decision-making shifts from application engineers to domain engineers, which could potentially cause delays that jeopardize time-to-market goals. This calls for additional research into a hybrid approach where a "continuous feedback model" could promptly incorporate the needs of the application engineers. Our ultimate goal is not only to encourage further investigation into this topic but also to inspire variability management vendors to incorporate this feature into their products.

Acknowledgement

We would like to extend our gratitude to *pure-systems'* employees for their invaluable contributions to this research. This work is co-supported by the "European Union NextGeneration EU/PRTR" under contract PID2021-125438OB-I00, FEDER/Ministry of Science and Innovation/Junta de Andalucía/State Research Agency with the following grants: *Data-pl*(PID2022-138486OB-I00), TASOVA PLUS research network (RED2022-134337-T) and *METAMORFOSIS* (FEDER_US-1381375). Raul Medeiros enjoys a doctoral grant from the Ministry of Science and Innovation - PRE2019-087324.

References

- [1] Mustafa Al-Hajjaji, Thomas Thüm, Malte Lochau, Jens Meinicke, and Gunter Saake. 2019. Effective product-line testing using similarity-based product prioritization. *Softw. Syst. Model.* 18, 1 (2019), 499–521. <https://doi.org/10.1007/s10270-016-0569-2>
- [2] Hamad I. Alsawalqah, Sungwon Kang, and Jihyun Lee. 2014. A method to optimize the scope of a software product platform based on end-user features. *J. Syst. Softw.* 98 (2014), 79–106. <https://doi.org/10.1016/j.jss.2014.08.034>
- [3] Maurizio Astegher, Paolo Busetta, Anna Perini, and Angelo Susi. 2021. Specifying Requirements for Data Collection and Analysis in Data-Driven RE. A Research Preview. In *Requirements Engineering: Foundation for Software Quality - 27th International Working Conference, REFSQ 2021, Essen, Germany, April 12-15, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 12685)*, Fabiano Dalpiaz and Paola Spoletini (Eds.). Springer, 182–188. https://doi.org/10.1007/978-3-030-73128-1_13
- [4] Goetz Botterweck and Andreas Pleuss. 2014. Evolution of Software Product Lines. In *Evolving Software Systems*, Tom Mens, Alexander Serebrenik, and Anthony Cleve (Eds.). Springer, 265–295. https://doi.org/10.1007/978-3-642-45398-4_9
- [5] Victor R Basili, Gianluigi Caldiera, and H Dieter Rombach. 1994. The goal question metric approach. *Encyclopedia of software engineering* (1994), 528–532.
- [6] Housseem Chemingui, Camille Salinesi, Inès Gam, Raul Mazo, and Henda Ghezala. 2021. Devising Configuration Guidance with Process Mining Support. (2021).
- [7] Paul Clements and Linda M. Northrop. 2002. *Software product lines - practices and patterns*. Addison-Wesley.
- [8] Anas Dakkak, David Issa Mattos, and Jan Bosch. 2021. Perceived benefits of Continuous Deployment in Software-Intensive Embedded Systems. In *IEEE 45th Annual Computers, Software, and Applications Conference, COMPSAC 2021, Madrid, Spain, July 12-16, 2021*. IEEE, 934–941. <https://doi.org/10.1109/COMPSAC51774.2021.00126>
- [9] Anas Dakkak, Hongyi Zhang, David Issa Mattos, Jan Bosch, and Helena Holmström Olsson. 2021. Towards Continuous Data Collection from In-service Products: Exploring the Relation Between Data Dimensions and Collection Challenges. In *28th Asia-Pacific Software Engineering Conference, APSEC 2021, Taipei, Taiwan, December 6-9, 2021*. IEEE, 243–252. <https://doi.org/10.1109/APSEC53868.2021.00032>
- [10] Oscar Díaz, Raul Medeiros, and Mustafa Al-Hajjaji. 2023. How can feature usage be tracked across product variants? Implicit Feedback in Software Product Lines. (2023), 43. <https://arxiv.org/abs/2309.04278> Manuscript submitted for publication.
- [11] Sebastian Eder, Henning Femmer, Benedikt Hauptmann, and Maximilian Junker. 2014. Which Features Do My Users (Not) Use?. In *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*. IEEE Computer Society, 446–450. <https://doi.org/10.1109/ICSME.2014.71>
- [12] José Angel Galindo, David Benavides, Pablo Trinidad, Antonio Manuel Gutiérrez-Fernández, and Antonio Ruiz-Cortés. 2019. Automated analysis of feature models: Quo vadis? *Computing* 101, 5 (2019), 387–433. <https://doi.org/10.1007/s00607-018-0646-1>
- [13] Loveleen Gaur, Gurinder Singh, Jeyta, and Shubhankar Kumar. 2016. Google Analytics: A Tool to make websites more Robust. In *ICTCS*. 45:1–45:7. <https://doi.org/10.1145/2905055.2905251>
- [14] Ruben Heradio, David Fernandez-Amoros, José A Galindo, David Benavides, and Don Batory. 2022. Uniform and scalable sampling of highly configurable systems. *Empirical Software Engineering* 27, 2 (2022), 44.
- [15] Philipp Hoffmann, Kai Spohrer, and Armin Heinzl. 2020. *Analyzing Usage Data in Enterprise Cloud Software: An Action Design Research Approach*. Springer International Publishing, Cham, 257–274. https://doi.org/10.1007/978-3-030-45819-5_11
- [16] José Miguel Horcas, Mónica Pinto, and Lidia Fuentes. 2023. Empirical analysis of the tool support for software product lines. *Softw. Syst. Model.* 22, 1 (2023), 377–414. <https://doi.org/10.1007/s10270-022-01011-2>
- [17] Jan Ole Johanssen, Anja Kleebaum, Bernd Bruegge, and Barbara Paech. 2018. Feature Crumbs: Adapting Usage Monitoring to Continuous Software Engineering. In *Product-Focused Software Process Improvement - 19th International Conference, PROFES 2018, Wolfsburg, Germany, November 28-30, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 11271)*, Marco Kuhrmann, Kurt Schneider, Dietmar Pfahl, Sousuke Amasaki, Marcus Ciolkowski, Regina Hebig, Paolo Tell, Jil Klünder, and Steffen Küpper (Eds.). Springer, 263–271. https://doi.org/10.1007/978-3-030-03673-7_19
- [18] Jan Ole Johanssen, Anja Kleebaum, Bernd Bruegge, and Barbara Paech. 2019. How do Practitioners Capture and Utilize User Feedback During Continuous Software Engineering?. In *27th IEEE International Requirements Engineering Conference, RE 2019, Jeju Island, Korea (South), September 23-27, 2019*, Daniela E. Damian, Anna Perini, and Seok-Won Lee (Eds.). IEEE, 153–164. <https://doi.org/10.1109/RE.2019.00026>
- [19] Azaz Ahmed Kiani, Yaser Hafeez, Muhammad Imran, and Sadia Ali. 2021. A dynamic variability management approach working with agile product line engineering practices for reusing features. *J. Supercomput.* 77, 8 (2021), 8391–8432. <https://doi.org/10.1007/s11227-021-03627-5>
- [20] Charles W. Krueger. 2006. New methods in software product line practice. *Commun. ACM* 49, 12 (2006), 37–40. <https://doi.org/10.1145/1183236.1183262>
- [21] Luciano Marchezan, Elder Rodrigues, Wesley Klewerton Guez Assunção, Maicon Bernardino, Fábio Paulo Basso, and João Carbonell. 2022. Software product line scoping: A systematic literature review. *J. Syst. Softw.* 186 (2022), 111189. <https://doi.org/10.1016/j.jss.2021.111189>
- [22] Jabier Martinez, Daniel Strüber, José Miguel Horcas, Alexandru Burdusel, and Steffen Zschaler. 2022. Acapulco: an extensible tool for identifying optimal and consistent feature model configurations. In *SPLC '22: 26th ACM International Systems and Software Product Line Conference, Graz, Austria, September 12 - 16, 2022, Volume B*, Alexander Felfernig, Lidia Fuentes, Jane Cleland-Huang, Wesley K. G. Assunção, Clément Quinton, Jianmei Guo, Klaus Schmid, Marianne Huchard, Inmaculada Ayala, José Miguel Rojas, Viet-Man Le, and José Miguel Horcas (Eds.). ACM, 50–53. <https://doi.org/10.1145/3503229.3547067>
- [23] Silverio Martínez-Fernández, Anna Maria Vollmer, Andreas Jedlitschka, Xavier Franch, Lidia López, Prabhat Ram, Pilar Rodríguez, Sanja Aarasmaa, Alessandra Bagnato, Michal Choras, and Jari Partanen. 2019. Continuously Assessing and Improving Software Quality With Software Analytics Tools: A Case Study. *IEEE Access* 7 (2019), 68219–68239. <https://doi.org/10.1109/ACCESS.2019.2917403>
- [24] Sarah Nadi, Thorsten Berger, Christian Kästner, and Krzysztof Czarnecki. 2015. Where Do Configuration Constraints Stem From? An Extraction Approach and an Empirical Study. *IEEE Trans. Software Eng.* 41, 8 (2015), 820–841. <https://doi.org/10.1109/TSE.2015.2415793>
- [25] Helena Holmström Olsson and Jan Bosch. 2013. Towards Data-Driven Product Development: A Multiple Case Study on Post-deployment Data Usage in Software-Intensive Embedded Systems. In *Lean Enterprise Software and Systems - 4th International Conference, LESS 2013, Galway, Ireland, December 1-4, 2013, Proceedings (Lecture Notes in Business Information Processing, Vol. 167)*, Brian Fitzgerald, Kieran Conboy, Ken Power, Ricardo Valerdi, Lorraine Morgan, and Klaas-Jan Stol (Eds.). Springer, 152–164. https://doi.org/10.1007/978-3-642-44930-7_10
- [26] Helena Holmström Olsson and Jan Bosch. 2015. Towards Continuous Customer Validation: A Conceptual Model for Combining Qualitative Customer Feedback with Quantitative Customer Observation. In *Software Business - 6th International Conference, ICSOB 2015, Braga, Portugal, June 10-12, 2015, Proceedings (Lecture Notes in Business Information Processing, Vol. 210)*, João M. Fernandes, Ricardo J. Machado, and Krzysztof Wnuk (Eds.). Springer, 154–166. <https://doi.org/10.1007/978->

- 3-319-19593-3_13
- [27] Marc Oriol, Melanie J. C. Stade, Farnaz Fotrousi, Sergi Nadal, Jovan Varga, Norbert Seyff, Alberto Abelló, Xavier Franch, Jordi Marco, and Oleg Schmidt. 2018. FAME: Supporting Continuous Requirements Elicitation by Combining User Feedback and Monitoring. In *26th IEEE International Requirements Engineering Conference, RE 2018, Banff, AB, Canada, August 20-24, 2018*, Guenther Ruhe, Walid Maalej, and Daniel Amyot (Eds.). IEEE Computer Society, 217–227. <https://doi.org/10.1109/RE.2018.00030>
- [28] Juliana Alves Pereira, Mathieu Acher, Hugo Martin, Jean-Marc Jézéquel, Goetz Botterweck, and Anthony Ventresque. 2021. Learning software configuration spaces: A systematic literature review. *J. Syst. Softw.* 182 (2021), 111044. <https://doi.org/10.1016/j.jss.2021.111044>
- [29] Juliana Alves Pereira, Pawel Matuszyk, Sebastian Krieter, Myra Spiliopoulou, and Gunter Saake. 2018. Personalized recommender systems for product-line configuration processes. *Comput. Lang. Syst. Struct.* 54 (2018), 451–471. <https://doi.org/10.1016/j.cl.2018.01.003>
- [30] Daniel Strüber, Mukelabai Mukelabai, Jacob Krüger, Stefan Fischer, Lukas Linsbauer, Jabier Martinez, and Thorsten Berger. 2019. Facing the truth: benchmarking the techniques for the evolution of variant-rich systems. In *Proceedings of the 23rd International Systems and Software Product Line Conference, SPLC 2019, Volume A, Paris, France, September 9-13, 2019*, Thorsten Berger, Philippe Collet, Laurence Duchien, Thomas Fogdal, Patrick Heymans, Timo Kehrer, Jabier Martinez, Raúl Mazo, Leticia Montalvillo, Camille Salinesi, Xhevahire Tërnavá, Thomas Thüm, and Tewfik Ziadi (Eds.). ACM, 26:1–26:12. <https://doi.org/10.1145/3336294.3336302>
- [31] Tassio Vale, Bruno Cabral, Loreno Freitas Matos Alvim, Larissa Rocha Soares, Alcemir Rodrigues Santos, Ivan do Carmo Machado, Iuri Santos Souza, Ivonei Freitas da Silva, and Eduardo Santana de Almeida. 2014. SPLICE: A Lightweight Software Product Line Development Process for Small and Medium Size Projects. In *Eighth Brazilian Symposium on Software Components, Architectures and Reuse, SBCARS 2014, Maceió, Alagoas, Brazil, September 29-30, 2014*. IEEE Computer Society, 42–52. <https://doi.org/10.1109/SBCARS.2014.11>
- [32] Simon van Oordt and Emitza Guzman. 2021. On the Role of User Feedback in Software Evolution: a Practitioners' Perspective. In *29th IEEE International Requirements Engineering Conference, RE 2021, Notre Dame, IN, USA, September 20-24, 2021*. IEEE, 221–232. <https://doi.org/10.1109/RE51729.2021.00027>
- [33] Mahsa Varshosaz, Mustafa Al-Hajjaji, Thomas Thüm, Tobias Runge, Mohammad Reza Mousavi, and Ina Schaefer. 2018. A classification of product sampling for software product lines. In *Proceedings of the 22nd International Systems and Software Product Line Conference - Volume 1, SPLC 2018, Gothenburg, Sweden, September 10-14, 2018*, Thorsten Berger, Paulo Borba, Goetz Botterweck, Tomi Männistö, David Benavides, Sarah Nadi, Timo Kehrer, Rick Rabiser, Christoph Elsner, and Mukelabai Mukelabai (Eds.). ACM, 1–13. <https://doi.org/10.1145/3233027.3233035>
- [34] Norha M. Villegas, Cristian Sánchez, Javier Díaz-Cely, and Gabriel Tamura. 2018. Characterizing context-aware recommender systems: A systematic literature review. *Knowl. Based Syst.* 140 (2018), 173–200. <https://doi.org/10.1016/j.knosys.2017.11.003>
- [35] Yu Zhou, Xinying Yang, Taolue Chen, Zhiqiu Huang, Xiaoxing Ma, and Harald C. Gall. 2022. Boosting API Recommendation With Implicit Feedback. *IEEE Trans. Software Eng.* 48, 6 (2022), 2157–2172. <https://doi.org/10.1109/TSE.2021.3053111>

Received 2023-07-14; accepted 2023-09-03

Virtual Domain Specific Languages via Embedded Projectional Editing

Niklas Korz
acm@korz.dev
Alugha
Germany

Artur Andrzejak
artur.andrzejak@uni-heidelberg.de
Heidelberg University
Germany

Abstract

Domain Specific Languages (DSLs) can be implemented as either *internal* DSL, i.e. essentially a library in a host general-purpose programming language (GPL), or as *external* DSL which is a stand-alone language unconstrained in its syntax. This choice implies an inherent trade-off between a limited syntactic and representational flexibility (internal DSLs), or an involved integration with GPLs and the need for a full stack of tools from a parser to a code generator (external DSLs).

We propose a solution which addresses this problem by representing a subset of a GPL - from simple code patterns to complex API calls - as GUI widgets in a hybrid editor. Our approach relies on matching parametrized patterns against the GPL program, and displaying the matched parts as dynamically rendered widgets. Such widgets can be interpreted as components of an external DSL. Since the source code is serialized as GPL text without annotations, there is no DSL outside the editor - hence the term 'virtual' DSL.

This solution has several advantages. The underlying GPL and the virtual DSL can be mixed in a compositional way, with zero cost of their integration. The project infrastructure does not need to be adapted. Furthermore, our approach works with mainstream GPLs like Python or JavaScript.

To lower the development effort of such virtual DSLs, we also propose an approach to generate patterns and the corresponding text-only GUI widgets from pairs of examples.

We evaluate our approach and its implementation on use cases from several domains. A live demo of the system can be accessed at <https://puredit.korz.dev/> and the source code with examples at <https://github.com/niklaskorz/puredit/>.

CCS Concepts: • Software and its engineering → Domain specific languages; Visual languages; Integrated and visual development environments; Programming by example.



This work is licensed under a Creative Commons Attribution 4.0 International License.

GPCE '23, October 22–23, 2023, Cascais, Portugal

© 2023 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-0406-2/23/10.

<https://doi.org/10.1145/3624007.3624059>

Keywords: DSLs, Projectional editing, Programming language integration, Programming by example, Assisted editing and IntelliSense

ACM Reference Format:

Niklas Korz and Artur Andrzejak. 2023. Virtual Domain Specific Languages via Embedded Projectional Editing. In *Proceedings of the 22nd ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '23)*, October 22–23, 2023, Cascais, Portugal. ACM, New York, NY, USA, 16 pages. <https://doi.org/10.1145/3624007.3624059>

1 Introduction

Domain Specific Languages (DSLs) have proven useful in developing software systems, and are increasingly adopted by practitioners in a multitude of application domains [16, 25, 43]. They can clearly communicate the intent of a part of a system, hide irrelevant implementation details, and make it harder to say "wrong things" in code. These properties help to improve development productivity and prevent defects. Finally, DSLs can effectively facilitate communication between domain experts and developers.

DSLs can be implemented as either *internal* DSL, i.e. essentially a library in a host general-purpose programming language (GPL), or as *external* DSL which is a stand-alone language unconstrained in its syntax. Internal DSLs can be implemented more easily yet and naturally integrate in the host GPL. However, they offer only a limited syntactic and representational flexibility which is tightly coupled with the syntax of their host language. On the other hand, while not limited in their syntax, the external DSLs face an issue of complex and possibly inefficient interfacing with other languages in the project. Moreover, they feature a high cost of implementation due to a need for a full stack of tools from a parser to code generator or compiler.

Projectional editors [56, 66, 67] solve the problem of integrating different languages by being able to render each language independently yet side-by-side. They can be also enhanced with interactive, graphical elements such as tables or dynamic diagrams that suit the specific needs of a certain domain. However, projectional editors come with an increased development and maintenance effort for the DSL developers, and have usability issues [67], such as challenges of efficient entering textual code, or code modifications. In


```

1 ((table) => {
2   console.log("Replacing ...");
3   table.column("name").replace("Mister", "Mr.");
4 })
5 (db["students"]);

```

Listing 1. An example code in TypeScript.

```

1 change table students
2   console.log("Replacing ...");
3   table.column("name").replace("Mister", "Mr.");
4 end change

```

Listing 2. Hybrid representation of code from Listing 1. Projections are shown in bold.

addition, code is typically serialized in a proprietary format, making infrastructure integration difficult.

To address these issues, we propose an approach which extends a textual code editor with the ability to represent parts of code as embedded textual or graphical GUIs. Listing 1 and Listing 2 illustrate this solution. Listing 1 shows a fragment of TypeScript code as rendered in a conventional editor or IDE. Lines 1 to 5 define an anonymous function with a parameter `table` and function body in lines 2 and 3. Line 5 applies this function to an object `db["students"]` representing a database table. Listing 2 displays how the same program fragment is shown in our *hybrid editor*. Lines 1 and 4 in the latter listing are part of a *projection*, an embedded GUI which offers a developer-friendly representation of the source code lines 1, 4, and 5 from Listing 1. Projections can be interpreted as components of an external DSL. Their representation can assume any form - from text to tables to diagrams to equations.

