Universität der Bundeswehr München

Universität der Bundeswehr München
Fakultät für Informatik

Tagungsband des 23. Kolloquium Programmiersprachen und Grundlagen der Programmierung

25–28 September 2025
Feldkirchen-Westerham, Deutschland

Edited by

Prof. Dr. Stefan Brunthaler

Universität der Bundeswehr München, Germany

ISBN: 978-3-98997-006-9
DOI: 10.18726/154170

Volume: 1
Year: 2026

Fakultät für Informatik
Universität der Bundeswehr München

Editors

Prof. Dr. Stefan Brunthaler

Universität der Bundeswehr München, Germany

Publication Information

Title: Tagungsband des 23. Kolloquium Programmiersprachen und Grundlagen der Programmierung
Date: 25–28 September 2025
Location: Feldkirchen-Westerham, Deutschland

ISBN: 978-3-98997-006-9
DOI: 10.18726/154170
Volume: 1

Published by

Fakultät für Informatik
Universität der Bundeswehr München
Werner-Heisenberg-Weg 39
85577 Neubiberg, Germany
https://www.unibw.de/inf

Typesetting

This volume was compiled using LaTeX and the `pdfpages` package.
Individual contributions were formatted using the ACM `acmart` document class.

Vorwort

Das 23. Kolloquium Programmiersprachen und Grundlagen der Programmierung (KPS 2025) findet vom 25. bis zum 28. September 2025 in Feldkirchen-Westerham (Bayern, Deutschland) statt. Die Veranstaltung dient dem zwanglosen Austausch neuer Ideen und Ergebnissen zu den Themen des Entwurfs und der Implementierung von Programmiersprachen einerseits sowie den Grundlagen und den Methodiken des Programmierens andererseits.

Erstmalig 1980 von den Forschungsgruppen der Professoren Friedrich L. Bauer (TU München), Klaus Indermark (RWTH Aachen) und Hans Langmaack (CAU Kiel) abgehalten, findet das KPS alle zwei Jahre an unterschiedlichen Orten Deutschlands und Österreichs statt und bietet ein offenes Forum für alle interessierten deutschsprachigen Wissenschaftler:innen. Insbesondere sollen junge Doktorand:innen ermutigt werden, ihre Arbeiten einem fachkundigen Publikum vorzustellen und mit diesem zu diskutieren.

Vorherige Kolloquien:

2023	Vaals	RWTH Aachen
2021	Kiel	Uni Kiel
2019	Baiersbronn	DHBW Stuttgart
2017	Weimar	Uni Jena
2015	Pörtschach am Wörthersee	TU Wien
2013	Lutherstadt Wittenberg	Uni Halle-Wittenberg
2011	Schloss Raesfeld, Raesfeld	Uni Münster
2009	Maria Taferl	TU Wien
2007	Timmendorfer Strand	Uni Lübeck
2005	Fischbachau	LMU München
2004	Freiburg-Munzingen	Uni Freiburg
2001	Rurberg in der Eifel	RWTH Aachen
1999	Kirchhundem-Heinsberg	FernUni Hagen
1997	Avendorf auf Fehmarn	Uni Kiel
1995	Alt-Reichenau	Uni Passau
1993	Garmisch-Partenkirchen	UniBw München
1992	Rothenberge bei Steinfurt	Uni Münster
1989	Hirschegg	Uni Augsburg
1987	Midlum auf Föhr	Uni Kiel
1985	Passau	Uni Passau
1982	Altenahr	RWTH Aachen
1980	Tannenfelde im Naturpark Aukrug	Uni Kiel

Neubiberg, Jänner 2026

Stefan Brunthaler

Inhaltsverzeichniss

LOOL: Low-Overhead, Optimization-Log-Guided Compiler Fuzzing

FELIX BERLAKOVICH, Universität der Bundeswehr, Germany
FLORIAN SCHWARCZ, Johannes Kepler Universität, Austria
GERGÖ BARANY, Oracle Labs, Austria
HANSPETER MÖSSENBÖCK, Johannes Kepler Universität, Austria
STEFAN BRUNTHALER, Universität der Bundeswehr, Germany

Compiler fuzzing with randomly generated input programs is a powerful technique for finding compiler crashes and miscompilation bugs. Existing fuzzers are either unguided or guided by low-level coverage metrics such as code coverage, which incur significant overhead and lack domain-specific context. We present Lool, an approach for fuzzing compilers with low overhead, guided by optimization log information produced by the compiler itself. The optimization log tracks program transformations at the method level, providing context-aware coverage with lower runtime overhead than off-the-shelf code coverage tools. We integrate Lool with a fuzzer for the GraalVM compiler, using a genetic algorithm to tune code generation parameters towards infrequently exercised optimizations. Preliminary experiments have identified 30 previously unknown bugs in the GraalVM compiler.

Additional Key Words and Phrases: fuzzing, JIT compiler, genetic algorithm, GraalVM, optimization log

1 Introduction

Bugs in compilers can have significant impact on users, especially when the symptom is not a crash but a silent miscompilation. Determining that a suspected program bug is actually a miscompilation is time-consuming for developers. Hand-written compiler tests typically cover cases engineers consider during development, but edge cases are often overlooked. Fuzzers can fill this gap by feeding compilers with automatically generated programs and testing the results for correctness.

Our work concerns testing of the GraalVM compiler, a JIT and AOT compiler for Java and other languages. While being written entirely in Java protects it from memory safety issues plaguing C/C++ compilers, GraalVM is not immune to implementation errors leading to miscompilations or crashes. We developed a compiler fuzzer targeting GraalVM and, in line with related work, found several previously unknown bugs.

However, the sustained effectiveness of a compiler fuzzer depends on the input code generator's capabilities. Certain compiler optimizations require very specific combinations of language features. A naïve approach using fixed probability distributions for language features leads to marginally small probabilities of certain combinations occurring, leaving specific optimizations untested.

Code coverage is a popular feedback mechanism for steering fuzzer input generation. However, for Java software, code coverage has two major drawbacks:

Overhead: Collecting code coverage incurs non-negligible overhead. We observed 17% throughput decline with JaCoCo coverage.

Loss of context: Existing coverage solutions merge information from compilations of different methods, losing the context of which optimizations occurred together during compilation of the same method.

Based on these observations, we propose Lool, which uses the GraalVM compiler's optimization log for coverage feedback. The precise, domain-specific information from the optimization log

Authors' Contact Information: Felix Berlakovich, Universität der Bundeswehr, Munich, Germany, felix.berlakovich@unibw.de; Florian Schwarcz, Johannes Kepler Universität, Linz, Austria, florian.schwarcz@jku.at; Gergö Barany, Oracle Labs, Vienna, Austria, gergo.barany@oracle.com; Hanspeter Mössenböck, Johannes Kepler Universität, Linz, Austria, hanspeter.moessenboeck@jku.at; Stefan Brunthaler, Universität der Bundeswehr, Munich, Germany, brunthaler@unibw.de.

guides input generation towards rare or uncovered compiler optimizations. During preliminary experiments, we found 30 previously unknown bugs in the GraalVM compiler.

Contributions.

- We present Lool, an optimization-log guided fuzzer that uses domain-specific knowledge to reach rare compiler optimizations.
- We describe a genetic algorithm for mutating code generation parameters based on optimization log feedback.
- Preliminary experiments demonstrate the effectiveness of varying generator parameters for finding bugs.

2 Background and Motivation

2.1 The Problem of Non-Uniform Coverage

Our goal is to change the code generator's parameters to produce code that triggers rare optimizations more often. Like other fuzzers based on fixed probability distributions, GraalVM fuzzing does not exercise all parts of the compiler uniformly. In our experiments with 1000 randomly generated Java programs using default configuration, the least frequent optimization occurred only 11 times, while others occurred millions of times.

2.2 Context-Aware Coverage

Consider a loop containing control flow and a method call, making loop vectorization inapplicable. However, applying *loop peeling* to separate the first iteration produces a loop that *can* be vectorized. Such pairs of optimizations are interesting—in our experience, many intricate compiler bugs involve interactions between multiple compilation phases.

Our optimization-log-based coverage can record that optimizations happened in the same *method* or even on the same *loop*. A context-aware metric that does not merge information from separate compilations can guide the fuzzer toward such interesting optimization pairs.

2.3 GraalVM Optimization Log

The GraalVM compiler generates an optimization log during compilation, including entries for every optimizing code transformation. Full log entries include the compiler phase, optimization name, and source code location. An abridged form only increments counters for each optimization, with very low overhead compared to code coverage.

3 GraalVM Compiler Fuzzing

3.1 Input Program Generation

Our fuzzer generates Java source code using liveness-driven random code generation, where all computed values are guaranteed to be used by following statements. This avoids unused computations being trivially eliminated, increasing the density of interesting code per test case. The generator supports a large subset of Java including class structures, control flow statements, expressions, and method calls.

The likelihood of generating each feature is parameterizable—we call a set of such parameters a *parameter vector*. Generated programs are fully self-contained and deterministic.

3.2 Test Execution

Our test harness:

(1) Receives a generated program from the input generator

(2) Executes the program in the bytecode interpreter, recording reference output (this also warms up VM profiles)
(3) Explicitly compiles methods with GraalVM up to a certain call graph depth
(4) Executes compiled code and compares output to the reference

The harness detects compiler crashes and output mismatches. Differential testing against the interpreter enables detection of miscompilations. This framework has been in regular use within GraalVM for three years and found many bugs.

4 Design of Lool

Lool uses the compiler's optimization log to select among suitable parameter vectors. The selection is driven by a genetic algorithm that breeds more desirable parameter vectors over time.

4.1 Genetic Algorithm Overview

We follow a parametric fuzzing approach but, unlike existing work that treats parameters as bit strings, we mutate the code generator's parameters *directly*. A population consists of different parameter vectors, each used to generate a number of input programs. A parameter vector is desirable if it triggers new optimizations or bugs. The genetic algorithm starts with baseline vectors, fuzzes with each, and builds new generations based on feedback.

4.2 Search Space Reduction

Our code generator has over 120 parameters. We restrict parameter vectors to a subset showing non-negligible correlation ($|r| > 0.3$) to optimizations:

- Five statement types: `if`, fold-style `for`, map-style `for`, `while`, `synchronized`
- Six expression types: local variable initialization/usage, parameter usage, binary expressions, method calls, ternary expressions

4.3 Fitness Function

We compute fitness relative to the entire generation along multiple dimensions:

- Number of different rare optimizations triggered
- Number of bugs discovered

In preliminary experiments, we identified seven rare optimizations (fewer than 100 occurrences in 1000 test cases). For each dimension, the ten best individuals receive scores from 10 to 1, multiplied by weights. Bug/timeout dimensions have higher weights than optimizations. Known bugs are deduplicated by scanning error messages.

4.4 Building Generations

During mutation, the algorithm adjusts parameters while staying in valid ranges. For probability distributions (statement/expression types), one entry "steals" from another to maintain constant sum. With 5% probability, we perform extreme mutations biasing heavily toward one property. Crossover chooses independent probabilities randomly from either parent, while distributions are chosen as a whole.

5 Preliminary Results

5.1 Bug Finding

Over approximately three months of experiments, we identified 30 new crashing bugs in the GraalVM compiler. About half were found by parameter tuning approaches, while the other half was found by prototype versions of the genetic algorithm. These bugs would likely not have been

found during routine fuzzing with unmodified parameters, as daily fuzzing generates new bug reports at a much lower rate.

5.2 Parameter-Optimization Correlation

Inspection of optimization logs confirmed that some parameters correlate with particular optimizations. This was expected at a high level—loop optimizations require certain loop shapes. Our goal is to let the mutation engine infer more complex relationships between parameter sets and optimizations.

5.3 Coverage Overhead

Recording line coverage with JaCoCo adds significant overhead to fuzzing runs. The optimization log counters have very low overhead by comparison, allowing more test cases in a given time.

6 Related Work

Swarm Testing. Lool follows the insight that inputs including more features are not necessarily beneficial [1]. Unlike directed swarm testing, Lool chooses targets automatically based on optimization log coverage.

JIT Compiler Fuzzing. Fuzzili generates JavaScript programs for V8 fuzzing [2]. JITFuzz uses coverage-guided mutation of Java class files [3]. JOpFuzz investigates relationships between code features and JIT compiler optimizations [4]. Lool differs by using optimization logs as domain-specific feedback.

Compiler Optimization Logs. V8's TurboFan, LLVM's Remarks, and Intel Compiler's Optimization Reports provide similar logging. Some include negative entries for attempted but unsuccessful optimizations, which could help fuzzers gradually approach triggering code shapes.

7 Conclusions

We have presented Lool, a low-overhead optimization-log-guided approach to compiler fuzzing. Lool uses domain-specific feedback from the compiler's optimization log, which can be collected with lower overhead than general code coverage. The optimization log naturally supports context-sensitive information, such as optimizations performed during compilation of a single method.

Our preliminary experiments within the GraalVM fuzzing infrastructure have shown promising results with 30 previously unknown bugs found. We are planning a thorough evaluation comparing different configurations: static parameters, AFL++-based mutation with code coverage, and Lool with context-aware optimization log feedback.

Acknowledgments. This research was partially funded by Oracle Labs. We thank all members of the Virtual Machine Research Group at Oracle Labs and researchers at Johannes Kepler University's Institute for System Software.

References

[1] Alex Groce, Chaoqiang Zhang, Eric Eide, Yang Chen, and John Regehr. 2012. Swarm Testing. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*. 78–88.

[2] Samuel Groß, Simon Koch, Lukas Bernhard, Thorsten Holz, and Martin Johns. 2023. Fuzzilli: Fuzzing for JavaScript JIT Compiler Vulnerabilities. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*.

[3] Mingyuan Wu, Minghai Lu, Heming Cui, Junjie Chen, Yuqun Zhang, and Lingming Zhang. 2023. JITFuzz: Coverage-Guided Fuzzing for JVM Just-in-Time Compilers. In *Proceedings of the International Conference on Software Engineering (ICSE)*. 56–68.

[4] Cong Li, Yanyan Jiang, Chang Xu, and Zhendong Su. 2023. Finding JIT Compiler Bugs via Automated Generation of Test Programs. In *Proceedings of the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*.
[5] Gergö Barany. 2017. Liveness-Driven Random Program Generation. *arXiv preprint arXiv:1709.06082*.

Semantic Patching: Language-based Automatic Program Repair

GIACOMO PRIAMO, La Sapienza, Italy
STEFAN BRUNTHALER, Universität der Bundeswehr München, Germany

Automatic Program Repair (APR) aims to automatically fix software defects by generating patches that correct erroneous program behavior. Existing approaches suffer from fundamental limitations: search-based methods like GenProg explore a vast space of potential patches through generate-and-validate strategies, while constraint-based approaches such as SemFix and Angelix use symbolic execution to derive repair constraints. Both paradigms face inherent incompleteness problems and scalability challenges. We present *Semantic Patching*, a novel language-based approach to automatic program repair that leverages rich semantic information from compiler infrastructures. By analyzing program dependency graphs and utilizing precise type information, control-flow graphs, data-flow analyses, and identifier semantics, our approach identifies semantic differences between program versions rather than merely syntactic ones. We implement our technique using Joern for program dependency graph extraction and clangd for symbol resolution, demonstrating that semantic patching offers a promising, practical, and scalable alternative to existing APR methods. Preliminary evaluation on the Codeflaws benchmark suite shows the viability of our approach across diverse defect classes.

Additional Key Words and Phrases: automatic program repair, semantic patching, program dependency graphs, static analysis, software maintenance

1 Introduction

Software defects remain one of the most persistent challenges in software engineering. Studies consistently show that debugging and maintenance activities consume a substantial portion of development resources, with estimates suggesting that fixing bugs accounts for 50% or more of total software development costs. As software systems grow in complexity, the burden of manual bug fixing becomes increasingly unsustainable, motivating research into automated approaches.

Automatic Program Repair (APR) has emerged as a promising research direction, aiming to automatically generate patches that correct faulty program behavior [1, 3]. The fundamental premise of APR is deceptively simple: given a buggy program and a specification of correct behavior (typically a test suite), automatically synthesize a patch that transforms the program to satisfy the specification.

Existing APR approaches can be broadly categorized into several paradigms. *Search-based methods*, exemplified by GenProg [6], employ genetic algorithms to explore the space of possible patches. These generate-and-validate approaches systematically modify program statements and evaluate candidate patches against test suites. While conceptually straightforward, the search space of possible modifications is astronomically large.

Constraint-based methods, such as SemFix [4] and Angelix [2], leverage symbolic execution to derive constraints that correct patches must satisfy. While more principled than pure search, these approaches are limited by the scalability of symbolic execution. More recently, *neural and LLM-based methods* [7] have shown promise but often struggle with semantic correctness.

All these paradigms share a *fundamental incompleteness problem*: the search space of potential repairs is inherently vast, and existing techniques can only explore a limited subset. We propose *Semantic Patching*, a novel approach that addresses these limitations by exploiting the rich semantic information available in modern compiler infrastructures.

Authors' Contact Information: Giacomo Priamo, La Sapienza, Rome, Italy, giacomo.priamo@unibw.de; Stefan Brunthaler, Universität der Bundeswehr München, Munich, Germany, brunthaler@unibw.de.

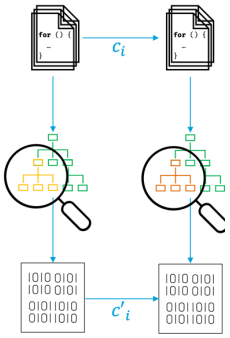


Fig. 1. The semantic patching pipeline. Two source file versions (top) undergo a syntactic change c_i. Each version is analyzed to extract its intermediate representation as a tree structure (middle), where magnifying glasses highlight the differing subtrees. The semantic change c'_i between the compiled representations (bottom, shown as binary) captures the behavioral modification independent of surface syntax.

2 Semantic Patching Approach

The key insight underlying semantic patching is the distinction between *syntactic* and *semantic* program differences. Traditional diff-based approaches operate at the syntactic level, but these differences often obscure the actual semantic modifications. Two syntactically different programs may exhibit identical behavior, while minimal syntactic changes can dramatically alter program semantics.

Figure 1 illustrates our approach. Semantic patching leverages compiler-derived information to analyze programs at a deeper level than surface syntax:

Control-Flow Graphs (CFGs) capture possible execution paths. By comparing CFGs between versions, we identify changes to control flow structure independent of syntactic representation.

Data-Flow Analysis tracks how values propagate through the program, identifying def-use chains and dependencies between program points.

Program Dependency Graphs (PDGs) combine control and data dependencies into a unified representation that captures the essential semantic structure while abstracting away irrelevant statement ordering.

Type Information provides precise semantic constraints on program values and operations.

Given a buggy program P and a reference program P', semantic patching proceeds in three phases. First, the *PDG differ* extracts and compares program dependency graphs, identifying semantic differences. Second, the *semantic patch generator* synthesizes a patch specification capturing the required transformation. Third, the *application sites detector* identifies locations where the semantic patch should be applied.

3 Implementation

We have implemented semantic patching as a prototype system targeting C programs. The implementation comprises approximately 1,600 lines of Python code, organized into three main components.

For program dependency graph extraction, we utilize **Joern** [8], an open-source platform for static code analysis. Joern provides robust PDG construction for C/C++ programs through its Code Property Graph (CPG) representation, which unifies abstract syntax trees, control-flow graphs, and program dependency graphs into a single queryable structure.

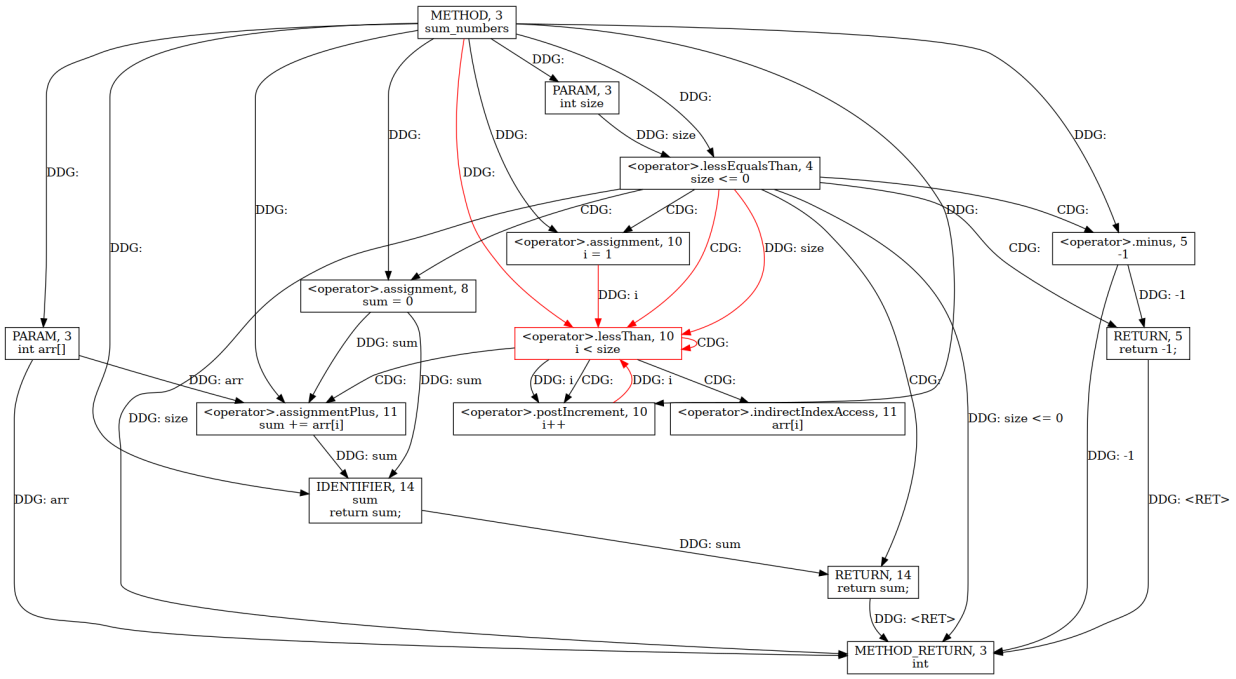


Fig. 2. Program Dependency Graph for a `sum_numbers` function. Each node represents a program statement labeled with its operation type and line number. Black arrows denote data dependencies (DDG edges) indicating value flow between statements—for example, the parameter `arr[]` flows to the array access and accumulator operations. Red arrows denote control dependencies (CDG edges) showing conditional execution—statements within the loop body depend on the loop condition `i < size`. The nodes highlighted with red borders mark the loop condition and its dependents, typical locations for boundary condition defects.

Figure 2 shows an example PDG generated by our system. The graph makes explicit the semantic structure—data dependencies showing how values flow from parameters through computations to the return value, and control dependencies showing how the loop condition governs execution of the loop body.

To obtain precise symbol information, we integrate with the **clangd** language server through a custom Python bridge (approximately 727 lines of code). This provides accurate type information, symbol resolution, and semantic analysis capabilities derived from the Clang compiler infrastructure.

The PDG differ component compares dependency graphs using graph matching algorithms, identifying corresponding nodes based on their semantic role rather than syntactic properties. The semantic patch generator abstracts identified differences into reusable patch patterns, while the application sites detector uses pattern matching on semantic structure to identify repair locations.

4 Preliminary Evaluation

We evaluate our approach on the **Codeflaws** benchmark suite [5], a comprehensive collection of real-world defects extracted from programming competition submissions. Codeflaws contains 7,436 programs encompassing 3,902 distinct defects classified into 40 categories, including statement-level changes, operator defects, operand defects, and higher-order mutations.

This benchmark provides an ideal testbed for semantic patching because it includes pairs of buggy and correct program versions, enabling direct analysis of semantic differences. Our preliminary results indicate that semantic patching can successfully identify and characterize repairs across multiple defect classes. The PDG-based analysis effectively captures the semantic essence of repairs, even when syntactic representations differ substantially.

We observe particularly strong results for operator and boundary condition defects, where the semantic change is localized and clearly visible in the dependency structure. The diversity of defect types allows us to assess the generality of our approach and identify areas where additional techniques may be needed.

5 Related Work

Our work builds on several lines of research. **Search-based APR** techniques like GenProg [6] use evolutionary algorithms but struggle with combinatorial explosion. Our semantic approach provides a more focused search space. **Constraint-based APR** methods including SemFix [4] and Angelix [2] provide stronger guarantees but face scalability challenges. Semantic patching shares the goal of semantic analysis but avoids path explosion through static dependency analysis. **Program differencing** research has developed techniques for comparing versions; we apply these ideas specifically to APR.

6 Conclusion and Future Work

We have presented Semantic Patching, a novel language-based approach to automatic program repair that leverages rich compiler-derived semantic information. By analyzing program dependency graphs rather than syntactic representations, our approach offers a promising alternative to existing APR techniques.

Our preliminary results suggest that semantic patching is: (1) **Promising**—successfully identifying semantic differences across diverse defect classes; (2) **Practical**—integration with Joern and clangd enables robust analysis; and (3) **Scalable**—PDG-based analysis avoids the exponential blowup of symbolic execution.

Future work will expand evaluation to additional benchmarks including Defects4J, extend language support to Java and Python, and develop more sophisticated patch synthesis algorithms that leverage semantic information for generating candidate repairs.

References

[1] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. 2019. Automatic Software Repair: A Survey. *IEEE Transactions on Software Engineering* 45, 1 (2019), 34–67.

[2] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis. In *Proceedings of the 38th International Conference on Software Engineering*. ACM, 691–701.

[3] Martin Monperrus. 2018. Automatic Software Repair: A Bibliography. *Comput. Surveys* 51, 1 (2018), 1–24.

[4] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. SemFix: Program Repair via Semantic Analysis. In *Proceedings of the 35th International Conference on Software Engineering*. IEEE, 772–781.

[5] Shin Hwei Tan, Jooyong Yi, Sergey Mechtaev, Abhik Roychoudhury, et al. 2017. Codeflaws: A Programming Competition Benchmark for Evaluating Automated Program Repair Tools. In *Proceedings of the 39th International Conference on Software Engineering Companion*. IEEE, 180–182.

[6] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically Finding Patches Using Genetic Programming. In *Proceedings of the 31st International Conference on Software Engineering*. IEEE, 364–374.

[7] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated Program Repair in the Era of Large Pre-trained Language Models. *arXiv preprint arXiv:2303.14396* (2023).

[8] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and Discovering Vulnerabilities with Code Property Graphs. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*. IEEE, 590–604.

Angluerus: Data-based Hardware-Software Binding Through Rowhammer

DANIEL DORFMEISTER, Software Competence Center Hagenberg, Austria
MATHÉO VERGNOLLE, Synacktiv, France
STEFAN BRUNTHALER, μCSRL, CODE Research Institute, University of the Bundeswehr Munich, Germany

Modern industrial machinery comes with advanced control software. Although this control software typically contains valuable intellectual property, it often lacks effective protection. Consider, for example, obfuscation: copying to cloned machinery suffices to bypass protection. Prior work has investigated possibilities to bind software to a specific device such that the software cannot be used independently of the device. GlueZilla, for example, binds software to hardware by making the execution of the software depend on device-specific Rowhammer-induced bit flips. The protected software thus exhibits a different behavior when executed on a device it was not compiled for.