Within the projection shown in Listing 2 users can change the argument `students` by typing it or selecting from a list of recommendations (not shown). They can also edit the textual code representation in lines 2 and 3 as usual. Users can enter new (textual) projections in their code by using the code recommendation feature of the hybrid editor. For example, after typing 'c' in an empty line in the live demo the projection 'change table' is proposed. It is also possible to edit the hybrid and the traditional form of GPL source code side by side, with updates being immediately synchronized. Grammar errors in the GPL code lead to failure of identifying the correct projection. As a consequence, the hybrid editor shows the original (erroneous) GPL code instead.

We illustrate the flexibility of representation in the hybrid editor with a more complex example - visual editing of mathematical expressions. We created a small Python library `mathdsl` (a wrapper of few large libraries) to support conversion of formulas expressed in LaTeX into executable NumPy code (see Section 5.2). Listing 3 shows code which implements a rotation of a 2-dimensional vector by angle

```

1 import mathdsl
2 rotate, args = mathdsl.compile("\\begin{pmatrix}\\cos\\theta & -\\sin\\theta\\\\\\sin\\theta & \\cos\\theta\\end{pmatrix}\\begin{pmatrix}x\\\\y\\end{pmatrix}")
3 print("rotate(x, y, theta):")
4 print(rotate(x=1, y=2, theta=0.5))

```

Listing 3. Rotation of a two-dimensional vector using library `mathdsl`.

```

1 import mathdsl
2 rotate, args =  $\begin{pmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$ 
3 print("rotate(x, y, theta):")
4 print(rotate(x=1, y=2, theta=0.5))

```

Figure 1. Code from Listing 3 as shown in the hybrid editor.

θ using `mathdsl`. While this representation relieves users familiar with LaTeX from coding in NumPy, the code in line 2 it is not easy to write and understand.

Figure 1 shows a representation of the same code in our hybrid editor. Users can enter each part of mathematical expression as a LaTeX code and/or via a visual palette/keyboard (opened by an icon at the end of line 2). Both code comprehension and the ease of editing are enhanced. A reader is encouraged to try out this example in the live demo. Our prototype also supports integration of such a solution in JupyterLab.

An essential feature of our approach is that the source code retains its traditional textual form while persisted, without any annotations or changes (internally, code is represented as an equivalent AST). In this way, the hybrid editor achieves full compatibility with existing infrastructure tools like linters, compilers, or other editors. Furthermore, developers do not need to adapt their code to use the system. Effectively, there are no traces of a DSL outside the editor - hence the term 'virtual' DSL.

Under the hood, the hybrid editor tries to find in the source code fragments which match one of the predefined *patterns*. In Listing 1, lines 1 and 5 match such a pattern. It is defined together with the corresponding *projection template*, here the one responsible for displaying lines 1 and 4 in Listing 2. Our hybrid editor renders a projection for each matching pattern (with corresponding parameters, e.g. `students`). As outlined in Section 2.3, edits of the textual or projectional representation update the internal AST model and might trigger a redraw of the view.

A pattern and a projection template constitute together a *DSL component* which is responsible for an alternative representation of a specific aspect of the GPL language or its library. Such components can be developed separately, allowing to incrementally grow a virtual DSL in a modularized

way. Compared to an external DSL, the development effort for a virtual DSL is a typically smaller.

To lower the development cost even further we propose an approach to generate patterns and the corresponding text-only projection templates from multiple examples. A DSL developer provides pairs of a GPL code to be matched and desired textual projection. Our algorithm either requests for more/other examples to resolve ambiguity or inconsistencies or generates a DSL component (i.e. pattern and corresponding projection template). These artifacts can be either used directly or can serve as a basis for further refinements.

In addition to work on projectional editing [9, 15, 37, 49, 52, 65, 67] there is already a substantial body of research on enriching *textual* editors with interactive GUI-based components for developer-friendly representation of code [1, 19, 54, 55, 59]. As discussed in Section 6, most approaches require adaptation of the source code, and only few target mainstream programming languages (and if so, predominantly Java). Our work focuses on practical usability of such a solution, in particular support for mainstream GPL languages like Python or JavaScript, compatibility with an existing infrastructure, and a moderate development effort of virtual DSLs. To further increase the practical value, our prototype uses web-based technologies and is implemented in TypeScript. It works almost out-of-the-box in JupyterLab and can be easily adapted for Visual Studio Code or other web-based editors.

The main contributions of this work are:

- Approach for specification of patterns for matching code fragments in GPL source code.
- Algorithm for matching code fragments and extraction of arguments from code.
- Approach for efficient rendering and editing of the projections associated with patterns, and bidirectional updating of textual and projectional code representation.
- Prototypical implementation of a web-based hybrid editor as a stand-alone tool and a JupyterLab extension.
- Approach and an implementation for synthesizing DSL components (with text-only projections) from samples of pairs GPL code/textual projection.
- An evaluation via a proof-of-concept for the domains spreadsheet processing and formula editing in GPL languages TypeScript and Python.

This paper has the following structure. Section 2 details the virtual DSL approach. Section 3 describes the algorithm for generating DSL components from examples. Section 4 outlines the implementation. Section 5 describes the evaluation. Section 6 discusses related work and Section 7 contains conclusions.

```

1 let db = contextVariable("db");
2 let tbl = arg("table", "string");
3 let [changePattern, changeDraft] =
4   statementPattern'
5   ((table) =>
6     ${ block({tbl:"table"}) })
7   (${ db })[${ tbl }];
8 ;

```

Listing 4. A description in the templating language of a pattern which matches parts of code in Listing 1.

2 Virtual DSL - Concepts and Algorithms

A virtual DSL is a system consisting of a hybrid editor (same for all projects), and a set of DSL components to be developed for a specific scenario. Recall from Section 1 that each DSL component is a pair including a pattern (defined rigorously in Section 2.1) and a template projection. A pattern matches in the edited program a code fragment called a *projected code (fragment)*. Such fragments can be simple expressions or statements, calls to APIs or libraries, or even statements in an internal DSLs. The projection templates are essentially GUI components (widgets), in our implementation using the Svelte web framework.

Section 2.1 describes how patterns are defined. In Section 2.2 we outline the process of compositional matching of these pattern and the extraction of arguments for projections. Section 2.3 describes the mechanisms related to rendering and editing projections.

2.1 Patterns and Templating Language

A *pattern* is a pair of an AST in a target GPL and an optional set of data structures called *active nodes*. An active node consists of a regular AST-node with a unique identifier and a separately maintained data structure referenced by this identifier. Active nodes play a special role during the matching process explained in Section 2.2 as they capture specific sets of AST nodes or provide context information.

Since it is difficult to create and edit a pattern directly, we use a subset of TypeScript called *templating language TL* for generating and representing patterns. A pattern description in TL is just a fragment of code in TypeScript which contains one *tagged template literal* [32, sections 13.2.9 and 13.3.11] and optionally declares some objects using functions `arg(argName, nodeType)`, `block(contextVars)`, or `contextVariable(varName)`. The role of these objects is to create active nodes in the described pattern.

We illustrate the above concepts on an example. Listing 4 shows a TL description of a pattern matching lines 1, 4, and 5 in Listing 1, i.e. code shown as a projection in lines 1 and 4 of Listing 2. Lines 4 to 7 in Listing 4 contain the tagged template literal which consists of a call to our custom function `statementPattern` and a template literal, i.e. essentially a string with parameters to be interpolated (parameters are

shown highlighted). In TypeScript, these parameters are any expressions enclosed in `{...}`.

The template literal represents the AST to be matched. All string parts (i.e. parts outside `{...}`) will be turned to AST nodes matched literally, and each parameter will give rise to an active node. Consequently, a pattern created by Listing 4 will have three active nodes. The latter are created by the calls: `block({tbl:"table"})` (line 6), `contextVariable("db")` (lines 1 and 7), and `arg("table", "string")` (lines 2 and 7), in this order. The details of these functions are described in Section 2.1.1. Note that the string parts of the template literal express the code of a target GPL language, and could be e.g. Python, not necessarily TypeScript as in this example.

In essence, the first active node captures an inner block of target GPL statements enclosed by a matched source code fragment. In Listing 1 this block is in lines 2-3. The second active node, `db`, is a context variable potentially initialized in a surrounding pattern match. It specifies a name of a variable giving access to a database specified in the surrounding code. This access can be used by the hybrid editor to e.g. dynamically provide a list of valid table names for parameter recommendations within the current projection. Finally, the third active node, `tbl`, has the task of capturing and updating a value of a source code AST node of type `string`. It is then used as a parameter to be shown and edited in the corresponding projection. In this case, the value is "students" (line 5 in Listing 1), and the same value is shown in the projection in Listing 2 (line 1).

A TL description like above is converted by our function `statementPattern` to a pattern. In this process, each parameter of a tagged literal template generates an active node. The corresponding internal data structure is kept for the matching process (Section 2.2), and the parameter location (substring `{...}`) is replaced by a unique identifier referencing this data structure. The literal template is then turned into a normal string containing a fragment of the target GPL code. We parse it subsequently into an AST using the parser generator *tree-sitter* [62] into TypeScript, Python, or (in future) other target GPL.

2.1.1 Details of Active Nodes. We describe in the following the three kinds of active nodes of a pattern and the corresponding functions used in the templating language TL to generate them.

Argument active node. This kind of node is specified in TL via the helper function `arg(argumentName, nodeType)`. Its purpose is capturing the text content of an identifier or literal nodes in the GPL source code as an argument for the corresponding projection. Moreover, depending on the value of `nodeType`, we can also capture arbitrary subtrees by specifying a filter function. The type specification `nodeType` allows to match AST-nodes by type and not their text content. Possible node types depend on the target GPL and are specified by

tree-sitter. Furthermore, the captured value is accessible to context variables under the `argumentName`. The parameter `argumentName` sets the name under which a projection can access the value captured in the source code.

Block active node. This kind of node is created in TL by the helper function `block(contextVariables)`. It captures nested blocks of code, such as the body of a function definition or the branch of an if-clause. Additionally, blocks themselves are recursively searched for pattern matches, in order to support compositionality. To this end optional context variables (explained below) can be passed down to patterns matching inner code.

Context variable active node. In TL, such nodes are expressed via the function `contextVariable(variableName)`. They declare context variables used in the pattern matches within inner code blocks. For example, the name of the currently specified table (in outer code block) can be passed via such variables to patterns matching code within inner code blocks. This can be helpful for e.g. recommendations of column names in the inner projections. When a pattern is initialized for the matching process, each such active node is replaced by a value specified by the corresponding context variable. The latter are set in a surrounding context by argument active nodes. Furthermore, context variables can also be declared and initialized globally for the whole editor, which is useful for providing the names of global variables to the projections.

2.2 Pattern Matching on Abstract Syntax Trees

Contrary to pure projectional editors which store programs in form of serialized trees (e.g. XML or JSON files), the hybrid editor serializes the source code in conventional text-based form, internally maintaining an equivalent AST. Rendering requires dynamic detection of projected code fragments matched by the patterns, extraction of the relevant information, and dynamic creation or update of each projection.

Algorithms 1 and 2 show the pseudocode of the approach for matching patterns in GPL code. We use the following symbols. *Src* is a syntax tree of the currently edited file and *Pat* is the set of the defined patterns. *Ctx* is a mapping from context variables to their values, initially populated with globally defined context variables.

Algorithm 1 visits each node *n* of the AST *Src* in a depth-first search fashion (line 3). For each *n* and each defined pattern *p* it checks via `MATCHPATTERN` whether the subtree of *Src* starting at *n* matches the pattern *p*. The pseudocode does not show a small optimization such that for *n* only patterns are considered with the same type of root node as the type of *n* (see [42] for all details).

If there is match of *p* at *n*, the result set *Res* is updated (line 8). Furthermore, for each inner code block identified by this match we issue a recursive call of `FINDPATTERNS` at

Algorithm 1 Top-level algorithm for matching patterns.

Require: Src, Pat, Ctx ▷ See text for definitions.

```

1: function FINDPATTERNS( $Src, Ctx$ )
2:    $Res \leftarrow \emptyset$  ▷ Resulting set of matches
3:   for all  $n \in \text{DFS}(Src)$  do
4:     for all  $p \in Pat$  do
5:        $Args \leftarrow \emptyset$  ▷ Set of captured arguments
6:        $Blks \leftarrow \emptyset$  ▷ Set of captured blocks
7:       if MATCHPATTERN( $p, n, Args, Blks, Ctx$ ) then
8:          $Res \leftarrow Res \cup \{(p, n, Args, Blks)\}$ 
9:         ▷ Recursively search all inner code blocks ◁
10:        for all  $(\hat{n}, \hat{C}) \in Blks$  do
11:           $\hat{S} \leftarrow \text{GETSUBTREEATNODE}(\hat{n}, Src)$ 
12:           $Res \leftarrow Res \cup \text{FINDPATTERNS}(\hat{S}, Ctx \cup \hat{C})$ 
13:        end for
14:        break ▷ At most one match per n
15:      end if
16:    end for
17:  end for
18:  return  $Res$ 
19: end function

```

a subtree \hat{S} , i.e. the root node of the code block, to support compositionality (lines 11-12).

The algorithm returns a set of *matches*. Each is a tuple $(p, n, Args, Blks)$ of a pattern p , root node n of an AST matching p , and sets $Args, Blks$ of captured arguments and captured blocks, respectively.

The pattern candidates are evaluated by function MATCHPATTERN. In essence, a node n of Src matches a pattern when it is deeply equivalent to the pattern's root node, see the recursive descend at lines 24-29. This function also updates the sets $Args$ and $Blks$ according to the meaning of the active nodes (lines 2-9), see Section 2.1, and checks that the context variables have correct values in the candidate AST subtree (lines 10-13).

2.3 Rendering and Editing Projections

Given a set of matches m_1, \dots, m_k (output of Algorithm 1) and the predefined projection templates the hybrid editor is capable of rendering the edited program S . Essentially, all statements (or AST subtrees) in S not included in any of the matchings are identified as non-projected code. These parts of S are shown in a traditional textual form. On the other hand, the rendering of each projected code fragment is delegated to the respective projection.

The process of updating internal model of the hybrid editor requires more detailed explanation. When the source code changes due to edits of the textual or the projectional parts the internal model (i.e. source code state) of the hybrid editor becomes invalid. Incremental, i.e. local updates of this

Algorithm 2 A function for testing whether an AST tree matches a pattern and extracting arguments and code blocks.

```

1: function MATCHPATTERN( $p, n, Args, Blks, Ctx$ )
2:   if isArgumentNode( $p$ )  $\wedge$  nodeType( $n$ ) = requiredType( $p$ ) then
3:      $Args \leftarrow Args \cup \{(\text{argumentName}(p), n)\}$ 
4:     return true
5:   end if
6:   if isBlockNode( $p$ ) then
7:      $Blk \leftarrow Blk \cup \{(n, \text{blockCxtSet}(p))\}$ 
8:     return true
9:   end if
10:  if isContextVariable( $p$ ) then
11:     $v \leftarrow \text{contextVariableName}(p)$ 
12:    return isIdentifier( $n$ )  $\wedge v \in \text{dom}(Ctx) \wedge Ctx(v) = \text{text}(n)$ 
13:  end if
14:  if nodeType( $p$ )  $\neq$  nodeType( $n$ ) then
15:    return false
16:  end if
17:  ▷ Atomi (leaf) nodes must be compared by their texts ◁
18:  if isAtomic( $p$ ) then
19:    return  $\text{text}(p) = \text{text}(n)$ 
20:  end if
21:  if  $|\text{children}(p)| \neq |\text{children}(n)|$  then
22:    return false
23:  end if
24:  ▷ Children of  $p$  and  $n$  are assumed to be aligned ◁
25:  for all  $\hat{p} \in \text{children}(p), \hat{n} \in \text{children}(n)$  do
26:    if  $\neg \text{MATCHPATTERN}(\hat{p}, \hat{n}, Args, Blks, Ctx)$  then
27:      return false
28:    end if
29:  end for
30:  return true
31: end function

```

model are difficult to achieve due to presence of context variables. Therefore we need to rerun Algorithm 1 in order to re-match the entire AST against the predefined patterns, and re-render the hybrid representation.

To optimize this process, we reuse the instantiated projections created by the previous model if possible. To this end we need to compare source code ranges r'_1, \dots, r'_k of the projected code (obtained from the matchings m'_1, \dots, m'_k) from the previous model against the projected code ranges r_1, \dots, r_k of the new model. If some r'_j and r_i overlap for the same pattern, we can reuse the previously instantiated projection after updating it with new argument values. In general, this provides a smooth user experience even for larger source code files.

```

1 tbl["name"] = tbl["name"].replace("Mister", "Mr.");
2 replace Mister with Mr. in column name
3
4 tbl["title"] = tbl["title"].replace("PhD", "Dr.");
5 replace PhD with Dr. in column title
6
7 tbl["adr"] = tbl["adr"].replace("Road", "Rd.");
8 replace Road with Rd. in column adr

```

Listing 5. Samples of projected code and the corresponding (textual) projection.

3 Generating DSL Components from Examples

In this section we propose an algorithm for synthesizing simple patterns and projections from multiple samples of GPL code and textual projections.

While the development effort of a pattern and a corresponding projection template is modest (below 100 LOC of a typical case, see Section 5), a developer of a virtual DSL might be faced with a steep learning curve due to multiple frameworks/technologies. In our implementation, developing a pattern requires knowledge of TypeScript and Javascript’s tagged template literals (see Section 2.1), and creating a projection template assumes familiarity with Svelte and optionally with HTML/CSS.

To this end we propose an approach to synthesize a basic form of a pattern and a corresponding projection template from multiple examples of pairs (projected code, projection). We assume that the projections are of textual form, like in lines 1 and 4 in Listing 2. Consequently, projection templates can be understood in context of this section as string templates where static text is interweaved with parts displaying variable and editable content called *placeholders*.

Listing 5 gives an example of 3 pairs of samples, each pair consisting of projected code and the corresponding expected textual projection. The code samples must be selected or created by the DSL-developer such that they differ in tokens indicating placeholders. Here the argument to `tbl["name"]` and both arguments to `replace()` indicate the differences. Analogously, differences between projection texts indicate positions of the placeholders. For the first pair, these positions are at the tokens `Mister`, `Mr.`, and `name`. By using the same placeholder values in a code sample and in its projection we indicate how the placeholders should be mapped.

Given this input, our algorithm is able to generate a *derived pattern* and corresponding *derived (projection) template*. Latter is the core part of a GUI widget (in our prototype, a Svelte component) rendering the projection in the hybrid editor. Despite of some limitations, this solution covers many simple cases without a need for any coding. The generated code can be useful even for more experienced developers as a skeleton for further extensions.

```

1 // Pattern prototype (AST unparsed)
2 tbl[<cp1>] = tbl[<cp2>].replace(<cp3>, <cp4>);
3
4 // Projection template prototype (list joined)
5 replace <pp1> with <pp2> in column <pp3>

```

Listing 6. Pattern and projection template prototypes derived from Listing 5. `<>` indicate placeholders.

The relevant limitations of the approach are (in addition to text-only form of projections) lack of context variables (Section 2.1.1). Moreover, a derived pair pattern/template can have at most one nested code block. A DSL-developer can still provide these features, they are not generally precluded. We opted for this simple form to reduce ambiguity and keep the number of required examples low.

In the following we detail our approach. Section 3.1 discusses how we identify constant and variable parts of GPL code by comparing the AST trees of the samples. In Section 3.2 we explain how projection samples are used to generate a textual template. Section 3.3 outlines mapping of placeholders found in code to those found in projections. Finally, Section 3.4 explains how this information is used to generate the pair pattern/projection template.

3.1 Analyzing Samples of Projected Code

The goal of analyzing samples is to obtain a *pattern prototype* defined as an AST-tree with some nodes marked as placeholders. Listing 6 (top) shows the unparsed, i.e. serialized pattern prototype derived from the code samples in Listing 5. Additionally, we compute a list of paths to the found placeholders in this prototype. This is required for mapping pattern and projection placeholders.

The algorithm works iteratively. Each sample code is parsed to an AST (which are formally also pattern prototypes). The comparison of two first ASTs yields a first candidate for pattern prototype, which is then compared against third code sample AST, yielding second candidate result etc. When all nodes of X are processed, we return the last candidate as the pattern prototype, and a list of paths to the placeholders.

A recursive comparison of two ASTs X and Y (or technically, two pattern prototypes) is the core of this routine. Given a list of AST nodes x_1, \dots, x_k (e.g. children of a node) from X and the same-length list y_1, \dots, y_k of nodes from Y , we test which node pairs (x_i, y_i) are identical and which differ. A pair is considered identical if both nodes have same AST-type, same content (e.g. identifier text), and are leaves. If x_i are y_i are same-type non-leaves with equal number of children, we perform a recursive descend on the set of their children.

Otherwise, x_i and y_i are considered different. They are marked as placeholders, and their tree path is recorded. If both have the same AST-node type, we inherit this type to

their placeholders. In other case it is considered a 'wildcard', i.e. the placeholder can match any type.

3.2 Analyzing Samples of Projections

Analogously to code samples, strings representing textual projections are compared for common (equal) and diverging parts. The goal is to obtain a (*projection*) *template prototype* which is an alternating list $c_0, v_0, c_1, v_1, \dots$ of static strings c_i (for common parts) and placeholders v_i (for diverging parts). We assume here that two subsequent variable parts are always separated by a static part, e.g. a comma. Listing 6 (bottom) shows the template prototype derived from the projection samples in Listing 5 (list joined to one string).

First, each sample projection text is turned into a list of tokens. Note that this list can be considered a special case of a prototype. We set the first sample as the initial solution and then iteratively refine the current version p of a template prototype by comparing it against the next sample s .

At each iteration step, we use the Meyer's Diff algorithm [51] for detecting the longest common token subsequences between p and s . We exploit the fact that each longest common token subsequence c_i must be followed by a diverging part v_i (v_i might be optional for the last common subsequence). For each c_i found by Meyer's Diff algorithm we can thus add to the updated version p' of the prototype the string c_i as the subsequent static part, and the v_i as the next placeholder.

3.3 Mapping Placeholders from Code to Projections

The next goal is to map placeholders found in code samples (Section 3.1) to those found in projection samples (Section 3.2). To make the presentation clearer, we call the earlier *c-placeholders* cp and the latter *p-placeholders* pp . In Listing 6 we have c -placeholders cp_1, \dots, cp_4 and p -placeholders pp_1, \dots, pp_3 .

Assume that we are given c -placeholders cp_1, \dots, cp_m , p -placeholders pp_1, \dots, pp_n , a projection template prototype t (found as in Section 3.2), a list V of tree paths to the c -placeholders (found as in Section 3.1), and the pairs of samples $(c_1, p_1), \dots, (c_k, p_k)$. We want to find a relation M which maps each cp_i to exactly one pp_j , and each pp_j appears in the image of M (in other words, M is surjective). It means that multiple placeholders in the code can be 'connected' by the same value, but this group corresponds to only one placeholder in the projection.

We propose the following algorithm to this end. We iterate over the pairs of samples and build M incrementally. For (c_x, p_x) we compute a list Q of strings of p_x which correspond to the placeholders in the template prototype t (the list is ordered such that i th element corresponds to pp_i). We can achieve this by using again the Meyer's Diff algorithm on p_x and t : each divergent sequence of tokens corresponds to one p -placeholder, since common subsequences are identical static parts of p_x and t . For example, p_1 in Listing 5 gives

$Q = ["Mister", "Mr.", "name"]$ corresponding to pp_1, pp_2, pp_3 in t .

The strings in Q create a link between p -placeholders and c -placeholders since they should also appear in the code sample c_x . To exploit this, we iterate over V (the tree paths to c -placeholders), and for each $v_i \in V$ we retrieve the AST node or AST-subtree in c_x referenced by v_i . The text obtained from this AST fragment is searched in Q . If found at position j in Q , we have obtained a mapping of a c -placeholder cp_i to p -placeholder pp_j : $M(cp_i) = pp_j$. For example, after processing the first pair (c_1, p_1) , we obtain the mapping M defined by $cp_1 \rightarrow pp_3, cp_2 \rightarrow pp_3, cp_3 \rightarrow pp_1, cp_4 \rightarrow pp_2$.

If different pairs of samples create incoherent mappings (i.e. a c -placeholder is mapped to different p -placeholders), the algorithm terminates with an error. We also check that all p -placeholders are covered. Finally, M is returned after all pairs of samples are processed.

3.4 Constructing a Pattern and a Projection Template

After the three processing steps outlined above a derived pattern and a corresponding projection template can be constructed. We save both artifacts a regular source code files which can be used or edited as any manually developed pattern/template pair.

A derived pattern is obtained from any code sample c_i (formally, a string) by replacing the substrings corresponding to c -placeholders by strings representing calls to helper functions discussed in Section 2.1.1. To this end we iterate over the AST paths identifying c -placeholders (found in Section 3.1), for each one we retrieve the substring range in c_i corresponding to this AST path, and replace it by the function call. Information about the source code range of each AST subtree is provided by default by tree-sitter.

A derived projection template is created from a skeleton of a GUI widget which displays text only and admits at most one nested block. The parametrized part is a list of tokens, where each token is either an HTML string for the static parts of the projection or a p -placeholder. Each p -placeholder is displayed as a `TextInput` component and 'wired' to the correct c -placeholder(s) (i.e. helper function calls) in the derived pattern according to the mapping M (Section 3.3).

4 Implementation Details

4.1 Overall Architecture

We use CodeMirror [8] web-based editor as a basis for our hybrid editor. In this environment, any components representable in HTML and JavaScript (in particular, any Svelte components) can be used as projection widgets. However, conceptually other extensible editors or IDEs such as Visual Studio Code or its basis library Monaco Editor [14] are a suitable basis for the hybrid editor.

Following this choice, the primary programming language of the project is TypeScript, with some complementary libraries like tree-sitter using other languages at their core. In particular, all algorithms described in Section 2 are implemented in TypeScript.

We make the hybrid editor available in two environments. The first is as a stand-alone browser-based editor with a web server as a backend (see the live demo). The other environment is JupyterLab [40], a popular tool data scientists. Since JupyterLab uses CodeMirror as its cell (code) editor, we can integrate our hybrid editor as a frontend extension, without affecting the Jupyter backend. This environment is used in both virtual DSLs discussed in Section 5.

4.2 Projections: Implementation and Editor Integration

Projections in the hybrid editor could be implemented by any of the web front end frameworks based on JavaScript, like React, Angular, Vue, or Svelte. Using such generic frameworks for projections offer flexibility, thus potentially allowing even more special projections such as diagrams or editors for graphical user interfaces. We choose Svelte [11] to its reactivity, small size of transferred files, ease of debugging, and lifecycle events.

A key feature needed in a hybrid editor is the ability to replace ranges of text with custom display and editing components. CodeMirror offers this ability via *replacement decorations* [33] which can render any HTML content instead of a specified text range. As CodeMirror renders its contents using conventional HTML and CSS, it is possible to turn these elements into interactive widgets (i.e. projections) using JavaScript.

When the document undergoes updates, previous decorations are retained if they remain relevant in the updated positions. Consequently, the Svelte properties update to the new match, leading to an automatic refresh of the widget's contents. This approach ensures continuity and alignment with user input.

Our system understands that CodeMirror expects widget classes to adhere to a specific element-producing interface. To streamline this process, we've designed a wrapper function that automatically morphs Svelte components into the desired widget classes. This wrapper plays a crucial role, ensuring that all critical data, such as the pattern match, context, and editor state, is transferred and updated as Svelte properties whenever changes occur.

4.2.1 Input Management. User input management is pivotal in our system. A shared `TextInput` component handles user input directly, replacing the source code at the corresponding AST node. This action triggers a syntax tree re-match, causing the system to update the components with fresh match data. The underlying principle is maintaining the textual document as the authentic source for projections

visible to the user, ensuring consistency. Replacing code with user input is a straightforward process, especially for string literals, where the requirement is merely replacing any quotation marks with their escaped variants. For more complex node types, we've implemented serialization functions that guarantee the produced code aligns with the field's pattern node.

To ensure the projections blend well with the rest of the document, cursor movement is monitored to provide a fluent transition between projection and code. We include a *focus manager* in every projection instance that is responsible for keeping track of the focusable inputs inside a projection widget and of the input that is currently focused. On internal cursor movement, the focus manager ensures the next or previous input element is focused, depending on whether the cursor has "left" the field on the start or end side. For the first and last input inside the projection, the cursor is moved to a position in the text editor adjacent to the projection instead. Overall, this improves the user experience by allowing modifications of the whole document without having to use a mouse.

New projections are inserted into the edited program through our hybrid editor's completion mechanism. As the user types a text sequence resembling a projection's name, they can choose a suggested projection to insert into the document. The system then generates a code snippet from the original code pattern template, replaces template placeholders with empty values, and integrates it into the document.

4.3 Simplified Definitions of Projections

The analysis of the development effort via counting non-comment Lines of Code (LOC) in Section 5 reveals that the majority of line count can be attributed to the code of a Svelte component. However, a closer look at the Svelte components reveals that these consist mostly of import statements and HTML code instead of logic. Consequently, we have introduced a wrapper function `simpleProjection`. It creates projectional widgets without need to understand Svelte components or the internals of the CodeMirror editor.

Listing 7, lines 12-15 shows an application of this function. It receives an array of tokens, where a token is either a string, a reference to a code placeholder `cp` (as defined in Section 3), or an array of such references if multiple source code parts are to be modified by a projection field. The placeholder reference must be named identically as the special element produced by the `arg` helper function for `cp` (Section 2.1) created when defining the associated pattern. In this way, we 'wire' the code placeholders with the projection placeholders and avoid typos.

Listing 7 shows a pattern definition in the TL (top) and a simplified projection definition (bottom) of a component for spreadsheet analysis DSL. The object references `fName`, `sName` specifying code placeholders (or special elements of type argument node) in the pattern definition are used in

```

1 // Pattern definition
2 const dsl = contextVariable("dsl");
3 const fName = arg("fName", "string");
4 const sName = arg("sName", "string");
5
6 export const [pattern, draft] =
7 pythonParser.statementPattern`
8 with ${dsl}.load_sheet(${fName}, ${sName}) as sheet:
9   ${block({ sheet: "sheet" })}
10 `;
11
12 // Simplified definition of a projection
13 export const widget = simpleProjection(
14   ["load sheet", sName, "from", fName, ":"]
15 );

```

Listing 7. Definitions of a pattern via Templating Language and its projection in a simplified form.

```

1 with dsl.load_sheet("Chromatography.xlsx", "raw data")
2 as sheet:
3   K, B = sheet.take("K4:L35", "isinstance(L,int)")
4   A, B = sheet.join("A4:B46704", K, "abs(K - A)",
5                     dsl.AggregationMethod["minimal"])
6   dsl.store_sheet("output.xlsx", "output", [K, L, B])

```

Listing 8. Example Python script for processing Excel data.

the call to `simpleProjection`. In this way, variable parts of the source code and the projection are uniquely connected.

5 Evaluation

5.1 DSL for Spreadsheet Analysis

We implemented a virtual DSL for analysis of Excel spreadsheets. The use case is motivated by a cooperation with researchers from a biomedical field. Their lab uses chromatography methods for genome analysis of viruses. The device performing the analysis exports its results as Excel files which need further analysis to extract relevant data. Due to data variations and lack of programming skills these files are processed manually, which is a repetitive and error-prone task. With the virtual DSL these researchers should be able to use and adapt scripts for automated processing yet retain the flexibility of a host GPL for a future automated processing pipeline.

The implementation uses as the environment JupyterLab and Python as the host GPL. We developed a simple package (library) in Python based on the module *openpyxl* [27] which performs several tasks like loading a spreadsheet, conditional cell selection, joining of cell ranges, and storing the results in a new spreadsheet. Listing 8 shows a typical Python script used in this scenario. Here `dsl` is a qualifier to our package.

For this virtual DSL we implemented four pattern/projection pairs, for the library calls *load*, *take*, *join*, and *store*. Figure 2 illustrates how the code from Listing 8 is actually shown in the JupyterLab hybrid editor.

```

1 Load sheet raw data from Chromatography.xlsx:
2   take K, B from K4:L35 where isinstance(L, int)
3   join A, B from A4:B46704 on K where abs(K - A) is minimal
4   store K, L, B in sheet output of output.xlsx

```

Figure 2. Code from Listing 8 as shown in the hybrid editor.

5.1.1 Discussion. Implementation of this virtual DSL has shown qualitatively that the proposed concept of a virtual DSL is useful. Moreover, the presented algorithms and the implementation have been verified as correct. The hybrid editor is responsive and due to the special care of input management (Section 4.2.1) offers a smooth editing experience.

However, we did not conduct a rigorous user study. We relied on feedback from one tester as well as our own experiences with the system to conclude high responsiveness and correctness of the implementation. The intended customers have not yet introduced the system (changes in the process require time investment on both sides) but are interested. A more thorough evaluation with a user study remains a part of the future work.

Note that the hybrid editor presentation in Figure 2 is not *radically* more user friendly than the textual representation of code in Listing 8. However, for researchers from non-CS fields the threshold for considering a software solution as ‘too complex’ is rather low. Consequently, even a moderate improvement of readability and editing experience (also due to parameter recommendations within projections) was relevant in this case.

5.2 DSL for Formula Editing

The next virtual DSL offers graphical formula (or equation) editing using mathematical notations. It demonstrates that our approach can also tackle non-textual projections, see Figure 3. Instead of implementing the desired projections from scratch, we leverage the fact that our hybrid editor can use any HTML/JavaScript components, and build upon Mathlive [28], a web-based graphical editor for mathematical equations. We choose LaTeX as its output format. This enables to use *latex2sympy2* [46], a library to transform LaTeX equations into SymPy [12] objects at runtime. SymPy can perform on these objects tasks such as simplification and solving of equations, but it can also generate executable Python functions.

We first implement a small Python library/module *mathdsl* that transforms LaTeX equations into performant (due to internal use of NumPy), executable Python functions. The module provides two functions: *compile* and *evaluate*. Earlier function transforms LaTeX equations into an executable form but does not evaluate them, while the other immediately performs the evaluation.

Both functions assume as their first argument a string containing the LaTeX code. The *compile* function returns


```

1 // Pattern for the compile function
2 expressionPattern'
3   ${contextVariable("mathdsl")}
4   .compile(${arg("latex", "string")})
5   '
6 // Pattern for the evaluate function
7 expressionPattern'
8   ${contextVariable("mathdsl")}
9   .evaluate(${arg("latex", "string")}, locals())
10  '

```

Listing 9. Two patterns for the graphical formula editor expressed in the Templating Language.

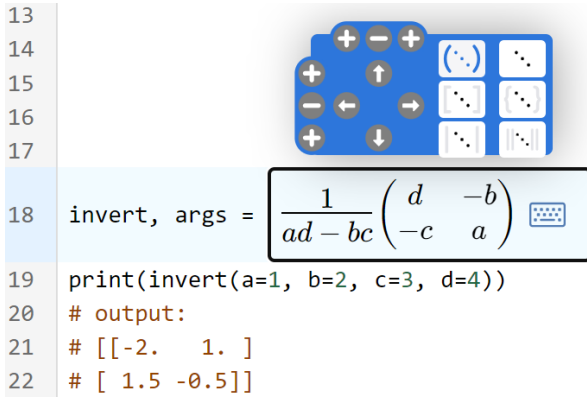


Figure 3. A hybrid editor view during editing a function to compute an inverse of a 2x2-matrix with mathdsl.

the generated function as well as a tuple of free symbols in the equation. For the evaluation function, an additional parameter must be provided next to the code that describes the variable scope to evaluate the term in. The patterns used to match these operations are defined in Listing 9 using the Templating Language. Unlike in previous examples, we declare these patterns to be expressions instead of statements. This makes it possible to embed the equation editor in normal code statements.

The corresponding hybrid editor has been integrated in JupyterLab [40] (see sources of our project) and in the live demo (tab 'Python'). Figure 1 in the introduction shows how the hybrid editor represents a function for rotating a vector. The corresponding Python source code is given in Listing 3. Another example is shown in Figure 3 with the corresponding source code in Listing 10. While editing the formula in the hybrid editor, interactive GUI elements support matrix editing and transformations. Also, visual keyboard is available for entering symbols and operators without knowledge of LaTeX (not shown; available in live demo).

Overall, we conclude that the choice HTML/JavaScript-based projection widgets, allows for a wide range of scenarios and types of projectional forms. Especially this enables access to a large ecosystem of open source libraries and components for web-based GUIs.

```

1 invert, args = mathdsl.compile("\\frac{1}{ad-bc}
2   {\\begin{pmatrix}d & -b \\\\ -c & a \\end{pmatrix}}")
3 print(invert(a=1, b=2, c=3, d=4))
4 # output:
5 # [[-2.  1. ]
6 # [ 1.5 -0.5]]

```

Listing 10. Python source code corresponding to Figure 3.

Table 1. Implementation sizes in non-blank LOC of four operations for the spreadsheet DSL. The total LOC for simplified definition of projections is shown in bold.

Operation	Pattern	Svelte	Svelte (simple)	Total
load	21	49	3	70/ 24
take	22	56	3	78/ 25
join	24	77	3	101/ 27
store	21	55	3	76/ 24

5.3 Analysis of the Development Effort

We analyze and discuss here the implementation size of the virtual DSL from Section 5.1 as a proxy for the development effort of such a DSL. Due to a limited scope of this project we could not yet conducted a controlled developer study on the coding effort. However, such a study can be pursued as a part of the future work.

Table 1 shows the number of non-comment Lines of Code (LOC) the four DSL components. Our original implementation of Svelte components produced a rather verbose code with large LOC values, see column "Svelte" in this table. These large LOC numbers can be significantly reduced by using the simplified definition of projections introduced in Section 4.3. While the simplified definitions slightly limit the flexibility of projections, it is possible to use them for the four DSL components constituting the virtual DSL for analysis of Excel spreadsheets. Listing 7 shows that even the most complex of the 4 projections can be defined in this way. The LOC count becomes for each projection only 3.

On average, the LOC for pattern code is 22 LOC. Using the simplified projection definitions we reach an average of 25 LOC as the implementation length of a DSL component, or 100 LOC for the whole spreadsheet DSL. This is already comparable to the size to the Python library code (69 LOC). The original 'naive' implementation of Svelte components gives an average of 59.25 LOC per projection and average of 81.25 LOC per DSL component. We conclude that the simplified definition of projection templates can significantly reduce the implementation effort and improve maintainability.

5.4 Generating Patterns and Projections from Samples: Advantages and Limitations

The approach for generating DSL components from examples introduced in Section 3 might even further reduce the development effort and make virtual DSL accessible for users

```

1 # Code sample 1
2 with dsl.load_sheet("chroma.xlsx", "raw data") as sheet:
3   sheet.do_something()
4 # Projection sample 1
5 load sheet raw data from chroma.xlsx :
6
7 # Code sample 2
8 with excel_dsl.load_sheet("cost.xlsx", "April") as sheet:
9   sheet.do_some_other_thing()
10 # Projection sample 2
11 load sheet April from cost.xlsx :
```

Listing 11. Samples for the *load* operation.

with limited programming skills. The majority of the code required for defining patterns and projections can be provided by the generator, thus decreasing the initial hurdle for developers to create a projectional editor for their DSL. The key question is whether the generator can create correct patterns and projection templates for realistic scenarios. To this end, we attempted to generate these components for all DSL components of the spreadsheet DSL from Section 5.1. Due to space limits, we describe here only the results for the *load* and *take* operations (results for *join* and *store* were similar).