We present Angluerus, which extends the idea of GlueZilla into the *data domain*. By aligning data with Rowhammer-induced bit flips, Angluerus is both more portable and able to tolerate unstable bit flips. To frustrate dynamic analysis, Angluerus uses debug blocking in combination with moving the Rowhammer part to a separate process. We evaluate Angluerus w.r.t. two properties: practicality and performance. Two case studies serve to demonstrate practicality. To evaluate performance, we use a matching subset of the SPEC CPU 2017 benchmarks and report an order of magnitude improvement over GlueZilla.

Acknowledgments

The research reported in this article has been funded by the Federal Ministry for Innovation, Mobility and Infrastructure (BMIMI), the Federal Ministry for Economy, Energy and Tourism (BMWET), and the State of Upper Austria in the frame of the COMET Module Dependable Production Environments with Software Security (DEPS) (FFG grant no. 888338) and the SCCH competence center INTEGRATE (FFG grant no. 892418) within the COMET - Competence Centers for Excellent Technologies Programme managed by Austrian Research Promotion Agency FFG.

Authors' Contact Information: Daniel Dorfmeister, Software Competence Center Hagenberg, Hagenberg, Austria, daniel.dorfmeister@scch.at; Mathéo Vergnolle, Synacktiv, Paris, France, matheo.vergnolle@synacktiv.com; Stefan Brunthaler, μCSRL, CODE Research Institute, University of the Bundeswehr Munich, Munich, Germany, brunthaler@unibw.de.

Code-Copying Compilation in Production

An Experience Report

M. ANTON ERTL, TU Wien, Austria
BERND PAYSAN, net2o, Germany

A code-copying compiler implements a programming language by concatenating code snippets produced by a different compiler. This technique has been used in Gforth since 2003, with code snippets generated by GCC. We have solved various challenges: in particular, which code snippets can be copied and what to do about the others; and challenges posed by changes in compilers. The performance of Gforth is similar to that of SwiftForth, a commercial system with a conventional compiler; the implementation effort is comparable to 1–2 targets for SwiftForth.

1 INTRODUCTION

Code copying is a programming language implementation technique where the compiler of the implemented languate A concatenates code snippets coming out of the compiler for language B. While there have been a number of research papers about this topic (see Section 8), we know of only one production language implementation that has used this approach for a long time: Gforth.

The present work is an experience report about the use of code copying in Gforth: How does it compare to a conventional compiler (Section 2)? Section 3 explains the concepts of code copying, while Section 4 discusses various implementation aspects. We also discuss the problems from changes in compilers (Section 5) and operating systems (Section 6) and how we overcame them.

In addition to this experience report, this paper also discusses alternative approaches (Section 7) and related work (Section 8).

The present work also appears in the EuroForth 2025 proceedings, with the same content and different formatting.

1.1 Is Gforth a production system?

Gforth is free software that has been developed since 1992 and first released in 1996. As it is free software, everybody can use it without contacting us, and few people do, so we do not know that much about who uses it for what purpose. However, we know that it has been used by IBM and Apple in their work on Open Firmware, and Forth, Inc. (who develop SwiftForth, but also give Forth courses) have given courses using Gforth, also in the Open Firmware context. So: Yes, Gforth is a production system.

2 WHY NOT JUST WRITE A CONVENTIONAL COMPILER?

One reason why people may have avoided going for a code-copying compiler is the assumption that writing a conventional compiler will produce better code, or require less effort. By "conventional" we mean that there is a large amount of hand-written architecture-specific code for each target architecture in the compiler. So before we go into details about code copying, we will address this concern.

2.1 Performance

Figure 1 shows the performance of the `gforth-fast` engine of Gforth[1] with various optimizations, of two commercial conventional Forth compilers (SwiftForth and VFX Forth), and, for of GCC-12.2

[1]Gforth also has an engine `gforth` intended for debugging. All referenences to Gforth performance refer to `gforth-fast`.

Authors' addresses: M. Anton Ertl, TU Wien, Vienna, Austria, anton@mips.complang.tuwien.ac.at; Bernd Paysan, net2o, Burladingen, Germany.

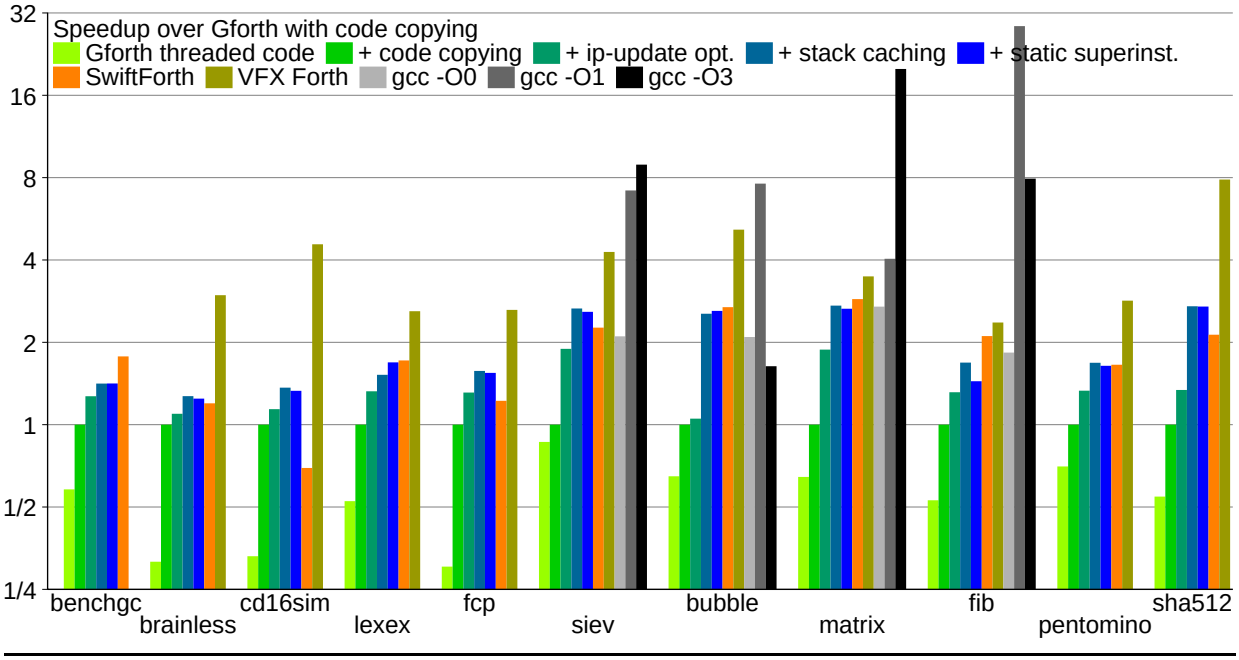


Fig. 1. Speedup factor of various systems over Gforth with code copying, on a Core i5-1135G7 (Tiger Lake)

`gcc -O0`, `-O1`, and `-O3`. All Forth systems use load-and-go compilers (compile time is included in the results), while GCC uses ahead-of-time compilation (only the run-time is shown in the results).

Not all benchmarks are available in C, and not all benchmarks run on all Forth systems, and the missing cases are reflected by missing bars.

The data shown is the median of 30 runs for each benchmark/system combination on a Core i5-1135G7 (Tiger Lake); each bar represents the number of cycles of Gforth with only code copying divided by the number of cycles of the system represented by the bar, i.e., the speedup of that system over Gforth with only code copying. The Gforth version used is `0.7.9_20250817`, commit `4224ab5fafea970dade64b04493ef690da8b3c32` compiled with gcc-11.4. The benchmarks are from the Forth appbench suite (benchgc–fcp), Gforth's small (and mostly loop-dominated) benchmarks (siev–fib), and two additional ones.

As can be seen, the performance of Gforth with all optimizations is similar to that of SwiftForth, which uses a conventional compiler, and typically around half of the performance of VFX Forth, which also uses a conventional compiler.

Before comparing Gforth with the others, let's first take a look at the variants of Gforth, starting with the one with the best performance/effort:

Threaded code This is a fast interpretation technique for virtual-machine (VM) code, without any machine-code generation (see Section 3.1).

Code copying This method concatenates code snippets from the threaded code engine (see Section 3). It requires an estimated 500 lines of code in the Gforth source code. With this method Gforth still accesses literal data and performs control flow by accessing the VM code; it therefore also maintains a VM instruction pointer (IP), and updates it once for every VM instruction.

IP-update optimization This optimizaton reduces these IP updates. It was added by inserting 864 lines and deleting 316 lines in the Gforth source code [EP24].

Stack caching (actually static multi-state stack caching) eliminates many memory accesses to stack items and stack-pointer updates [EG04a, EG05]. The way this optimization as implemented in Gforth requires code copying to work.

Static superinstructions replace a sequence of Forth words with an optimized sequence [EGKP02]. Many of the benefits that static superinstructions have originally provided are now provided by code copying, the IP-update optimization and static stack caching; there are still cases where static superinstructions result in shorter code, but this has not led to consistent speedups in these measurements.

The code implementing stack caching and static superinstructions is quite interweaved with the rest of the code, so it is hard to give precise numbers for their size, but we estimate [Ert24] that all four optimizations combined require an estimated total of 5000 lines of code.

SwiftForth's compiler can be seen as a copy-and-patch compiler, but with the code snippets written by hand in assembly language and better resulting code than when patching using object file linkage imformation (see Section 7.3). SwiftForth does not have a VM interpreter substrate, and therefore does not have IP updates, so it gains the benefits of the IP update optimization without having to do anything. It deals with literal values and control flow by patching the code. SwiftForth does not perform multi-state stack caching, but it makes extensive use of static superinstructions (346 rules in 1819 lines). Overall each of the IA-32 and AMD64 targets of SwiftForth has about 7000 lines of architecture-specific code [Ert24].

Gforth with all optimizations is competetive in speed with SwiftForth, so apparently Gforth's stack caching provides enough speedup to compensate the costs that Gforth incurs for literals and control flow.

VFX Forth performs register allocation of data-stack items within a basic block, and inlines aggressively; inlining is very helpful for idiomatic Forth code, where calls and returns are the most frequent basic block boundaries. Therefore inlining also enhances the effectiveness of VFX's register allocator. The speed advantage of VFX over Gforth and SwiftForth is a result of these optimizations. In particular, for the cd16sim benchmark there is one call site that calls an empty definition and that is responsible for 2/3 of Gforth's run-time on this benchmark, while VFX inlines it away. We have no source code for VFX and therefore cannot report numbers about the size of its compiler. When asked about the effort to port VFX to ARM A64 (a currently ongoing project), Stephen Pelc gave the qualitative statement "far too much".

VFX is faster than Gforth by typically around a factor of 2. However, it is possible to perform inlining in Gforth, too, with direct performance benefits as well as indirect benefits from better stack caching. It will be interesting to see how far Gforth (and code copying) can close the gap.

Gforth's performance with all optimizations is roughly comparable to that of `gcc -O0` on those benchmarks that are also available in C. `gcc -O1` and `gcc -O3` often produce significantly faster code; sometimes they don't, but the reasons for that are beyond the scope of this paper.

2.2 Portability

A major reason to avoid implementing a conventional compiler is portability/retargetability.

Gforth has supported as many architectures as we could get our hands on, as long as gcc and something Unix-like (e.g., Cygwin for Windows) is available on the architecture. Gforth has supported the following architectures with a code copying compiler: Alpha, ARM A32/T32, ARM A64, HPPA, IA-32, IA-64, Loongarch, SPARC, PowerPC, PowerPC64 (but we no longer can check for all architectures that they still work). Gforth supports all architectures it does not know about by falling back to threaded code, which is slower, but still works.

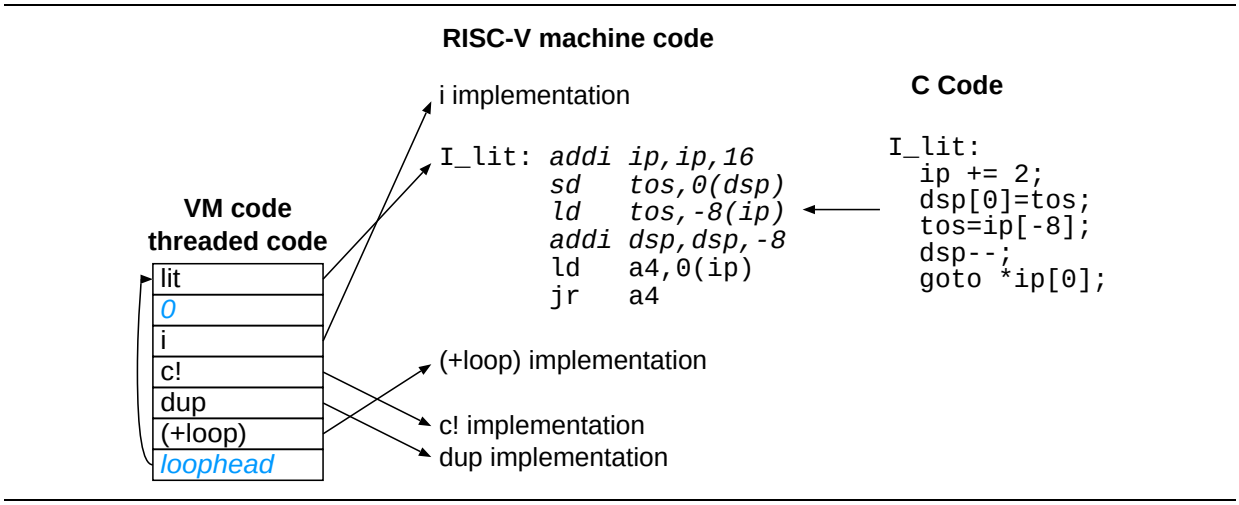


Fig. 2. Threaded-code representation of VM code. Each box is a machine word. *Slanted light blue* indicates an immediate operand of the preceding VM instruction.

In particular, when IA-64 (launched 2001) and AMD64 (launched 2003) became available to us in 2003, Gforth worked out of the box on these architectures[2] using the unknown-architecture support, likewise for ARM A64 in 2014 and RISC-V in 2017. A few small changes enabled code copying[3], and a one-line change for configuring the number of registers for stack caching.

The benefit of code copying is that it reuses the retargeting efforts of the compiler it is based on (GCC or Clang in case of Gforth).

By contrast, SwiftForth has supported only IA-32 until the 2020s, when they started working on an AMD64 port (released on 2025-10-22). VFX has supported IA-32 initially, later ARM A32, and, also starting in the 2020s, AMD64. Both systems have interactive cross-compilers for a number of embedded targets.

The low number of desktop ports and the late support for AMD64 may be due to lack of commercial interest, but we think that the larger effort required to retarget and maintain the compiler for another architecture has something to do with it. iForth, another conventional Forth compiler, got an AMD64 port in 2009, but the IA-32 port was subsequently dropped (last release with IA-32 support in 2011).

2.3 Incremental development

Another benefit of code copying over writing a conventional compiler is that it can be done step-by-step: First add code copying, then add one optimization (e.g., IP-update optimization), then the next, etc., always with the fallback options of disabling the optimization or completely falling back on threaded-code interpretation.

By contrast, when coming from an interpreter, the conventional model requires a big-bang approach where a complete code generator for one target has to be developed without reusing much from an existing interpreter; and as long as you do not develop code generators for all targets, you still need to maintain the interpreter, as well as all the compiler targets. The latter will hopefully be helped by designing the compiler for retargetability, but that increases the complexity of the compiler framework.

[2]We added 64-bit support in 1996 while doing the Alpha port.

[3]For RISC-V, this was our first encounter with gcc-7 and its more aggressive code duplication (Section 5.4); we needed a little longer to find a workaround for that, but that's not specific to the architecture.

3 WHAT IS CODE COPYING COMPILATION?

3.1 Threaded Code

The basis for Gforth's code-copying implementation is a threaded-code interpreter [Bel73] for Gforth's virtual machine (VM).

As a running example, we look at the VM code in Fig. 2. The first VM instruction in the example is `lit`, which has an immediate operand *0*. This VM instruction pushes its immediate operand on the data stack. It is represented by the address of the machine code that implements it; in direct-threaded code, every VM instruction is represented by the address of the machine code that implements it. In the case of `lit`, the implementation for RISC-V (RV64G) is:

```
                # //C code
addi ip,ip,16   # ip += 2;
sd   tos,0(dsp) # dsp[0] = tos;
ld   tos,-8(ip) # tos = ip[-1];
addi dsp,dsp,-8 # dsp--;
ld   ca,0(ip)   # ca = ip[0];
jr   ca         # goto *ca;
```

This code uses register names that reflect their roles: `ip` is the VM instruction pointer; `tos` is the top of the data stack; `dsp` is the data stack pointer; `ca` is the code address (of the next VM instruction).

The *slanted blue* instructions are the payload which perform the actual work of the VM instruction as far as code copying is concerned. Other optimizations reduce that part further; e.g. the first instruction updates IP, and the IP-update optimization often optimizes it away.

The third instruction loads the immediate operand (0) from the VM code by accessing it through IP. This access of immediate operands and control-flow operations through IP is still in Gforth with all optimizations applied, and is the difference between an interpreter-based code-copying system and a copy-and-patch system (Section 7.3).

The bottom two (black) instructions perform the dispatch to the next VM instruction. The first instruction loads the machine code address of the next VM instruction, and the second instruction jumps to it.

This assembly-language code can be generated from the C code shown in the comments of the assembly language. It uses the GNU C extension "Labels as Values",[4] which allows jumping to the address in `ca` with `goto *ca`[5]; this extension is also supported by Clang, tcc, and icc.

The other VM instruction implementations have the same pattern of *payload*, and dispatch. The last VM instruction in our example, `(+loop)` is notable: it is a VM-level conditional branch that branches back to *loophead* (given as immediate operand) or falls through to the next instruction. It is implemented with the following code

[4] https://gcc.gnu.org/onlinedocs/gcc/Labels-as-Values.html

[5] The GCC maintainers call this a computed goto, although it is more like a Fortran assigned goto.

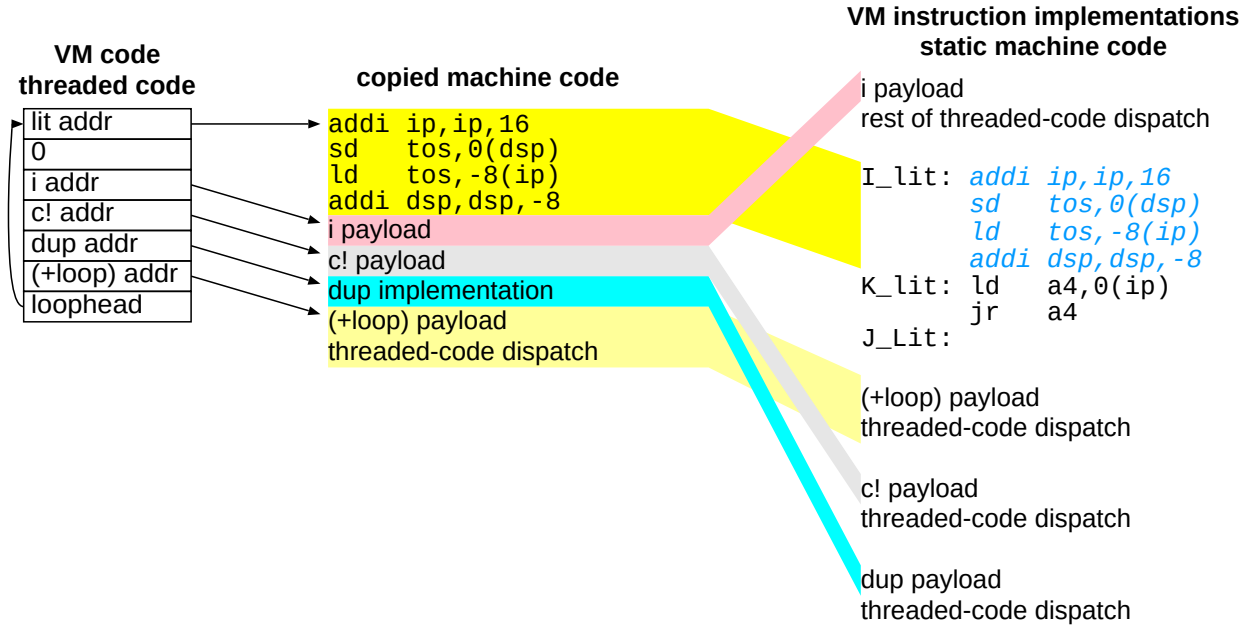


Fig. 3. Code copying.

```
 addi ip,ip,16            # ip += 2;
 ...compute condition...
 blt  a5,zero,fallthrough # if (taken) {
 ld   ip, -8(ip)          #   ip = ip[-1];
 ld   ca, 0(ip)           #   ca = ip[0];
 jr   ca                  #   goto *ca;
fallthrough:              # }
 ld   ca,0(ip)            # ca = ip[0];
 jr   ca                  # goto *ca;
```

If the conditional branch is taken, the new IP is loaded from the immediate operand and a dispatch is performed. It is better to have separate dispatches for the taken and the fallthrough cases for branch prediction[6] and because it allows to leave away the fallthrough dispatch in code-copying.

3.2 Code copying

Most VM instructions do not perform VM-level control flow, but just continue with the next VM instruction. Code copying copies and concatenates the machine code implementing the VM instructions, but in most cases without the dispatch code at the end. Only taken branches (i.e. VM instructions that change IP to point to some other VM instruction than the next one) need to perform a dispatch.

Figure 3 shows this for our running example. The VM code is conceptually the same as before, but for each VM instruction the machine word now points to the copied machine code instead of the original.

[6]Even with history-based indirect-branch prediction, branch predictors have an easier time if there are fewer targets for each indirect branch

In particular, the copied code still has the IP, which points to the threaded (VM) code, and it accesses the immediate operands *0* and *loophead* through it. The threaded code is also used on control flow: the VM-level conditional branch `(+loop)` is taken, loads the target threaded-code address *loophead* into IP, and then performs a threaded-code dispatch, which loads the code address at loophead, which points to the start of the concatenated code. All control flow in Gforth is performed with threaded-code dispatches in this way.

The threaded-code slots for instructions other than `lit` in this example are not accessed during execution. Gforth keeps them around to simplify the implementation.

At the end of the shown sequence the threaded-code dispatch is copied. While this is necessary for unconditional branches, it is not generally necessary for conditional branches such as `(+loop)` (as discussed above). However, the following VM instruction may make it necessary to perform a dispatch after the `(+loop)`.

Code copying has also been called the `memcpy()` method [RS96], selective inlining [PR98] and (especially in Gforth) dynamic superinstructions [EG03a].

3.3 Benefits over threaded code

The obvious benefit of code copying is that it eliminates most threaded-code dispatches and results in straight-line execution of VM-level straight-line code, avoiding the limit of typically one taken branch per cycle. Another benefit is that the indirect branches in most of the remaining dispatches have only one target, vastly improving branch prediction accuracy in CPUs without sophisticated indirect-branch predictors, and still making life easier (and faster) for hardware with such branch predictors.

Another benefit is that code copying enables additional optimizations that require code snippets that are not represented as VM instructions (and where introducing additional VM instructions with threaded-code dispatch would make the optimization unprofitable).

E.g., the IP update optimization [EP24] leaves the IP update in front of most VM instruction implementations away and replaces it with an IP update by a larger amount for VM instructions that actually use the IP.

As another example, stack caching as implemented in Gforth inserts transitions between stack-cache states where necessary. These transitions do not have a VM instruction slot and therefore can only be inserted when code-copying is enabled. Gforth's stack-caching implementation relies on being able to insert the transitions, so stack caching is disabled when code copying is disabled [EG04a].

3.4 When is code copying appropriate?

The shorter the VM instruction implementations are, the larger the benefit of code copying over threaded code, because the overhead of threaded-code dispatch is relatively larger then.

Conversely, with long VM instruction implementations as in Tcl, whose VM instructions "can average hundreds of [machine] instructions" [VA04] the benefit is small, and often does not amortize the cost of copying the code or of increased I-cache misses [VA04].

Another aspect is that a compiler (to VM code) that uses more VM instructions, with each doing less, has more opportunities to optimize the VM code. This has been done for CPython recently[7]. With expensive VM instruction dispatch, splitting an existing VM instruction into several simpler ones increases the cost, and the opimization must be very good and must be applicable often to amortize this cost. With code-copying, the dispatch cost approaches 0, and such transformations become less of a gamble.

[7] https://github.com/faster-cpython/

4 IMPLEMENTATION OF CODE COPYING

4.1 Code organization

Gforth has a big function `engine()` that contains all the code snippets (implementations of all VM instructions, and additional snippets used by optimizations), and little else.

Every code snippet has a label in front of it and behind it:

```
L_before:
  code snippet in C;
L_after:
  threaded-code dispatch;
```

You can see that more concretely in Fig. 3.

The label before it obviously points to the start of the code snippet.

Getting the right label for the end of the code snippet was initially straightforward (up to gcc-3.1), but later required extra work. If the source code falls through to the label (i.e., it does not end in an unconditional branch), like for the payload of most VM instructions in Gforth, with some extra help (see Section 5.4), the following label points right behind the code snippet, but if the code snippet cannot reach the label (e.g., because it ends in an unconditional branch, e.g, in a threaded code dispatch), gcc-3.2 and following have reordered code. We solved this problem by taking the values of all the labels, sorting them, and searching for the first label behind the label at the start of the snippet. This might include some unrelated code in cases where the code snippet does not fall through to the label, but in that case this is not a problem for correctness (but possibly for relocatability, see Section 4.5).

The function `engine()` has two code paths: the first just returns a table containing all the labels, for use in threaded-code generation and code-copying; the second starts the execution of the code by performing a threaded-code dispatch.

If code copying is disabled,[8] the threaded code address for each VM instruction just points to the implementation of that instruction inside `engine()`, and every threaded-code dispatch jumps around within this function.

With code copying, the first threaded code dispatch in `engine()` jumps to the copy of the VM instruction implementation and continues running there, with control-flow changes by performing a threaded-code dispatch.

4.2 Why does it work?

Why can we concatenate the code snippets produced in the way described above, and get code that works?

In particular, won't the register allocator have different register allocations for the different code snippets? Actually, at the start and, for fallthrough snippets, the end of the snippet, the register allocation has to be the same as at the start of every other snippet, because the compiler has to consider the possibility that every `goto *` jumps to every label whose address is taken. And the addresses of all labels before and after all code snippets are taken (to determine the code snippet address and length).

The code snippets that do not fall through end in a `goto *` in Gforth. And the register allocation at the `goto *` has to be compatible with that of all the labels whose address is taken, or it would not work even in ordinary use.

[8] Gforth option `--no-dynamic`.

More precisely, `engine()` is compiled separately from the code dealing with the threaded code, so the C compiler has to assume that every `goto *` in `engine()` can jump to any label whose address is taken.