Running the samples from Listing 11 through our generator algorithm yields a pattern and projection template whose placeholders are correctly connected. Note the space character before the colon in the projection samples, which is required to satisfy the tokenizer used in the generator.

This evaluation reveals some shortcomings of the approach from Section 3.1. In the chosen samples, the nested statements included in the respective *with* statements' blocks only differ by the method identifier after the dot infix. Thus, the algorithm generates a pattern that includes two variable identifier nodes at these positions instead of a block. If we replace these two nested statements by code without any AST overlap, such as `x = 42` and `print("Hello")`, then a wildcard placeholder is generated which would match an arbitrary statement, but not a block with multiple statements. To express the intent of including multiple statements in the block, one more statement must be added to one of the samples. Now, the algorithm correctly detects a block at this position in the AST.

Furthermore, the generator does not perform any kind of context analysis. Thus, the variable nodes `dsl` and `excel_dsl` are not identified as context variables but simply as placeholder nodes of type *identifier*. Similarly, the generated pattern does not provide any context variables to the nested block. These two parts must be edited by hand in the generated pattern.

Finally, it is advisable to change the generated variable names to more meaningful names to help with later maintenance. We call the two variables that map to parts of the projection `fileName` and `sheetName`. The context variable

```

1 # Pattern
2 with ${contextVariable("dsl")}.load_sheet(
3   ${arg("fileName", "string")},
4   ${arg("sheetName", "string")}
5 ) as sheet:
6   ${block({ sheet: "sheet" })}
7
8 # Projection
9 load sheet sheetName from fileName :
```

Listing 12. Final pattern and projection for the *load* operation.

```

1 # Code sample 1
2 K, L = sheet.take("K4:L35", "isinstance(L, int)")
3 # Projection sample 1
4 take K, L from K4:L35 where isinstance(L, int)
5
6 # Code sample 2
7 A, B, C = the_sheet.take("A1:C100", "A + B == C")
8 # Projection sample 2
9 take A, B, C from A1:C100 where A + B == C
```

Listing 13. Samples for the *take* operation.

for the DSL module name is simply called `dsl` and the sheet context variable passed to the nested block becomes `sheet`.

The final pattern and projection are shown in Listing 12. For sake of brevity, we show only a textual form of the projection instead of the code of the generated Svelte component.

Listing 13 shows samples for the operation *take*. Similar problems with *load* occur here at the attempt to generate a pattern and a projection template. Instead of a problem with detection of a block, the code samples must have a different amount of identifiers before the assignment to express that the pattern should not match the identifiers themselves as placeholders but the whole list of identifiers. Also here context variables are incorrectly detected and must be replaced by the context variable `sheet` in the generated pattern. As before, we rename the variables to be more meaningful.

This case study revealed how our generator can be further improved. Context variables in particular are not covered by the generator and require manual editing of the generated code. Currently, the code samples provided to the generator are isolated per DSL operation, preventing a connection of placeholders that appear in multiple operations. Comparing the code samples of different operations should allow the generator to identify placeholder nodes as context variables if they appear in multiple operations but not in any of the projections. Furthermore, the declarations of nested blocks could automatically be filled with the context variables by analyzing the identifier or parameter nodes affecting the variable scope inside the block. This however requires knowledge about the host language's scope resolution behavior.

6 Related Work

Research areas relevant to our work are *hybrid textual editors*, *projectional editors*, *Domain Specific Languages*, and *language workbenches*. Furthermore, work in *programming by example* is connected to our code generation approach.

Hybrid textual editors relate directly to our work. We use this term to bracket contributions which extend textual code editors with the ability to express parts of the code in an alternative form, typically as visual GUI elements. Contrary to our work, the approaches described below do not address generation of such GUI components from examples.

Eisenberg and Kiczales [18–20] propose *ETMOP*, an Eclipse-based editor which renders selected parts of Java code as interactive graphical elements. Using Java 5 annotations, a developer specifies which methods and classes should be shown in a graphical way. Similarly to our work, ETMOP uses pattern matchers to recognize these annotations. However, our approach matches directly the code parts to be shown as projections and does not require annotations or other developer intervention. Our technique also works for host GPL languages which do not support annotations.

Later works [1, 54, 59] share the same concept of specifying which parts of code are represented in an alternative form via annotations or syntax extension of the edited code.

Work of Renggli *et al.* propose *language boxes* [59], a modular mechanism to encapsulate language extensions such as DSLs. The key idea is to extend the grammar of the host language (here: Smalltalk) using the concept of executable grammars [5]. The approach can combine different textual notations using delimiters inserted explicitly using a special-purpose editor. It assumes that the grammar, compiler and editor of the host language can be extended, which incurs a considerable effort for mainstream languages like Python.

Andersen *et al.* propose syntactic extensions for the Racket language [24] in terms of *interactive syntax* [1]. Such (textual) syntax extensions are rendered as interactive GUIs by an appropriate editor, here DrRacket. The authors posit that the approach is compatible with other languages with a macro system, such as Closure, Julia, Rust, or even C++. Contrary to this, our method does not require a macro system and works with languages like Python or JavaScript. Our approach also supports compositionality, i.e. projections can be nested.

Omar *et al.* introduce live literals, or *livelits* which embed user-defined GUIs into textual code [54]. The implementation in Hazel/OCaml extends the above-cited work [1] with compositionality, type safety, and liveness. The latter means that the evaluation of livelits occur in the runtime environment of the program being written. As [1], livelits require a host language with a macro system and a pure (functional) host language, differently to our work. However, the authors note that imperative languages can be also covered, with more effort.

A slightly different flavor of a hybrid textual editor is described in [55]. Upon triggering the code completion functionality of the editor, the developer is presented with a context-sensitive palette, i.e. a GUI which allows for an interactive specification of expressions, statements or API parameters. A prototype called *Graphite* provides palettes for regular expressions and for color selection in Java. Differently to our approach, palettes must be associated with a target class via annotations or explicitly via menus. Also, Graphite shows the enriched code representation only locally and during editing, which limits code comprehension.

Barista [41] is an implementation framework designed for creating projectional editors with a strong support for textual input. Editors constructed with Barista represent code as a proprietary data structure (a model), but they deploy parsing techniques that treat the structure as if it were textual. For example, equations are displayed visually as math, but when a caret is placed on them, a textual view is shown.

French *et al.* [26] propose a similar system which can embed interactive graphical objects in Java or Python textual code. The internal representation takes form of a modified AST. Contrary to our work, the last two approaches must be aware of all syntactic structures of a target language, leading to a considerably larger development effort.

Further systems add visual programming to textual code. Examples include Boxer [17], Scratch [60], or Smalltalk with its live programming capability [48]. Mathematica [35] supports expressions which include visual elements such as images or diagrams. These can be copied and pasted into textual expressions but also manipulated in a GUI-fashion. Jupyter and JupyterLab [40] allow inserting widgets like sliders that modify parameters of code. The Jupyter API extension *mage* [39] enables creation of tools that can represent themselves as both code and GUI as needed.

Our previous work [2] enhances text editing by embedding user-defined DSLs as code comments. During editing, a code completion action on these comments expands a DSL expression into regular Python or R code. Our tool *NLDSL* [10] is implemented as an extension for Visual Studio Code and can be easily ported to all IDEs which support Microsoft's Language Server Protocol [13]. Contrary to work presented here, NLDSL requires developers to edit DSL expressions and manually trigger code generation. Moreover, NLDSL supports only textual DSLs.

Projectional editors were pioneered in the 1970s and 1980s by projects like *Cornell Program Synthesizer* [65], *Incremental Programming Environment* [49], and *GANDALF* [52]. A key feature of such systems is the ability to provide different ways of viewing and editing program components. For example, embedded data can be shown as a spreadsheet, state machine as a diagram, and other parts of code as text. In particular, they can express code fragments in a simplified form resembling a DSL. Projectional editors store and manipulate programs directly as an abstract syntax tree, so

parsing is not necessary [9, 15, 37]. Consequently, they can mix arbitrary programming languages (multiple GPLs and DSLs) in a single view without ambiguity.

While text-based languages can be edited in any text editor, projectional editing ties the language to a specialized editor that is aware of the language's syntax and possibly semantics [66, Chap. 7.3]. Voelter *et al.* [67] conducts an analysis of user-friendliness of projectional editing based on JetBrains MPS. Their conclusion is that projectional editors do not lead to code being written faster than in conventional text editors. The authors identify usability challenges in the areas of (i) efficiently entering textual code, (ii) selecting and modifying code, and (iii) infrastructure integration.

Approaches such as *grammar cells* [68] address the challenges (i) and (ii) and attempt to improve editing experience in such editors. The key idea is a formalism for textual notations via declarative specification of the projectional editor's behavior. In our work we use a sophisticated input management via focus managers (Section 4.2.1) to improve the editing experience in the projections.

Domain Specific Languages (DSLs) [16, 25, 38, 43, 66] focus on a limited domain but allow its modeling in a concise and readable way. By 'making it hard to do something wrong' [25], they can also reduce the amount of defects and improve maintainability.

Research activities related to DSLs include extensible languages like Racket [23, 24], language-oriented programming and language workbenches [31, 44], analysis of the usage and properties of DSLs [25, 43, 50], syntactic macros [4, 24, 34], or enhancing libraries by "syntactic sugar" [22].

Modern DSL engineering frameworks like textX [16] or Spoofox [38] significantly lower the cost of developing a DSLs. An even higher level of productivity and toolchain integration can be achieved by language workbenches discussed below.

Language workbenches such as JetBrains' MPS [7], MontiCore [44, 61], or Xtext [3] complement and extend DSL development frameworks by providing editors with syntax checking and code completion for created DSL or GPL languages. They also facilitate code parsing and generation.

JetBrain's *Meta-Programming System (MPS)* [7, 36, 56] is probably the most popular publicly available language workbench. Since it uses projectional editing, users can switch between different notations or visualizations of the same program. The AST manipulated by the editor is stored in form of an XML file. It is then translated into a target language such as Java or C by the MPS code generator.

Lafontant *et al.* propose *Gentleman*, a lightweight Web-based language workbench [45]. It offers commonly used interface layouts, such as tables, or horizontal and vertical stacks to design custom visual representations of code. It is similar to our approach by using web-based components as projections and their styling through CSS.

Contrary to our approach, these systems persist source code in a proprietary format. This makes it harder to use other editors and complicates integration with existing tools.

Programming by example (PBE) is a form of program synthesis which uses input and/or output examples to specify the desired code, typically a DSL expression [6, 21, 30, 63, 64]. In the past decade, many sophisticated approaches have been developed, including FlashMeta [57], a framework for designing and implementing program synthesis engines for custom DSLs. PBE has been commercially exploited in e.g. Excel, PowerShell and other products to automate string manipulation, data preprocessing, and other tasks [29, 30, 47, 58]. Recently introduced Large Language Models have also been applied in the field of program synthesis, typically in the context of input specification by natural language [53].

Our method for generating patterns and projection templates from examples (Section 3) is a form of highly specialized PBE. However, instead of a DSL we generate domain-specific GPL code. For patterns, the algorithm described in Section 3 creates an AST-based data structure which is then converted into a Template Language, i.e. annotated JavaScript code. For projections, we use code skeletons which are populated with the corresponding placeholders (see Section 3.4). Compared to work referenced above, our approach is more narrow and less flexible but has lower complexity and implementation effort.

7 Conclusions and Future Work

We proposed an approach and a tool for hybrid editing of code by mixing a textual representation and projectional views. Latter can assume virtually any form - from textual, DSL-like formats to tables to equations. The editor directly manipulates the source code of the host GPL. This enables seamless integration of the virtual DSL and the GPL, and facilitates integration in the development toolchain. Our evaluation shows that the approach provides a good developer's experience and requires an acceptable effort of developing DSL components. To reduce this effort, we have proposed an approach to generate virtual DSL components from samples of code and corresponding textual projection.

Our future work will include an implementation of a larger use scenario in order to refine the approach and the evaluation. Another task is to offer our hybrid editor as an extension for Visual Studio Code. We will also improve the approach for generating DSL components from samples by cross-component analysis as discussed in Section 5.3. An interesting but rather challenging problem is generation of non-textual projections from examples. Furthermore, a more rigorous analysis of the soundness and completeness of the proposed algorithms is needed. Finally, adding type checking for projectional editing could improve user experience.

References

- [1] Leif Andersen, Michael Ballantyne, and Matthias Felleisen. 2020. Adding Interactive Visual Syntax to Textual Code. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 222 (nov 2020), 28 pages. <https://doi.org/10.1145/3428290>
- [2] Artur Andrzejak, Kevin Kiefer, Diego Elias Costa, and Oliver Wenz. 2019. Agile Construction of Data Science DSLs (Tool Demo). In *Proceedings of the 18th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences* (Athens Greece). ACM, 27–33. <https://doi.org/10.1145/3357765.3359516>
- [3] Lorenzo Bettini. 2016. *Implementing Domain Specific Languages with Xtext and Xtend - Second Edition* (2nd ed.). Packt Publishing.
- [4] Jeffrey Werner Bezanon. 2015. *Abstraction in technical computing [Julia language]*. Thesis. Massachusetts Institute of Technology. <http://dspace.mit.edu/handle/1721.1/99811>
- [5] Gilad Bracha. 2007. Executable Grammars in Newspeak. *Electronic Notes in Theoretical Computer Science* 193 (2007), 3–18. <https://doi.org/10.1016/j.entcs.2007.10.004>
- [6] José Cambroner, Sumit Gulwani, Vu Le, Daniel Perelman, Arjun Radhakrishna, Clint Simon, and Ashish Tiwari. 2023. FlashFill++: Scaling Programming by Example by Cutting to the Chase. *Proc. ACM Program. Lang.* 7, POPL, Article 33 (jan 2023), 30 pages. <https://doi.org/10.1145/3571226>
- [7] Fabien Campagne. 2016. *The MPS Language Workbench Volume I: The Meta Programming System (Volume 1)* (3rd ed.). CreateSpace Independent Publishing Platform, USA.
- [8] Sergei Chestakov. 2022. *Betting on CodeMirror*. Retrieved September 10, 2023 from <https://blog.replit.com/codemirror>
- [9] Tony Clark. 2015. A General Architecture for Heterogeneous Language Engineering and Projectional Editor Support. (2015). arXiv:1506.03398 [cs]
- [10] NLDL contributors. 2023. NLDL Overview. <https://aip.ifi.uni-heidelberg.de/software/nldl> Accessed on September 10, 2023.
- [11] Svelte contributors. 2023. Svelte - Cybernetically enhanced web apps. <https://svelte.dev/> Accessed on September 10, 2023.
- [12] SymPy contributors. 2023. SymPy - a Python library for symbolic mathematics. <https://www.sympy.org> Accessed on September 10, 2023.
- [13] Microsoft Corp. 2023. Language Server Protocol Specification. Retrieved September 10, 2023 from <https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification/>
- [14] Microsoft Corp. 2023. Monaco - The Editor of the Web. <https://microsoft.github.io/monaco-editor/> Accessed on September 10, 2023.
- [15] Riwan Cuinat, Ciprian Teodorov, and Joel Champeau. 2020. SpecEdit: Projectional Editing for TLA+ Specifications. In *2020 IEEE Workshop on Formal Requirements (FORMREQ)*. 1–7. <https://doi.org/10.1109/FORMREQ51202.2020.00008>
- [16] I. Dejanović, R. Vadera, G. Milosavljević, and Ž. Vuković. 2017. TextX: A Python tool for Domain-Specific Languages implementation. *Knowledge-Based Systems* 115 (Jan. 2017), 1–4. <https://doi.org/10.1016/j.knosys.2016.10.023>
- [17] A. diSessa and H. Abelson. 1986. Boxer: A Reconstructible Computational Medium. *Commun. ACM* 29, 9 (sep 1986), 859–868. <https://doi.org/10.1145/6592.6595>
- [18] Andrew David Eisenberg. 2008. *Presentation techniques for more expressive programs*. Ph.D. Dissertation. University of British Columbia. <https://doi.org/10.14288/1.0051292>
- [19] Andrew D. Eisenberg and Gregor Kiczales. 2006. A Simple Edit-Time Metaobject Protocol: Controlling the Display of Metadata in Programs. In *Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications* (Portland, Oregon, USA) (OOPSLA '06). Association for Computing Machinery, New York, NY, USA, 696–697. <https://doi.org/10.1145/1176617.1176679>
- [20] Andrew D. Eisenberg and Gregor Kiczales. 2007. Expressive Programs through Presentation Extension. In *Proceedings of the 6th International Conference on Aspect-Oriented Software Development* (Vancouver, British Columbia, Canada) (AOSD '07). Association for Computing Machinery, New York, NY, USA, 73–84. <https://doi.org/10.1145/1218563.1218573>
- [21] Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sablé-Meyer, Lucas Morales, Luke Hewitt, Luc Cary, Armando Solar-Lezama, and Joshua B. Tenenbaum. 2021. DreamCoder: Bootstrapping Inductive Program Synthesis with Wake-Sleep Library Learning. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) (PLDI 2021). Association for Computing Machinery, New York, NY, USA, 835–850. <https://doi.org/10.1145/3453483.3454080>
- [22] Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann. 2011. SugarJ: Library-based Syntactic Language Extensibility. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications* (OOPSLA '11). ACM, New York, NY, USA, 391–406. <https://doi.org/10.1145/2048066.2048099>
- [23] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt. 2015. The Racket Manifesto. In *1st Summit on Advances in Programming Languages (SNAPL 2015)* (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 32), Thomas Ball, Rastislav Bodik, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett (Eds.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 113–128. <https://doi.org/10.4230/LIPIcs.SNAPL.2015.113>
- [24] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt. 2018. A Programmable Programming Language. *Commun. ACM* 61, 3 (feb 2018), 62–71. <https://doi.org/10.1145/3127323>
- [25] Martin Fowler. 2010. *Domain Specific Languages* (1st ed.). Addison-Wesley Professional.
- [26] G.W. French, J.R. Kennaway, and A.M. Day. 2014. Programs as Visual, Interactive Documents. *Softw. Pract. Exper.* 44, 8 (aug 2014), 911–930. <https://doi.org/10.1002/spe.2182>
- [27] Eric Gazoni and Charlie Clark. 2023. openpyxl - A Python library to read/write Excel 2010 xlsx/xlsm files. <https://openpyxl.readthedocs.io/en/stable/> Accessed on September 10, 2023.
- [28] Arno Gourdol. 2023. Mathlive - Equations served with a side of interaction. <https://cortexjs.io/mathlive/> Accessed on September 10, 2023.
- [29] Sumit Gulwani. 2011. Automating String Processing in Spreadsheets Using Input-Output Examples. *SIGPLAN Not.* 46, 1, 317–330. <https://doi.org/10.1145/1925844.1926423>
- [30] Sumit Gulwani. 2016. Programming by Examples (and its Applications in Data Wrangling). In *Verification and Synthesis of Correct and Secure Systems*. IOS Press. <https://www.microsoft.com/en-us/research/publication/programming-examples-applications-data-wrangling/>
- [31] Gopal Gupta. 2015. Language-based Software Engineering. *Sci. Comput. Program.* 97, P1 (Jan. 2015), 37–40. <https://doi.org/10.1016/j.scico.2014.02.010>
- [32] Jordan Harband, Shu-yu Guo, Michael Ficarra, and Kevin Gibbons (Eds.). 2020. *ECMAScript® 2021 Language Specification* (12 ed.). Ecma International.
- [33] Marijn Haverbeke. 2022. *CodeMirror Decoration Example*. Retrieved September 10, 2023 from <https://codemirror.net/examples/decoration/>
- [34] David Herman and Mitchell Wand. 2008. A Theory of Hygienic Macros. In *Programming Languages and Systems*, Sophia Drossopoulou (Ed.). Lecture Notes in Computer Science, Vol. 4960. Springer Berlin Heidelberg, 48–62. https://doi.org/10.1007/978-3-540-78739-6_4
- [35] Wolfram Research Inc. 2008. *Dynamic Interactivity – Wolfram Mathematica Tutorial Collection*. Wolfram Research Inc. <https://library.wolfram.com/infocenter/1/1/1176617.1176679.pdf>