Therefore, at a `goto *` all variables are alive (i.e., read before being overwritten) that are alive at any label whose address is taken, and each variable has to be in the same location at all those labels and all the instances of `goto *`. The code snippets that fall through to their second label are followed by a threaded-code dispatch:

```
ca = ip[0];
goto *ca;
```

so at the label between the code snippet and the dispatch, all the same variables are alive as at the `goto *`, except possibly `ca`, but that is not alive before the threaded-code dispatch, either. These variables also all have to reside at the same locations, because the `goto *` could jump to them.

4.3 Fallback

There are cases where certain code snippets cannot be copied (usually because they are not relocatable, see Section 4.5). How does Gforth deal with that?

Gforth falls back to plain threaded code in these cases: Append a threaded-code dispatch to the previous copied code snippet (unless the code snippet already ends with a threaded-code dispatch), and let the machine word representing the current VM instruction point to the original implementation of the VM instruction (inside `engine()`) rather than a copy). At run-time, the code performs the threaded-code dispatch, which then jumps to the original; that ends in another threaded-code dispatch, which may jump to code coming out of code-copying, or to another original implementation.

If other optimizations are active, the preparation for the fallback may require appending additional code. E.g., the IP needs to be up-to-date before the threaded-code dispatch, so in the presence of IP-update optimization, an IP update may be inserted before the threaded-code dispatch. Also, in Gforth the plain threaded code always expects the stack in the canonical state, so in the presence of stack caching, a transition from the current stack state to the canonical stack state may need to be inserted before the threaded-code dispatch.

Gforth may also find that it cannot copy the threaded-code dispatch. In that case it disables code copying completely and falls back to threaded code not just for individual VM instructions, but for all of them.

The option to fall back to threaded code has helped in various cases where things did not work according to our expectations (e.g., see Section 5.4). It means we always have a way to make Gforth work, albeit not as fast as we would like.

4.4 Instruction sets

Code copying is based on the assumption that the code snippets are independent and concatenable. At the instruction-set level this is satisfied if individual instructions are independent and concatenable. Some instruction sets have restrictions between groups of instructions. In this case a code snippet must not contain a partial group, i.e., there must not be a label within a group.

There are a few cases of such instruction-set restrictions:

Branch delay slots This is a misfeature of some early RISC architectures, in particular, HPPA, MIPS and SPARC: The branch instruction performs the instruction behind it before continuing at the target. This does not work with code copying if the compiler puts a label

between the branch and the instruction behind it. However, the compilers we have used (most recently gcc-14.2) do not do that.

Load delay slots This is a restriction of the MIPS I instruction set (eliminated in MIPS II). The instruction behind a load instruction is not allowed to read the register written by the load instruction. MIPS I also has some placement restrictions on reading and writing the `hi` and `lo` registers. Having labels right after the load or in the shadow of `hi`/`lo` reads can result in violating these restrictions in code copying. We have not tested if compilers actually place labels in a way that would lead to such violations. Instead, these concerns along with the relocatability problems (Section 4.5) and the lack of relevance of MIPS in Unix systems around 2003 were the reasons why we just configured Gforth to fall back to threaded code on MIPS (including the 64-bit MIPS port).

Instruction groups This is an IA-64 (aka Itanium processor family) property. Instructions within a group have restrictions on register usage that are intended to ensure that the instructions can be performed in one cycle without register renaming.[9] If a compiler put a label inside a group, code copying could violate these restrictions. Apparently the compilers we used (gcc-3.3, gcc-4.1.3, gcc-4.3.2) put stops (group boundaries) at labels, because in our testing IA-64 has always worked fine. If they did not, an easy fix would be to insert the stops using `asm` statements or at the assembly-language stage.

Based on the experiences with branch delay slots and instruction groups, it seems that gcc developers also avoid splitting groups of instructions with interdependencies by inserting a label inside these groups, but if these instruction sets still were important targets, that might change.

The problematic restrictions/features have not spread to newer architectures and all the architectures with these restrictions in general-purpose computers have been canceled in the meantime, while older or contemporary architectures without these restrictions thrive. So apparently the idea of independent, concatenable instructions has some merit, and we can expect that future instruction sets will also exhibit this property and thus support code copying.

4.5 Relocatability

A code snippet must be relocatable in order to be used in code copying, i.e., it must behave the same way in the original place and when copied.

Non-relocatable code. The main problems here are references to addresses: The code in the snippet must refer to addresses inside the snippet in a PC-relative way, and must *not* refer to addresses outside the snippet in a PC-relative way. Most architectures refer to other code addresses in a PC-relative way, so the most common reason for non-relocatability is when the VM instruction implementation performs a call to some function (e.g., for performing I/O).

Accesses to global constants or to global variables in a PC-relative way can also cause non-relocatability. Gforth avoids global variables for that reason and because of multi-threading; it stores some formerly global variables in a struct whose address is stored in a local variable inside `engine()`. However, computing the FP negation and the FP absolute value implicitly involve a constant that resides in memory on AMD64 (with SSE2 FP), making the implementations of these VM instructions (`fnegate` and `fabs`) non-relocatable on this architecture.

The pointer-to-struct approach could also be used for invoking functions without making the calling code non-relocatable, but for now we have not done that.

Note that asking the C compiler for position-independent code does not mean that individual code snippets are relocatable, even though the binary as a whole is, because position-independent

[9] Groups are often confused with bundles, which are IA-64's encoding of three instructions in 128 bits. By contrast, groups can be arbitrarily long, and can start and end somewhere in the middle of a bundle.

code may refer to code or data outside the code snippet in a PC-relative way (and usually does), while a relocatable code snippet must not do this.

Determining relocatability. How do we find out if a code snippet is relocatable or not? The implementations of the VM instructions actually look as follows:

```
L_skip:
  asm("SKIP4");
  asm("SKIP4");
  asm("SKIP4");
  asm("SKIP4");
L_before:
  code snippet in C
L_after:
  asm("SKIP4");
  asm("SKIP4");
  asm("SKIP4");
  asm("SKIP4");
  threaded-code dispatch
```

We compile `engine()` with these pieces to assembly language. Then we assemble the result twice: Once with `SKIP4` defined as empty string, so the `SKIP4`s assemble to nothing, and the result is as discussed earlier; and once with `SKIP4` defined as `.skip 4`, and with `engine` defined as `engine2`, so as a result the object file contains a function `engine2()` that has 16 bytes of padding before and after each code snippet.[10] We link both object files into the final executable. The addresses of the `L_skip` labels are taken and passed outside `engine()`, so gcc cannot optimize the initial skip away as dead code, and also because that usually is the next label after a threaded-code dispatch.

We now have a function `engine()` without the skips before and after the code snippets, and a function `engine2()` that has 16-byte skips before and after each code snippet. We extract the labels from each of the functions, and then compare the code snippets: If a code snippet from `engine()` contains exactly the same bytes as the corresponding code snippet from `engine2()`, then the code snippet is relocatable, otherwise it is not.

How does this work? If code from inside the code snippet references a code or data address outside the code snippet through a PC-relative address, the offset of the relative address will be different between `engine()` and `engine2()`, because the target label will be farther away in `engine2()` thanks to the skips. If there is an absolute reference (e.g., MIPS `j` instruction) to inside the code snippet, it will be different between `engine()` and `engine2()`, because the respective targets are at different addresses.

Even if the code snippet ends in an unconditional branch and the C compiler puts some other code behind that unconditional branch,[11] this scheme works: If the two code snippets compare equal, the code is relocatable. When used in a code-copying system, the code snippet may have some unused code behind the unconditional jump, but the generated code is still correct.

The reason for skipping 16 bytes is that this is a common code-alignment value, so the skips would not result in altered alignment (these days we ask the compiler to align to 1-byte boundaries, so skipping less might be sufficient). The reason for performing the 16-byte skip as 4 4-byte skips

[10] In earlier times we compiled twice rather than assembling twice, but compiling once is faster, and we do not need to worry if the two compilation runs introduce unintended differences in addition to the intended ones.

[11] We have not seen such an occurence yet.

is that for some targets gcc counts the number of instructions in `asm` statements, assumes that each instruction takes at most 4 bytes, and generates code that relies on this assumption.

The absolute target addresses for the MIPS `j` and `jal` instructions have a catch: They work only for targets in the same 256MB segment of the address space. When we last looked, the functions `engine()` and `engine2()` were linked in the same 256MB segment as the functions called by some of the code snippets, and the code snippets would have been classified as relocatable. However, they were only relocatable within this 256MB segment. This is another reason why we disabled code copying for MIPS. An alternative would have been to allocate the memory for the copied code in the same 256MB segment as the original. Fortunately, among the architectures we have looked at, only MIPS has this property.

5 COMPILER ISSUES

In the previous section we have already mentioned a few caveats about how compilers have interfered with our initial assumptions about the generated code, and what we do about that. This section discusses additional issues.

We had quite a few problems with various gcc versions in the 2000s, and for some we found ways to deal with them, while some others were eventually fixed (after reappearing for several years). Also, the rethoric about undefined behaviour started at around that time and has spread and become more aggressive since then,[12], so at some point we expected to have to switch from using GNU C to assembly language as a more reliable foundation at some point [Ert14], essentially switching to a conventional compiler. But this has not happened (yet?), and actually, in the 2010s and 2020s only few new problems have appeared, and we found ways to deal with them. So GNU C seems to be a relatively stable foundation after all, once one has implemented various workarounds.

5.1 Code reordering

When we started, gcc arranged the basic blocks in source order. This changed with gcc-3.2. This has an effect on how we find the next label (Section 4.1). But we also saw cases where the compiler moved basic blocks from between `L_before` and `L_after` to outside these labels, which caused problems.

To avoid such problems, we tried to have only straight-line code in the VM instruction implementations. We extracted loops and most `if`-statements into functions that are compiled separately, and the VM instruction implementation only contains a call to this function. This costs a little performance (from the function call as well as turning the VM instruction implementation into non-relocatable code on most architectures), but fortunately the VM instruction affected by this are executed relatively rarely.

However, conditional VM branches are executed frequently, and in the ideal case they contain a conditional branch, in the following form (also seen for `(+loop)` in Section 3):

[12] http://blog.llvm.org/2011/05/what-every-c-programmer-should-know.html

```
    ... skips ...
  L_before:
    ... stack handling etc. ...
    if (VM_branch_taken)
      ip = ip[-1]; /*VM-branch target*/
      threaded-code dispatch;

  L_after:
    ... skips ...
    threaded-code dispatch;
```

Ideally such VM-instruction implementations are compiled such that the basic blocks in the machine code are in the same order as in the source code, so that the code controled by the `if` is between `L_before` and `L_after`, and the second threaded-code dispatch can be left away by code-copying in the usual case. For now, gcc does it that way for our code. But if gcc ever started changing this, a possible way to steer it back on the right path may be to use `__builtin_expect(VM_branch_taken,` instead of just `VM_branch_taken`.

5.2 Code alignment

Compilers insert padding to align branch targets to instruction-fetch boundaries or cache-line boundaries. In particular, they do this for branch targets behind unconditional branches and loop heads.

When code copying, the padding inserted for the original code is often inappropriate for the target code. Therefore, we suppress this padding by compiling `engine()` with the options `-falign-labels=1 -f -falign-jumps=1`.

Instead, our code-copying implementation performs its own alignment (but on 2007-era processors where we measured the effects, the effects were in the noise).

5.3 Code deduplication

Starting with gcc-3.0, gcc started to compile all the `goto *` instances to an unconditional jump to one instance of an indirect branch. The reason for this probably was to reduce the control-flow edges in the data-flow analysis, for m `goto *` and n labels from nm to $n + m$.

In a number of gcc versions (up to the early gcc-4.x releases), gcc then did not eliminate the unconditional jump afterwards, with some versions eliminating them and some versions regressing, but eventually the gcc maintainers managed to make the unconditional-branch elimination stick, for our code.

So if that is a solved problem, why do we mention it here? We occasionally see this problem reappear in some form, so it's not completely gone.

E.g., when we managed to extend stack-caching support on AMD64 to three registers, we found that on AMD64 gcc compiled the `goto *` to an unconditional branch to common code that contains a lot of register shuffling (with no overall effect) and finally the indirect branch. Apparently the register shuffling made the common code so long that the branch-elimination heuristic decided not to eliminate the branch.

Fortunately, we found out that the register shuffling (and, consequently, the unconditional branch) go away with the compilation option `-fno-tree-vectorize`. Apparently without this option gcc tries to vectorize loads and stores of adjacent values, and is less precise in the data flow analysis for that than for individual values, leading to the register shuffling.

For the problems in the gcc-3.x and 4.x era, Gforth contains a workaround that has just one threaded-code dispatch and jumps there from all the VM instruction implementations. Gforth has labels before and after this dispatch, and because there is only one, gcc does not deduplicate it; this allows Gforth to use it as a code snippet that is appended whenever a threaded-code dispatch is needed.

In order to work with this workaround and still be relocatable, we implemented conditional VM branches to just set the IP on a taken branch, and then continue through `L_after` to the dispatch code. This results in worse code than we would have liked, but it was the best that was possible on these compiler versions. This approach remains an option when building Gforth,

5.4 Code duplication

On our first encounter with gcc-7, we found that the generated code looked as a straightforward compiler would generate for:

```
L_skip:
  ... skipping ...
  code snippet in C;
  threaded-code dispatch;
L_before:
  code snippet in C;
  threaded-code dispatch;
L_after:
  threaded-code dispatch;
```

I.e., gcc-7 duplicated code reached by jumping to a label and the same code being reached in a straight-line way. This may be a useful optimization, but it means that our code snippets now contain the dispatch code, which is contrary to our intentions.

We found the following workaround: In order to convince gcc that this code duplication does not pay off, after each label we insert 8 `asm` statements, each containing a comment with a text unique to that label (so gcc hopefully will not try to deduplicate the code). Currently this is enough to convince gcc to avoid the code duplication

5.5 Register allocation

Virtual machines have a number of "registers", which are implemented in C code as C (local) variables. At least for the frequently-used variables, it would help performance if they were allocated to real-machine registers.

Up to and including gcc-9, we explicitly assigned registers to several of these variables on many platforms with GNU C's feature "Explicit Register Variables". In gcc-10 and later, disabling the explicit register variables produced better results than enabling them.

With either approach, we have the following problem: In the Gforth engine, gcc only used callee-saved registers for these variables. With explicit register variables, because gcc does not accept caller-saved registers for those. But if left to itself, gcc does not use caller-saved variables, either, because `engine()` contains about 100 VM instruction implementations that perform calls, and these calls apparently cause the compiler to avoid using caller-saved registers for these variables, especially for those that are used in < 100 VM instructions, such as the return-stack pointer of Gforth. A problem here is that gcc does not know that VM instructions that access the return stack are used frequently, while VM instructions that perform calls tend to be used rarely. This is a problem even for architectures like Alpha that have a lot of registers in principle, but a calling convention with relatively few callee-saved registers.

For being able to use additional registers for stack caching without spilling other VM registers, we use the following observation: All VM instruction implementations that contain a call only use the canonical state with one stack item in a register, due to non-relocatability. So additional stack cache registers are dead at the end of these VM instruction implementations, and there is no reason to preserve these registers across the calls. But how do we tell gcc about that?

```
L_skip:
  ... skipping ...
L_before:
  code snippet containing a call;
  asm(""":"=X"(spb));
  asm(""":"=X"(spc));
L_after:
  threaded-code dispatch;
```

The empty `asm` statements right before `L_after` claim to overwrite `spb` and `sbc` (the variables holding the additional stack-cache items in some stack-cache states). Therefore, these variables are dead at the call and do not need to be preserved. This means that this VM instruction implementation is no hindrance to allocating `spb` and `spc` in a caller-saved register. And indeed, one of these variables is allocated by gcc in a caller-saved register.

Another way to influence the register allocator that we have not used is the GNU C extension "Label Attributes" (available since gcc-5). We can declare the VM instruction implementations with calls as being `cold`, and/or declare frequently-used VM instruction implementations to be `hot` by following the label with an attribute:

```
L_skip:
  ... skipping ...
L_before: __attribute__((cold));
  code snippet containing a call;
L_after:
  threaded-code dispatch;
```

With that, the register allocator is hopefully more willing to use caller-saved registers for local variables of the VM.

5.6 Cache consistency

Many architectures do not guarantee cache consistency between data and instruction caches, and require a special piece of code between generating code and executing code; this incantation typically consists of a few lines of architecture-specific (or, on some architectures worse, implementation-specific or OS-specific) code, and for a long time has been the only non-portable part of Gforth's code copying implementation. Gcc-4.3 introduced `__builtin___clear_cache()`, which would eliminate this last piece of non-portability. We use `__builtin___clear_cache()` on RISC-V.

Unfortunately, `__builtin___clear_cache()` is not implemented correctly on at least PowerPC64.[13] We have switched Gforth back to using architecture-specific implementations of this functionality (except on RISC-V). When implementing your own code-copying compiler, check if `__builtin___clear_cache()` is compiled to non-empty code on each architecture that requires special code to make the caches consistent. If it compiles to non-empty code, that code will hopefully be correct.

[13] https://gcc.gnu.org/bugzilla/show_bug.cgi?id=93811

Another problem with such architectures is multi-threading: The code-generating thread must ensure that the D-cache lines are written to a common memory, and then the code-executing threads must invalidate these regions in the I-cache (to get rid of stale I-cache lines); due to prefetching and branch prediction, this may even be necessary if code in the address range has never been executed.

Until now we have ignored this problem, and relied on our luck. Typically Gforth programs only start subthreads after finishing compiling the source code (and thus code generation), which may explain why we have not seen any problems from that. A system with on-demand code generation (the narrow meaning of JIT) may be more likely to encounter such problems, however.

5.7 Spectre

GCC offers mitigations against Spectre v2 [KHF+19]. While all of these mitigations are expensive, because they disable indirect-branch prediction, the option `-mindirect-branch=thunk-inline` is less expensive than `-mindirect-branch=thunk`, because the latter makes the code snippets non-relocatable, so every VM instruction performs an indirect branch, while with the former option the relocatability of the code snippets is not affected, resulting in fewer indirect branches and therefore less slowdown.

On a Ryzen 3900X, we see slowdowns by a factor of 2.1–7.6 from using `-mindirect-branch=thunk-inline` and slowdown factors of 7.5–18.1 from using `-mindirect-branch=thunk`.

However, if you want to implement your programming language with Spectre mitigations, you will prefer approaches such as copy-and-patch compilation that avoid performing so many indirect branches. You will also want to use mitigations against other Spectre vulnerabilities (e.g., speculative load hardening [ZBC+23] against Spectre v1), which will introduce additional slowdowns for any approach, but unfortunately, these other mitigations require more work than just setting a C compiler flag.

5.8 Control-flow protection

There are exploit techniques such as return-oriented and jump-oriented programming that work by returning or jumping to arbitrary code. To make it more difficult to use these techniques, architectures and compilers offer ways to check that branches and returns only jump to targets that the compiler had in mind. E.g., gcc with the option `-fcf-protection=full` inserts an `endbr64` instruction at every indirect-branch target (i.e., every label in `engine()`), and the CPU can be told to report an error on an indirect branch to some other code. `Endbr64` is an AMD64 instruction, some other architectures have similar features.

This works with code copying: It copies the `endbr64` instruction to those places that the dispatch code will later indirect-branch to (and to additional places).

We use `-fcf-protection=none` in Gforth, however, because Gforth offers enough gadgets[14] already at the intended targets of indirect branches: All the VM instructios; moreover, Gforth and its VM is a low-level language that allows arbitrary memory access within the process. So a Gforth program that is exposed to untrusted input has to successfully defend against an attacker at the front line (source-level bounds checks etc.) and cannot make life harder for the attacker who has breached the front-line defense.

However, if your language is better suited to defense-in-depth, you can enable `-fcf-protection=full`, and they will work with code copying. This feature may cost a little

[14] In the context of return-oriented and jump-oriented programming, a gadget is a machine-code sequence that an attacker may want to return/jump to.

performance, though: All the `endbr64` instructions need to be decoded and executed. In a small experiment with Gforth on a Ryzen 8700G (Zen4), we saw an increase in instruction count by a factor 1.45 and an increase in cycle count by a factor 1.04 from `-fcf-protection=full`. Narrower processors may see a bigger slowdown (the instructions per cycle on Zen4 increased from 3.83 to 5.34). VM implementations with more machine instructions per VM instruction will see a smaller effect.

5.9 Clang

Clang supports "Labels as Values", and Gforth is built with clang on platforms where GCC is not available. However, using Clang poses a number of problems:

- Clang wants to understand the assembly language in `asm` statements, and stops compiling when it sees `asm("SKIP4")`. One can work around that, and that is done in the ports that need clang, but we have not done that for the experiments on Debian Linux in the following.
- Clang takes much longer than gcc to compile Gforth's `engine()` and also needs more memory. As an example, for `gforth-itc` (an indirect-threaded-code Gforth without code copying nor other optimizations, and therefore without `SKIP4`), on a Ryzen 5800X gcc-12.2 takes 3s and 346MB to compile `engine()`, while clang-14.0.6 takes 699s and 5603MB. For `engine()` for `gforth-fast` (with all optimizations enabled), clang takes 3399s and 18264MB before it stops compiling because of `SKIP4` (gcc takes 26s and 1804MB).
- Clang generates a lot of register and memory shuffling code, similar to what we have seen with gcc-3.0. As a result, runnung the small benchmarks on Clang-compiled `gforth-itc` executes 6.4 times more AMD64 instructions than on GCC-compiled `gforth-itc` and consumes 4.2 times more Ryzen 5800X cycles.

As a result, Gforth selects GCC whenever it can. We expect that the clang compilation speed will be a problem for other code-copying compilers. The bad code generation may be less pronounced in language implementations that rely less on copy propagation than Gforth. Clang may be more viable when using tail calls instead of using one function and "Labels as Values" (see Section 7.1).

6 OS ISSUES

Over the years operating systems have restricted executing dynamically-generated code more and more. In the beginning, all memory was allocated with read, write, and execute (RWX) permissions; later, `malloc()` only allocated RW memory, and one has to use `mmap()` to get RWX memory.

Recently, some operating systems (in partcular MacOS on Apple silicon) do not serve `mmap()` calls that ask for RWX memory (this restriction is also known as `W^X`). This is a problem for all systems with run-time code generation, not just code-copying compilers, but, e.g., Java JITs as well. For a single-threaded language implementation, one can `mprotect()` the memory to W when generating the code, and to X when executing it, but that does not work for multi-threaded code, unless you want to start a new page whenever you generate a new piece of code.

MacOS provides a MacOS-specific API for JIT compilers that supports switching the memory into W in the code-generating thread and keeping it X in the other threads, and Bernd Paysan has actually invested the time to use this API.

Several of the BSDs also has `W^X` by default, but allows to mark binaries such that RWX works. The command for marking the binary is short, but specific to the BSD variant.[15]

An approach that may work without special APIs is to have the code generation in one process and the execution in a different process, both mapping the same memory, but with different permissions. Another option may be to map the same memory within one process twice, at one

[15] https://www.reddit.com/r/BSD/comments/10isrl3/notes_about_mmap_mprotect_and_wx_on_different_bsd/

address range with W permission, and at the other address range with X permission. We have not tried either approach.

If all else fails or you don't want to jump through the hoops that these operating systems put up, code-copying based on threaded code always allows you to fall back to plain threaded code, which works fine on operating systems with the `W^X` restriction. E.g., Gforth-0.7 (which was not specifically designed for this circumstance) automatically falls back to plain threaded code on MacOS on Apple silicon: the `mmap()` call for allocating the code memory fails, so Gforth-0.7 falls back to using `malloc()`, and because that does not produce executable memory on modern OSs, Gforth-0.7 turns off dynamic code generation.

7 ALTERNATIVE APPROACHES

In this section we describe approaches that are interesting but that are not implemented in production Gforth.

7.1 Tail calls

Instead of putting all VM-instruction implementations in one function and using `goto *` for threaded-code dispatch, one can also put each VM instruction implementation in a separate function and use optimized tail-calls for threaded-code dispatch, as follows:

```
typedef void (*vm_inst)(void **ip, long *dsp, long tos);

void lit(void **ip, long *dsp, long tos)
{
  ... payload including ip update ...;
  (*(((vm_inst *)ip)[0]))(ip,dsp,tos);
}
```

The last line of the function performs the threaded-code dispatch. The tail-call must be optimized into a jump, otherwise the C stack grows and eventually overflows. When we first considered this approach [Ert95], GCC did not tail-call optimize such code, but in the meantime it does, as does Clang [XK21]; Clang even provides a way to require that a call is tail-call-optimized, and will report an error if it cannot meet this requirement.

The VM registers are passed as parameters, at least as long as the calling convention supports passing them in machine registers. With gcc, additional VM registers could be stored in global explicit register variables; on AMD64 this results in 12 general-purpose and 8 floating-point registers available for VM registers. Clang does not support explicit register variables, but it supports using a calling convention for these functions and calls that uses as many registers as possible for parameter-passing.

So for dealing with VM registers efficiently, one has to pass VM-registers in parameters or keep them in global register variables with compiler-dependent and ABI-dependent code, but that is a relatively small effort.

With the tail-calling approach, there is a fixed allocation of VM registers to machine registers, either coming from the position in the parameter list, or from the explicit register allocation.

We expect that the VM instruction implementations can be compiled faster and with less memory with the tail-calling approach, because the compiler will hopefully not try to perform data-flow analysis between the functions, while it tries to do it when the implementations are all contained in one function. We can then squander the compilation speed gain on introducing more code snippets, for various optimization purposes (Xu and Kjolstad report using 98831 code snippets [XK21]).

Another benefit is that we should see no or little of the register-and-memory shuffling that we see with Clang, or with gcc without `-fno-tree-vectorize`.

So far you have only seen how tail calls can be used to implement threaded code. How can it be used for code-copying compilation?

In order to do that, we need a way to get rid of the dispatch part of the implementation. Unfortunately, compilers tend to mix the instructions from the payload part with those from the dispatch part; just inserting a label between them will not work, because there is nothing that jumps to this label. Maybe an `asm` statement can be made to act as a barrier, but preliminary experiments failed to produce satisfying results.

One way that may be more promising is to have, in addition to functions that end in a threaded-code dispatch (to have a fallback option), variants intended only for code copying that end in a direct [XK21]) or indirect tail-call without threaded-code dispatch. On many architectures this is just one instruction, that must be last in the function. However, there are exceptions: Some architectures have delayed branches (HPPA, MIPS, SPARC); some architectures require two instructions for indirect branches (PowerPC, IA-64). In some programming models, a direct jump to a function is expressed as an indirect jump to a target loaded from the global offset table (GOT), and as a result the direct jump also is expressed with more than one instruction.

Once we have solved the problem of keeping the payload separate from the tail call, how do we know where the tail call starts so that we can use the code between the start of the function and this instruction as code snippet? Xu and Kjolstad extract the function size (and the code) from the object file (see Section 7.2), and apparently use their own architecture-specific knowledge about the size of the last instruction to determine where it starts. A way to determine the size of this last instruction may be to have a function that performs only this tail-call, and look at its size.

7.2 Snippets from object files

Gforth extracts code snipets from the executable at run-time and has some startup overhead while it examines all the code snippets for relocatability and performs its table setup.

An alternative is to extract code snippets from object files [NHCL98, XK21] at system build time using the Binary File Descriptor library (GNU BFD). One advantage of this approach is that the object file contains additional information, such as the function size, or linkage information for symbols external to the object file.

7.3 Copy-and-patch compilation

Gforth accesses immediate operands and control-flow information through IP. This requires a register for IP, results in less efficient accesses to immediate operands and less efficient control flow than with ordinary compilers, and requires keeping the VM code around.

An alternative is to have code snippets that contain dummy immediate arguments and perform control flow directly to dummy targets, and then patch the constants or target addresses in these code snippets with the actual values, resulting in copy-and-patch compilation.

One approach for copy-and-patch compilation has been based on using the linkage information in object files [NHCL98, TCL+00, XK21]. References to external symbols are used for patchable immediate operands and patchable control-flow targets. The linkage information describes where to patch and how to patch (e.g., absolute or relative address). This requires some architecture/ABI-specific work, but ABIs have a finite number of relocation types (e.g., 52 in the AMD64 ABI [LMG+]) and only a few are actually used in the code snippets.

However, by refering to an external symbol the copy-and-patch compiler usually cannot patch the immediate operand of instructions like RISC-V's `addi`. The external symbol is a 64-bit (or

32-bit) value, while the immediate operand of `addi` is 12 bits long, so the addition of a constant (whatever its size) is compiled to several instructions.

Another approach is to start with code snippets delimited by labels in one C function, like Gforth's code copying uses, but perform patching in addition [VA04, EG04b].

We implemented copy-and-patch compilation for Gforth in a prototype for IA-32 and PowerPC using the latter approach [EG04b]. This work was based on Gforth's approach of extracting code snippets from the executable at system startup time. The `engine()` function was compiled thrice, twice with the same immediate arguments, and once with different immediate arguments. The first two versions were compared to determine relocatability, the third version was compared to find out the placeholders of the immediate arguments.

This approach can make use of the RISC-V `addi` instruction, but needs to fall back to code that uses several instructions when the immediate operand becomes too large. It needs quite a bit of knowledge about the instruction encodings, in particular, the sizes of the immediate-operand fields. We considered determining the encoding and size by varying the immediate operands a lot more, but did not implement that idea; dealing with each architecture manually is probably less work.

We originally intended to turn this copy-and-patch compiler into a production engine for Gforth, but in those years several GCC releases resulted in falling back to threaded code, so the copy-and-patch approach looked too brittle, and we let it bit-rot. Later, the rethoric by the advocates of C code without undefined behaviour kept the distrust in GCC high. If we had continued to maintain this engine, maybe we could now report on its success and the hurdles we had to overcome. Or maybe it would have been a bridge to far.

8 RELATED WORK

GCC-2.0 (released February 1992) introduced "Labels as Values", which not only proved useful for implementing threaded code (we started the Gforth project[Ert93] in July 1992), but also for compiling by copying compiler-generated code snippets between two labels, with all the code snippets being within a function. This method was first outlined by Rossi and Sivalingam [RS96, Section 2.5], who refer to an unpublished discussion between Xavier Leroy and Kenneth Oksanen. Piumarta and Riccardi provided a more elaborate treatment [PR98], with deduplication of code sequences.

Ertl and Gregg implemented code-copying in Gforth, and in the beginning the main benefit was in indirect branch prediction accuracy [EG03a, EG03b, CEG07]; it turned out that leaving away deduplication (or conversely, introducing replication, as we framed it) helped the branch predictors at the time. Indirect branch predictors have improved a lot in general-purpose processors [RSS15], but code copying still provides a good speedup.[16]

Once you have code copying, you can eliminate instruction-pointer (IP) updates, either by leaving away the unneeded VM instruction slots [PR98], or by replacing several IP updates with a combined one [EP24]. While IP updates play a minor role for performance on CPUs from the 2000s, they can be the decisive bottleneck on loop-dominated benchmarks in the 2020s.

Another optimization that was facilitated by code copying is multi-state stack caching [Ert95, EG04a, EG05].

Tempo is a partial evaluator that uses code copying and patching by extracting information from object files [NHCL98]; Tempo was later used to specialize an interpreter into a compiler [TCL+00].

Iliasov [Ili03] describes a copy-and-patch compiler with a minimal patching component: Only literals need to be patched; control flow is performed by performing indirect jumps to addresses provided as literals.

[16] See Section 2.1 and http://www.complang.tuwien.ac.at/anton/interpreter-branch-pred.txt.

QEMU is a full-system emulator. It is a production system with a long history, and has many more users than Gforth. QEMU can emulate machines with a different instruction set than the host machine. It uses dynamic translation techniques for that, originally implemented in its Dyngen component [Bel05] using code-copying and patching, similar to what we described in Section 7.2 and 7.3. But Dyngen uses ordinary functions, not tail-calling functions, and has to get rid of the function prologue and epilogue. Dyngen is gcc-3.x-specific, and it apparently was too difficult to adapt it to newer gcc versions or other compilers, so it was replaced with TCG in QEMU-0.10.0 released in 2009. TCG is based on QOP by Paul Brook, who described it as "Hand written code generator"[17], so TCG probably is not based on copying and pasting compiler-generated code.

In Gforth we have dealt with changes in GCC by finding workarounds, or, for versions where we were not successful, by falling back to threaded code. Another approach is to actually define the properties that a compiler's code generation should have to support code copying; then modify a compiler to provide those properties (when asked for it), and report an error if it fails to provide the properties. This approach has been explored by Prokopski and Verbrugge [PV07, PV08], but their patches have not been integrated into GCC.

Several code-copying JavaVM implementations have been implemented, among them SableVM [GH03] and the Cacao interpreter [ETK06]. A particular challenge solved by these implementations was quickening of VM instructions, where VM instructions rewrite themselves into faster code on first execution. SableVM stopped being maintained after the research project ended (last release 2007). The Cacao interpreter bit-rotted while the main thrust of Cacao continued to use conventional code generation technology.

Maxine is a Java VM implementation with two-level compilation (baseline and optimizing compiler), where the baseline compiler is a copy-and-patch compiler that uses templates written in Java and where the code is generated by the optimizing compiler (which uses conventional compiler techniques) or by HotSpot [WHV+13].

Xu and Kjolstad implement two copy-and-patch compilers: One that directly compiles from the abstract syntax tree (AST) without going through a VM and one for WebAssembly. Their technique works by having each code snippet (called stencil in the paper) in a tail-calling function with references to external symbols as placeholders for patching, and extracting the code snippets from object files. They use 1666 code snippets for the WebAssembly compiler, and 98831 code snippets for the AST compiler; the latter is notable, because it is beyond practical for the technique where all code snippets are in one function.

9 CONCLUSION

Code-copying compilers make retargeting of the compiler much easier by using code snippets coming from a different compiler. Gforth demonstrates that code-copying without patching can produce code with similar performance as a compiler with a hand-written architecture-specific code generator. Gforth has used code copying since 2003, on many architectures, and has dealt with many GCC versions in those years. If all else fails, Gforth can fall back to threaded code, but it usually does not have to.

Copy-and-patch compilation promise an improvement in performance over copying without patching (as in Gforth) at a moderate increase in architecture-specific code. However, while there have been a number of publications about this technology, no production system is known to us that currently uses it.

[17] https://qemu-devel.nongnu.narkive.com/bCtjCaPs/hand-written-code-generator-2

REFERENCES

[Bel73] James R. Bell. Threaded code. *Communications of the ACM*, 16(6):370–372, 1973.

[Bel05] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *Freenix Track of Usenix Annual Technical Conference*, pages 41–46, 2005.

[CEG07] Kevin Casey, M. Anton Ertl, and David Gregg. Optimizing indirect branch prediction accuracy in virtual machine interpreters. *ACM Transactions on Programming Languages and Systems*, 29(6):37:1–37:36, October 2007.

[EG03a] M. Anton Ertl and David Gregg. Optimizing indirect branch prediction accuracy in virtual machine interpreters. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI'03)*, 2003.

[EG03b] M. Anton Ertl and David Gregg. The structure and performance of *Efficient* interpreters. *The Journal of Instruction-Level Parallelism*, 5, November 2003. http://www.jilp.org/vol5/.

[EG04a] M. Anton Ertl and David Gregg. Combining stack caching with dynamic superinstructions. In *Interpreters, Virtual Machines and Emulators (IVME '04)*, pages 7–14, 2004.

[EG04b] M. Anton Ertl and David Gregg. Retargeting JIT compilers by using C-compiler generated executable code. In *Parallel Architecture and Compilation Techniques (PACT' 04)*, pages 41–50, 2004.

[EG05] M. Anton Ertl and David Gregg. Stack caching in Forth. In M. Anton Ertl, editor, *21st EuroForth Conference*, pages 6–15, 2005.

[EGKP02] M. Anton Ertl, David Gregg, Andreas Krall, and Bernd Paysan. vmgen — a generator of efficient virtual machine interpreters. *Software—Practice and Experience*, 32(3):265–294, 2002.

[EP24] M. Anton Ertl and Bernd Paysan. The Performance Effects of Virtual-Machine Instruction Pointer Updates. In Jonathan Aldrich and Guido Salvaneschi, editors, *38th European Conference on Object-Oriented Programming (ECOOP 2024)*, volume 313 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 14:1–14:26, Dagstuhl, Germany, 2024. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[Ert93] M. Anton Ertl. A portable Forth engine. In *EuroFORTH '93 conference proceedings*, Mariánské Láznè (Marienbad), 1993.

[Ert95] M. Anton Ertl. Stack caching for interpreters. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI'95)*, pages 315–327, 1995.

[Ert14] M. Anton Ertl. How to get rid of C. In *30th EuroForth Conference*, pages 63–65, 2014.

[Ert24] M. Anton Ertl. Interpreter vs. compiler performance at run-time. In *Tagungsband des Jahrestreffens 2024 der GI-Fachgruppe "Programmiersprachen und Rechenkonzepte"*, INSIGHTS — Schriftenreihe der Fakultät Technik, pages 7–12, 2024.

[ETK06] M. Anton Ertl, Christian Thalinger, and Andreas Krall. Superinstructions and replication in the Cacao JVM interpreter. *Journal of .NET Technologies*, 4:25–32, 2006. Journal papers from *.NET Technologies 2006* conference.

[GH03] Etienne Gagnon and Laurie Hendren. Effective inline-threaded interpretation of Java bytecode using preparation sequences. In *Compiler Construction (CC '03)*, volume 2622 of *LNCS*, pages 170–184. Springer, 2003.

[Ili03] Alex Iliasov. Templates-based portable just-in-time compiler. *SIGPLAN Notices*, 38(8):37–43, August 2003.

[KHF+19] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[LMG+] H.J. Lu, Michael Matz, Milind Girkar, Jan Hubička, Andreas Jaeger, and Mark Mitchell, editors. *System V Application Binary Interface — AMD64 Architecture Processor Supplement (With LP64 and ILP32 Programming Models)*.

[NHCL98] François Noël, Luke Hornof, Charles Consel, and Julia L. Lawall. Automatic, template-based run-time specialization: Implementation and experimantal study. In *IEEE International Conference on Computer Languages (ICCL '98)*, pages 123–142, 1998.

[PR98] Ian Piumarta and Fabio Riccardi. Optimizing direct threaded code by selective inlining. In *SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 291–300, 1998.

[PV07] Gregory B. Prokopski and Clark Verbrugge. Towards GCC as a compiler for multiple VMs. In *Proceedings of the GCC Developers' Summit*, pages 117–129, 2007.

[PV08] Gregory B. Prokopski and Clark Verbrugge. Compiler-guaranteed safety in code-copying virtual machines. In *Compiler Construction (CC'08)*, pages 163–177. Springer LNCS 4959, 2008.

[RS96] Markku Rossi and Kengatharan Sivalingam. A survey of instruction dispatch techniques for byte-code interpreters. Technical Report TKO-C79, Faculty of Information Technology, Helsinki University of Technology, May 1996.

[RSS15] Erven Rohou, Bharath Narasimha Swamy, and André Seznec. Branch prediction and the performance of interpreters — don't trust folklore. In *Code Generation and Optimization (CGO)*, 2015.

[TCL+00] Scott Thibault, Charles Consel, Julia L. Lawall, Renaud Marlet, and Gilles Muller. Static and dynamic program compilation by interpreter specialization. *Higher-Order and Symbolic Computation*, 13(3):161–178, September 2000.

[VA04] Benjamin Vitale and Tarek S. Abdelrahman. Catenation and specialization for Tcl virtual machine performance. In *IVME '04 Proceedings*, pages 42–50, 2004.

[WHV+13] Christian Wimmer, Michael Haupt, Michael L. Van De Vanter, Mick Jordan, Laurent Daynès, and Douglas Simon. Maxine: An approachable virtual machine for, and in, Java. *ACM Transactions on Architecture and Code Optimization*, 9(4):30:1–30:24, January 2013.

[XK21] Haoran Xu and Fredrik Kjolstad. Copy-and-patch compilation. *Proc. ACM Program. Lang.*, 5(OOPSLA):136:1–136:30, October 2021.

[ZBC+23] Zhiyuan Zhang, Gilles Barthe, Chitchanok Chuengsatiansup, Peter Schwabe, and Yuval Yarom. Ultimate SLH: Taking speculative load hardening to the next level. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 7125–7142, Anaheim, CA, August 2023. USENIX Association.

Designing Real-time Mission-critical Systems with the TeamPlay Coordination Language

CLEMENS GRELCK, Friedrich-Schiller-Universität Jena, Deutschland

It is estimated that 98% of the world's computing devices operate in embedded or cyber-physical systems, where non-functional properties of program execution, such as energy (budgets), time (budgets), security or fault-tolerance can be as crucial as functional correctness. The rise of the internet-of-things with the edge-fog-cloud computing continuum and the increasing heterogeneity and parallelism of computing platforms rather solidify than change the need for resource-aware software. These developments create new challenges for software engineering.

We introduce the coordination language TeamPlay that introduces energy, time, security and fault-tolerance as first-class citizens into the software design process. Following the concept of exogeneous coordination, TeamPlay enforces a stringent software architecture with strict separation of concerns between operational detail and application-level design. We discuss selected aspects of the TeamPlay compiler and runtime environment as well as the scheduling and mapping problem that today's heterogeneous parallel architectures for the cyber part of cyber-physical systems create.

Author's address: Clemens Grelck, Friedrich-Schiller-Universität Jena, Jena, Deutschland, clemens.grelck@uni-jena.de.

Advanced Syntax Extensions with MOSTflexiPL

CHRISTIAN HEINLEIN, Hochschule Aalen (University of Applied Sciences), Germany
christian.heinlein@hs-aalen.de

MOSTflexiPL is a general-purpose programming language, whose syntax can be freely extended and customized by every programmer. Based on a small set of predefined operators, it is possible to define new operators with arbitrary syntax, which do not only cover prefix, infix, and postfix operators, but also control structures, type constructors, and declaration forms. While the author's KPS 2023 paper gave an overview of major concepts of MOSTflexiPL, this paper focuses on more advanced syntax extensions which are possible with MOSTflexiPL.

1 Introduction

MOSTflexiPL, which is an acronym for **mo**dular, **s**tatically **t**yped, **fl**exibly **ex**tensi**i**ble **p**rogramming **l**anguage, is a general-purpose programming language, whose syntax can be freely extended and customized by every programmer. The logo used in the paper title shall express the extreme flexibility provided by the language allowing even fancy constructions unimaginable with conventional languages. (Therefore, it might be advisable to forget almost all familiar and seemingly necessary limitations of other languages to be able to fully recognize MOSTflexiPL's capabilities.)

A basic principle enabling that flexibility is: Everything is an expression, i. e., the application of an operator to subexpressions, where operators might possess any number of names and operands in an arbitrary order. Apart from well-known prefix, infix, and postfix operators, this also includes "circumfix" operators such as `(•)` (an operand depicted by the bullet sign • enclosed in parentheses), control structures such as `if•then•else•end`, declaration forms such as `•:•` (a name and a type separated by a colon), and so on. Another basic principle is, that the language provides only a small set of predefined operators covering arithmetic and logic operations as well as basic control structures, which can be used to define arbitrary new operators.

As the name indicates, the language is statically typed – which imposes numerous challenges with respect to the already mentioned flexibility –, and it is currently implemented by a compiler and a run-time system written in C++.

While the author's KPS 2023 paper [5] gave an overview of major concepts of MOSTflexiPL, the primary goal of this paper is to show examples of more advanced syntax extensions which are possible with MOSTflexiPL. To make the current paper self-contained, several parts of the previous paper, which are necessary to understand the advanced examples, are repeated.

2 Simple Operator Declarations

To give a first example, the following simple declarations define operators computing the square and the absolute value, respectively, of an integer value `x`, which can afterwards be applied using well-known mathematical syntax, e. g., 5^2 or $|2-7|^2$:

```
(x:int) "²" -> (int = x * x);
"|" (x:int) "|" -> (int = if x > 0 then x else -x end)
```

An operator declaration generally consists of a *signature*, an arrow and a *result declaration*, where the signature is a sequence of *names* and *parameter declarations*, while the result declaration consists of a *type* (the result type of the operator), an equality sign, and the *implementation* of the operator, enclosed in parentheses. A name is either a sequence of letters and digits starting with a letter (denoting exactly this sequence of characters, e. g., `abc`) or a sequence of arbitrary characters en-

closed in quotation marks (denoting this sequence of characters without the quotation marks, e. g., `"²"` denoting ²). A parameter declaration consists of a name, a colon, and a type, enclosed in parentheses. Finally, the implementation and the types mentioned above are – according to the basic principle mentioned in Sec. 1 – expressions. (At the moment, types are atomic expressions such as `int` or `bool`, but see Sec. 3 and Sec. 5 for more complex type expressions.)

When an operator application such as `|2-7|²` is evaluated at run time, the parameters of the operator are initialized from left to right by recursively evaluating the corresponding operands, and then the value of the expression is determined by evaluating the implementation of the operator.

In the examples above, the implementation of the square operator uses the predefined multiplication operator `•*•`, while the implementation of the abs operator uses the predefined change sign operator `-•` as well as the conditional operator `if•then•else•end` that returns, according to the truth value of its first operand, either the value of its second or its third operand.

The semicolon used to separate the two operator declarations is a simple predefined infix operator that evaluates its left and right operand and returns the value of the latter and, therefore, is typically used to denote sequential execution of subexpressions. But – again according to the basic principle mentioned in Sec. 1 – since declarations are expressions, too, the semicolon is also used to separate multiple declarations. In contrast to many other programming languages, however, semicolon must not be used at the end of a sequence of subexpressions, because it is an infix operator.

To give another example, the following declaration defines an operator that recursively computes the factorial of an integer value `n`, which can also be applied using well-known mathematical syntax, e. g., `5!` or `5²!`:

```
(n:int) "!" -> (int = if n <= 1 then 1 else (n-1)! * n end)
```

The particular challenge for the compiler with a declaration like this is to already recognize and accept the new syntax defined by the declaration inside of its own implementation to allow recursive applications of the operator.

3 Constants and Variables

A declaration of the form *`name : type = init`* declares a constant with the given name and type whose value is obtained by evaluating the initializer expression *`init`*, e. g., `N : int = 5²`. If the type is omitted, e. g., `N := 5²`, it is automatically deduced from the type of the initializer. If the initializer is omitted, the constant receives a unique new "synthetic" value that is different from every other value of the type. While this is of limited usefulness for numeric types such as `int`, it is crucial for variable types described below and for user-defined types described in the KPS 2023 paper [5].

For any type `T`, the type `T?` denotes memory cells containing values of type `T`. Therefore, a declaration such as `x : T?` defines `x` as a constant referring to a unique new memory cell that contains a value of type `T`, i. e., `x` actually denotes a variable with content type `T`. The current value contained in such a variable can be queried with the prefix question mark operator `?•`, and it can be changed with the assignment operator `•=!•`. The initial value of a variable is *nil*, which is a predefined value for any type denoting the absence of a real value, that is different from every other value of the type. Because a variable with content type `T` is itself a value of type `T?`, it might itself be stored in a variable of type `T??`. If the content of such a variable is queried prior to any assignment to the variable, the returned value is the nil value of type `T?`. If the content of this nil variable is queried in turn, it will be the nil value of type `T`, and assigning any value to such a nil variable has no effect.

(This behaviour is roughly comparable to reading and writing the Unix special file `/dev/null`: read operations return EOF, i. e., nothing, and write operations are discarded.)

By using variables and the predefined loop operator `while•do•end`, the factorial operator mentioned in Sec. 2 can also be implemented in a more procedural style:

```
(n:int) "!" -> (int =
    f : int?; f =! 1;
    i : int?; i =! 2;
    while ?i <= n do
        f =! ?f * ?i;
        i =! ?i + 1
    end;
    ?f
)
```

In the implementation of the operator, the variables `f` and `i` are declared and assigned their initial values as described above, where `i` is used as a loop counter running from `2` to `n`, while `f` accumulates the factorial value that is finally returned.

4 Optional, Alternative, and Repeatable Syntax Parts

To provide even more syntactic flexibility, the signature of an operator declaration might also contain optional, alternative, and repeatable parts using well-known EBNF syntax.

For example, the following declaration defines a variadic maximum operator that can be applied to any number of operands, e. g., `max of 1`, `max of 1 and 2`, `max of 1 and 2 and 3`, and so on:

```
max of (x:int) { and (y:int) } -> (int =
    m : int?; m =! x;
    { if y > ?m then m =! y end };
    ?m
)
```

According to EBNF, the curly brackets in the signature indicate that an application of this operator might contain the word `and` followed by an operand corresponding to the parameter `y` any number of times (zero or more). To access the different values of this parameter in the implementation of the operator, a corresponding curly bracket operator `{•}` is provided there, whose operand is repeatedly evaluated for every value of `y`. For the particular application `max of 1 and 2 and 3` this means, that the variable `m` declared in the implementation is initialized with the value of `x` (i. e., `1`), and then the `if` expression inside the curly brackets is evaluated in turn for `y` equal to `2` and to `3`, changing the value of the variable `m` to `2` and to `3`, respectively. Finally, the resulting value of `m` is returned.

To give another example, the following operator performs arbitrary calculations consisting of additions and subtractions, e. g., `calc minus 1 plus 2` or `calc 1 minus 2 plus 3`:

```
calc [minus] (x:int) { (plus|minus) (y:int) } -> (int =
    res : int?;
    res =! [-x | x];
    { res =! (?res + y | ?res - y) };
    ?res
)
```

In addition to the curly brackets denoting a repeatable part, the signature of this operator also contains square brackets denoting an optional part as well as round brackets containing two or more alternative parts separated by vertical bars. To find out in the implementation of the operator, whether the optional word `minus` after the word `calc` is present or not in a particular application of the operator, a corresponding square bracket operator `[•|•]` is provided, whose first or second operand, respectively, is evaluated accordingly. Similarly, a round bracket operator `(•|•)` with two operands corresponding to the round brackets with two alternatives in the signature is provided, whose first or second operand is evaluated according to whether the first or second alternative has been chosen in a particular application of the operator or – because in this example the round brackets are nested inside the curly brackets – in the respective pass through the curly brackets. The result value of a square or round bracket operator is the value of the operand that has been evaluated, while the result value of a curly bracket operator is the number of passes through these brackets. Taken together, these bracket operators allow the implementation of the operator to exactly determine the structure of a particular operator application and to process the values of its operands in a rather concise manner.

Generally speaking, all three kinds of brackets can have any number of alternatives separated by vertical bars, except that round brackets must contain at least two, because round brackets with just one alternative are useless. Therefore, the corresponding bracket operators provided in the implementation of the operator have a corresponding number of operands separated by vertical bars, where the *i*-th operand is evaluated if the *i*-th alternative has been chosen in a particular operator application or pass through curly brackets. As an exception, an operator corresponding to square brackets has an additional optional operand, that is evaluated (if it is present) if none of the alternatives has been chosen.

Therefore, the `calc` operator could also be defined as follows:

```
calc [minus] (x:int) { plus (y:int) | minus (z:int) } -> (int =
    res : int?;
    res =! [-x | x];
    { res =! ?res + y | res =! ?res - z };
    ?res
)
```

5 Generic Operators

If an optional parameter appears in the type of another parameter of the same operator, its value can be automatically deduced from the type of the operand corresponding to the other parameter and, therefore, the former parameter is called a *deducible parameter*. This can be used to define generic operators similar to C++ templates and Java generics, for example:

```
$$ Definition.
[(T:type)] (x:T?) "<->" (y:T?) -> (T? =
    z := ?x; x =! ?y; y =! z; y
);

$$ Application.
v1 : int?; v1 =! 1;
v2 : int?; v2 =! 2;
v1 <-> v2
```

Because `v1` and `v2` both have type `int?`, `v1 <-> v2` is a correct application of the previously defined swap operator, where the parameters `x` and `y` are initialized with the explicit operands `v1`

and v2, respectively, while the optional parameter T is implicitly initialized with the type int causing the type int? of the operands v1 and v2 to match the type T? of the corresponding parameters x and y. The implementation of this operator swaps the values contained in the variables x and y and returns the variable y.

6 Lambda Parameters

6.1 Problem

The following example shows an operator defining the syntax of for loops as well as a typical application of the operator:

```
$$ Definition.
for (var:int?) "=" (lower:int) ".." (upper:int) do
                                    [(B:type)] (body:B) end -> (int =
    var =! lower;
    while ?var <= upper do
        body;
        var =! ?var + 1
    end
);

$$ Application: Should print the numbers 1 to 10.
i : int?;
for i = 1 .. 10 do
    print ?i
end
```

According to Sec. 2, an application of this operator is evaluated by first initializing its parameters from left to right with the values of the corresponding operands, that means:

- The Parameter var is initialized with the variable i.
- The parameters lower and upper are initialized with the values 1 and 10, respectively.
- The deducible parameter B is initialized with the type bool of the subexpression print ?i.
- The parameter body is initialized with the value resulting from the evaluation of this subexpression, which prints the current value nil of the variable i (i. e., a blank line) and returns the bool value true.

Afterwards, the operator's implementation is evaluated, which means:

- Variable var (i. e., i) is assigned the value lower (i. e., 1).
- While the value of that variable is less or equal to upper (i. e., 10), the subexpression body; var =! ?var + 1 is evaluated repeatedly:
  - The evaluation of the parameter body simply yields its constant value true, i. e., it has no effect at all. In particular, it does *not* evaluate the subexpression print ?i.
  - The evaluation of the assignment var =! ?var + 1 increments the variable var (i. e., i) by 1, so that its final value after the last iteration will be 11.

In summary that means, that the above application of the for loop does not print the numbers 1 to 10, but just a blank line.

6.2 Solution

The problem with the above definition of the `for` operator is, of course, that the subexpression `print ?i` is evaluated *once before* the evaluation of the operator's implementation, instead of being evaluated *repeatedly during* the evaluation of this implementation.