- wolfram.com/infocenter/Books/8513/
- [36] JetBrainsTV. 2017. *Why JetBrains MPS*. Retrieved September 10, 2023 from https://www.youtube.com/watch?v=XGm_khXZl44
 - [37] Ján Juhár and Liberios Vokorokos. 2015. A Review of Source Code Projections in Integrated Development Environments. In *2015 Federated Conference on Computer Science and Information Systems (FedCSIS)*. 923–927. <https://doi.org/10.15439/2015F289>
 - [38] Lennart C.L. Kats and Eelco Visser. 2010. The Spoofox Language Workbench: Rules for Declarative Specification of Languages and IDEs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '10)*. ACM, New York, NY, USA, 444–463. <https://doi.org/10.1145/1869459.1869497>
 - [39] Mary Beth Kery, Donghao Ren, Fred Hohman, Dominik Moritz, Kanit Wongsuphasawat, and Kayur Patel. 2020. Mage: Fluid Moves Between Code and Graphical Work in Computational Notebooks. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology (Virtual Event, USA) (UIST '20)*. Association for Computing Machinery, New York, NY, USA, 140–151. <https://doi.org/10.1145/3379337.3415842>
 - [40] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica Hamrick, Jason Grout, Sylvain Corlay, Paul Ivanov, Damián Avila, Safia Abdalla, and Carol Willing. 2016. Jupyter Notebooks – a publishing format for reproducible computational workflows. In *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, F. Loizides and B. Schmidt (Eds.). IOS Press, 87 – 90.
 - [41] Amy J. Ko and Brad A. Myers. 2006. Barista: An Implementation Framework for Enabling New Tools, Interaction Techniques and Views in Code Editors. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Montréal, Québec, Canada) (CHI '06). Association for Computing Machinery, New York, NY, USA, 387–396. <https://doi.org/10.1145/1124772.1124831>
 - [42] Niklas Korz. 2022. *Projectional Editing of Internal Domain-Specific Languages*. Master's thesis. Heidelberg University.
 - [43] Tomaž Kosar, Sudev Bohra, and Marjan Mernik. 2016. Domain-Specific Languages: A Systematic Mapping Study. *Information and Software Technology* 71 (March 2016), 77–91. <https://doi.org/10.1016/j.infsof.2015.11.001>
 - [44] Holger Krahn, Bernhard Rumpe, and Steven Völkel. 2010. MontiCore: A Framework for Compositional Development of Domain Specific Languages. *Int. J. Softw. Tools Technol. Transf.* 12, 5 (sep 2010), 353–372. <https://doi.org/10.1007/s10009-010-0142-1>
 - [45] Louis-Edouard Lafontant and Eugene Syriani. 2020. Gentleman: A Light-Weight Web-Based Projectional Editor Generator. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings* (New York, NY, USA) (MODELS '20). Association for Computing Machinery, 1–5. <https://doi.org/10.1145/3417990.3421998>
 - [46] latex2sympy2 contributors. 2023. latex2sympy2 - Parse LaTeX math expressions. <https://github.com/OrangeX4/latex2sympy> Accessed on September 10, 2023.
 - [47] Vu Le and Sumit Gulwani. 2014. FlashExtract: A Framework for Data Extraction by Examples. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 542–553. <https://doi.org/10.1145/2594291.2594333>
 - [48] John Maloney, Kimberly M. Rose, and Walt Disney Imagineering. 2001. An Introduction to Morphic: The Squeak User Interface Framework. In *Squeak: Open Personal Computing and Multimedia*, Mark Guzdial and Kimberly Rose (Eds.). Prentice Hall, 39–77.
 - [49] R. Medina-Mora and P.H. Feiler. 1981. An Incremental Programming Environment. *IEEE Transactions on Software Engineering* SE-7, 5 (1981), 472–482. <https://doi.org/10.1109/TSE.1981.231109>
 - [50] Marjan Mernik, Jan Heering, and Anthony M. Sloane. 2005. When and how to develop domain-specific languages. *Comput. Surveys* 37, 4 (dec 2005), 316–344. <https://doi.org/10.1145/1118890.1118892>
 - [51] Eugene W. Myers. 1986. AnO(ND) difference algorithm and its variations. *Algorithmica* 1, 1-4 (nov 1986), 251–266. <https://doi.org/10.1007/bf01840446>
 - [52] David Notkin. 1985. The GANDALF project. *Journal of Systems and Software* 5, 2 (1985), 91–105. [https://doi.org/10.1016/0164-1212\(85\)90011-1](https://doi.org/10.1016/0164-1212(85)90011-1)
 - [53] Augustus Odena, Charles Sutton, David Martin Dohan, Ellen Jiang, Henryk Michalewski, Jacob Austin, Maarten Paul Bosma, Maxwell Nye, Michael Terry, and Quoc V. Le. 2021. Program Synthesis with Large Language Models. arXiv:2108.07732 [cs.PL]
 - [54] Cyrus Omar, David Moon, Andrew Blinn, Ian Voysey, Nick Collins, and Ravi Chugh. 2021. Filling Typed Holes with Live GUIs. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) (PLDI 2021). Association for Computing Machinery, New York, NY, USA, 511–525. <https://doi.org/10.1145/3453483.3454059>
 - [55] Cyrus Omar, Young Seok Yoon, Thomas D. LaToza, and Brad A. Myers. 2012. Active code completion. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE. <https://doi.org/10.1109/icse.2012.6227133>
 - [56] Vaclav Pech, Alex Shatalin, and Markus Voelter. 2013. JetBrains MPS as a tool for extending Java. In *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools, Stuttgart, Germany, September 11-13, 2013*, Martin Plümcke and Walter Binder (Eds.). ACM, 165–168. <https://doi.org/10.1145/2500828.2500846>
 - [57] Oleksandr Polozov and Sumit Gulwani. 2015. FlashMeta: A Framework for Inductive Program Synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Pittsburgh, PA, USA) (OOPSLA 2015). Association for Computing Machinery, New York, NY, USA, 107–126. <https://doi.org/10.1145/2814270.2814310>
 - [58] Mohammad Raza and Sumit Gulwani. 2017. Automated Data Extraction Using Predictive Program Synthesis. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA.*, Satinder P. Singh and Shaul Markovitch (Eds.). AAAI Press, 882–890. <http://aaai.org/ocs/index.php/AAAI/AAAI17/paper/view/15034>
 - [59] Lukas Renggli, Marcus Denker, and Oscar Nierstrasz. 2010. Language Boxes. In *Software Language Engineering*, Mark van den Brand, Dragan Gašević, and Jeff Gray (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 274–293.
 - [60] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and Yasmin Kafai. 2009. Scratch: Programming for All. *Commun. ACM* 52, 11 (nov 2009), 60–67. <https://doi.org/10.1145/1592761.1592779>
 - [61] Bernhard Rumpe, Katrin Hölldobler, and Oliver Kautz. May 2021. *MontiCore Language Workbench and Library Handbook: Edition 2021* (2021 ed.). Shaker Verlag. <https://www.se-rwth.de/research/MontiCore/AachenerInformatik-Berichte,SoftwareEngineering,Band48>
 - [62] Tree sitter contributors. 2023. Tree-sitter - a parser generator tool and an incremental parsing library. <https://tree-sitter.github.io/tree-sitter/> Accessed on September 10, 2023.
 - [63] Calvin Smith and Aws Albarghouthi. 2016. MapReduce Program Synthesis. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. ACM, New York, NY, USA, 326–340. <https://doi.org/10.1145/2908080.2908102>
 - [64] Armando Solar-Lezama. 2008. *Program Synthesis by Sketching*. PhD Thesis. University of California at Berkeley, Berkeley, CA, USA.

- [65] Tim Teitelbaum and Thomas Reps. 1981. The Cornell Program Synthesizer: A Syntax-Directed Programming Environment. *Commun. ACM* 24, 9 (sep 1981), 563–573. <https://doi.org/10.1145/358746.358755>
- [66] Markus Voelter, Sebastian Benz, Christian Dietrich, Birgit Engelmänn, Mats Helander, Lennart C. L. Kats, Eelco Visser, and Guido Wachsmuth. 2013. *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*. dslbook.org.
- [67] Markus Voelter, Janet Siegmund, Thorsten Berger, and Bernd Kolb. 2014. Towards User-Friendly Projectional Editors. In *Software Language Engineering*, Benoît Combemale, David J. Pearce, Olivier Barais, and Jurgen J. Vinju (Eds.). Lecture Notes in Computer Science, Vol. 8706. Springer International Publishing, 41–61. https://doi.org/10.1007/978-3-319-11245-9_3
- [68] Markus Voelter, Tamás Szabó, Sascha Lisson, Bernd Kolb, Sebastian Erdweg, and Thorsten Berger. 2016. Efficient Development of Consistent Projectional Editors Using Grammar Cells. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering* (Amsterdam, Netherlands) (*SLE 2016*). Association for Computing Machinery, New York, NY, USA, 28–40. <https://doi.org/10.1145/2997364.2997365>

Received 2023-07-14; accepted 2023-09-03

Generating Constraint Programs for Variability Model Reasoning: A DSL and Solver-Agnostic Approach

Camilo Correa Restrepo
camilo.correa-restrepo@univ-
paris1.fr
Université Paris 1 Panthéon-Sorbonne
Paris, France

Jacques Robin
jacques.robin@esiea.fr
Ecole Supérieure d'Informatique-
Electronique-Automatique
Paris, France
Université Paris 1 Panthéon-Sorbonne
Paris, France

Raul Mazo
raul.mazo@ensta-bretagne.fr
Lab-STICC, ENSTA-Bretagne
Brest, France

Abstract

Verifying and configuring large **Software Product Lines (SPL)** requires automation tools. Current state-of-the-art approaches involve translating variability models into a formalism accepted as input by a constraint solver. There are currently no standards for **variability modeling languages (VML)**. There is also a variety of constraint solver input languages. This has resulted in a multiplication of ad-hoc architectures and tools specialized for a single pair of VML and solver, fragmenting the SPL community. To overcome this limitation, we propose a novel architecture based on model-driven code generation, where the syntax and semantics of VMLs can be declaratively specified as data, and a standard, human-readable, formal pivot language is used between the VML and the solver input language. This architecture is the first to be fully generic by being agnostic to both VML and the solver paradigm. To validate the genericity of the approach, we have implemented a prototype tool together with declarative specifications for the syntax and semantics of two different VMLs and two different solver families. One VML is for classic, static SPL, and the other for run-time reconfigurable *dynamic* SPL with soft constraints to be optimized during configuration. The two solver families are **Constraint Satisfaction Programs (CSP)** and **Constraint Logic Programs (CLP)**.

CCS Concepts: • Software and its engineering → Software product lines; Software architectures; • Computing methodologies → Model verification and validation.

Keywords: Software Product Lines, Automated Reasoning, Generic Architecture, Configuration Automation

Publication rights licensed to ACM. ACM acknowledges that this contribution was authored or co-authored by an employee, contractor or affiliate of a national government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.

GPCE '23, October 22–23, 2023, Cascais, Portugal

© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 979-8-4007-0406-2/23/10...\$15.00

<https://doi.org/10.1145/3624007.3624060>

ACM Reference Format:

Camilo Correa Restrepo, Jacques Robin, and Raul Mazo. 2023. Generating Constraint Programs for Variability Model Reasoning: A DSL and Solver-Agnostic Approach. In *Proceedings of the 22nd ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '23)*, October 22–23, 2023, Cascais, Portugal. ACM, New York, NY, USA, 15 pages. <https://doi.org/10.1145/3624007.3624060>

1 Introduction

SPL Engineering (SPLE) [40] is a collection of engineering techniques for managing the design, development, and subsequent evolution of large sets of software products that share partially overlapping sets of requirements and reusable software assets implementing them. Considered as one whole, these products form the eponymous SPL. Crucially, each product in an SPL is a *variant* that includes a specific subset of the requirements and reusable assets. This variability makes SPLE unique: it needs to be managed across the entire product line to control the quality and consistency of the products. To carry this out, SPLE prescribes the construction, verification and maintenance of an additional software artifact called a *variability model*, that represents the relationships holding among the SPL's variable requirements and the reusable assets implementing them. These models have been used for three main distinct purposes:

- *Software Mass Customization* [30] from a baseline software version with automated variant code generation from reusable artifacts.
- *Design space exploration* [33], *i.e.*, determining which set of design choices best satisfies¹ a set of hard (*i.e.*, must have) and soft (*i.e.*, nice to have) requirements.
- Context-aware *autonomic adaptation* [32, 54] through run-time self-reconfiguration.

The third purpose involves embarking the variability model as a runtime artifact to support re-exploring the design space during operations to find an alternative configuration that restores requirement satisficing following an operational

¹*Satisfice* is a portmanteau word derived from *satisfying* and *suffice*, conveying satisfaction to a sufficient extent rather than necessarily completely.

context change that rendered the current configuration inadequate. SPLs for such self-adaptive systems are called **Dynamic SPLs (DSPL)** [25].

In all three cases, the models support the semi-automatic derivation of correct *configurations*, i.e., choices of cohesive and consistent requirements that satisfy the business, technical and regulatory constraints captured by the variability model. Real-life, industrial SPLs are too large and complex to be *manually* verified, troubleshoot and correctly configured [34]. To make things worse, these tasks need to be repeated after each evolution of the SPL during its life-cycle that routinely spans over multiple decades [10]. Automated reasoning must thus be used to repeatedly verify, troubleshoot and (re)configure an SPL at initial design-time, evolution-time and even runtime throughout its life-cycle. In most cases, the *augmented* intelligence [20] flavor of automated reasoning is required, in which the reasoning must be explainable to the team of human engineers and operators managing the SPL and that have the final say concerning each configuration or defect correction choice.

Today, there is no accepted standard SPL VML, so every SPLE automation tool uses its own **Domain-Specific VML (DSVML)**. However, despite superficial syntactic differences, at the semantic level, most DSVMLs used for mass customization define cohesive requirement sets, generally called *features*, and share four key expressive capabilities:

- SFM1** Organize these features into abstraction and composition hierarchies and associate the lowest level ones with reusable and composable concrete software assets implementing them.
- SFM2** Distinguish between *mandatory* features shared by all configurations from *optional elements* specific to strict sub-spaces of the whole configuration space.
- SFM3** Specify ranges of alternative possibilities for the refinement of a higher-level feature into a set of lower-level features.
- SFM4** Specify simple required co-occurrence or exclusion constraints between two features across the abstraction hierarchy.

DSVML providing such expressive capabilities are generally referred to as **Simple Feature Models (FM)**. Four additional expressive capabilities are provided by DSVML which are often called *Extended FM*.

- EFM1** Structure features into sets of attributes of various types rather than limiting them to Boolean variables.
- EFM2** Specify ranges of alternative possible values for those attributes.
- EFM3** Specify multiplicity constraints on those attributes and on relationships among features.
- EFM4** Specify complex constraints on the values of attributes of features located anywhere in the abstraction hierarchy.

Whether simple or extended, all FMs used for mass customization only contain *hard* constraints that must be collectively consistent and fully satisfied by all valid configurations. In contrast, models used for design space exploration and autonomic adaptation also contain *soft* constraints to satisfy as much as possible rather than necessarily fully; thus their modeling languages need to be semantically more expressive.

A wide variety of approaches have been proposed to implement SPL model verification and model-guided SPL configuration in SPLE automation tools. Just like for VMLs, there is also currently no accepted standard API for such tools. Nevertheless, the overwhelming majority of them include a translator of the DSVML to some logical **Knowledge Representation Language (KRL)**. This allows them to reuse practically scalable **Inference Engines (IE)** developed over the last 50 years for formal software engineering and artificial intelligence. Four main paradigms of such IEs have been used to automate SPL model verification and model-guided configurations:

- **SATisfiability (SAT)** solvers [24] and their extensions with **Satisfiability Modulo Theories (SMT)** [16].
- CSP solvers and their extensions for *Constraint Optimization Problems* [17].
- Logic programming engines and their CLP extensions [21].
- Description logic engines and their semantic web ontology reasoning extensions [4].

Notably, the foundational feature-based SPLE tool FODA used a logic programming engine [28], the original version of VariaMos [51] used a CLP engine, COFFEE used a CSP solver [52], AUFG used a description logic engine [38], FeatureIDE [47], FlamaPy [22], Splot [36], Glencoe [44], Kernel Haven [29] and pure::variants [9], all use a SAT solver, while Familiar can use either an SMT or a CSP solver [1].

Each pair of KRL and IE from these paradigms corresponds to a different trade-off in terms of semantic expressiveness, inference scalability and reasoning explainability. The best KRL for a given SPL reasoning task is thus very much dependent on both the nature of that task and the semantic expressiveness of the DSVML used to model variability [6]. Since SPLE is a heavy upfront investment method whose return on investment takes a fairly long time before becoming tangible [40], SPLE projects have long life cycles. Therefore, both the expressiveness requirements of a DSVML and the automated reasoning tasks to analyze the variability model and correctly (re)configure the SPL can evolve significantly during its life cycle.

The main common limitation of the state-of-the-art SPLE automation tools listed above is their *ad-hoc* architectures that tightly couples a single DSVML with an IE from a given automated reasoning paradigm. This impedes one from choosing the IE in lockstep with the evolution of the DSVML

and reasoning task requirements at a cost that is low enough to avoid denting the long-term benefits of adopting SPLE.

In an attempt to overcome this severe limitation of current SPLE tools, we try, in this paper, to answer the following open research question:

How to architect an SPLE automation tool, in which IEs from various paradigms can be seamlessly plugged in and out, to adapt the tool's reasoning capabilities to the evolution of both (a) the semantic expressiveness of the DSVMLs that it accepts as input and (b) the analysis and configuration tasks that need to be automated on an SPL, throughout its lifecycle.

Our first research hypothesis on this question is that such an architecture must satisfy the following requirements:

- REQ1** Support low-cost extension of existing DSVMLs and addition of new DSVMLs.
- REQ2** Support low-cost addition of new automated reasoning tasks to run on the variability model.
- REQ3** The architecture must be agnostic *w.r.t.* the logical KRL and IE paradigm used for automated reasoning on the variability model, supporting low-cost addition of interoperability with solvers from different paradigms.
- REQ4** The architecture must be agnostic *w.r.t.* the DSVML editor tool, accepting the variability model as input data exportable from multiple popular editors.