This can be remedied by defining the parameter `body` as a *lambda parameter*:

```
for (var:int?) "=" (lower:int) ".." (upper:int) do
                              [(B:type)] (\ body -> (B)) end -> (int =
    var =! lower;
    while ?var <= upper do
        body;
        var =! ?var + 1
    end
);

i : int?;
for i = 1 .. 10 do
    print ?i
end
```

Omitting several technical details, this means:

- In the implementation of the `for` operator, there is a local operator `body` defined as `body -> (B)`.
- The implementation of this local operator is provided by the corresponding operand of an application of the operator, i. e., the subexpression `print ?i` in the above example.
- Because `body` is now an operator instead of a constant (as in Sec. 6.1), each evaluation of `body` inside the `while` loop causes a new evaluation of of this subexpression which will print the current value of the variable `i`.
- Because the variable `var` (i. e., `i`) is incremented in each iteration of the `while` loop, the above application of the `for` operator now in fact prints the numbers 1 to 10 instead of a single blank line.

In other words, operands corresponding to a lambda parameter are passed *unevaluated* and will be evaluated every time the lambda parameter is evaluated during the evaluation of the operator's implementation.

The backslash resembling a λ character is necessary to distinguish a lambda parameter from a regular parameter whose type is an operator type:

- An operand corresponding to a lambda parameter `\ body -> (B)` must be an expression with type `B`, which is used as the implementation of an implicitly generated local operator defined as `body -> (B)`.
- On the other hand, an operand corresponding to a regular parameter `body -> (B)` must be an expression that directly yields an operator of a compatible type, i. e., a parameterless operator with result type `B`. (This is described in more detail in the KPS 2023 paper [5].)

7 Lambda Parameters with Parameters

If a lambda parameter – which actually denotes an operator as described in Sec. 6.2 – has itself parameters, they are made visible in the operands corresponding to the lambda parameter, for example:

```
$$ Definition.
for i "=" (lower:int) ".." (upper:int) do
                    [(B:type)] (\ body (i:int) -> (B)) end -> (int =
    var : int?;
    var =! lower;
    while ?var <= upper do
        body ?var;
        var =! ?var + 1
    end
);

$$ Application: Prints the numbers 1 to 10.
for i = 1 .. 10 do
    print i
end
```

This example differs from the example given in Sec. 6.2 in several important details:

- The `for` operator defined here requires the fixed name `i` following the initial name `for` instead of an operand yielding an arbitrary variable of type `int?`. Therefore, no variable is required for applications of this operator.
- Instead, there is a local variable `var` defined in the operator's implementation.
- The lambda parameter `body` defined here has itself a parameter `i` with type `int` and, therefore, requires an operand of type `int` (actually `?var`) when being applied.
- The operand corresponding to the lambda parameter in the application of the `for` operator is `print i` instead of `print ?i`, where `i` is actually the parameter `i` of the lambda parameter which is visible in this operand.
- In each evaluation of the lambda parameter `body`, its parameter `i` receives the value of the corresponding operand `?var`, i. e., the current value of the variable `var`.
- Because each evaluation of `body` causes an evaluation of the corresponding operand `print i` with the respective value of the parameter `i`, the entire `for` loop will again print the numbers 1 to 10.

8 Passing Names to Operators

Of course, the fixed name `i` used to denote the current iteration value in the `for` operator defined in Sec. 7 is somewhat artificial.

To define a more realistic operator that can instead be used with an arbitrary user-defined name, it is necessary to pass that name to the operator. For that purpose, the fixed name `i` is replaced with a parameter named `#name` with the special type `sym`, that is in turn used as the name of the parameter of the lambda parameter `body`:

```
$$ Definition.
for ("#name":sym) "=" (lower:int) ".." (upper:int) do
                [(B:type)] (\ body (#name:int) -> (B)) end -> (int =
    var : int?;
    var =! lower;
    while ?var <= upper do
        body ?var;
        var =! ?var + 1
    end
);

$$ Applications.
for j = 1 .. 10 do
    print j
end;
for k = 1 .. 10 do
    print k
end
```

This requires some explanations:

- According to Sec. 2, names containing special characters such as `#` must be enclosed in quotation marks in a declaration, but used without the quotation marks in applications of the constant, parameter, or operator defined by the declaration.
- Therefore, `#name` in the declaration `#name:int` of the parameter of the lambda parameter `body` denotes an application of the parameter defined as `"#name":sym`.
- If this parameter would be defined as `name:sym`, the declaration `name:int` of the parameter of the lambda parameter `body` would be ambiguous because `name` could denote either an application of the parameter `name` with type `sym` or directly the name `name`. To avoid such ambiguities, the names of parameters that shall be used in the declaration of other parameters must contain at least one special character.
- Expressions with type `sym` – in particular applications of parameters with type `sym` – can be used instead of names in the signature of a constant, parameter, or operator declaration. Therefore, `#name:int` is in fact a correct parameter declaration.
- Operands corresponding to parameters with type `sym` can be names as described in Sec. 2 (e. g., `xyz`, `"N′"` etc.) or arbitrary expressions with type `sym`, in particular applications of other parameters with type `sym` whose operands are in turn either names or such expressions etc. Therefore, such operands eventually denote names.
- When an operator that has parameters with type `sym` is applied, applications of these parameters are replaced with the names denoted by the corresponding operands.

Therefore, the expression

```
for j = 1 .. 10 do
    print j
end
```

is processed at compile time as follows:

- The parameter named `#name` of the `for` operator is initialized with the name `j`.
- The parameters `lower` and `upper` are initialized with the values 1 and 10, respectively.
- Before the operand corresponding to the lambda parameter `body` is parsed, the application `#name` of the aforementioned parameter is replaced with the name `j`, causing the lambda parameter to actually have a parameter `j:int` which will be visible in the corresponding operand.
- Therefore, `print j` is a correct subexpression in this context (which would not be the case before or after the application of the `for` operator, because the parameter `j` is not visible there) which is used as the implementation of the lambda parameter.

At run time, the application of the `for` operator is evaluated in the same way as in Sec. 7.

9 Virtual Operators

9.1 Basic Principle

Applications of the following operator consist of an arbitrary type `T` followed by an arbitrary name, e. g., `int num` or `bool flag`, i. e., they denote declarations of variables `num` and `flag` with content type `int` and `bool`, respectively, in languages such as C and Java:

```
(T:type) ("#name":sym) -> (T? =
    #name : T?
)
```

In fact, the implementation of the operator actually contains a declaration of a variable with name `#name` and content type `T`, which are replaced with the name and type, respectively, which are passed to an application of the operator. This variable, however, is a local variable which is not visible outside of the operator's implementation.

To make it visible there, the above operator might be turned into a *virtual operator* by simply replacing the arrow `->` with a double-headed arrow `->>`:

```
(T:type) ("#name":sym) ->> (T? =
    #name : T?
)
```

Again omitting several technical details, an application of a virtual operator such as `int num` is replaced *at compile time* with the implementation of the operator, i. e., `#name : T?` in this example, where applications of the operator's parameters (`#name` and `T`) are in turn replaced with the corresponding operands (`num` and `int`, respectively), finally yielding the declaration `num : int?` which logically replaces the original expression `int num`. Therefore, the variable `num` defined by this declaration will be visible in the sequel:

```
int num;
num =! 1;
print ?num
```

9.2 Haskell-like Function Definitions

Haskell provides a very concise syntax for function definitions and applications, e. g.:

```
$$ Definition.
square x = x * x

$$ Application.
square 5
```

Doing the same with MOSTflexiPL's predefined declaration syntax is much more verbose:

```
$$ Definition.
square (x:int) -> (int = x * x);

$$ Application.
square 5
```

By omitting the parameter and result type of the `square` operator, which can be automatically deduced by the compiler, the definition becomes a little bit shorter:

```
square (x:) -> (= x * x)
```

To make it as concise as in Haskell, a virtual operator typically defined in a library can be used:

```
$$ Virtual operator defined in a syntax library.
("#func":sym) ("#par":sym) "="
                [(P:type) (R:type)] (\ impl (#par:P) -> (R)) ->> (=
    #func (par:P) -> (R = impl par)
);

$$ Application of the virtual operator
$$ to define function square.
square x = x * x;

$$ Application of the function square.
square 5
```

At compile time, the expression `square x = x * x` is recognized as follows as an application of the virtual operator defined before:

- Its parameters `#func` and `#par` are initialized with the names `square` and `x`, respectively.
- Before the operand `x * x` corresponding to the lambda parameter `impl` is parsed, `#par` is replaced with the name `x` in the declaration of its parameter, making the parameter `x` with type `P` (which is currently unknown, but will be deduced later) visible in this operand.
- Because `x` with unknown type `P` as well as the predefined multiplication operator `(int) "*" (int) -> (int)` are visible, `x * x` is recognized as a correct subexpression with type `int` by deducing `P` as `int` along the way.
- This subexpression is used as the implementation of the lambda parameter `impl` deducing `R` also as `int` along the way.

Because the operator is virtual, its application is replaced at compile time with its implementation, in which applications of the operator's parameters (`#func`, `P` and `R`) are in turned replaced by the corresponding operands, yielding the declaration

```
square (par:int) -> (int = impl par)
```

where `impl par` is an application of the following local operator that is implicitly generated for the lambda parameter `impl` using the operand `x * x` as its implementation:

```
impl (x:int) -> (int = x * x)
```

In summary, `square x = x * x` is replaced with a declaration of an operator `square (int) -> (int)` whose implementation computes the square of its parameter value and, therefore, this operator can be used in the sequel, e. g.:

```
square 5
```

9.3 Type Aliases

Another important use case of virtual operators, which cannot be achieved with normal operators, are type aliases.

If, for example, a programmer wants to define an alternative syntax `var of T` for variable types, because he dislikes the predefined syntax `T?`, this can be easily achieved with the following virtual operator:

```
$$ Definition.
var of (T:type) ->> (type = T?);

$$ Application.
x : var of int;
x =! 1;
print ?x
```

Because an application of a virtual operator is replaced at compile time with the operator's implementation where applications of the operator's parameters are in turn replaced with the corresponding operands, `var of int` is replaced with `int?` and, therefore, `x` has actually type `int?`.

If the operator `var of •` would be a normal, non-virtual operator, the expression `var of int` would still return the type `int?` at *run time*, but at compile it would be different from `int?` and, therefore, the expressions `x =! 1` and `?x` would not be type-correct, because they require an operand with type `T?` for some type `T`.

10 Outlook

During the last years, MOSTflexiPL has reached a certain degree of maturity and stability, which allows it to be employed for teaching purposes and for the implementation of small to medium-sized real world projects. In particular, the compiler messages in case of errors and ambiguities have been significantly improved during the last months. Furthermore, the compiler has been prototypically integrated into Microsoft's Visual Studio Code Editor and into the well-known text editor VI Improved (VIM) to make practical program development with MOSTflexiPL more convenient.

A topic that is neither covered by the KPS 2023 paper [5] nor by this paper are visibility declarations, which are required to define user-defined scoping rules and locally confined syntax extensions.

To make the language even more flexible and powerful, several features are still desirable:

- Meta-operators, which are required to pass the values of a repeatable parameter to another operator accepting repeatable parameters.
- User-defined literals, e. g., for types representing date and time values.

- User-defined whitespace and comments to allow any desired syntax for both block and line comments.
- Dynamic redefinitions of operators [4], which allow amongst other things strictly modular extensions of existing code and thus support unanticipated software evolution.
- Basic operators for parallel execution and synchronization, which can then be used to define more convenient and advanced constructs for parallel programming.

Some of these features have already been developed and prototypically implemented in earlier versions of the compiler, but have not been integrated into the current compiler yet.

11 Related Work

During the history of programming language development, the idea of an extensible programming language has appeared every now and then.

One of oldest and most well-known examples is Lisp [9] with its different dialects and flavors. Similar to MOSTflexiPL, Lisp does neither distinguish between operators and functions nor between predefined and user-defined operators/functions. By defining new functions – or macros, whose syntactic appearance is identical to that of functions – a programmer is actually extending the language all the time. Another parallel to MOSTflexiPL is the fact that language extensions are defined in the language itself, and that a very small language core is sufficient for that purpose. However, there are also essential differences: First of all, Lisp does not possess a static type system. Furthermore, Lisp expressions must always be parenthesized, which significantly restricts the possibilities for defining new syntax. Finally, MOSTflexiPL does not have a "procedural" macro engine, i. e., no user code will be executed at compile time in order to perform syntactic transformations. In contrast, the compile time transformations provided by virtual operators are completely declarative as well as statically type-safe. In summary, MOSTflexiPL has considerable advantages over Lisp (complete syntactic freedom and static type safety), while the deliberately omitted procedural macro facility has not been perceived as a major limitation yet.

Dylan [3] is a more modern language that has been strongly influenced by Lisp's ideas. It also supports syntactic extensibility in the language itself (actually in a rewrite macro system which is an integral part of the language). Even though the programmer has more freedom than with Lisp's simple s-expressions, there are also strict syntactic limitations which cannot be exceeded. In contrast, the operator concept of MOSTflexiPL offers virtually unlimited syntactic freedom. Apart from that, Dylan does not have a static type system either.

Many different languages, e. g., Haskell [7], Prolog [2], and Scala [8], allow the user to extend at least the syntax of expressions by defining new operator symbols. Since functional languages, just as MOSTflexiPL, do not distinguish between expressions and statements, the syntax of statements (e. g., control structures) becomes also extensible in principle. However, the syntax of types and declarations still remains fixed. In MOSTflexiPL, however, the basic principle "*everything* is an expression" is also applied to types and declarations, whose syntax can be extended by means of virtual operators.

An approach whose basic ideas and objectives are almost identical to that of MOSTflexiPL is "π – a Pattern Language" [6]. The concept called pattern there – which is "the only language construct in π" – directly corresponds to an operator in MOSTflexiPL: It possesses a syntax, composed of names (or symbols) and placeholders for operands, and an associated meaning corresponding to the implementation of a MOSTflexiPL operator. Thus, both approaches provide the same virtually unlimited syntactic flexibility that ultimately stems from the lack of any predefined grammar.

A significant difference and advantage of MOSTflexiPL over π is once again the static type system, since π is completely dynamically typed. In fact, the endeavour to reconcile extreme flexibility on the one hand with a maximum of static checkability on the other hand has been and still is one of the most ambitious challenges in the development of MOSTflexiPL.

Apart from that, MOSTflexiPL provides several other useful facilities not found in π, e. g., implicit and deducible parameters (where the latter are dispensable in a dynamically typed language) or visibility and exclude declarations which allow, amongst others, user-defined scoping rules and locally confined syntax extensions.

Finally, MOSTflexiPL might also be considered an adaptive grammar formalism [1, 10]. Because "everything is an expression," there is a single non-terminal symbol `X` denoting expressions. Every operator declaration induces a new production for `X` whose right hand side can be derived from the operator's signature by treating the operator's names as terminal symbols and replacing explicit parameters with the non-terminal `X`. The type information associated with the parameters and the result type of the operator can be added as grammar attributes. Visibility declarations control the set of currently active productions, while exclude declarations can be used to rule out some otherwise possible derivations.

12 Conclusion

MOSTflexiPL is a programming language currently under development whose syntax can be extended and customized by its users in a virtually unlimited way, where a rather small number of core constructs is sufficient to support a broad range of different programming styles. Therefore, it can be used, amongst others, as an extensible general purpose programming language, but also as a host language for developing domain-specific languages. It possesses a static type system and is implemented by a compiler and a run-time system written in C++.

References

[1] H. Christiansen: "A survey of adaptable grammars." *ACM SIGPLAN Notices* 25 (11) November 1990, 35–44.

[2] W. F. Clocksin, C. S. Mellish: *Programming in Prolog* (Fourth Edition). Springer-Verlag, Berlin, 1994.

[3] I. D. Craig: *Programming in Dylan*. Springer-Verlag, London, 1997.

[4] C. Heinlein: "Global and Local Virtual Functions in C++." *Journal of Object Technology* 4 (10) December 2005, 71–93, http://www.jot.fm/issues/issue_2005_12/article4.

[5] C. Heinlein: "MOSTflexiPL – An Extremely Flexible Programming Language." In: T. Noll, I. Fesefeldt (eds.): *22. Kolloquium Programmiersprachen und Grundlagen der Programmierung*. Aachener Informatik-Berichte AIB-2023-02, Department of Computer Science, RWTH Aachen, 2023, 55–72.

[6] R. Knöll, M. Mezini: "π – a Pattern Language." In: *Proc. 24th Ann. ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2009)* (Orlando, FL, October 2009). *ACM SIGPLAN Notices* 44 (10) October 2009, 503–521.

[7] S. Marlow (ed.): *Haskell 2010 Language Report*. HaskellWiki, 2010. http://haskell.org/definition/haskell2010.pdf

[8] M. Odersky: *The Scala Language Specification* (Version 2.9). Programming Methods Laboratory, Ecole Polytechnique Fédérale de Lausanne (EPFL), May 2011.

[9] G. L. Steele Jr.: *Common Lisp: The Language* (Second Edition). Digital Press, Bedford, MA, 1990.

[10] Wikipedia Contributors: *Adaptive Grammar*. https://en.wikipedia.org/wiki/Adaptive_grammar (2023-08-30)

Programming Joins

FRITZ HENGLEIN, University of Copenhagen, Dänemark

In programming one often has to implement queries such as joins on in-memory data. Joins, and relational queries in general, have been considered difficult to program efficiently. It turns out they are not.

We show that worst-case optimal joins, also for cyclic joins, are straightforward to program using a few basic programming techniques usually taught and learned in the first year of a computer science bachelor program: immutable dictionaries such as hash tries or radix trees, choosing the smallest set to iterate over when intersecting sets, and iterating over the variables in a join query in any nesting order.

We provide a novel proof of worst-case optimality by amortization, where the execution cost is charged to the output generated from input that is extended with ghost data. We furthermore show experimentally that widely used SQL engines generate asymptotically and practically inferior code on fundamental cyclic joins. For example, for triangle queries on certain sparse input relations, our 10-line standard Python program (or a corresponding 20 line Haskell program) executes 400 to 50.000 times faster than employing a variety of SQL engines.

This suggests that dispatching queries on program data to an SQL database engine is sometimes, and we suspect often, not a good idea for one of the motivations behind language-integrated querying: computational efficiency.

This is joint work with Changjun Li, Mikkel Kragh Mathiesen and Mads Rehof.

Author's address: Fritz Henglein, University of Copenhagen, Kopenhagen, Dänemark, fritz@henglein.com.

Towards Language Product Line Engineering Using Classic Compiler Generators

Teaching Old Horses New Tricks

THOMAS KÜHN, Martin-Luther University Halle-Wittenberg, Germany

In recent years, research on language product line (LPL) engineering has emerged. Building on the ideas of modular compiler construction and software product line engineering (SPLE), LPLs enable language users to choose and pick language features from which a corresponding compiler, interpreter and/or integrated development environment is composed. While several LPL engineering approaches showcased their general applicability to individual cases, most approaches rely on their own especially tailored language workbench. Hence, I aim to identify design principles and techniques in classic compiler generators that support the development of LPLs. In this talk, I present ongoing work creating an LPL for the Sample Programming Language (LAX) [22] using the compiler construction toolkit Eli [10] as a classic compiler generator.

Additional Key Words and Phrases: Compiler Construction, Software Product Lines, Language Product Lines

Most compilers for programming languages are developed using compiler construction tools (or language workbenches [7]) that provide various domain-specific specification languages, e.g., for grammars, attribute equations, or transformation rules, to generate almost all parts of the compiler. However, as everything is generated, reusing parts of a compiler is generally to either copying these definitions to another compiler specification or reusing an entire compilation phase or the whole back-end. To remedy this, researchers focused on modular compiler constructions, e.g., [9, 11, 17, 16], to establish more fine-grained reusable specifications. In the past decades, compiler construction tools and language workbenches included means for systematically reusing compiler specifications, e.g., [6, 19, 21, 4, 2, 8], ultimately leading to the notion of language components as "a reusable unit encapsulating a potentially incomplete language definition [comprising] the realization of syntax and semantics of a (software) language." [3, p. 243]. Building on the ideas of modular compiler construction and language components, more recently, researchers coined the term language product line* (LPL), e.g., [5, 24, 13], to denote the systematic development of a set of compilers for a family of languages by composing reusable language components. Adopting ideas from software product line (SPL) engineering, language users should be able to choose and pick needed language features[1] from which a corresponding compiler is constructed.

While several LPL approaches appeared over the years, their scope and the level of abstraction on which they operate differs. In detail, while many approaches compose languages on the concrete or abstract syntax level only, e.g. [24, 14, 12, 15], few include the composition of intermediate or machine code generation, e.g., [3]. Moreover, while some showed their approaches' applicability to general programming languages, e.g., [20, 13], many relied on their own specialized and tailored language workbenches, e.g. [20, 8, 2]. Thus, it is difficult to identify fundamental underlying principles and techniques for the modular development and composition of languages uncovered in recent years.

[1]Following Vacchi and Cazzola [18], language features are either language constructs, e.g., if then else, or language concepts (without concrete syntax), e.g., recursion.

Author's Contact Information: Thomas Kühn, Software Engineering and Programming Languages Group, Martin-Luther University Halle-Wittenberg, Halle, Saxony Anhalt, D-06120, Germany, thomas.kuehn@informatik.uni-halle.de.

To remedy this, in my ongoing work I aim to answer the following research questions:

- (RQ1) How suitable are classic compiler generators for the creation of an LPL of a programming language?
- (RQ2) What are fundamental principles and techniques for the modular development and composition of programming languages?
- (RQ3) What properties must a language component exhibit to be composable with another component?

In pursuit of answering these questions, I have started to develop an LPL of the Sample Programming Language (LAX) [22, Appendix A] using the classic compiler construction toolkit Eli [10]. Eli provides a rich set of domain-specific languages to specify all phases of a compiler, ranging from concrete and abstract syntax via attribute grammars and definition tables to code selection and code generation [10]. Moreover, its ability to integrate auxiliary C code makes it very flexible. Following the fine-grained iterative development approach proposed by Zimmermann, Kühn, and WeiSSbach [23], I started with an initial compiler covering the smallest LAX program and systematically implementing language features and extending the feature model in small increments.[2] Thus far, I have followed the iteration plan outlined in [kuehn2023kps] and employed one of the earliest additive SPL implementation technique, i.e., preprocessor annotations [1, Cha. 5.3], whereas previously implemented features as well as the initial compiler version shall not be changed during development.

In this talk, I provide insights into how the different specification languages permit or hinder modular compiler development. Moreover, I highlight the implementation techniques used to allow for extending the initial compiler with additional language components, focusing on the composability of the different domain-specific specification languages provided by the Eli toolkit. I will outline the development of the initial compiler version and the implementation of the first features, highlighting the employed extension points. In particular, I will not only focus on grammars for the concrete and abstract syntax but also on attribute grammars through to code generator specifications. Furthermore, for each compilation phase, I discuss when a corresponding specification or code fragment is composable.

This, in turn, will enable researchers and practitioners to use existing compiler generators for developing LPLs and extend them with composable specification languages for the individual phases. Overall, guiding practitioners towards more modular compilers or full-fledged language product lines.

Acknowledgments

I want to thank my supervisor Prof. Wolf Zimmermann for establishing the fine grained iterative development approach and fruitful discussions regarding compiler development. Moreover, I want to thank Edward Sabinus who reworked the iteratively developed LAX compiler, my work is based on. Last but not least, I thank Elisabeth Fritsch for the many discussions about the LAX LPL during her masters thesis.

References

[1] Sven Apel et al. Feature-Oriented Software Product Lines: Concepts and Implementation. Springer, 2013. ISBN: 9783642375200.

[2] Arvid Butting et al. “Systematic composition of independent language features”. In: Journal of Systems and Software 152 (2019), pp. 50–69.

[2] This approach is a well-established SPL development approach denoted the reactive approach [1, Cha. 2.4].

[3] Arvid Butting and Andreas Wortmann. “Language Engineering for Heterogeneous Collaborative Embedded Systems”. In: Model-Based Engineering of Collaborative Embedded Systems: Extensions of the SPES Methodology. Cham: Springer International Publishing, 2021, pp. 239–253. ISBN: 978-3-030-62136-0. DOI: 10.1007/978-3-030-62136-0_11. URL: https://doi.org/10.1007/978-3-030-62136-0_11.

[4] Thomas Degueule et al. “Melange: a Meta-Language for Modular and Reusable Development of DSLs”. In: 8th International Conference on Software Language Engineering (SLE’15). Ed. by Davide Di Ruscio and Markus Völter. Pittsburgh, PA, USA: ACM, 2015, pp. 25–36.

[5] Benjamin Delaware, William Cook, and Don Batory. “Product lines of theorems”. In: 2011 ACM international conference on Object oriented programming systems languages and applications. 2011, pp. 595–608.

[6] Torbjörn Ekman and Görel Hedin. “The JastAdd System — Modular Extensible Compiler Construction”. In: Science of Computer Programming 69.1-3 (2007), pp. 14–26.

[7] Sebastian Erdweg et al. “Evaluating and Comparing Language Workbenches: Existing Results and Benchmarks for the Future”. In: Computer Languages, Systems and Structures 44 (2015), pp. 24–47.

[8] Luca Favalli, Thomas Kühn, and Walter Cazzola. “Neverlang and FeatureIDE just married: Integrated language product line development environment”. In: 24th ACM Conference on Systems and Software Product Line (SPLC’20): Volume A-Volume A. 2020, pp. 1–11. DOI: 10.1145/3382025.3414961.

[9] Harald Ganzinger. “Increasing modularity and language-independency in automatically generated compilers”. In: Science of Computer Programming 3.3 (1983), pp. 223–278.

[10] Robert W. Gray et al. “Eli: A complete, flexible compiler construction system”. In: Communications of the ACM 35.2 (1992), pp. 121–130.

[11] Uwe Kastens and William M. Waite. “Modularity and reusability in attribute grammars”. In: Acta Informatica 31 (1994), pp. 601–627.

[12] Thomas Kühn and Walter Cazzola. “Apples and Oranges: Comparing Top-Down and Bottom-Up Language Product Lines”. In: 20th International Software Product Line Conference (SPLC’16). Beijing, China: ACM, 2016, pp. 50–59.

[13] Thomas Kühn, Walter Cazzola, and Diego Mathias Olivares. “Choosy and Picky: Configuration of Language Product Lines”. In: 19th International Software Product Line Conference (SPLC’15). Ed. by Goetz Botterweck and Jules White. Nashville, TN, USA: ACM, 2015, pp. 71–80.

[14] Thomas Kühn et al. “A Metamodel Family for Role-Based Modeling and Programming Languages”. In: 7th International Conference Software Language Engineering (SLE’14). Lecture Notes in Computer Science 8706. Västerås, Sweden: Springer, 2014, pp. 141–160.

[15] Manuel Leduc, Thomas Degueule, and Benoit Combemale. “Modular Language Composition for the Masses”. In: Proceedings of 11th International Conference on Software Language Engineering (SLE’18), SLE 2018. Boston, MA, USA: ACM, 2018, pp. 47–59. ISBN: 9781450360296. DOI: 10.1145/3276604.3276622. URL: https://doi.org/10.1145/3276604.3276622.

[16] Marjan Mernik and Viljem umer. “Incremental Programming Language Development”. In: Computer Languages, Systems and Structures 31.1 (2005), pp. 1–16. DOI: 10.1016/j.cl.2004.02.001.

[17] Nathaniel Nystrom, Michael R Clarkson, and Andrew C Myers. “Polyglot: An extensible compiler framework for Java”. In: International Conference on Compiler Construction. Springer. 2003, pp. 138–152.

[18] Edoardo Vacchi and Walter Cazzola. “Neverlang: A Framework for Feature-Oriented Language Development”. In: Computer Languages, Systems & Structures 43.3 (2015), pp. 1–40. DOI: 10.1016/j.cl.2015.02.001.

[19] Markus Völter and Vaclav Pech. “Language Modularity with the MPS Language Workbench”. In: 34th International Conference on Software Engineering (ICSE’12). Zürich, Switzerland: IEEE, 2012, pp. 1449–1450.

[20] Markus Völter et al. “mbeddr: an Extensible C-Based Programming Language and IDE for Embedded Systems”. In: 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity (SPLASH’12). Tucson, AZ, USA: ACM, 2012, pp. 121–140. DOI: 10.1145/2384716.2384767.

[21] Guido H. Wachsmuth, Gabriël D. P. Konat, and Eelco Visser. “Language Design with the Spoofax Language Workbench”. In: IEEE Software 31.5 (2014), pp. 35–43.

[22] William M Waite and Gerhard Goos. Compiler construction. Springer Science & Business Media, 1995. ISBN: 0-387-90821-8.

[23] Wolf Zimmermann, Thomas Kühn, and Mandy WeiSSbach. “(Almost) Agile Development of Verified Compilers”. In: 22. Kolloquium Programmiersprachen und Grundlagen der Programmierung. Ed. by Thomas Noll and Ira Fesefeldt. AIB-2023-03. 2023, pp. 176–185. URL: https://publications.rwth-aachen.de/record/972197/files/972197.pdf.

[24] Steffen Zschaler et al. “VML*–a family of languages for variability management in software product lines”. In: Software Language Engineering: Second International Conference, SLE 2009. Springer. Denver, CO, USA, 2010, pp. 82–102.

d2d = XML für Autoren

MARKUS LEPPER, semantics gGmbH, Deutschland

Die Codierung XML hat sich in vielen Bereichen als Standard für die Modellierung und Verarbeitung von Textdokumenten bewährt.

Auch semi-formale Fließtexte (Berichte, Bedienungsanleitungen, Gedichte, Romane, Briefe) können von automatischer Verarbeitung profitieren. D2d gibt den Autorinnen solcher Dokumente ein Format, welches mit wenig syntaktischem Ballast, auch ganz ohne technische Hilfsmittel und mit möglichst geringer Störung des kreativen Flows erlaubt, in gewohntem Schreibtempo formal-korrekte XML-Dokumente zu notieren.

Weiterhin sollen auch Domain-Experten für ihre Fachgebiete mit einfachen Mitteln verständliche Text-Typ-Definitionen erstellen können. Für beide Zwecke müssen sehr unterschiedliche avancierte Techniken kombiniert werden.

Author's address: Markus Lepper, semantics gGmbH, Berlin, Deutschland, post@mlepper.de.

Is this a Language Killed by Incremental Improvements? Python, Can We Help?

STEFAN MARR, University of Kent, United Kingdom

In the Python community, two major players having investing into the future of Python over the past years. Microsoft's Faster CPython team pushed ahead with impressive performance improvements for the CPython interpreter, which has gotten at least 2x faster since Python 3.9 and they have a baseline JIT compiler for CPython, too. At the same time, Meta is worked hard on making free-threaded Python a reality and bring classic shared-memory multithreading to Python, without being limited by the still standard Global Interpreter Lock, which prevents true parallelism.

Both projects delived major improvements to Python, and the wider ecosystem. So, it's all great, or is it?

In this talk, I'll discuss some of the aspects the Python core developers and wider community seem to not regard with the same urgency as it seems necessary from my personal perspective. Concurrency bugs scare me, and I strongly believe the Python ecosystem should be scared, too, or look forward to the 2030s being "Python's Decade of Concurrency Bugs". We'll start out reviewing some of the changes in observable language semantics between Python 3.9 and today, discuss their implications, and because of course I have some old ideas lying around, I'll propose a way forward. In practice though, this isn't a small well-defined research project. So, I hope to find people that might want to follow me down the rabbit hole of Python's free-threaded future.

Author's address: Stefan Marr, University of Kent, Canterbury, United Kingdom, s.marr@kent.ac.uk.

Bisimulation: Analyzing Divergent Interpreter Implementations

TIM MATUSSEK, Universität der Bundeswehr München, Germany
STEFAN BRUNTHALER, Universität der Bundeswehr München, Germany

Ensuring semantic equivalence between interpreter implementations poses a fundamental challenge in programming language engineering. When reimplementing an interpreter—whether for performance optimization, platform portability, or security hardening—developers face the problem of verifying that the new implementation faithfully preserves the semantics of the reference implementation. In the absence of a formal specification, the reference implementation *is* the specification. This paper presents a practical bisimulation-based approach for systematically detecting semantic divergences between interpreter implementations. Our method executes both interpreters in lockstep, observing bytecode instructions, stack states, and function invocations at each step. When divergences occur, we aggregate and analyze them to identify root causes in the interpreter source code. We demonstrate our approach by comparing CPython and GraalPy, revealing previously unknown semantic differences. Our tooling includes a bisimulation driver, a cross-interpreter protocol, test case generation, and divergence aggregation infrastructure. Initial results show that bisimulation provides finer-grained semantic analysis than traditional testing approaches.

Additional Key Words and Phrases: bisimulation, interpreter semantics, Python, CPython, GraalPy, semantic equivalence, differential testing

1 Introduction

The proliferation of programming language implementations presents a unique verification challenge. For widely-used languages like Python, multiple interpreters exist: CPython (the reference implementation), PyPy (a JIT-compiled implementation), GraalPy (part of the GraalVM ecosystem), and MicroPython (for embedded systems). Each implementation makes different trade-offs regarding performance, memory usage, and platform support, yet all must preserve the language's semantics.

The fundamental problem we address is: *How can we systematically verify that two interpreter implementations exhibit equivalent semantics?* Traditional approaches rely on test suites, but these have inherent limitations. Test coverage is bounded, edge cases may remain untested, and a failing test only provides indirect evidence of the underlying bug. More critically, tests approximate the *specified* behavior, whereas bisimulation approximates the *implemented* behavior—a subtle but crucial distinction when the reference implementation serves as the implicit specification.

Our approach leverages bisimulation, a well-established technique from concurrency theory, to analyze behavioral equivalence at the instruction level. By executing both interpreters in lockstep and observing their internal states, we can detect divergences that might escape conventional testing. When divergences are found, we aggregate them across many test cases to identify patterns that point to root causes in the interpreter implementation.

This paper makes the following contributions:

- A practical bisimulation framework for comparing interpreter implementations without requiring formal specifications
- A protocol for coordinating execution between disparate interpreter architectures (demonstrated with CPython and GraalPy)
- Techniques for divergence aggregation and root-cause analysis
- Strategies for resynchronization when interpreter instruction sequences diverge structurally

Authors' Contact Information: Tim Matussek, Universität der Bundeswehr München, Munich, Germany; Stefan Brunthaler, Universität der Bundeswehr München, Munich, Germany.

2 Background and Motivation

2.1 The Semantic Equivalence Problem

When implementing a new interpreter for an existing language, developers face a specification problem. For languages like Python, the formal specification is incomplete—the CPython implementation effectively *is* the specification for edge cases and corner behaviors. This creates a verification challenge: how do we know the reimplementation is correct?

Formal verification against a specification is ideal but often impractical. Recent work by Desharnais and Brunthaler [1] explores verified virtual machines for dynamic languages, but the effort required is substantial. For many practical scenarios, we need lightweight approaches that can find semantic divergences without full formal verification.

2.2 Bisimulation

Bisimulation is a foundational concept from process algebra and concurrency theory [2]. Two systems are bisimilar if they can simulate each other step by step: any action one system can take, the other can match with an equivalent action, and vice versa. This provides a strong notion of behavioral equivalence.

We adapt bisimulation for interpreter comparison with a relaxed goal: rather than proving full bisimilarity, we aim to *find divergent behavior*. The interpreters execute the same program, and we observe whether they take equivalent steps. Any divergence indicates a semantic difference worth investigating.

2.3 Why Not Just Tests?

Traditional testing has fundamental limitations for semantic equivalence checking:

- **Coverage bounds:** Maximum test coverage is inherently limited; some behaviors remain unexercised
- **Edge case difficulty:** Certain bugs (e.g., host floating-point representation differences) are notoriously hard to test
- **Indirect error indication:** A failing test reveals that something is wrong but not necessarily what or where

Bisimulation addresses these limitations by operating at instruction granularity. It observes actual implemented behavior rather than specified behavior, and divergences directly indicate the point and nature of the difference.

A key insight motivates our approach: tests approximate the implicit specification of *tested behavior*, while bisimulation approximates the implicit specification of *implemented behavior*.

3 Approach

3.1 Architecture Overview

Our bisimulation framework consists of four main components: (1) a *bisimulation driver* that coordinates execution, (2) a *protocol* for interpreter control and observation, (3) a *test case builder* for generating input programs, and (4) a *divergence aggregator* for analysis. The driver orchestrates lockstep execution: both interpreters execute one instruction, report their observations, and the driver compares them before proceeding.

Figure 1 illustrates the high-level architecture. The bisimulator sits between the two interpreters, receiving protocol messages from each and coordinating their execution.

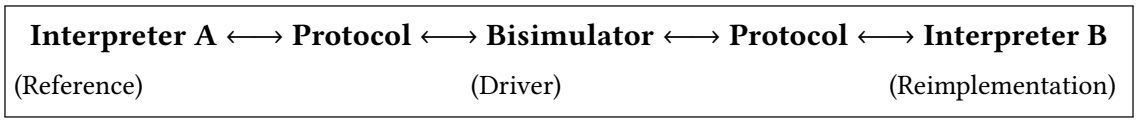


Fig. 1. Bisimulation architecture: the driver coordinates lockstep execution between interpreters via a common protocol.

3.2 Observable Behavior

At each step, we observe three aspects of interpreter state:

- **Instructions:** The bytecode instruction being executed (accounting for different instruction sets)
- **Stack state:** The operand stack contents, particularly the top-of-stack value
- **Function calls:** Entry and exit of function invocations

We classify divergences into *instruction divergences* (the interpreters execute semantically different operations) and *stack divergences* (the interpreters produce different values).

3.3 Divergence Analysis

A single divergence is rarely informative in isolation. We aggregate divergences across many test case executions, looking for patterns. For instance, if stack divergences consistently occur after `BINARY_OP` instructions with floating-point operands producing values like 1.02 vs. 1.019, this pattern suggests a floating-point handling difference in the interpreter's fast-path code.

Our analysis pipeline includes: (1) aggregation by instruction type and operand patterns, (2) filtering of known benign differences, and (3) root-cause correlation with interpreter source code. This transforms raw divergence data into actionable bug reports.

4 Illustrative Example

Consider the following Python snippet: `myfunc("test")`. Table 1 illustrates the bisimulation trace comparing CPython and GraalPy execution.

Table 1. Bisimulation trace comparing CPython and GraalPy instruction sequences. Minor naming differences are normalized; structural equivalence is maintained.

Step	CPython	GraalPy
0	LOAD_CONST	LOAD_CONSTANT
1	MAKE_FUNCTION	MAKE_FUNCTION_
2	LOAD_CONST	LOAD_CONSTANT
3	CALL	CALL_VARARGS_METHOD_

The traces show semantically equivalent behavior despite different instruction naming conventions. However, divergences can reveal real bugs. For example, we observed consistent stack divergences at `BINARY_OP`/`PY_NUMBER_ADD_` instructions where floating-point values differed (e.g., 1.02 vs. 1.019). Tracing this to the GraalPy source revealed a fast-path optimization in the double-handling code:

```
if (child1Value instanceof Double) {
    double child1Value_ = (double) child1Value;
    state_0 = state_0 | 0b100000000;
    this.state_0_ = state_0;
```

```
    quicken(state_0, $bc, $bci);
    return PyNumberAddFastPathsBase.doDD(
        child0Value_, child1Value_);
}
```

This fast-path introduced subtle precision differences compared to CPython's implementation.

5 Technical Challenges

5.1 Instruction Mapping

Different interpreters use different bytecode instruction sets. CPython's `LOAD_CONST` corresponds to GraalPy's `LOAD_CONSTANT`. We maintain an instruction mapping table that normalizes semantically equivalent operations, allowing meaningful comparison despite surface-level differences.

5.2 Resynchronization

A more challenging problem arises when instruction sequences differ structurally. Consider class definition: CPython uses the sequence `PUSH_NULL; LOAD_BUILD_CLASS; ...; PRECALL; CALL` while GraalPy uses `BUILD_CLASS_; STORE_LOCAL; ...; CALL_VARARGS_METHOD_`. The interpreters execute different numbers of instructions for the same semantic operation.

Table 2 shows an example of structural divergence during class construction.

Table 2. Structural divergence in class definition requiring resynchronization. The interpreters use different instruction counts for equivalent semantics.

CPython		GraalPy	
0	PUSH_NULL	0	—
1	LOAD_BUILD_CLASS	1	BUILD_CLASS_
2	—	2	STORE_LOCAL
⋮	⋮	⋮	⋮
14	PRECALL	14	—
15	CALL	15	CALL_VARARGS_METHOD_

Our resynchronization strategies include: (1) instruction grouping based on semantic boundaries, (2) synchronization at function call/return points, and (3) heuristic alignment based on stack state similarity. This remains an active area of development.

6 Implementation Status

Our current implementation includes:

- A bisimulation driver for coordinating CPython and GraalPy execution
- A protocol implementation for interpreter instrumentation and control
- A test case generation system based on Python's test suite and synthetic programs
- Divergence aggregation and filtering infrastructure
- Initial resynchronization strategies for common divergence patterns

7 Related Work

Our work builds on several research threads. Sangiorgi's comprehensive treatment of bisimulation and coinduction [2] provides the theoretical foundation. Desharnais and Brunthaler [1] explore verified virtual machines for dynamic languages, representing the formal verification end of the spectrum.

Differential testing, as pioneered by McKeeman [3] and applied to compilers by Yang et al. [4], shares our goal of finding semantic differences without formal specifications. Our approach differs by operating at the interpreter instruction level rather than program output level, enabling finer-grained analysis.

Fuzzing techniques from the security domain, particularly coverage-guided fuzzing [5], suggest avenues for systematic test case generation. Combining bisimulation with fuzzing-inspired program mutation is a promising direction for future work.

8 Conclusion and Future Work

We have presented a practical bisimulation approach for detecting semantic divergences between interpreter implementations. By observing instruction execution, stack states, and function calls in lockstep, we can identify differences that escape traditional testing. Our divergence aggregation and analysis techniques help trace these differences to root causes in interpreter source code.

Future work includes:

- **Root-cause analysis tooling:** Automated correlation of divergence patterns with interpreter source code locations
- **Fuzzer integration:** Systematic test case mutation and automatic program generation, guided by coverage metrics
- **Broader interpreter support:** Extending the protocol to PyPy (JIT), MicroPython (embedded), and potentially other language ecosystems (Lua, WebAssembly, SQL interpreters)
- **Debugging integration:** Using bisimulation divergence data to assist developers in debugging interpreter reimplementations

References

[1] Martin Desharnais and Stefan Brunthaler. Towards efficient and verified virtual machines for dynamic languages. In *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '21)*, 2021.

[2] Davide Sangiorgi. *Introduction to Bisimulation and Coinduction.* Cambridge University Press, 2011.

[3] William M. McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1):100–107, 1998.

[4] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*, pages 283–294, 2011.

[5] Michal Zalewski. American fuzzy lop (AFL). https://lcamtuf.coredump.cx/afl/, 2014.

T3 - Tree Transformation Tool

STEPHAN MITTE, Martin-Luther University Halle-Wittenberg, Germany

Das Werkzeug *T3 - Tree Transformation Tool* wurde für die Transformation von Abstrakten Syntaxbäumen entwickelt. Im Gegensatz zu anderen Werkzeugen generiert *T3* eine Attributierte Grammatik, mit der spezifizierte Transformationen auf dem Baum angewendet werden. Zur einfachen Beschreibung dieser Transformationen wird die neue Transformationssprache *T2L - Tree Transformation Language* genutzt. Diese enthält allgemeine Sprachfeatures zur Berechnung der Baumtransformationen und kann zusätzlich auf die Attribute des abstrakten Syntaxbaums zugreifen, wodurch komplexe Transformationen definiert werden können.

1 Einleitung

Abstrakte Syntaxbäume (AST) sind eine verbreitete Art der Repräsentation von Programmcode. Diese werden insbesondere in Übersetzerbau-Werkzeugen eingesetzt, um verschiedene Analysen und Operationen durchzuführen. Die von Knuth in [11] vorgestellten Attributierten Grammatiken (AG) haben sich für die Spezifikation von Eigenschaften in den Bäumen etabliert. Diese repräsentieren in kompakter Form die semantischen Informationen der AST-Knoten und werden auch für Berechnungen auf den Bäumen genutzt. Im Übersetzerbau ist der AST eine Zwischendarstellung des Programmcodes, welcher während der Übersetzung in andere Repräsentationen, wie zum Beispiel einer *Intermediate Language*, umgewandelt wird. Diese Transformationen werden typischerweise manuell implementiert, was fehleranfällig ist und bei größeren Bäumen sehr unübersichtlich werden kann. Zudem muss der Entwickler selbst festlegen, welche Transformation auf welchen Teilbaum angewendet werden soll, obwohl dieses Problem mithilfe von *Pattern-Matching* Algorithmen gelöst werden kann.

Eine andere Möglichkeit ist die automatische Generierung der Transformationen aus einer Spezifikation. In der Literatur gibt es bereits viele Beispiele für solche Transformationswerkzeuge. Im Kontext des Übersetzerbaus existieren individuelle Lösungen für AST-Transformationen wie zum Beispiel [7, 8, 15]. Andere Baumtransformationen wurden im Rahmen der modellgetriebenen Softwareentwicklung entwickelt. Dabei wird der AST durch ein Meta-Modell beschrieben und mithilfe von Modelltransformationen umgewandelt.

In vielen Übersetzerbau-Werkzeugen, wie zum Beispiel Eli [6], werden AG als formaler Rahmen für die statische Semantik genutzt. Bei der Transformation des AST wurden diese aber bisher gar nicht ([8]) oder nur mit Einschränkungen ([7, 15]) genutzt. In der modellbasierten Softwareentwicklung existieren Arbeiten für Modelltransformationen basierend auf AG ([3], [4]). Diese erfordern aber mit den zugrundeliegenden Meta-Modellen eine zusätzliche Abstraktionsebene, die für die Transformation von Abstrakten Syntaxbäumen nicht benötigt wird. Das Werkzeug *T3* adressiert dieses Problem und bietet einen direkteren Weg. Dabei werden die Transformationen durch generierte Attribute einer AG beschrieben und mithilfe der Berechnungsstrategie der AG automatisch auf den AST angewendet. Für die einfachere Abbildung der Transformationen auf Attribute und Regeln einer AG wurde die Spezifikationssprache *T2L - Tree Transformation Language* entworfen.

2 Verwandte Arbeiten

Die Transformation von attributierten Abstrakten Syntaxbäumen wurde bereits in vielen Arbeiten untersucht. Für einige Übersetzerbau-Werkzeuge wurden spezielle Transformationssprachen entwickelt, die in dem jeweiligen Werkzeug fest integriert sind. Eine Alternative dazu sind Modelltransformationen in der modellgetriebenen Softwareentwicklung, die ein deutlich größeres

Author's Contact Information: Stephan Mitte, Martin-Luther University Halle-Wittenberg, D-06120 Halle, Germany.

Anwendungsfeld abdecken. Im Folgenden werden wichtige Arbeiten aus beiden Bereichen vorgestellt und diskutiert.

2.1 Spezifikationssprachen

Im Rahmen des Projekts PROSPECTRA [12] wurde die Spezifikationssprache OPTRAN [15] entwickelt, mit der statische Transformationen des AST beschrieben werden können. Transformationsregeln bestehen aus einem Baummuster, Kontextbedingungen über den Attributen des Baummusters und Aktionen. Durch die Anwendung der Transformation müssen aber unter Umständen Attribute und deren Abhängigkeiten neu berechnet werden. In [14] wird die automatische Neuberechnung der Attribute in OPTRAN diskutiert, wobei diese im schlechtesten Fall nach jeder Transformation neu berechnet werden müssen. Ein weiterer Nachteil ist, dass die Reihenfolge, in der die Regeln angewendet werden, nur durch globale Strategien wie zum Beispiel: *top-down, left-right* festgelegt werden kann. Durch den Einsatz einer AG bei der Transformation des AST wird automatisch eine Berechnungsstrategie ermittelt, sodass Attribute nicht neu berechnet werden müssen. Durch zusätzliche Abhängigkeiten zwischen den Attributen können auch andere Berechnungsreihenfolgen erzwungen werden. Dies spricht für den Einsatz einer AG bei der Transformation.

Das Werkzeug Puma ist Bestandteil des Projekts *A Tool Box for Compiler Construction* [5] und dient ebenfalls der Transformation und Manipulation des AST. In Puma werden Transformationen in Subroutinen (Prozeduren, Funktionen) eingebettet. Jede Subroutine definiert eine Menge von Transformationsregeln. Wenn ein Teilbaum auf das Baummuster einer Regel passt, werden die entsprechenden Anweisungen der Regel ausgeführt. In Beispiel 2.1 wird die Subroutine `ILCode` auf beliebige Bäume angewendet. Der Zwischencode wird durch die Funktion `Emit` aber nur für binäre Ausdrücke und Konstanten erzeugt. Dadurch können in Puma komplexere Transformationen beschrieben werden als in OPTRAN. Jedoch benötigt Puma ein spezielles Format des AST, welches in [5] genutzt wird. Zudem werden die Transformationen in einem generierten Programm-Modul implementiert, welches ebenfalls nur in [5] verwendet werden kann. Dadurch ist Puma sehr stark an das Übersetzerbau-Werkzeug gebunden und eignet sich nicht für die Verwendung in anderen Werkzeugen.

```
1 PROCEDURE ILCode(t: TREE)
2     Binary (Lop, Rop, {Plus}, TypeCode) :-
3         ILCode(Lop); ILCode(Rop);
4         Coerce(Lop::Type, Rop::Type);
5         Emit(ADD, TypeCode); .
6     IntConst  (Value := val) :- Emit(IntType, val); .
7     RealConst (Value := val) :- Emit(RealType, val); .
```

Beispiel 2.1. Auszug aus einem Puma-Programm zur Zwischencode-Generierung

2.2 Modelltransformationen

In der modellgetriebenen Softwareentwicklung ist der AST ein Modell für den Programmcode. Baumtransformationen werden in diesem Kontext als Modelltransformationen bezeichnet und in vielen Arbeiten thematisiert. Für abstrakte Syntaxbäume wurde ein Meta-Modell im MOF-Standard [17] unter dem Namen *Abstract Syntax Tree Metamodeling* (ASTM) [16] veröffentlicht, um neben Transformationen auch Programmanalysen über verschiedene Programmiersprachen hinweg zu erleichtern. Dazu wird der Programmcode zunächst in einen sprachspezifischen AST (SAST) umgewandelt, welcher durch ein SAST Meta-Modell beschrieben wird. Anschließend wird der SAST auf einen Generic AST (GAST) abgebildet, welcher seinerseits durch ein GAST Meta-Modell beschrieben wird. Ein weiteres Meta-Modell wird für den Target-Code benötigt, um auch

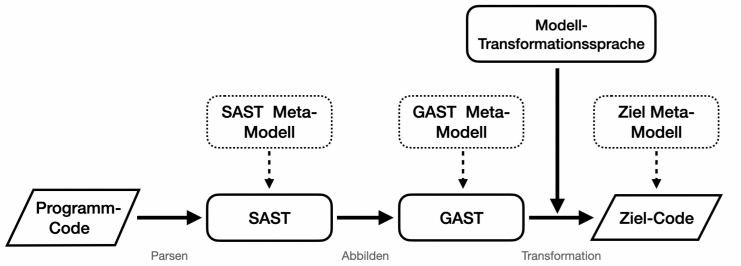


Fig. 1. Modelltransformation eines Abstrakten Syntaxbaums mit dem *Abstract Syntax Tree Metamodeling*

dessen Struktur zu definieren. Nachdem die verschiedenen Meta-Modelle beschrieben und die entsprechenden Modelle daraus abgeleitet wurden, kann die Transformation des GAST in den Target-Code mit etablierten Modelltransformationssprachen definiert werden. Der gesamte Ablauf dieser Transformation ist in Abbildung 1 skizziert.

Andere Modelltransformationssprachen wie zum Beispiel ATL [10], GReAT [1] oder auch YAMTL [2] funktionieren in ähnlicher Weise. Zunächst wird das Meta-Modell für den AST und den Target-Code beschrieben. Dann muss der AST entsprechend des Meta-Modells aufgebaut werden, bevor die Transformationen zwischen den Modellen definiert werden können.

In [3, 4] werden Modelltransformationen mithilfe von Attributierten Grammatiken beschrieben. Dazu werden die Meta-Modelle zunächst in eine AG überführt. Die Korrektheit dieser Abbildung wird ausführlich in [3] gezeigt. Anschließend werden die Transformationen mithilfe einer weiteren AG beschrieben. Während beide Arbeiten die automatische Berechnung der Attribute durch die AG hervorheben, werden keine Informationen zur Transformationsreihenfolge und Auswahl der Regeln angegeben. In den Beispielen in [3] wird stets eine *top-down* Traversierung genutzt. Allerdings geht aus der Arbeit keine globale Strategie hervor.

2.3 Zusammenfassung

Die vorgestellten Spezifikationssprachen bieten auf unterschiedliche Art und Weise Möglichkeiten zur Transformation des AST an. Die Spezifikationssprache OPTRAN [15] erfordert aber unter Umständen die Neuberechnung von Attributen und deren Abhängigkeiten. Zudem wird ein systemspezifisches Modul generiert, welches nur im Kontext des Übersetzerbau-Werkzeugs verwendet werden kann, was die Integrierbarkeit stark einschränkt. Mit Puma [7] können komplexere Transformationen beschrieben werden. Jedoch wird eine spezielle Repräsentation des AST benötigt, welche nur im Kontext von [5] genutzt wird, was ebenfalls die Integrierbarkeit in andere Übersetzerbau-Werkzeuge erschwert.

Modelltransformationen erfordern ein eigenes Meta-Modell für den AST und ein weiteres für den Target-Code. Standards, wie MOF [17], werden im Übersetzerbau nur selten berücksichtigt. Stattdessen werden viele Aufgaben durch manuelle Implementierungen bearbeitet oder Generatoren aus einer Spezifikation erzeugt und genutzt, wie im Beispiel von Lex [13] und Yacc [9] bei der lexikalischen und syntaktischen Analyse. Zudem wird mit den Meta-Modellen eine zusätzliche Abstraktion eingeführt, die für die Transformation des AST an sich nicht benötigt wird, da dieser in Kombination mit einer AG bereits alle wichtigen Informationen über den Programmcode enthält. Auch wenn dieser nicht durch ein standardisiertes Meta-Modell beschrieben wird, so ist er ein weit verbreitetes Modell im Übersetzerbau.