To satisfy these requirements, we propose the following principles inspired from model-driven engineering [43]:

- DP1** The concrete and abstract syntaxes of the DSVML should be decoupled from one another.
- DP2** They both should be *declaratively* specified as data in a widely used exchange format, rather than hard-coded in the SPLE tool.
- DP3** The semantics of the DSVML should also be *declaratively* specified as data in a widely used exchange format encoding a mapping from the abstract syntax elements to expressions in a formal KRL.
- DP4** The set of reasoning automation tasks to be run on the variability model should also be *declaratively* specified as data in a widely used exchange format, which must furthermore be decoupled from any specific IE KRL.
- DP5** The many-to-many translation from the multiple DSVMLs to the multiple IE KRLs should be decoupled into a pipeline of N many-to-one transformations to a *standard pivot intermediate language* followed by M one-to-many transformations from this pivot, to avoid the combinatorially explosive cost of developing and evolving $N \times M$ direct DSVML to IE KRL transformations.
- DP6** This standard pivot language must be easily interpretable by a wide range of stakeholders.

Our second research hypothesis is that **REQ1** and **REQ2** can be satisfied by the combination of **DP1**, **DP2** and **DP3**,

REQ2 can be satisfied by **DP4** and **REQ3** by **DP5**. In the rest of the paper, we attempt to verify these two hypotheses.

To do so we proceed as follows. In section 2, we start by presenting two SPLE reasoning task examples that we have used as a first step toward validating these hypotheses. While they are small enough to fit in this article, they are purposely representative of very different variability modeling language families used for very different purposes. In that section, we also explain how these tasks can be carried out by leveraging, as intermediate pivot language between the two different input DSVMLs and any logical IE input KRL, the ISO standard for logical IE interoperability **CLIF (Common Logic Interchange Format)** [27] following the original proposal of [13]. Next, in section 3, we propose a detailed SPLE tool architecture following the design principles **DP1** to **DP5** listed above. We then discuss the prototype SPLE tool **PLEIADES (Product Line Engineering Intelligent Assistant for Defect detection Explanation and Solving)** that we (a) implemented to show the practical feasibility of this architecture and (b) tested on the two example tasks presented in section 2. In section 5, we then compare PLEIADES with state-of-the-art SPLE tools. Since none of them aimed to satisfy requirements **REQ1** through **REQ4**, nor explicitly followed design principles **DP1** to **DP5**, this comparison is grounded on various tool versatility criteria which are met by following these principles. In section 6 we discuss the limitations of both the presented architecture and its current implementation in terms of satisfaction of requirements **REQ1** to **REQ4** and the future work that we intend to carry out to overcome them. Finally in section 7, we conclude by recapitulating the original contributions of the presented research.

2 Background and Running Examples

To illustrate our approach on concrete examples, we now present two of them. The first concerns a defect detection verification task on a simple FM. The second concerns an optimal reconfiguration search for a DSPL that leverages a DSVML specifically designed for that purpose and presented in [42]. They are thus two intentionally distant points in the space of variability models encountered in the literature.

2.1 Variability Modeling Languages

2.1.1 Simple Feature Models. In Fig. 1, we present a minimalist FM example. Its graphical concrete syntax shows features as rectangular vertices in a directed graph and the hierarchical decomposition of features in mandatory and optional sub-features as edges ending with a filled or empty circle (respectively). This decomposition forms a tree. Edges ending with an arrow represent exclusion or co-occurrence constraints among feature pairs in different tree branches. The widgets at the top-right of each graph node allows

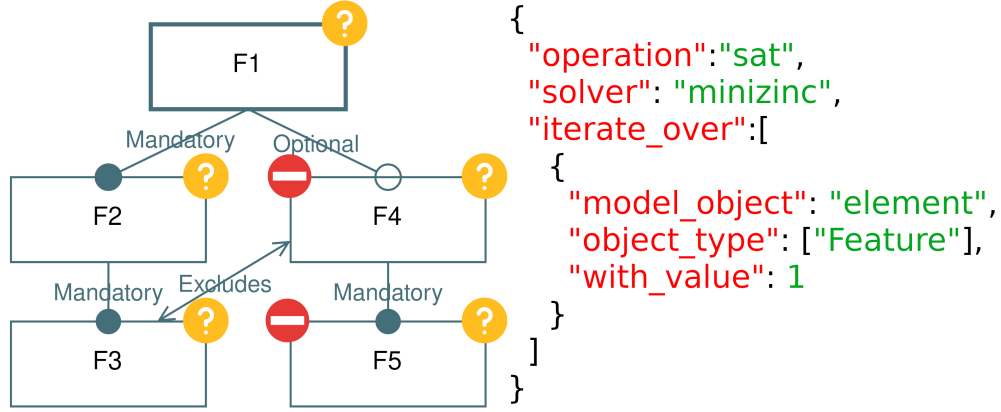


Figure 1. First part of our running example with a synthetic Feature Model. The query specifying the iterative search for dead features in the model is beside the model.

reusing the FM diagram to overlay partial or complete configurations on top of it: a question mark in an orange circular widget means that the node element has not yet been selected nor excluded from the configuration, either manually by the user or by automated constraint propagation. Since this first example illustrates an FM verification task prior to configuration, all the widgets are of this type in Fig. 1. As exemplified in Fig. 2, selected elements are annotated by a check sign on a blue circular widget while excluded elements are annotated by a cross sign on a red circular widget.

Verification reasoning tasks on variability models generally search for various defects, which were comprehensively categorized in [6]. In figure 1 we show a simple FM with two dead feature defects. A feature is said to be "dead" if, although it is present in the variability model, it can never be selected in any valid configuration due to contextual constraints relating to other features. The one-way widgets on the top-left of features F4 and F5 visualize the result of this task. It detects that F4 and F5 are dead. F4 is dead as excluded by the selection of F3, which must be selected in all configurations as a mandatory descendant of the top-level feature F1. F5 is also dead as a child of F4, a dead feature.

Following principle DP4, the text beside the simple FM diagram is the declarative specification in JSON [14] of the search for dead feature defects reasoning task. It reads as follows:

- Line 2: the reasoning task being carried out is a satisfiability check denoted as *sat*, i.e., determining if there are valid solutions for the model;
- Line 3: the IE to be used is the CSP solver *minizinc*[37];
- Lines 4-10: this solver must be iteratively called on each of the feature selected (i.e., with value 1) in the graph.

Section 2.3 further details the reasoning task query specification language.

2.1.2 Sawyer et al.'s DSPL VML. In Fig. 2, we present a second example of reasoning task. This time it is a search

for an optimal reconfiguration leveraging the context-aware DSPL Variability Model of a flood early-warning system. The vocabulary of concepts and relations of this DSPL DSVML [42] is much more extensive than that of simple FM used in the first example. They are the following:

- *Hard goals*, shown as green parallelograms, determine the functional requirements of the system and are analogous to features in FMs. Hard goals are structured in a decomposition hierarchy where higher level ones can be achieved by achieving all their lower level components.
- *Soft Goals*, shown as blue clouds, encode the non-functional requirements of the system and can be satisfied on a 0 to 4 scale, which is encoded as "--", "-", "=", "+", "++" in the model. They are themselves structured into a decomposition hierarchy. The level of satisficing of a higher level soft goal in this hierarchy is the average of the satisficing level of its lower level components.
- *Context Variables*, shown in blue rectangles on the right, encode the state of the system's context among symbolic value enumerations.
- *Operationalizations*, shown as gray hexagons, are concrete software assets that can implement the hard goals.
- *Bundles*, shown as white circles, contain integer range expressions for the multiplicity constraints on the operationalizations that can implement a hard goal.
- *Claims*, shown as white trapezoids, express the extent to which operationalizations satisfy soft goals as a function of which have been selected.
- *Soft Influences*, shown as grey ellipses, relate the context variables to the soft goals, and determine the required level of satisfaction when the given state is determined by the context, e.g., if the context variable *BatteryHealth* is set to "Low", the required level of satisfaction of *EnergyEfficiency* is "++".

The Variability Model in Fig. 2 represents the various redundant means of communications available to transfer sensor

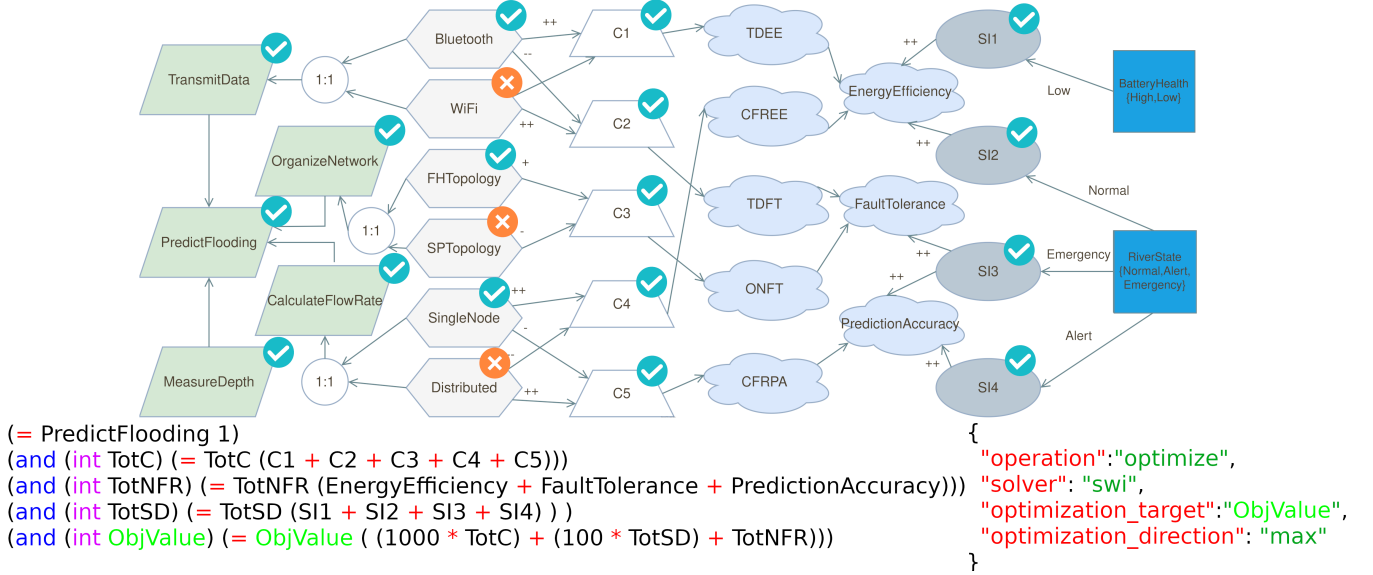


Figure 2. Second part of our running example with a model using Sawyer et al.’s DSPL variability modeling Language. This model reproduces the original example from the paper [42] for a flood early-warning system. As with our other example, in addition to a reasoning task to solve (on the bottom right), this time, an optimization problem to find an optimal configuration whose additional constraints are shown below the model (bottom left).

data in a distributed autonomic flood early-warning system. It also supports determining which set of means represent the best trade-off between energy consumption, fault tolerance and prediction accuracy in various flood risk contexts. The reasoning task specification shown on the bottom-right of Fig. 2 can be read as follows:

- Line 2: The reasoning task is to find an *optimal* solution (as opposed to merely checking for satisfiability as in Fig. 1), denoted as *optimize*;
- Line 3: the IE to use is *swi*, the CLP solver SWI-Prolog with its finite domain constraints library CLP(FD) [55];
- Line 4: the optimization *target*, i.e., the variable whose value must be optimized is *ObjValue* (defined as equal to the weighted sum of other variables in the model, the details of which are outlined in section 2.3);
- Line 5: the optimization direction is maximization.

The contrast between the models of Figs. 1 and 2 highlights the diversity of DSVMLs and reasoning tasks that can be carried out by our PLEIADES framework and tool.

2.2 CLIF and Model Transformation

In PLEIADES, we encode the semantics of the different variability modeling languages using the CLIF ISO standard for logical IE interoperability [27]. It evolved from the earlier Knowledge Interchange Format [23] and uses a LISP [35] derived syntax to represent logical expressions. For example, $(a \wedge (b \vee c))$ is written as `(and a (or b c))` where a , b and c can also be nested subexpressions. The semantics of a variability model is expressed as a set of CLIF sentences.

The solvers are employed to determine the satisfiability of this set sentences or to derive from it concrete models of them, in the formal logic model-theoretic semantics sense of the word. Complex or long range constraints relating distant nodes in the variability model graph cannot be concisely and intuitively represented by a graphical concrete syntax. This is a recurrent issue of graphical software modeling that is not specific to SPLE. For example, it is addressed by the textual *Object Constraint Language (OCL)* [39] complementing the graphical *Unified Modeling Language (UML)* [45] in the Object Management Group standard ecosystem. Since CLIF is already a textual ISO standard and the target of the first step of the VML to solver input translation in PLEIADES, we opted to use it also as the default language to express constraints relating both arbitrary model elements and those elements with elements of the reasoning task specification. For example, the CLIF sentence shown at the bottom left of Fig. 2 defines the optimization target property of the task specification to its right, in terms of various model elements.

Figure 3 exemplifies PLEIADES’ implementation of design principle DP5. On the left side of the diagram, two fragments of the models in Figs. 1 and 2 are shown. For the former we reproduce the exclusion constraint between Features F3 and F4, and, for the latter, we reproduce *TransmitData* as one of the four sub-goals of the *PredictFlooding* hard goal. The second column shows the DSVML to CLIF semantic JSON translation rules for each of the elements in the model fragments to their left. Both the nodes and the edges of each graph have their own semantics. As a consequence, the JSON



Figure 3. Concrete examples of translation using the translation rules specification for Feature Models (top) and Sawyer et al.’s DSPL variability modeling Language (bottom).

[14] translation rules include rules for both “**elements**” (the nodes) and “**relations**” (the edges). These rules encode a template for representing the semantics in CLIF, and contain both a template for the shape of the CLIF sentence corresponding to the semantics and a parameter which is where the name or identifier of the element will be inserted (*c.f.*, [13] for a more in depth look as to how this is achieved). The third column shows the representation of the concrete semantics of the model fragments in CLIF. In these examples, all the nodes are represented as declarations of boolean variables. For the fragment from Fig. 1, the exclusion constraint is rendered as a sum that must be less than one, meaning that at most one of the boolean variables can be 1, with the convention of representing the boolean values false and true by integer values 0 and 1, respectively. For the fragment from Fig. 2, the goal decomposition is rendered as equality, denoting that either both goals are fulfilled or none are. The final column shows the semantics rendered in the input KRLs for both SWI-Prolog [55] and Minizinc [37].

2.3 Reasoning Tasks and Query Specification

Design principle DP4 was illustrated by the declarative JSON specification of reasoning examples given at the bottom right of Fig. 1 and Fig. 2. The two first attributes of these specifications, operation and solver, are compulsory. In contrast, the third attribute of example 1, *iterate_over*, and the third and fourth of example 2 are optional, as they are specific

to respective tasks of global FM defect search and context-aware search for optimal DSPL configuration.

At the very bottom of Fig. 2 are the textual constraints in CLIF that complement the graphical model and permit the construction of the optimization problem. These constraints define a set of equations related to the model elements and equate the optimization variable to a complex expression. In this case, our target variable is **ObjValue** which is defined as a weighted sum² (shown below) over the satisfaction level of claims, soft influences and soft goals as defined in section 2.1.2. Since we seek to simulate the behaviour of a context-aware Dynamic SPL, we also provide the context values to influence the search for an optimal solution by setting the context variables (blue boxes on the right) to one of their possible values.

```
(and (int ObjValue)
  (= ObjValue ((1000*TotC)+(100*TotSD)+TotNFR)))
```

The operations allowed, beyond those shown for Figs 1 and 2, are finding a specific solution (denoted as *solve*), finding a fixed number of solutions (denoted as *nsolve* with an additional attribute “**operation_n**” that is the number of solutions to be enumerated, though there may be less), or even performing a “*dry-run*” (denoted as *get_model*) where

²The weighted sum defining our optimization target is a faithful translation of the original illustrative optimization function defined in the original paper by Sawyer et al. [42].

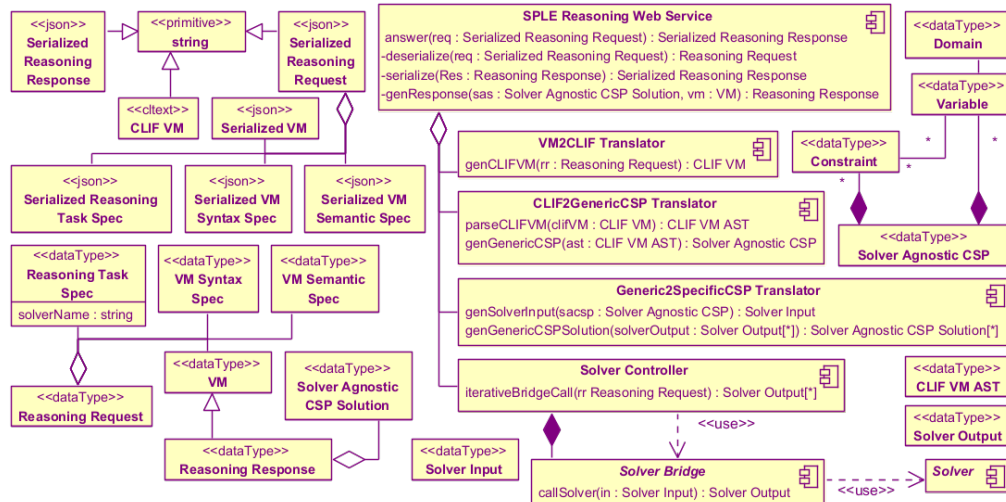


Figure 4. High-level components and data types of the PLEIADES architecture

only the semantics are calculated for the model, which is particularly useful when developing a language for variability modeling. The operations that can be expressed with the current iteration of the query language are essentially those offered by the underlying solvers, that is, checking satisfiability, enumerating solutions or ordering solutions w.r.t. to an optimization criteria. Iterative operations using the “**iterate_over**” are achieved through the architecture’s ability to control the execution of the underlying solvers.

3 Overview of the Architecture

In this section we detail the architecture of PLEIADES. The variability model verification and configuration reasoning functionalities are provided as web services that are accessible through a REST API endpoint. With this approach, multiple variability model editor clients can send HTTP requests for variability model verification or SPL configuration tasks and receive as response the result of the automated reasoning performed by the constraint solvers hosted by the server. This choice of a web service architecture allows fully separating the concern of editing a model from the concern of reasoning on it. It also insures full decoupling of the implementation platforms respectively used for (a) model editing, (b) translation of variability model reasoning requests into constraint solver inputs and back from constraint solver outputs into reasoning responses and (c) constraint solving. Additionally, it provides an installation-free usage of the reasoning services.

3.1 High-Level Component Structure and Control Flow of the Architecture

The UML diagram in Fig. 4 shows the structural model of the PLEIADES architecture for SPLE reasoning services. It

shows its main components, the signatures of the operations that they implement, together with the data types of each signature parameter. To avoid implying the adoption of class-based object-orientation to implement our architecture, this diagram only contains the very general concepts of component (*a.k.a.*, service, module or package in different implementation platforms) and data type, rather than classes.

The SPLE Reasoning Web Service is the top-level component of the architecture. It answers web requests that contain as their payload a serialized JSON representation of the reasoning request to execute on the server. As shown at the top-left of Fig. 4, this Serialized Reasoning Request includes four top-level properties: (a) the variability model VM to analyze, (b) the specification of the reasoning task to carry out for the analysis, (c) the abstract syntax specification of the VML used for the variability model, and (d) the semantic specification of this VML in the form of a mapping between VML model elements and logical sentences in CLIF. It is the fact that the reasoning request comes with declarative specifications of the abstract syntax and formal semantics of the VML that allows reasoning services following this PLEIADES architecture to be VML agnostic. And it is the fact that the VML’s formal semantics are expressed in CLIF, a standard for logical inference engine interoperability, whose expressiveness subsumes that of all constraint solvers widely used for SPLE automated reasoning, that allows variability model reasoning services following the PLEIADES architecture to be solver agnostic.

In addition to its public answer operation endpoint, the SPLE Reasoning Web Service also encapsulates three private operations: one to deserialize the reasoning request JSON string into an instance of the Reasoning Request data type, one to generate the instance of the Reasoning

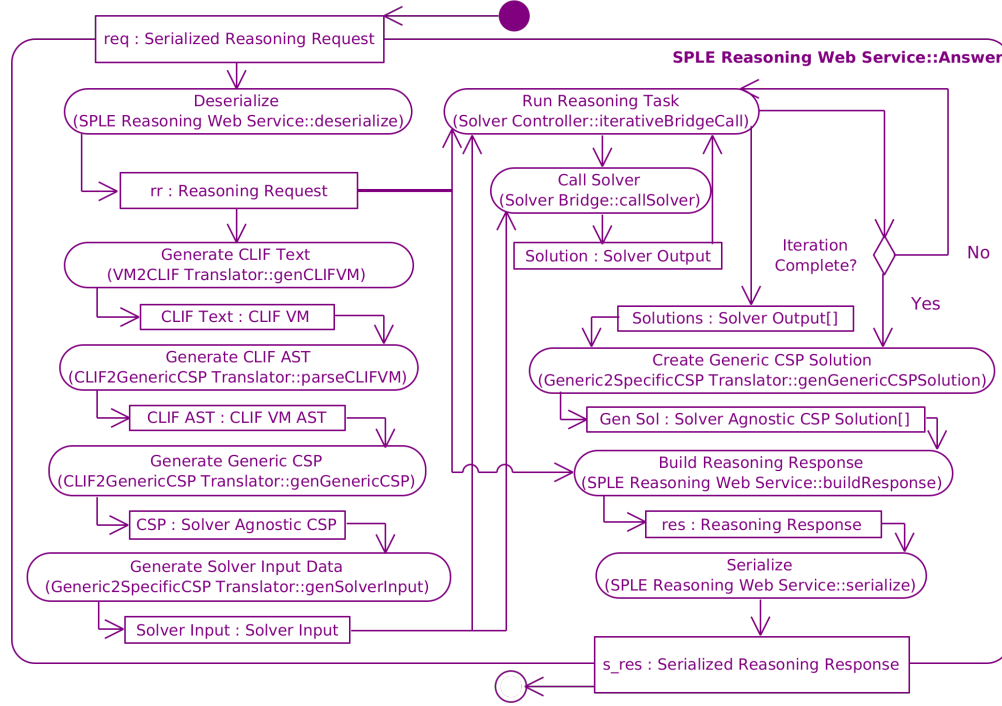


Figure 5. Activity diagram detailing the operation of our architecture.

Response data type from the reasoning task result in a solver agnostic format, and one to serialize this Reasoning Response data type instance into a JSON string.

This top-level component also has access to the operations of its nested components. Let us now review them in top-to-bottom order as depicted in Fig. 4. The first is the VM2CLIF Translator which translates the VM data structure contained in the Reasoning Request data structure into a CLIF text by simultaneously leveraging the model’s syntax and semantic specifications that accompany the variability model in the Reasoning Request.

The second SPLE Reasoning Web Service sub-component is the CLIF2GenericCSP Translator which translates the CLIF text representing the semantics of the VM into a semantically equivalent solver-agnostic CSP representation of the VM. As shown at the top right of Fig. 4, this representation is simply a set of constraints relating variables, each one associated with its domain of possible values. This translation occurs in two steps. The first is to parse the CLIF text into an *Abstract Syntactic Tree (AST)* and the second is to generate, from this AST, the constraints, variables and domain data types of the solver-agnostic CSP.

The third SPLE Reasoning Web Service sub-component is the Generic2SpecificCSP Translator which translates this solver-agnostic representation of a CSP into one accepted as input by the solver chosen in the Reasoning Task Specification. This component is also used to translate the solver output back in the other direction into a Solver

Agnostic CSP Solution. Note that thanks to their purely declarative, relational and intentional nature, a CSP and its solution can be uniformly represented by the same three data types: Constraint, Variable and Domain. A CSP solution is merely a CSP with less constraints and more variables with domains reduced to a singleton [17]. As shown at the bottom-left of Fig. 4, the Reasoning Response data type associates the value of those singleton domains with the model element represented by the CSP variable whose domain has been reduced to a single value. That value is injected in the Reasoning Response that is then serialized and sent back to the client model editor.

The fourth and last SPLE Reasoning Web Service sub-component is the Solver Controller, the most complex component of the architecture. Understanding its role requires realizing that many variability model verification tasks cannot be directly executed through a single call to a constraint solver. They rather require meta-programming an iteration over the model’s elements in which, at each step, the initial CSP representation of the variability model is modified by adding or removing some constraints and the solver is called on this modified CSP [41]. Depending on the result returned by the solver at each step, the iteration proceeds to the next step or stops. During the iteration, the results of each solver call are accumulated in a set. The IterativeBridgeCall operation of the Solver Controller implements this iteration. It returns a set of Solver Output data type instances.

As an example of this need for iteration, let us consider the search for so-called *dead features* in a feature model: *i.e.*, those that cannot be selected in any valid configuration. The declarative specification of this task as a JSON object is shown beside the model of Fig. 1. It defines a strategy for the Solver Controller to find dead features in the variability model [6]. It consists of iterating over the variability model features, and, for each of them, adding to the CSP, representing the VM, the new constraint that this feature is selected (`"with_value": 1`) and checking whether this makes the CSP unsatisfiable. To be able to repeatedly call the requested solver, the Solver Controller encapsulates a set of bridges, one for each solver to be integrated in the SPLE Reasoning Web Service. The role of each bridge is to start a new instance of the solver, call its API to pass the CSP to solve at each iteration of the `IterativeBridgeCall` operation and pass the result of each such call back to the Solver Controller.

The control flow of the top-level answer operation of the root SPLE Reasoning Web Service component is modeled by the UML activity diagram of Fig. 5. It shows the input and output parameter of the answer operation of this component and in what order it internally calls the other operations of the architecture. It also shows the data structures that these calls exchange as arguments. At the highest level, this activity is divided in three main phases. The first is the pipeline on the left side of Fig. 5 that progressively transforms the Serialized Reasoning Request received from the web client SPLE GUI into an input for the solver (Solver Input) chosen as a property of the Serialized Reasoning Request. The second is the loop of calls to the solver made by the Solver Controller shown in the top right quadrant of Fig. 5. The third and last phase is another pipeline, shown in the bottom right quadrant of Fig. 5, that translates the solver outputs accumulated during the iteration into a Serialized Reasoning Response to send back to the web client editor.

3.2 Revisiting Requirements and Design Principles

In light of the requirements and design principles we outlined in the introduction, it is important to highlight how our architecture reflects these principles and meets these requirements. Design principles **DP1** and **DP2** are reflected first in the clear separation of the data types shown in Fig. 4, where every aspect of the DSVML has its own dedicated data type and dedicated component operation. They are also reflected in the envisioned operation of the architecture shown in Fig. 5, where they are taken as user-specified specifications (as part of the Reasoning Request) instead of being hard-coded into the architecture. The same is also true for Principle **DP3** when it comes to the specification of the semantics. As for the widely used exchange format, JSON [14] is well supported and certainly widely used. In addition, concerning the encoding into a formal KRL, this is covered by the

translation into CLIF based on the semantics specification, allowing virtually any DSVML with first order logic semantics to be translated into it. Considering that our architecture follows these principles, it also covers requirements **REQ1** and **REQ3**.

Design Principle **DP4** is handled in much the same way as **DP2**. JSON is used again to allow the user to tailor the specification for their reasoning task with no need to manually hard-code it in the system, as described in Section 2.3. This in turn allows fulfilling requirement **REQ2**. Principle **DP5** is, firstly, contingent on the earlier principles being met, and, second, is covered by the variability model transformation outlined in Fig. 5 and section 2.2. Indeed, a significant portion of the architecture is dedicated to enabling this translation, as evidenced by both the component solely dedicated to it (VM2CLIF Translator shown in Fig. 4) and the subsequent components that are dedicated to the treatment of the generated CLIF to arrive at the solvers' input. To be clear, what this achieves is to completely decouple integrating DSVMLs and IEs into the architecture by linking both through CLIF and meeting requirement **REQ3**.

Design principle **DP6** stands alone as it is not tied to specific functional requirements but rather aims to ease the analysis of variability models, as well as the definition and debugging of new DSVML by different stakeholders without requiring them to dive into the details of a specific solver.

4 Prototype

To validate our architecture and its feasibility for constructing a reasoning platform that covers all of the requirements outlined above, we have implemented a prototype in Python based on this architecture. Fig. 6 shows how the prototype specializes the components outlined in the architecture. This prototype's internal component structure closely reflects the architecture. A key element of our prototype is the integration of multiple solvers from different solver families: SWI Prolog [55] and MiniZinc [37] (which allows us to target multiple Constraint and Integer Programming Solver libraries). To implement the pivot language, we wrote our own "CLIF Parser" with the TextX library [18] based on the grammar described in the latest version of the ISO standard [27].

It is worth highlighting, regarding our implementation, how the `CLIF2GenerciCSP` component was implemented. Fundamentally, after the CLIF semantics have been generated, these are parsed into an AST. Though the CLIF standard has no "distinguished" predicates besides equality, we have added into the grammar the ability to recognize several predicates which are particularly relevant for our purposes. In particular, we distinguish predicates that "type" the variables in the semantics, for example `int`, as in `(int ObjValue)` shown in Fig. 2, that declares `ObjValue` to be an integer variable. We also distinguish other operators beyond equality

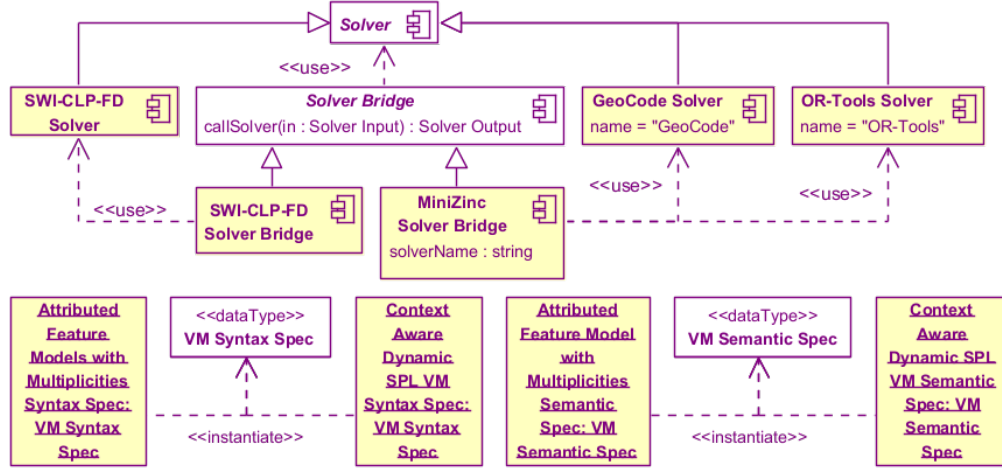


Figure 6. Current PLEIADES prototype instantiating the generic architecture

such as the basic arithmetic operators and arithmetic comparison predicates (such as the different inequality operators). These extensions to the CLIF grammar provide us with a fully typed AST that then maps directly to the constructs supported by the solvers. In this way, we construct the Generic representation of the constraint problem, with variables (and their domains) and constraints. This “GenericCSP” is then used to generate the concrete solver input in its particular input KRL to then begin the reasoning process on the solver.

We demonstrate our approach within an open-source tool called VariaMos [51] for all visualization concerns, and for the specification of the concrete visual syntax, the abstract syntax and the formal semantics in JSON format. The examples shown in Figs. 1 and 2, and the results shown, were all done within this tool. We have modified the tool so that the reasoning requests can be made directly from it by specifying the URL of the reasoning web service. We have made the prototype available on GitHub as an open source tool³.

5 Related Work

5.1 State-of-the-art Variability Modeling Tools

In this section, we examine state-of-the-art tools with aims similar to ours that are currently available, mature and well documented as reported by a recent survey [26], with the addition of some others that we consider particularly relevant. The tools we’ve identified are: Feature IDE [47], its related project FAMILIAR [1], FlamaPy [22], the COFFEE Framework [52], SPLOT [36], Glencoe [44], ClaferTools [3], Kernel Haven [29], pure::variants [9], and Gears [31]. Though most of these tools also implement modeling, visualization or code generation, we focus on their verification functionality.

In Table 1 we present an analysis of the characteristics of these tools. We divide our analysis into five main dimensions:

- First, we examine the different approaches from a purely architectural perspective. Of interest are the following characteristics: whether this architecture is well documented with standard modeling languages like the UML [45]; the detail and granularity of the models as a function of the quantity of elements they contain; whether the models are structural, behavioral, or both; and, finally, what general architectural pattern is applied for their tool. We also analyze the interaction with model editors as part of their architectural patterns related to requirement REQ4.
- Next, we analyze the support their respective approaches have for the diversity of variability modeling languages. This includes the treatment of integers, and first-order constraints among the elements of a variability model. Two other key factors are whether the languages support concepts of context-aware dynamic SPLs and whether the semantics of the variability models is expressed in a human-interpretable, declarative language, or rather buried inside the tool’s imperative code details. This is related to requirement REQ1 and the types of languages supported by competing tools.
- We analyze in a similar way the support for different solver features among the tools. We must, nevertheless, highlight the fact that our analysis of these features relates to the capabilities of the solvers themselves and not necessarily of their use within the tool. For instance, though FeatureIDE allows edition of variability models with numeric attributes, it cannot verify constraints on them due to the limitation of its modified SAT4J [8] solver that can only reason on Boolean variables. The characteristics we examine are: (a) treatment of integer domains, (b) treatment of first order constraints with universally quantified variables, (c) whether optimization can be run in addition to pure solving; (d) whether there are meta-programming mechanisms to control the behavior of the solvers; and (e)

³https://github.com/ccr185/semantic_translator. The user manual is in the repository’s wiki page: https://github.com/ccr185/semantic_translator/wiki

Table 1. Characteristics of State of the Art Tools

		Tools										
		(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)
Architecture	Standard Modeling Language	P	N	N	P	P	N	N	N	N	N	Y
	Architecture Complexity	36	14	9	36	27	?	4	32	?	?	50
	Structural(a) & Behavioral Models(b)	(a)	(a)	N	(a)	Y	N	N	N	N	N	Y
	Architectural Pattern	A	B	C	D	E	F	F	G	H	?	I
Variability Model Support	Integer Attributes	Y	N	Y	Y	N	N	N	Y	N	?	Y
	First Order Constraints	N	N	Y	P	N	N	Y	N	N	N	Y
	Context Aware Dynamic SPL	P	P	N	N	N	N	N	Y	N	N	Y
	Human Interpretable Variability Model	P	N	N	N	N	N	N	P	N	N	Y
	Semantic Language											
Solver Support	Integer Domains	P	P	Y	Y	N	N	N	Y	N	?	Y
	First Order Constraints	N	N	N	N	N	N	N	Y	N	?	Y
	Optimization	Y	?	Y	Y	Y	Y	?	Y	N	?	Y
	Meta-Programming	Y	Y	Y	P	Y	?	Y	?	Y	?	Y
	Incremental Solutions	Y	Y	Y	Y	Y	Y	Y	Y	Y	?	Y
Reasoning Task Spec Support	Declarative Verification Task	N	N	P	N	Y	N	N	N	N	?	Y
	Specification Language											
	Declarative Configuration Task	N	N	P	N	Y	N	N	N	N	?	Y
	Specification Language											
Standards	Leverages Existing Standards	N	N	N	P	N	N	N	N	N	?	Y

Tools: (1) FeatureIDE; (2) FlamaPy; (3) FAMILIAR; (4) COFFEE; (5) Kernel Haven; (6) SPLOT; (7) Glencoe; (8) ClaferTools; (9) Pure::Variants; (10) Gears; (11) Our Approach
Architectural Patterns: ^A Java API + Eclipse Plugin; ^B CLI + Python API (Plugin-Based); ^C CLI + Java API + Eclipse Plugin; ^D 3-Layered Web Service; ^E 3 Layered Monolith with Plugin System; ^F Client/Server (With no further details); ^G Haskell API/Web Server + Web Client; ^H IDE Plugins (With no further details); ^I Multi Layer Python API/Web Server + Pivot Language

Legend: Y – Yes; N – No; P – Partial Fulfillment; ? – Unknown/Unclear;

whether the solvers can present series of solutions without restarting the search space exploration from scratch. This is tied to our requirement **REQ3** and the backend support offered by the different tools.

- We also analyze the support for complex but nonetheless declarative specifications as a parameter of the reasoning tasks to be performed. This is related to requirement **REQ2**.
- Finally, we analyze the use of internationally recognized standards as part of the proposals as part of our analysis tied to our design principle **DP6**.

From this one can conclude that all of the approaches cited follow an approach of transforming the variability model into an input for a constraint solver. PLEIADES aims to generalize this idea to provide a truly generic architecture to perform these tasks and, provide details of how these transformations can be performed.

5.2 Domain Specific Languages in SPLE

A key feature of our architecture is its agnosticity w.r.t. DSVMLs; while one can make use of simple and extended

FMs, the use of DSVML as alternative to FMs is well attested in the literature. Their fit as alternatives to feature models is highlighted in [11], with some of the first proposals aiming to integrate textual DSVMLs directly into artefacts [5]. This was further explored in a survey article [53] where all the possible combinations of DSVMLs and FMs were explored, concluding that they may coexist to different degrees in a project, or that one may outright replace the FM with a DSVML model. Though their focus was primarily on textual-based DSVMLs, other authors have found it important to construct graphical DSVML based proposals. In this vein, the work of Demuth et al. [19] is particularly important. They proposed a tool that should allow one to construct DSVMLs and their meta-models all as part of a single graphical modeling tool. This allows the generation of artefacts from the models since all of their elements are ultimately mapped to a restricted subset of the UML. The analysis capabilities of the resulting tool were, however, quite limited.

There have been other approaches to use and integrate graphical DSVMLs for variability modeling, such as [48], presenting a case study of a DSVML for creating variants

of robot software. Several approaches hinged on creating custom UML profiles [12, 56] to aid in the automated generation of websites. A large set of industrial case studies where graphical DSVMLs for managing variability were employed has also been presented [50], highlighting the use of a commercial DSL development tool [49]. This tool has also been used for creating a graphical DSVML for an automatic performance test generation approach based on a product line approach [7].

These approaches, however, are all held back by one fundamental limitation of not supporting an explicit, declarative, parametric formal semantics for the DSVML. The semantics is only operational and imperatively hard-coded inside the tool. They prevent tailoring the underlying reasoning to the particular characteristics of the DSVML, as we have done with Sawyer et al.'s language [42], which, though originally a graphical DSVML for DPSL without any code generation capabilities, we have brought directly into our approach as a key demonstrator of its capabilities.

6 Evaluation and Discussion

6.1 Requirements Coverage

Given the novelty of our approach, and the mechanism employed, we begin first with an analysis of the coverage of our requirements by our approach. This analysis is informed both by our analysis of the architecture in Section 3 and the lessons learned from the prototype implementation.

REQ1 *Support low-cost extension of existing DSVMLs and addition of new DSVMLs:* Our approach is specifically designed to use a declarative specification of the modeling languages, and, within reason, any VML could be handled by our approach. We illustrate this variety by implementing and testing our implementation with two languages that differ greatly syntactically and semantically.

REQ2 *Support low-cost addition of new automated reasoning tasks to run on the variability model:* Our architecture is designed to handle precisely this, with a view on exposing a large variety of operations to the user so that as new DSVMLs are created, it is straightforward to create the corresponding reasoning tasks. The PLEIADES prototype reflects this and has been demonstrated to be robust enough to support very different reasoning tasks through the same mechanism.