Andererseits wird bereits in [3, 4] beschrieben, wie Modelltransformationen auch mit AG durchgeführt werden können, indem das Meta-Modell auf eine AG abgebildet wird. Dies zeigt die

Möglichkeiten von Attributierte Grammatiken bei der Transformation von Abstrakten Syntaxbäumen.

3 Methodik

Die Werkzeuge Puma [7] und OPTRAN [15] sind aufgrund der verschiedenen Repräsentationen des AST und systemspezifischen Modulen stark mit dem jeweiligen Übersetzerbau-Werkzeug gekoppelt. Dies hat negative Auswirkungen auf die Integrierbarkeit. Andererseits verwenden viele Übersetzerbau-Werkzeuge bereits eine Attributierte Grammatik, um auf die Attribute des AST zuzugreifen und die Semantik zu beschreiben. Eine AG kann um neue Attribute erweitert werden, ohne die anderen Attribute neu berechnen zu müssen oder bestehende Abhängigkeiten zu verletzen. Aufgrund dieser Vorteile werden die Transformationen von *T3* auf Attribute und Regeln einer AG abgebildet, welche in dem entsprechenden Übersetzerbau-Werkzeug bereits verwendet wird.

3.1 Muster und Baumtransformationen

Bei der Transformation des AST in Zwischencode werden die Baumtransformationen anhand eines Musters der abstrakten Syntax und der Attribute, die in diesem Muster vorkommen, ausgewählt. Ein Muster wird durch Definition 1 beschrieben.

Definition 1. Sei $G \triangleq (T, N, P, Z)$ eine kontextfreie Grammatik zur Definition einer abstrakten Syntax. Ein **Muster über** G ist induktiv wie folgt definiert:

i. Jedes $A \in N$ ist ein Muster vom Typ A
ii. Falls $A ::= A_1 \ldots A_n$ und $m_1, \ldots, m_n$ Muster vom Typ $A_1, \ldots, A_n$ sind, dann ist $A(m_1, \ldots, m_n)$ Muster vom Typ A.

Damit eine Baumtransformation angewendet wird, muss das Muster auf einen Teilbaum des AST passen. Diese Relation ist in Definition 2 erläutert.

Definition 2. Sei t ein Unterbaum eines abstrakten Syntaxbaums zu G mit einer Wurzel vom Typ A. Der **Unterbaum t passt auf ein Muster** m genau dann, wenn:

i. $m = A$ oder
ii. $m = A(m_1, \ldots, m_n)$ und die Unterbäume $t_1, \ldots, t_n$ passen auf die Muster $m_1, \ldots, m_n$.

Wenn das Muster einer Transformationsregel auf einen Teilbaum passt, werden die Bedingungen innerhalb der Regel ausgewertet und die Aktionen des ersten erfüllten Wächters ausgeführt. Diese Aktionen können entweder Zwischencode generieren, oder weitere Regeln aufrufen. Definition 3 vereint all diese Eigenschaften einer Baumtransformation.

Definition 3. Sei $AG \triangleq (G, A, R, B)$ eine attributierte Grammatik. Eine **Baumtransformation** ist eine Funktionsdefinition

$$\begin{aligned} \mathcal{T}[\![m]\!]a_1 \ldots a_n &= Aktionen_1 && \textbf{if } Bedg_1 \\ &= Aktionen_2 && \textbf{if } Bedg_2 \\ &\ldots \\ &= Aktionen_k && \textbf{if } Bedg_k \end{aligned}$$

wobei

- $\mathcal{T}$ der Name der Transformation,
- m ein Muster über der kontextfreien Grammatik G,
- $a_1 \ldots a_n$ die beteiligten Attribute,
- $Aktionen_i$ eine Folge von Aktionen und

- $Bedg_i$ boolesche Terme über den im Muster beteiligten Attributen ist.

Beispielsweise würde die Baumtransformation $\mathcal{B}$ in Abbildung 2 auf alle Teilbäume angewendet werden, welche eine binäre Addition beschreiben. Je nachdem, ob das Attribut *type* den Wert *integer* oder *float* hat, werden zunächst die Transformationen von $Expr_1$ und $Expr_2$ durchgeführt und anschließend der Zwischencodebefehl für Integer-Addition bzw. Float-Addition generiert.

$$
\begin{aligned}
&\mathcal{B}[\![BinOp(Expr_1, Op(+), Expr_2)]\!] \quad type, val \\
&\quad = \mathcal{B}[\![Expr_1]\!] && \textbf{if } Op.type = integer \\
&\quad\ \ \mathcal{B}[\![Expr_2]\!] \\
&\quad\ \ BinOp.val = \text{IntADD}(\text{Expr}_1\text{.val}, \text{Expr}_1\text{.val}) \\
&\quad = \mathcal{B}[\![Expr_1]\!] && \textbf{if } Op.type = float \\
&\quad\ \ \mathcal{B}[\![Expr_2]\!] \\
&\quad\ \ BinOp.val = \text{FltADD}(\text{Expr}_1\text{.val}, \text{Expr}_1\text{.val})
\end{aligned}
$$

Fig. 2. Baumtransformation einer binären Addition

3.2 Abbildung von Baumtransformationen

Aus Definition 1 folgt, dass ein Muster über G sowohl ein einzelnes Nichtterminal $A \in N$, als auch ein Term $A ::= A_1 \ldots A_n$ sein kann. Terme können direkt auf die Produktionen der AG abgebildet werden. Dagegen müssen einzelne Nichtterminale zunächst den entsprechenden Produktionen zugeordnet werden. Sei $\mathcal{T}[\![A]\!]$ eine Transformation mit dem Muster $m = A$. Dann muss $\mathcal{T}$ auf alle Produktionen $p \in P$ abgebildet werden, für die A auf der rechten Seite vorkommt: $p : X ::= \ldots A \ldots$. Abgesehen von dem Wurzelkonten können so alle Vorkommen eines Nichtterminals abgebildet werden.

Bei der Transformation des AST in Zwischencode müssen die Aktionen einer Baumtransformation den Wert eines Attributs berechnen, welches den Zwischencode repräsentiert. Dafür wird in der AG ein neues Attribut definiert, welches die Berechnungsreihenfolge der anderen Attribute nicht verändern darf, damit keine zyklischen Abhängigkeiten entstehen. Gleichzeitig dürfen auch bestehende Abhängigkeiten nicht verletzt werden. Damit dies gelingt, wird in dem Werkzeug *T3* nur lesend auf die anderen Attribute des AST zugegriffen.

Die booleschen Terme einer Baumersetzungsregel müssen in einer Fallunterscheidung ausgewertet werden. Diese muss bei der Berechnung der Attribute in der AG ebenfalls abgebildet werden. Wenn die Sprachdefinition der AG kein passendes Sprachkonzept bereitstellt, kann diese Fallunterscheidung alternativ über externe Funktionen realisiert werden.

Die Reihenfolge, in der die Baumtransformationen ausgewertet werden, kann über die Berechnungsreihenfolge der AG und zusätzlich eingeführte Attributierungsregeln beeinflusst werden. Diese Regeln erzwingen aufgrund von Abhängigkeiten zwischen den Attributen eine bestimmte Auswertereihenfolge, dürfen aber bestehende Abhängigkeiten nicht verletzen.

3.3 Spezifikation von Baumtransformationen

Die Beschreibung eines Musters einer Baumtransformation muss alle Informationen enthalten, um bei der Abbildung die Aktionen den richtigen Produktionen zuzuordnen. Neben den bereits betrachteten Mustern aus Definition 1 können in anderen Werkzeugen, wie zum Beispiel Puma [7], auch Variablen und *Don't Care* Symbole zur Beschreibung von komplexen Mustern genutzt

werden. Bei der Abbildung auf die AG müssen die Variablen und *Don't Care* Symbole dann wieder durch die entsprechenden Terme und Nichtterminale ersetzt werden. Eine Möglichkeit, um diese Informationen kompakt darzustellen, bieten Funktionssignaturen und Typ-Konstruktoren aus funktionalen Programmiersprachen. Dabei wird das Muster einer Baumtransformation als Funktionssignatur verstanden. Damit können Terme beschrieben werden, welche direkt auf eine Produktion abgebildet werden. Einzelne Nichtterminale, sowie Muster mit Variablen und *Don't Care* Symbole werden dagegen durch Typ-Muster beschrieben, welche analog zu Typ-Konstruktoren zu verstehen sind.

Für die Auswahl der auszuführenden Aktionen innerhalb einer Baumtransformation muss eine Syntax definiert werden, mit der sowohl die booleschen Terme, als auch die zugehörigen Aktionen spezifiziert werden können. Dieses Sprachkonstrukt hat im Kontext von Baumtransformationen Ähnlichkeit mit Wächtern in funktionalen Sprachen.

Eine andere Schwierigkeit betrifft die Grundblöcke im Zwischencode, welche bei der Transformation des AST definiert werden müssen. Alle Befehle im Zwischencode sind einem Grundblock zugeordnet. Diese Zuordnung muss zunächst aus dem AST ermittelt werden. Weiterhin endet jeder Grundblock mit einem Sprungbefehl, dessen Sprungziel ebenfalls berechnet werden muss. Zwischencodebefehle können durch spezielle Sprachkonstrukte Grundblöcken explizit zugeordnet werden. Andernfalls muss die Zuordnung aus dem Kontext ermittelt werden.

Um diese Schwierigkeiten zu überwinden und die Abbildung zu erleichtern, wurde die Baumtransformationssprache T2L entwickelt, welche die Beschreibung von Baumtransformationen als funktionales Programm ermöglicht. Die einzelnen Transformationen werden dabei durch Funktionen dargestellt. Die Muster der Transformationen werden durch Funktionssignaturen und Typ-Konstruktoren definiert. Mithilfe spezieller Sprachkonstrukte können Grundblockgraphen konstruiert und Fallunterscheidungen ausgewertet werden. Das Werkzeug *T3* generiert dann automatisch aus dieser Spezifikation eine AG, mit der die beschriebenen Transformationen berechnet werden.

4 Fallstudie zur Generierung von Zwischencode

In diesem Abschnitt wird gezeigt, wie das Werkzeug *T3* für die Generierung von Zwischencode innerhalb des Übersetzerbauwerkzeugs Eli [6] eingesetzt werden kann. Die Zwischencodebefehle sollen als Folge von Tupeln dargestellt werden und das Attribut `symReg` für die Tupelnummern. Die Baumtransformation in Beispiel 4.1 transformiert den Teilbaum einer bedingten Fallunterscheidung, wobei auch der Grundblockgraph verändert wird. Im Folgenden werden wichtige Auszüge der T2L-Spezifikation dieser Transformation beschrieben.

4.1 Spezifikation der Transformationen

Export. Der Zwischencode soll in dem Attribut `il` berechnet werden. In dem Übersetzerbauwerkzeug Eli [6] können Zwischencodebefehle als strukturierter Text mithilfe eines *Pattern-Based-Text-Generator* erzeugt werden. Dieser strukturierte Text wird als Attribut vom Typ *PTGNode* dargestellt und mit dem Wert *PTGNULL* initialisiert. Die Transformationen, welche dieses Attribut berechnen, haben den Namen `T` und werden in Listing 1 definiert.

```
1 EXPORT
2     il::PTGNode("PTGNULL") as T
```

Listing 1. Spezifikation des Attributs und Transformationsname für den Zwischencode

Data. Während Terme als Muster einer Transformation direkt auf eine Produktion abgebildet werden können, müssen einzelne Nicht-Terminale durch ein Typ-Muster, wie in Listing 2 dargestellt,

$$
\begin{aligned}
&\mathcal{T}[\![Expr(Expr, Stats, Stats)]\!] \quad symReg, priType \\
&\quad = \quad \mathcal{T}[\![Expr_2]\!] \quad CondIL(Expr_2.symReg, b1, b2) \\
&\qquad \text{b1:} \\
&\qquad\quad \mathcal{T}[\![Stats_1]\!] \quad JumpIL(b3) \\
&\qquad \text{b2:} \\
&\qquad\quad \mathcal{T}[\![Stats_2]\!] \quad JumpIL(b3) \\
&\qquad \text{b3:} \\
&\qquad\quad IntPhiIL(Stats_1, Stats_2) \qquad \textbf{if } \text{Expr}_1\text{.priType} = \text{integer}
\end{aligned}
$$

Beispiel 4.1. Baumtransformation einer bedingten Fallunterscheidung

spezifiziert werden. Das Typ-Muster `Expression` soll auf alle Nichtterminale `Expr` angewendet werden, um Zwischencodebefehle für eine mögliche Typanpassung zu generieren.

```
1 DATA
2     data Expression = Expr
```

Listing 2. Spezifikation der Typ-Muster

Transformationen. Die Spezifikation der Transformationsregel ist in Listing 3 dargestellt. In Zeile 3 wird zunächst `Expr[2]` durch den Aufruf einer anderen Transformation bearbeitet. In Zeile 5 wird dann der aktuelle Grundblock beendet und anschließend ein neuer, mit dem symbolischen Label `b1`, begonnen. Dabei handelt es sich um vordefinierte Aktionen, welche den Aufbau des Grundblockgraphen und die Berechnung der konkreten Labels durchführen. Das Schlüsselwort `CASE` in Zeile 15 generiert die Fallunterscheidung für die beschriebenen Wächter und deren Aktionen.

```
1  RULE ifExpr
2  TRANSFORM T: Expr(Expr, Stats, Stats) COMPUTE
3      Eval.T(Expr[2])
4      PTGCondIL(Expr[2].symReg + 1, Expr[2].symReg, b1, b2)
5    ENDBB
6    BEGINBB b1:
7      Eval.T(Stats[1])
8      PTGJumpIL(Stats[1].symReg + 1, b3)
9    ENDBB
10   BEGINBB b2:
11     Eval.T(Stats[2])
12     PTGJumpIL(Stats[2].symReg + 1, b3)
13   ENDBB
14   BEGINBB b3:
15     CASE
16       | Expr[1].priType == OilTypeintType
17         = PTGIntPhiIL(Expr[1].symReg, Stats[1].symReg, Stats[2].symReg)
18       /* ... */
19       otherwise = PTGError()
20     END
21 END
```

Listing 3. Transformation von bedingten Fallunterscheidungen

Das Werkzeug *T3* generierte aus dieser Spezifikation die in Listing 4 dargestellte Produktion. In Zeile 3 wird zunächst die Transformation von `Expr[2]` berechnet. In den Zeilen 5, 9, 13 werden die vordefinierten Aktionen zum Beenden eines Grundblocks durchgeführt und in den darauffolgenden Zeilen ein neuer Grundblock mit dem entsprechenden Label begonnen. Die einzelnen Grundblöcke

werden auf einem Stack verwaltet und nachdem der letzte Grundblock geschlossen wurde zu einem Grundblockgraphen vereinigt.

```
1  RULE ifExpr: Expr ::= Expr Stats Stats
2  COMPUTE
3      Expr[2].ilIn = Expr[1].ilIn;
4      % End current BB and Begin new BB
5      BBPushStack(ilCodeGenifExpr_ilEndBB_0(Expr[2].ilOut,Expr[2].pri,Expr[2].post,...));
6      Stats[1].ilIn = BeginBB();
7      Stats[1]._T3lab = ADD(Expr[2]._T3lab,1);
8      % End current BB and Begin new BB
9      BBPushStack(ilCodeGenifExpr_ilEndBB_1(Stats[1].ilOut, Stats[1].symReg, ...));
10     Stats[2].ilIn = BeginBB();
11     Stats[2]._T3lab = ADD(Stats[1]._T3lab, 1);
12     % End current BB and Begin new BB
13     BBPushStack(ilCodeGenifExpr_ilEndBB_2(Stats[2].ilOut, Stats[2].symReg, ...));
14     Expr[1].ilOut = ilCodeGenifExpr_il(BeginBB(), Expr[1].pri, Stats[1].symReg, ...);
15     Expr[1]._T3lab = ADD(Stats[2]._T3lab, 1);
16 END;
```

Listing 4. Generierte AG der bedingten Fallunterscheidung

Die anderen Aktionen werden in den jeweiligen `ilCodeGen`-Funktionen implementiert, welche als externe C-Funktionen eingebunden werden. Die Aktionen des ersten Grundblocks sind in Listing 5 dargestellt. Die externen PTG-Funktionen berechnen in der Variable `code` einen konkreten *PTGNode*-Wert. Die Fallunterscheidung in den Zeilen 3-7 implementiert das Typ-Muster `Expression` für `Expr[2]`.

```
1  extern PTGNode ilCodeGenifExpr_ilEndBB_0(...)
2  {
3      if(Expression_pri==OilTypeintType && Expression_post==OilTypefltType){
4          code = PTGSeq(code, PTGIntToFlt(Expression_symReg, Expression_symReg - 1));
5      } else {
6          code = PTGSeq(code, PTGSkip());
7      }
8      code = PTGSeq(code, PTGCondIL(Expr_2_symReg + 1, Expr_2_symReg, b1, b2));
9      return code;
10 }
```

Listing 5. Generierte Hilfsfunktion für die bedingte Fallunterscheidung

Das Typ-Muster `Expression` wird durch die in Listing 6 dargestellte `SYMBOL` Transformation spezifiziert. Diese generiert Attribute und Regeln für Zwischencodebefehle, welche die automatischen Typanpassung von `int` zu `float` berechnen und in alle Transformationen eingesetzt werden, in denen das Nichtterminal `Expr` durch eine `T` Transformation ausgewertet wird. Dies ist in Listing 3 für `Expr[2]` der Fall.

```
1  SYMBOL
2  TRANSFORM T: Expression COMPUTE
3      Eval.T(Expression)
4      CASE
5          | Expression.priType == OilTypeintType && Expression.postType == OilTypefltType
6              = PTGIntToFlt(Expression.symReg)
7          otherwise = PTGSkip()
8      END
9  END
```

Listing 6. Transformation des Typ-Musters `Expression`

Auf ähnliche Weise können mithilfe des Werkzeugs *T3* alle benötigten Attribute und Produktionen einer AG für die gesamte Berechnung von Zwischencode generiert werden.

5 Fazit

Die Transformation des Abstrakten Syntaxbaums wurde in der Vergangenheit durch fehleranfällige manuelle Implementierungen gelöst. Daher wurden Spezifikationssprachen entwickelt, welche aber

schlecht in andere Übersetzerbau-Werkzeuge zu integrieren sind. Auch Modelltransformationen haben das Problem nicht gelöst, sondern nur eine zusätzliche Abstraktionsebene hinzugefügt. Das Werkzeug *T3* bildet die Transformationen auf eine Attributierte Grammatik ab, welche in vielen Übersetzerbau-Werkzeugen bereits Verwendung findet. Dabei können die neu generierten Attribute zur Beschreibung der Transformationen genutzt werden, ohne die anderen Attribute des AST neu berechnen zu müssen. Aufgrund der Abhängigkeiten zwischen den Attributen, welche die Transformationen berechnen, erfolgt die Auswertereihenfolge stets bottom-up. Jedoch kann die Reihenfolge zwischen Knoten mit gleichem Eltern-Knoten beliebig definiert werden. Zudem können Transformationen in *T2L* auch kombiniert werden. Obwohl das Werkzeug *T3* für den Einsatz in dem Übersetzerbau-Werkzeug Eli [6] entwickelt wurde, können die beschriebenen Transformationen auch in anderen Übersetzerbau-Werkzeugen genutzt werden. Als Voraussetzung muss eine AG in dem Werkzeug verwendet werden, sodass die generierten Attribute und deren Abhängigkeiten in diese AG integriert werden können.

Neben der Integrierbarkeit hat dieser Ansatz auch den Vorteil, dass die Korrektheit der Transformationen gezeigt werden kann. Dieser Nachweis und weitere Analysen sollen in weiterführenden Arbeiten erbracht werden, um die korrekte Funktionsweise des Werkzeugs festzustellen. Aktuell können nur Transformationen mit flachen Mustern und Wächtern (Eltern → Kindknoten) beschrieben werden. Aber bereits für den Aufbau von Zwischencodebäumen oder optimierenden Transformationen müssen auch verschachtelte Muster und Wächter beschrieben werden können. Dies erfordert komplexere Typ-Muster.

Literatur

[1] Daniel Balasubramanian, Anantha Narayanan, Christopher van Buskirk, and Gabor Karsai. 2006. The Graph Rewriting and Transformation Language: GReAT. *Electronic Communications of the EASST* 1 (2006), 1–15. doi:10.14279/tuj.eceasst.1.30

[2] Artur Boronat. 2018. Expressive and efficient model transformation with an internal DSL of Xtend. In *Proceedings of the 21st ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS '18)*. Association for Computing Machinery, New York, NY, USA, 78–88. doi:10.1145/3239372.3239387

[3] Daniel Calegari and Marcos Viera. 2015. Model-driven engineering based on attribute grammars. In *Programming Languages: 19th Brazilian Symposium (SBLP 2015) (Lecture Notes in Computer Science, Vol. 9325)*. Springer International Publishing, Cham, Switzerland, 112–127. doi:10.1007/978-3-319-24012-1_8

[4] May Dehayni and Louis Féraud. 2003. Attribute grammars as tools for model-to-model transformations. In *Proceedings of the European Workshop on Model Driven Architecture: Foundations and Applications (MDAFA 2003) (CTIT Technical Report, 03-25)*. University of Twente, Enschede, The Netherlands, 73–82.

[5] H Emmelmann and J Grosch. 1990. *A tool box for compiler construction.* Technical Report. GMD Karlsruhe, Technical Report.

[6] Robert W Gray, Steven P Levi, Vincent P Heuring, Anthony M Sloane, and William M Waite. 1992. Eli: A complete, flexible compiler construction system. *Commun. ACM* 35, 2 (1992), 121–130.

[7] Josef Grosch. 1991. *Puma — A Generator for the Transformation of Attributed Trees.* Cocktail Document 26. GMD Forschungsstelle an der Universität Karlsruhe, Karlsruhe, Germany.

[8] Reinhold Heckmann and Georg Sander. 1993. TrafoLa-H Reference Manual. In *Program Development by Specification and Transformation: The PROSPECTRA Methodology, Language Family, and System*, Berthold Hoffmann and Bernd Krieg-Brückner (Eds.). Lecture Notes in Computer Science, Vol. 680. Springer-Verlag, Berlin, Heidelberg, 275–313. doi:10.1007/3-540-56733-X_154

[9] Stephen C. Johnson and Ravi Sethi. 1990. Yacc: Yet Another Compiler-Compiler. *Unix Programmer's Manual, Supplementary Documents* 2 (1990), 347–374.

[10] Frédéric Jouault and Ivan Kurtev. 2005. Transforming Models with ATL. In *Model Driven Engineering Languages and Systems (MoDELS 2005) (Lecture Notes in Computer Science, Vol. 3713)*. Springer, Berlin, Heidelberg, 128–138. doi:10.1007/11557432_10

[11] Donald E Knuth. 1968. Semantics of context-free languages. *Mathematical systems theory* 2, 2 (1968), 127–145.

[12] Bernd Krieg-Brückner, Einar W. Karlsen, Junbo Liu, and Owen Traynor. 1991. The PROSPECTRA Methodology and System: Uniform Transformational (Meta-) Development. In *VDM '91. Formal Software Development Methods (Lecture Notes in Computer Science, Vol. 551)*. Springer, Berlin, Heidelberg, 363–397. doi:10.1007/BFb0020005

[13] Michael E. Lesk and Eric Schmidt. 1975. *Lex: A Lexical Analyzer Generator.* Number 39 in Computing Science Technical Report. Bell Laboratories, Murray Hill, NJ, USA.

[14] Peter Lipps, Ulrich Möncke, Matthias Olk, and Reinhard Wilhelm. 1988. Attribute (Re) evaluation in OPTRAN. *Acta Informatica* 26 (1988), 213–239.

[15] Peter Lipps, Ulrich Möncke, and Reinhard Wilhelm. 1991. An Overview of the OPTRAN System. In *Attribute Grammars, Applications and Systems (Lecture Notes in Computer Science, Vol. 545).* Springer-Verlag, Berlin, Heidelberg, 505–506. doi:10.1007/3-540-54572-7_19

[16] OMG. 2011. *About the Abstract Syntax Tree Metamodel Specification Version 1.0.* https://www.omg.org/spec/ASTM/1.0/About-ASTM

[17] OMG. 2016. *About the Meta Object Facility Specification Version 2.5.1.* https://www.omg.org/spec/MOF

Ein Language-Server für Java-TX

RUBEN KRAFT, Baden-Wuerttemberg Cooperative State University, Germany
MARTIN PLÜMICKE, Baden-Wuerttemberg Cooperative State University, Germany

Um den Herausforderungen der zunehmenden Komplexität moderner Entwicklungsumgebungen gerecht zu werden, sind leistungsstarke Tools notwendig, die Entwicklerinnen und Entwickler bei der Nutzung von Sprachen mit Typinferenz unterstützen. Die Konzeption und der Aufbau eines Language Servers für Java-TX, einer auf Java basierenden, zwar statisch getypten aber ohne jede Typannotation auskommende Sprache, werden in dieser Arbeit behandelt. Der Server verwendet das Language Server Protocol (LSP), um gängige Integrated Development Environment (IDE)s wie Visual Studio Code und IntelliJ IDEA mit Funktionen wie Syntaxhervorhebung, Fehlerdiagnosen, Typinferenz, Quick-Fixes und Inlay-Hints zu versorgen. Die Architektur ist modular aufgebaut und nutzt LSP4J. Diese Integration ermöglicht es Entwickelnden, typlose Programme effizienter zu erstellen und zu debuggen. Am Ende werden die Herausforderungen, Designentscheidungen und mögliche Erweiterungen, wie Performanceverbesserungen und zusätzliche Editor-Features, besprochen.

1 Einleitung

1.1 Motivation

Java-TX ist eine statisch getypten Sprache, die es erlaubt, im Quellcode auf explizite Typangaben verzichtet. Der Compiler ermittelt die Typen während der Kompilierung automatisch. Das reduziert den syntaktischen Overhead.
Ein Problem dieser Herangehensweise ist, dass eingefügten Typen für den Entwickler während des Programmierens nicht direkt ersichtlich sind. Es ist daher notwendig, dem Entwickler die Typen, die der Compiler berechnet hat, zurückzugeben, um die Nachvollziehbarkeit zu verbessern. Wenn es mehrere gültige Typen gibt, ist es sinnvoll, eine Auswahlmöglichkeit zu schaffen, damit der Entwickler den passenden Typ bewusst auswählen kann.
Java-TX ist, im Gegensatz zu etablierten Sprachen wie Java, noch nicht in moderne Entwicklungsumgebungen integriert. In Java-TX jedoch müssen Entwickler oft manuell mit der Kommandozeile arbeiten und verzichten somit auf wichtige Entwicklungsunterstützung.
Ein Language Server soll diese Lücke schließen und Java-TX in moderne IDEs integrierbar machen. Dieser Server bietet Funktionen wie Syntaxhervorhebung, Typinferenzanzeige, Typauswahl und Fehlerdiagnosen. Die Absicht ist es, Java-TX in Bezug auf die Entwicklerfreundlichkeit auf das Niveau aktueller Programmiersprachen zu bringen, ohne die Vorzüge der typlosen Programmierung zu verlieren.