REQ3 *The architecture must be agnostic w.r.t. the logical KRL and IE paradigm used for automated reasoning on the variability model, supporting low-cost addition of interoperability with solvers from different paradigms:* Given that multi-solver support has been a key design goal for the architecture, this is covered by the N-to-one-to-M translation approach through

CLIF, so that there is no hard-coded bias towards any IE and allowing easy integration of new IEs.

REQ4 *The architecture must be agnostic w.r.t. the DSVML editor tool, accepting the variability model as input data exportable from multiple popular editors:* Our architecture seeks to be as independent as possible from any particular modeling tool by relying on the declarative specifications of the languages to perform all processing instead of relying on a particular set of technologies. Our prototype implementation has been developed and tested initially on the VariaMos [51] modeling framework which has been modified to allow for the declarative specifications needed for the prototype. Further integration with other variability model editors will require determining how to add this functionality to them.

6.2 Limitations of the Architecture and Prototype

The architecture has one important omission that is worth discussing. It relies on the assumption that the concrete syntax of the input DSVML has a diagrammatic component possibly extended by complementary textual expressions. This has an important effect on the abstract syntax and therefore interpretation of the models: it is not yet clear how to deal with purely textual VMLs that do not use a standard interchange format like JSON. The reason for this is simple. Textual VMLs with their own grammar and structure would need their parser to be integrated into the architecture. In addition to this, a more abstract, common representation of both the hybrid and textual models would need to be constructed such that it would serve the base for constructing the semantics. Nevertheless, this seems possible with some modifications to the architecture, with, in particular, a more capable input management system.

The prototype we present is limited in some key ways. We have developed the prototype utilizing the VariaMos modeling environment, which has the underlying assumption that all the graphical models are directed graphs where the nodes can be typed and have attributes. In addition, we have only implemented two input DSVMLs and 2 output solvers so far.

6.3 Threats to Validity

In addition to the limitations above, our work is subject to internal and external threats to validity. In terms of **Internal Validity** our primary concern is how feasible our architecture is. To counter this threat, we have created a prototype that covers as closely as possible the proposed architecture. The converse of this is the correctness of the prototype, that we've endeavored to test extensively. We have sought, therefore, to demonstrate its capacity to provide the functionality envisioned in the architecture. Another further threat is tied to the correctness of our architectural design. We have utilized a standard and well-understood modeling language (UML) for its definition and have sought to collect feedback

from colleagues and users of the tool, both from a developer’s perspective and from an end-user perspective.

Now, in terms of **external validity**, we recognize that we cannot provide guarantees on the exhaustiveness of the tools surveyed, more recent efforts that the authors are unaware of might have been proposed in the meantime. Nevertheless, we have sought to base our overview on a recent and well sourced survey. An additional threat concerns our ability to manage purely textual languages within the architecture, as mentioned in the previous section. To combat this we have designed a flexible and modular architecture capable of being modified in this direction.

Finally, we also recognize a threat to **construct validity** due to the informal measurement of the notion of cost in our requirements. A quantitative measure of this is a necessary next step that we intend to undertake.

6.4 Future Work

To overcome our limitations, we are currently working in several directions. The first is supporting purely textual VMLs, beginning with the Universal Variability Language (UVL) [46]; The second is enlarging the set of reasoning tasks supported, including having a more fine grained control over the variables used for iteration. The third is supporting more output KRLs and solvers, beginning with the Z3 SMT solver [15]. The fourth is to add an incremental solving component in order to make interactions far more efficient. This will all imply minor refinements of the architecture, though we believe the core will remain unchanged.

Another future work we envision is to treat the architecture itself as a software product line, such that we could create fully configurable distributions for particular needs, *i.e.*, include commercial solvers if the licenses are present, and, more importantly, give a larger freedom on the reasoning task formulations and solver’s parameter tuning. We see value in this direction because the decision system for DSPLs can be modeled and then exported for whatever solver is supported, or even combinations of solvers, which could enable portfolio [2] constraint solving using multiple engines.

7 Conclusion

In this paper, we have presented an innovative software architecture to provide automated reasoning for SPLE. As we have seen in section 5, its main contribution is to be the first proposal focused on being agnostic with respect to both the VML used to model variability, and the constraint solving paradigm used to implement the reasoning. Such agnosticism allows fully decoupling the variability model and SPL asset editing tools from the automated reasoning tools. We have put forward four requirements for an architecture to achieve such agnosticism. We then have presented fairly detailed structural and behavioral models for the architecture and discussed why it satisfies those requirements. Its key

ideas are (a) to avoid combinatorial explosion of components by using a pivot standard semantic language and (b) pass as parameters declarative specifications of (i) the abstract syntax and semantics of the VML used by the variability model to analyze and (ii) the analysis task itself.

We have validated this architecture by quickly refining and instantiating these models allowing for four distinct SPLE reasoning pipelines already available in our PLEIADES framework prototype. Each one combines any of two VML languages, respectively feature models and a context-aware dynamic SPL VML, with solvers from two logical reasoning paradigms, respectively CLP, through libraries of SWI-Prolog, and CSP, through the CSP solver integration layer MiniZinc. We hope that our demonstration of the feasibility of VML and solver agnostic SPLE reasoning services together with the simplicity of instantiating the PLEIADES architecture we propose into a working implementation, reusing various VML and solvers, will foster more reuse of third party SPL model editors and third party solvers in the SPLE community.

References

- [1] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert B. France. 2013. FAMILIAR: A Domain-Specific Language for Large Scale Management of Feature Models. *Science of Computer Programming* 78, 6 (June 2013), 657–681. <https://doi.org/10.1016/j.scico.2012.12.004>
- [2] Roberto Amadini, Maurizio Gabbriellini, and Jacopo Mauro. 2015. SUNNY-CP: a sequential CP portfolio solver. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*. 1861–1867.
- [3] Michał Antkiewicz, Kacper Bąk, Alexandr Murashkin, Rafael Olachea, Jia Hui (Jimmy) Liang, and Krzysztof Czarnecki. 2013. Clafer Tools for Product Line Engineering. In *Proceedings of the 17th International Software Product Line Conference Co-Located Workshops*. ACM, Tokyo Japan, 130–135. <https://doi.org/10.1145/2499777.2499779>
- [4] Franz Baader, Ian Horrocks, Carsten Lutz, and Uli Sattler. 2017. *Introduction to description logic*. Cambridge University Press.
- [5] Don Batory, Clay Johnson, Bob MacDonald, and Dale Von Heeder. 2000. Achieving Extensibility through Product-Lines and Domain-Specific Languages: A Case Study. In *Software Reuse: Advances in Software Reusability*, Gerhard Goos, Juris Hartmanis, Jan Van Leeuwen, and William B. Frakes (Eds.). Vol. 1844. Springer Berlin Heidelberg, Berlin, Heidelberg, 117–136. https://doi.org/10.1007/978-3-540-44995-9_8
- [6] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. 2010. Automated Analysis of Feature Models 20 Years Later: A Literature Review. *Information Systems* 35, 6 (Sept. 2010), 615–636. <https://doi.org/10.1016/j.is.2010.01.001>
- [7] Maicon Bernardino, Avelino F. Zorzo, and Elder M. Rodrigues. 2016. Canopus: A Domain-Specific Language for Modeling Performance Testing. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, Chicago, IL, USA, 157–167. <https://doi.org/10.1109/ICST.2016.13>
- [8] Daniel Le Berre and Anne Parrain. [n. d.]. The Sat4j Library, Release 2.2. ([n. d.]).
- [9] Danilo Beuche. 2011. Modeling and Building Software Product Lines with Pure::Variants. In *Proceedings of the 15th International Software Product Line Conference, Volume 2*. ACM, Munich Germany, 1–1. <https://doi.org/10.1145/2019136.2019190>
- [10] Goetz Botterweck and Andreas Pleuss. 2014. Evolution of Software Product Lines. In *Evolving Software Systems*, Tom Mens, Alexander Serebrenik, and Anthony Cleve (Eds.). Springer Berlin Heidelberg,

- Berlin, Heidelberg, 265–295. https://doi.org/10.1007/978-3-642-45398-4_9
- [11] Barrett R. Bryant, Jeff Gray, and Marjan Mernik. 2010. Domain-Specific Software Engineering. In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*. ACM, Santa Fe New Mexico USA, 65–68. <https://doi.org/10.1145/1882362.1882376>
- [12] Juan José Cadavid, Juan Bernardo Quintero, David Esteban Lopez, Jesus Andrés Hincapié, Antonio Brogi, Araújo João, and Raquel Anaya. 2009. A Domain Specific Language to Generate Web Applications.. In *CibSE*. 139–144.
- [13] Camilo Correa, Raul Mazo, Andres O. Lopez, and Jacques Robin. 2023. A Lightweight Method to Define Solver-Agnostic Semantics of Domain Specific Languages for Software Product Line Variability Models. In *SOFTENG 2023 - The 9th International Conference on Advances and Trends in Software Engineering*. IARIA: International Academy, Research and Industry Association, Venise, Italy.
- [14] Douglas Crockford. 2006. *The Application/Json Media Type for JavaScript Object Notation (JSON)*. Request for Comments RFC 4627. Internet Engineering Task Force. <https://doi.org/10.17487/RFC4627>
- [15] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.
- [16] Leonardo De Moura and Nikolaj Bjørner. 2011. Satisfiability modulo theories: introduction and applications. *Commun. ACM* 54, 9 (2011), 69–77.
- [17] Rina Dechter and David Cohen. 2003. *Constraint Processing*. Morgan Kaufmann.
- [18] I. Dejanović, R. Vadera, G. Milosavljević, and Ž. Vuković. 2017. TextX: A Python tool for Domain-Specific Languages implementation. *Knowledge-Based Systems* 115 (2017), 1–4. <https://doi.org/10.1016/j.knosys.2016.10.023>
- [19] Andreas Demuth, Roberto E. Lopez-Herrejon, and Alexander Egyed. 2011. Cross-Layer Modeler: A Tool for Flexible Multilevel Modeling with Consistency Checking. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*. ACM, Szeged Hungary, 452–455. <https://doi.org/10.1145/2025113.2025189>
- [20] Douglas C Engelbart. 1962. Augmenting Human Intellect: A Conceptual Framework. *Menlo Park, CA* 21 (1962).
- [21] Thom Frühwirth and Slim Abdennadher. 2003. *Essentials of constraint programming*. Springer Science & Business Media.
- [22] José A Galindo and David Benavides. 2020. A Python Framework for the Automated Analysis of Feature Models: A First Step to Integrate Community Efforts. In *Proceedings of the 24th Acm International Systems and Software Product Line Conference-Volume b*. 52–55.
- [23] Michael R Genesereth and Richard E Fikes. 1992. Knowledge Interchange Format-Version 3.0: Reference Manual. (1992).
- [24] Carla P Gomes, Henry Kautz, Ashish Sabharwal, and Bart Selman. 2008. Satisfiability Solvers. *Foundations of Artificial Intelligence* 3 (2008), 89–134.
- [25] Svein Hallsteinsen, Mike Hinchey, Sooyong Park, and Klaus Schmid. 2008. Dynamic Software Product Lines. *Computer* 41, 4 (April 2008), 93–95. <https://doi.org/10.1109/MC.2008.123>
- [26] José Miguel Horcas, Mónica Pinto, and Lidia Fuentes. 2023. Empirical Analysis of the Tool Support for Software Product Lines. *Software and Systems Modeling* 22, 1 (Feb. 2023), 377–414. <https://doi.org/10.1007/s10270-022-01011-2>
- [27] International Organization for Standardization. 2018. *Information Technology – Common Logic (CL) – A Framework for a Family of Logic-Based Languages – ISO/IEC 24707:2018*. Technical Report. International Organization for Standardization, Geneva, CH. 70 pages.
- [28] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. S. Peterson. 1990. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Technical Report. Defense Technical Information Center, Fort Belvoir, VA. <https://doi.org/10.21236/ADA235785>
- [29] Christian Kröher, Sascha El-Sharkawy, and Klaus Schmid. 2018. KernelHaven – An Experimentation Workbench for Analyzing Software Product Lines. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings*. 73–76. <https://doi.org/10.1145/3183440.3183480> arXiv:2110.05858 [cs]
- [30] CharlesW Krueger. 2001. Easing the transition to software mass customization. In *International Workshop on Software Product-Family Engineering*. Springer, 282–293.
- [31] Charles Krueger and Paul Clements. 2018. Feature-Based Systems and Software Product Line Engineering with Gears from BigLever. In *Proceedings of the 22nd International Systems and Software Product Line Conference-Volume 2*. 1–4.
- [32] Philippe Lalanda, Julie A McCann, and Ada Diaconescu. 2013. *Autonomic computing: principles, design and implementation*. Springer Science & Business Media.
- [33] Shih-Hsi Liu. 2010. *Design Space Exploration for Distributed Real-Time*. VDM Verlag Dr. Müller.
- [34] Mike Mannion. 2002. Using First-Order Logic for Product Line Model Validation. In *Software Product Lines: Second International Conference, SPLC 2 San Diego, CA, USA, August 19–22, 2002 Proceedings*. Springer, 176–187.
- [35] John McCarthy. 1960. Recursive functions of symbolic expressions and their computation by machine, part I. *Commun. ACM* 3, 4 (1960), 184–195.
- [36] Marcilio Mendonca, Moises Branco, and Donald Cowan. 2009. S.P.L.O.T.: Software Product Lines Online Tools. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*. ACM, Orlando Florida USA, 761–762. <https://doi.org/10.1145/1639950.1640002>
- [37] Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. 2007. MiniZinc: Towards a Standard CP Modelling Language. In *Principles and Practice of Constraint Programming – CP 2007*, Christian Bessière (Ed.). Vol. 4741. Springer Berlin Heidelberg, Berlin, Heidelberg, 529–543. https://doi.org/10.1007/978-3-540-74970-7_38
- [38] Mahdi Noorian, Alireza Ensar, Ebrahim Bagheri, Harold Boley, and Yevgen Biletskiy. 2011. Feature Model Debugging Based on Description Logic Reasoning.. In *Proceedings of the 17th International Conference on Distributed Multimedia Systems, DMS 2011, October 18–20, 2011, Convitto della Calza, Florence, Italy*, Vol. 11. Citeseer, 158–164.
- [39] Object Management Group. 2014. *Object Constraint Language (OCL), Version 2.4*. Technical Report. Object Management Group.
- [40] Klaus Pohl, Günter Böckle, and Frank van der Linden. 2005. *Software Product Line Engineering: Foundations, Principles, and Techniques* (1st ed ed.). Springer, New York, NY.
- [41] Camille Salinesi and Ral Mazo. 2012. Defects in Product Line Models and How to Identify Them. In *Software Product Line - Advanced Topic*, Abdelrahman Elfaki (Ed.). InTech. <https://doi.org/10.5772/35662>
- [42] Pete Sawyer, Raul Mazo, Daniel Diaz, Camille Salinesi, and Danny Hughes. 2012. Using Constraint Programming to Manage Configurations in Self-Adaptive Systems. *Computer* 45, 10 (2012), 56–63.
- [43] DC Schmidt. 2006. Model-Driven Engineering. *Computer* 39, 2 (2006), 25–31.
- [44] Anna Schmitt, Christian Bettinger, and Georg Rock. 2018. Glencoe—a Tool for Specification, Visualization and Formal Analysis of Product Lines. In *Transdisciplinary Engineering Methods for Social Innovation of Industry 4.0*. IOS Press, 665–673.
- [45] Steve Cook, Conrad Bock, Pete Rivett, Tom Rutt, Ed Seidewitz, Bran Selic, and Doug Tolbert. 2017. *Unified Modeling Language (UML), Version 2.5.1*. Technical Report. Object Management Group.

- [46] Chico Sundermann, Kevin Feichtinger, Dominik Engelhardt, Rick Rabiser, and Thomas Thüm. 2021. Yet Another Textual Variability Language? A Community Effort towards a Unified Language. In *Proceedings of the 25th ACM International Systems and Software Product Line Conference - Volume A* (Leicester, United Kingdom) (SPLC '21). Association for Computing Machinery, New York, NY, USA, 136–147. <https://doi.org/10.1145/3461001.3471145>
- [47] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. 2014. FeatureIDE: An Extensible Framework for Feature-Oriented Software Development. *Science of Computer Programming* 79 (Jan. 2014), 70–85. <https://doi.org/10.1016/j.scico.2012.06.002>
- [48] Susumu Tokumoto. 2010. Product Line Development Using Multiple Domain Specific Languages in Embedded Systems. (2010).
- [49] Juha-Pekka Tolvanen and Steven Kelly. 2018. Describing Variability with Domain-Specific Languages and Models. In *Proceedings of the 22nd International Systems and Software Product Line Conference - Volume 1*. ACM, Gothenburg Sweden, 300–300. <https://doi.org/10.1145/3233027.3233059>
- [50] Juha-Pekka Tolvanen and Steven Kelly. 2019. How Domain-Specific Modeling Languages Address Variability in Product Line Development: Investigation of 23 Cases. In *Proceedings of the 23rd International Systems and Software Product Line Conference - Volume A* (Paris, France) (SPLC '19). Association for Computing Machinery, New York, NY, USA, 155–163. <https://doi.org/10.1145/3336294.3336316>
- [51] VariaMos Team. 2023. VariaMos Framework. <https://variarnos.com/>. Accessed: 2023-03-27.
- [52] Angela Villota. 2022. *Coffee : A Framework Supporting Expressive Variability Modeling and Flexible Automated Analysis*. Ph. D. Dissertation. Université Panthéon-Sorbonne - Paris I.
- [53] Markus Voelter and Eelco Visser. 2011. Product Line Engineering Using Domain-Specific Languages. In *2011 15th International Software Product Line Conference*. IEEE, Munich, Germany, 70–79. <https://doi.org/10.1109/SPLC.2011.25>
- [54] Danny Weyns. 2020. *An introduction to self-adaptive systems: A contemporary software engineering perspective*. John Wiley & Sons.
- [55] Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager. 2010. SWI-Prolog. arXiv:1011.5332 [cs]
- [56] Tewfik Ziadi, Loïc Hélouët, and Jean-Marc Jézéquel. 2004. Towards a UML Profile for Software Product Lines. In *Software Product-Family Engineering*, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Dough Tygar, Moshe Y. Vardi, Gerhard Weikum, and Frank J. van der Linden (Eds.). Vol. 3014. Springer Berlin Heidelberg, Berlin, Heidelberg, 129–139. https://doi.org/10.1007/978-3-540-24667-1_10

Received 2023-07-14; accepted 2023-09-03

Author Index

Andrzejak, Artur	122	Hatch, William Gallard	86	Polgreen, Elizabeth	42
Bach Poulsen, Casper	14	Hochrainer, Christoph	29	Porncharoenwase, Sorawee	86
Basso, Matteo	1	Hübner, Paul	14	Robin, Jacques	138
Benavides, David	113	Korz, Niklas	122	Sayilir, Ömer Faruk	72
Binder, Walter	1	Krall, Andreas	29	Tian, Zilu	100
Bonetta, Daniele	1	Magalhães, José Wesley de Souza	42	Van Assen, Mart	72
Broman, David	57	Mazo, Raul	138	Watson, Guy	86
Correa Restrepo, Camilo	138	Medeiros, Raul	113	Woodruff, Jackson	42
Darragh, Pierce	86	Ntagengerwa, Manzi Aimé	72	Zaytsev, Vadim	72
Díaz, Oscar	113	O’Boyle, Michael F. P.	42	Zwaan, Aron	14
Eide, Eric	86	Palmkvist, Viktor	57		
Eriksson, Oscar	57				