1.2 Zielsetzung

Die Funktionen, die in der Motivation vorgestellt wurden, sollen innerhalb eines zeitgemäßen, wiederverwendbaren Moduls umgesetzt werden. Hierbei wird besonders darauf geachtet, wie die Typinformationen, die der Compiler ermittelt, dargestellt werden, weil diese ein zentrales Merkmal und eine große Herausforderung von Java-TX sind.
Das Ziel ist es diese Erweiterung auf Basis des LSP zu bauen, um eine standardisierte und editorunabhängige Integration zu schaffen. Mit LSP ist es möglich, die entwickelte Lösung in verschiedenen Entwicklungsumgebungen zu nutzen.
Außerdem ist es wichtig, dass die Umsetzung eine geringe Latenz aufweist, wenn es um die

Authors' Contact Information: Ruben Kraft, Baden-Wuerttemberg Cooperative State University, Horb, Germany, rkraft@hb.dhbw-stuttgart.de; Martin Plümicke, Baden-Wuerttemberg Cooperative State University, Horb, Germany, pl@dhbw.de.

Bereitstellung von Typinformationen und anderen sprachspezifischen Funktionen geht. Es ist wichtig, dass Entwicklerinnen und Entwickler in Echtzeit Feedback bekommen, ohne dass ihre Arbeitsgeschwindigkeit dadurch beeinträchtigt wird.

2 Grundlagen

2.1 JavaTX - Eine typlose aber statisch getypte Sprache auf Basis von Java

Java-TX ist eine aus Java hervorgegangene Programmiersprache [4]. Sie ermöglicht es, Quellcode ohne typisierte Methoden oder Parameter zu schreiben. Während der Kompilierung ermittelt der zugrunde liegende Typinferenz- bzw. Typunifikationssalgorithmus die fehlenden Typen und setzt diese ein.
Java-TX vereint daher die Vorzüge von typlosen Sprachen, wie die hohe Ausdrucksstärke und Reduktion, mit der Sicherheit und Robustheit, die eine statische Typisierung bietet.
Außerdem weist Java-TX eine leicht angepasste Syntax im Vergleich zu klassischem Java auf, um typbedingte Konstrukte zu vermeiden oder zu vereinfachen. Alles in allem soll die Sprache den Entwicklungsprozess vereinfachen und beschleunigen, während sie dennoch die Vorteile der Typüberprüfung und einer sicheren Kompilierung nutzt.

2.2 Herausforderungen typloser Sprachen

Typlosen Programmiersprachen besitzen im Vergleich zu statisch typisierten Sprachen Vorteile. Hierzu gehören gesteigerte Flexibilität und verkürzte Entwicklungszyklen, da auf explizite Angaben von Typen im Quellcode verzichten wird.
Es ergibt sich eine Reduzierung der Entwicklungszeit, weil typbezogene Überlegungen wegfallen und die Programmlogik in den Vordergrund tritt.
Ein zentrales Problem typloser Sprachen zeigt sich jedoch in der mangelnden Transparenz während der Entwicklungszeit: Während der Entwicklungszeit, wissen Entwicklerinnen und Entwickler oft nicht, welche Typen der Compiler letztlich inferieren wird. Das macht es schwierig, das Programm zu analysieren und zu verstehen. Es wird besonders kritisch, wenn es mehrere gültige Typen gibt und der Entwickler bewusst auswählen muss.
Ein bedeutendes Ziel von typlosen, aber typsicherer Sprachen wie Java-TX sollte es daher sein, dass sie den Entwicklern geeignete Werkzeuge bieten, um inferierte Typen sichtbar zu machen. So kann man die Vorzüge von dynamischer und statischer Typisierung vereinen.

2.3 Das Language Server Protocol (LSP)

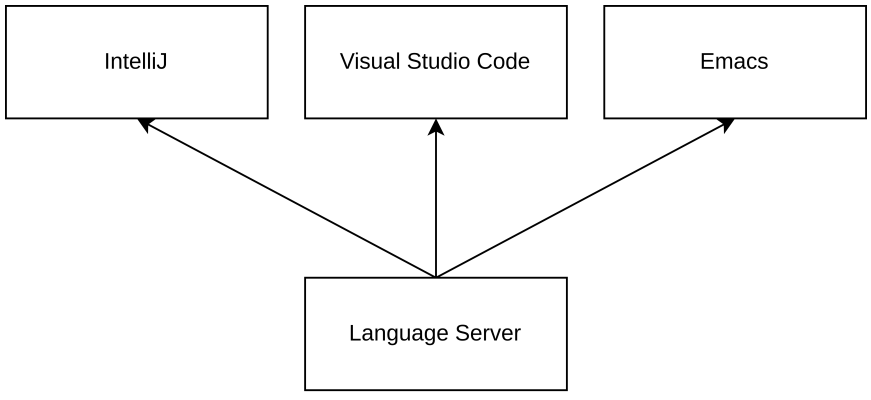


Fig. 1. Übersicht über Zweikomponenten-Architektur.

Plugins oder Extensions werden erstellt, um IDEs um neue Funktionen zu erweitern. In der Vergangenheit war es notwendig, für jede IDE eine eigene Erweiterung zu erstellen, weil sich die internen Architekturen und die Erweiterungsschnittstellen teils erheblich unterscheiden. Die Microsoft Corporation hat das LSP eingeführt, um die Wiederverwendbarkeit zu verbessern [3]. Es schafft eine weitere Abstraktionsebene zwischen der IDE (Client) und der sprachspezifischen Logik (Server). So kann ein Language Server für mehrere Clients und somit IDEs genutzt werden, ohne dass die Logik des Servers mehrfach entwickelt werden muss. Nur der Client muss an die Umgebung angepasst werden. Das Protokoll sieht vor, dass die Logik von der Anzeige getrennt wird. Dabei entstehen zwei Komponenten. Diese sind in Abbildung 1 dargestellt.

(1) Der Language-Server: Der Langauge Server enthält alle Logik und kommuniziert mit dem Client durch das LSP definierte Nachrichten.
(2) Der Client: Der Client enthält lediglich die konkrete Anzeigelogik für die IDE. Dabei besitzt jede IDE einen eigenen Client.

Generische LSP-Client-Implementierungen oder Frameworks ermöglichen eine schnelle Erstellung eines Clients. Die Kommunikation läuft dabei über LSP. Das erlaubt eine einheitliche und wiederverwendbare Interaktion, unabhängig von der spezifischen Entwicklungsumgebung.

2.3.1 Funktionsweise von LSP. Um ein Verständnis über das LSP zu bekommen, erfordert es zunächst, die Funktionen zu betrachten, die das Protokoll bietet. Das LSP legt eine standardisierte Schnittstelle fest, die es einem Editor (Client) und einem sprachspezifischen Server (Language Server) ermöglicht, miteinander zu kommunizieren. Das Ziel ist es, den Entwicklern Informationen ihres Programmcodes zu liefern, beispielsweise Fehler oder inferierte Typen.

Anschließend werden die wichtigsten LSP Funktionen erläutert.

(1) Inlayhints: Inlay Hints sind Hinweise, die im Quellcode angezeigt werden. Diese zeigen Informationen, wie etwa inferierte Typen an. Im Falle von Java-TX sind sie wichtig, um Entwicklern die durch den Compiler ermittelten Typen darzustellen, ohne dass der Quellcode verändert wird. Deswegen sind sie oft unter dem Namen Type Hints bekannt.

```
public class Test{
    public  java.lang.Integer method1(){

        return 1;
    }
}
```

Fig. 2. Beispielhaftes Inlayhint

(2) Diagnostics: Eines der wichtigsten Merkmale des LSP sind die Diagnostics. Die meist durch Unterstreichung dargestellten Diagnostiken machen auf Fehler, Informationen oder zusätzlichen Hinweisen aufmerksam. Dabei werden verschieden Farben verwendet.

Fig. 3. Beispielhaftes Diagnostic

(3) Text-Edits: Text-Edits werden verwendet, um auf mögliche Anpassungen am Quellcode hinzuweisen. Sie beinhalten unter anderem Quick Fixes, die fehlende Imports ergänzen, Typen hinzufügen oder andere Verbesserungen vorschlagen. Werden die TextEdits eingefügt, so verändert sich der Programmcode.

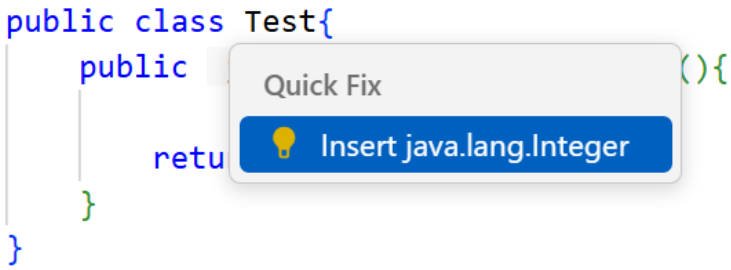


Fig. 4. Beispielhafter Quick-Fix

(4) Hovers: Wenn man mittels des Mauszeiger über bestimmte Codepositionen fährt, erscheinen zusätzliche Informationen. Diese sind beispielsweise die Typdefinition einer Variablen oder eine Dokumentation. Diese Informationen verbessern das Verständnis des Codes, ohne dass er explizit nachgeschlagen werden muss.

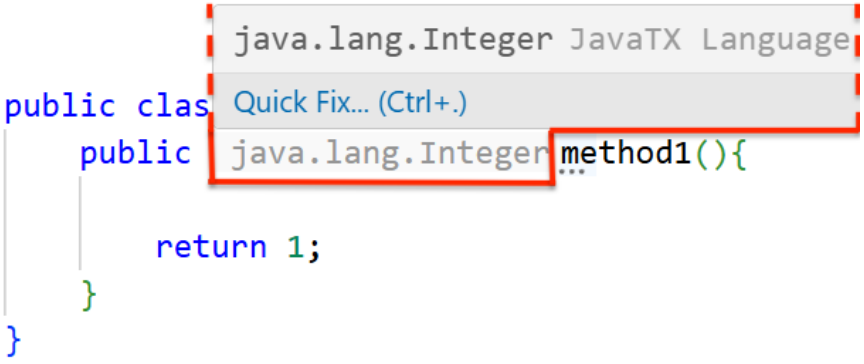


Fig. 5. Beispielhafter Hover-Effekt

Über das standardisierte JSON-RPC-Protokoll (Version 2.0) erfolgt die Kommunikation zwischen Client und Server, wobei alle Nachrichten als strukturierte JSON-Objekte gesendet werden. Nachrichten enthalten typischerweise diese Felder:

(1) `jsonrpc`: Zeigt die JSON-RPC-Version an, die verwendet wird. Bei LSP ist "2.0" der Standard.
(2) `id`: Eine eindeutige Kennung, um Anfragen und Antworten miteinander zu verbinden.
(3) `method`: Der Name der Methode, die ausgeführt werden soll, wie z. B. textDocument/hover.
(4) `params`: Ein Objekt, welches die Parameter für die Methode beinhaltet. Je nach Kontext variieren sie und sind oft der umfangreichste Teil der Nachricht.

```
Content-Length: ...\r\n
\r\n
{
        "jsonrpc": "2.0",
        "id": 1,
        "method": "textDocument/completion",
        "params": {
                ... //Parameter je nach Methode
        }
}
```

Listing 1. Beispiel für JRPC

Jede Nachricht im Protokoll erhält zudem einen Header mit dem Feld Content-Length, der die Größe der Nachricht festlegt. Ein komplettes Beispiel befindet sich in Listing 1. Nachdem der Client die Anfrage verschickt hat, bearbeitet der Language Server die übergebene Methode und sendet eine passende JSON-RPC-Antwort zurück. Je nachdem, welche Methode verwendet wird, wird im Language Server unterschiedliche Funktionalität ausgeführt. Eine Vielzahl von standardisierten Methoden werden in der LSP-Spezifikation definiert, um die Kommunikation zwischen Client und Language Server zu ermöglichen. Einige der wichtigsten und für diese Arbeit relevanten Methoden sind im Folgenden beispielhaft aufgeführt:

(1) `sendMessages`: Diese Methode ermöglicht es dem Language Server, Nachrichten direkt an den Client zu übermitteln. Dabei können sowohl allgemeine Statusinformationen als auch konkrete Hinweise, wie beispielsweise Fehlermeldungen oder Benachrichtigungen über erfolgreiche Kompilierungen, übertragen werden.
(2) `inlayHint`: Mithilfe dieser Methode werden Inlay Hints an den Client gesendet. Diese enthalten Informationen wie berechnete Typen oder ergänzende Hinweise und werden direkt im Quelltext an den entsprechenden Positionen angezeigt. Für die in dieser Arbeit entwickelte Erweiterung stellt diese Methode eine zentrale Grundlage dar, da die Typinferenz-Ergebnisse auf diesem Weg an den Entwickler übermittelt werden.
(3) `hover`: Diese Methode ermöglicht es dem Client, detaillierte Zusatzinformationen über eine bestimmte Codeposition abzurufen. Wird der Mauszeiger beispielsweise über eine Variable oder Methode bewegt, sendet der Client eine Anfrage an den Server, der daraufhin weiterführende Informationen wie Typdefinitionen, Dokumentationen oder Aufrufhierarchien zurückliefert.

Diese und viele weitere Methoden sind Teil der offiziellen LSP-Spezifikation und bilden die Grundlage für die standardisierte Kommunikation zwischen dem Language Server und den unterstützten IDEs. Insbesondere die Methoden `inlayHint` und `hover` sind essenziell für die Integration der in dieser Arbeit umgesetzten Typinferenzmechanismen.

2.3.2 LSP4J - LSP in Java. Das LSP stellt eine Spezifikation dar, aber es ist keine konkrete Implementierung. In der Programmiersprache Java gibt es mit LSP4J eine offizielle Implementierung, die von der Eclipse Foundation zur Verfügung gestellt wird [1]. Die grundlegende Kommunikation zwischen Client und Server wird von LSP4J übernommen. Es trennt die Protokolllogik von der Anwendungslogik. Indem es vordefinierte Interfaces bereitstellt, abstrahiert es die standardisierten LSP-Nachrichtenformate. Es gibt eigene Methoden innerhalb der Interfaces, die je nach dem welche Nachricht durch den Client an den Server gesendet wurde ausgeführt werden. Die Rückgabewerte der Methoden werden wieder an den Client übermittelt. Die Parameter werden in Java Objekte umgewandelt. Eine solche Methode kann in Listing 2 nachvollzogen werden. Beispielhaft wird hier die Methode für die Anforderung der InlayHints verwendet. Der Rückgabetyp beinhaltet eine Liste an Inlayhints. Das vereinfacht die Entwicklung erheblich und erlaubt es gleichzeitig, auf stabile, getestete Kommunikationspfade zurückzugreifen. Ein weiterer Vorteil von LSP4J in diesem Zusammenhang ist, dass der Compiler von Java-TX ebenfalls in Java verfasst ist. So ist es möglich, dass die Language Server und der Compiler direkt miteinander kommunizieren, ohne dass es weitere Übersetzungs- oder Integrationsschichten gibt.

```
1 @Override
2 public CompletableFuture<List<InlayHint>> inlayHint(InlayHintParams
      params) {
3     ...
4 }
```

Listing 2. Beispielmethode von LSP4J

Der Datenaustausch erfolgt standardmäßig über die Standard-Ein- und -Ausgabe. LSP4J unterstützt jedoch auch andere Transportprotokolle wie TCP oder UDP. Um einen voll funktionsfähigen Language Server zu erstellen, müssen innerhalb von LSP4J verschiedene Komponenten implementiert werden, die jeweils unterschiedliche Funktionen bieten, etwa für Textverarbeitung, Diagnostik oder Code-Aktionen.

3 Projektüberblick

Der Language Server, der hier vorgestellt wird, hat als Hauptzweck, dem Entwickler Typinformationen anzuzeigen und ihm die Wahl zwischen mehreren möglichen Typen zu lassen. Außerdem ist es das Ziel des Language Servers, dass er syntaktische Fehler in Java-TX erkennt und sie visuell hervorhebt. Weil die reguläre Java-Syntax in fast allen modernen Entwicklungsumgebungen standardmäßig unterstützt wird, führen die spezifischen Sprachkonstrukte von Java-TX häufig dazu, dass sie innerhalb der IDE fehlerhaft markiert werden.

3.1 Überblick über die Hauptkomponenten

Die gesamte Extension setzt sich aus drei klar abgegrenzten Komponenten zusammen, die zwar miteinander kommunizieren, aber funktional unabhängig sind:

(1) **Compiler-Interface:** Das Compiler-Interface ist ein Bestandteil des Java-TX-Compilers und stellt die Verbindung zu ihm her. Es erlaubt den Zugriff auf die vom Compiler erstellten ResultSets und den abstrakten Syntaxbaum (AST), den der Compiler erzeugt hat. Außerdem ist das Interface dafür zuständig, aus dem vollständigen, typisierten Quellcode den Bytecode zu erstellen.

(2) **Language Server:** Die zentrale Business-Logik der Anwendung wird im Language Server verwaltet. Er untersucht die Informationen, die der Compiler bereitstellt, legt die Positionen für diagnostische Hinweise fest und liefert Typhinweise (Inlay Hints). Dank dieser Abstraktion können ihn verschiedene Entwicklungsumgebungen entkoppelt genutzt werden.
(3) **Clients:** Die Clients agieren als Darstellungsschicht und kommunizieren über das LSP mit dem Language Server. Sie empfangen und fragen die Informationen, die der Server liefert, wie etwa Typhinweise, Fehlermeldungen oder Quick-Fixes ab und visualisieren diese innerhalb der jeweiligen IDE.

3.2 Unterstützte Entwicklungsumgebungen

Selbst wenn die Clients meist auf bestehenden Frameworks beruhen, erfordert ihre Entwicklung dennoch einen gewissen Aufwand. Die jeweiligen Client-Erweiterungen müssen den Language Server initialisieren und seinen Lebenszyklus verwalten. Obwohl diese Aufwände normalerweise gering sind, benötigen sie dennoch Tests und Wartung bei zukünftigen Änderungen.
In Anbetracht dieser Einschränkungen wurde die Entscheidung getroffen, nur eine Auswahl der gängigen Entwicklungsumgebungen zu unterstützen. Es wurden gängige und praxisnahe IDEs ausgewählt, um eine hohe Nutzbarkeit zu gewährleisten.
Die Entwicklungsumgebungen, die ausgewählt wurden, sind:

(1) **Visual Studio Code:** Visual Studio Code ist ein populärer Editor, der vor allem für seine hohe Erweiterbarkeit bekannt ist. Diese Umgebung wurde als Hauptzielplattform ausgewählt, weil sie eine große Nutzerbasis hat und Extensions leicht integriert werden können.
(2) **IntelliJ IDEA:** Die IDE IntelliJ IDEA gehört zu den meistgenutzten Optionen für Java [2]. Durch die einfache Installation von Plugins und die große Verbreitung im Java-Umfeld ist sie eine wichtige Zielplattform für Java-TX.
(3) **Emacs:** Emacs ist im Linux-Umfeld ein weitervererbter konfigurierbarer Editor, der insbesondere im Programmiersprachenforschung häufig eingesetzt wird. Er kann über `lsp-mode` mit der Language Server verbunden werden.

4 Funktionalität des Language Servers

4.1 Syntaxhighlighting

```
import java.lang.String;
import java.lan
import java.lan
                extraneous input 'Test' expecting {'extends', 'implements', 'permits', '{', '<'}
                View Problem (Alt+F8)   No quick fixes available

public class t Test{
    public  java.lang.Comparable<? extends java.lang.c… method1(Boolean decider){

        if(decider){
            return "Hello World!";
        }
        return 1;
    }
}
```

Fig. 6. Syntaxcheck innerhalb der IDE

Das Syntaxhighlighting hebt fehlerhafte Syntax hervor. Fehler, welche durch das Parsen von ANTLR erhalten werden, werden unterstrichen und angezeigt (vgl. Abbildung 6). Die durch ANTLR enthaltenen Fehlerbeschreibungen werden angezeigt. Da Java-TX zwar eine Java-ähnliche Syntax besitzt, sie jedoch ohne Typen arbeitet, ist es essentiell dieses Feature zu unterstützen. Dadurch kann eine sinnvolle Arbeit mit Java-TX in modernen IDEs garantiert werden. Wenn dieses Feature nicht existieren würde, würden gar keine Syntaxfehler erkannt, oder eine Syntaxerkennung der Java-Syntax stattfinden. Ein Syntaxfehler ist in Abbildung 6 dargestellt.

4.2 Typehints und Typanzeige

Die vom Typinferenzalgorithmus ermittelten Typen werden dem Entwickler als Typhinweise direkt vor den entsprechenden Variablen angezeigt. Es ahmt die Typdeklarationen der klassischen Java-Syntax nach, was die Lesbarkeit des Codes verbessert, ohne dass der Entwickler die Typen ausdrücklich angeben muss. Die berechneten Typen werden zusätzlich als Informations-Diagnostiken unterhalb der Variablen dargestellt, um eine konsistente und redundante Darstellung der Ergebnisse zu gewährleisten. Es werden nur valide Inlayhints angezeigt. Falls eine Variable mehrere mögliche Typen besitzt, werden sie im Inlayhint zusammen aufgeführt. Die verschiedenen Möglichkeiten werden durch das Zeichen | getrennt. Außerdem wird für jeden einzelnen Typ eine eigene Diagnostik erstellt, damit der Entwickler eine gezielte Auswahl treffen kann. Außerdem ermöglicht die Erweiterung das automatische Einfügen eines vorgeschlagenen Typs über einen Quick-Fix in den Programmcode.

4.3 Einfügen von Typen

Fig. 7. Inlayhints mit Diagnostiken.

Ein Quick-Fix wird automatisch für jeden möglichen Typ einer Variablen erstellt, den der Language Server ermittelt. Mit dieser Funktion kann der Entwickler einen konkreten Typ direkt im Programmcode einfügen, ohne dass er ihn manuell eingeben muss.

Wenn man einen Quick-Fix auswählt, wird nicht nur der passende Typ in den Code eingefügt, sondern auch die interne Logik zur Typauswahl aktualisiert. Das heißt: Wenn man einen konkreten Typ für eine Variable festlegt, werden alle nun ungültigen Typkombinationen ausgeschlossen. So wird garantiert, dass nur gültige und konsistente Typkombinationen für die verbleibenden Typplatzhalter verbleiben. Die inhärente Typabhängigkeit innerhalb der vom Compiler erstellten ResultSets ist der Grund für dieses Verhalten: Wenn man einen Typ auswählt, beeinflusst das die Gültigkeit der anderen Typzuweisungen. Eine Auswahl und die Inlayhints können in Abbildung 7 nachvollzogen werden.

Der Quellcode wird beim Einfügen direkt verändert, im Gegensatz zu den rein informativen Inlay-Hints, die nur zusätzliche Informationen bieten, aber keinen Einfluss auf den Programmtext haben.

4.4 Fehlererkennung und Diagnostik

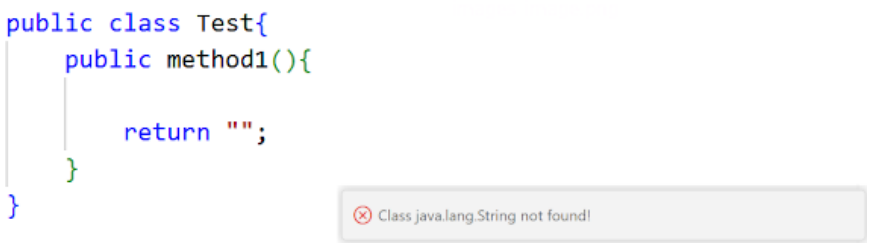


Fig. 8. Fehlermeldung bei fehlenden Imports.

Wenn ein Programm nicht erfolgreich mit dem Java-TX-Compiler kompiliert werden kann, wird eine Fehlermeldung erstellt, als Nachricht an den Client gesendet wird. Die genaue Beschreibung des Kompilierungsfehlers werden in dieser Meldung angezeigt.
Typische Ursachen für Fehler sind zum Beispiel fehlende Import-Anweisungen für bestimmte Typen oder nicht auflösbare Referenzen. Dies kann in Abbildung 8 gesehen werden. Indem diese Fehler direkt angezeigt werden, hat der Entwickler die Möglichkeit, genau auf die Ursache zu reagieren und entsprechende Anpassungen vorzunehmen. Das ist ein wichtiger Faktor für die Effizienz der Entwicklungsarbeit mit Java-TX.

4.5 Live-Compilation und Ausgabe

Sobald der Programmcode in der Entwicklungsumgebung gespeichert wird, initiiert das Compiler-Interface automatisch einen Kompilierungsvorgang. Durch diesen Prozess wird ausführbarer Bytecode erstellt, der als `.class`-Dateien ausgegeben wird.
Die Dateien, die erstellt werden, landen im gleichen Verzeichnis wie die Quelldateien, strukturiert in einem separaten `/out`-Ordner. Dies kann in Abbildung 9 gesehen werden. In diesem Ordner sind alle kompilierten Klassen abgelegt, sodass das Programm ohne weiteren manuellen Eingriff direkt ausgeführt werden kann.

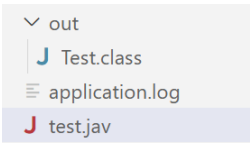


Fig. 9. Bytecode innerhalb des /out Ordners.

Durch die Automatisierung der Kompilierung ist kein manuelles Kompilieren über die Konsole notwendig. Das vereinfacht den Entwicklungsprozess mit Java-TX erheblich und sorgt dafür, dass die Sprache gut in moderne Entwicklungsumgebungen integriert ist.

5 Architektur des Systems

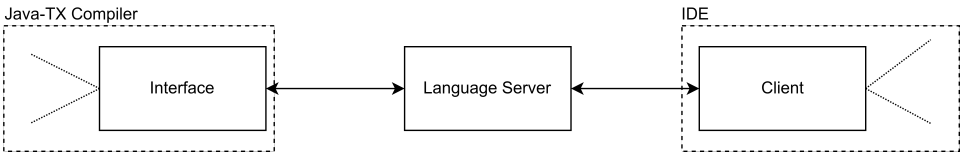


Fig. 10. Gesamtarchitektur des Systems.

Die Architektur des Language Servers ist insgesamt aufgebaut auf den drei Hauptkomponenten, die vorher beschrieben wurden. Diese können in Abbildung 10 nachvollzogen werden. Das Compiler-Interface, das Teil des Compilers ist, wird als Abhängigkeit dem Language Server hinzugefügt und kommuniziert innerhalb der Java-Anwendung direkt mit ihm.
Der Language Server ist das Bindeglied zwischen dem Compiler-Interface und den unterschiedlichen Clients. Er kümmert sich um die Aufbereitung und Weiterverarbeitung der Daten, die der Compiler liefert. Um die Zugriffe auf das Compiler-Interface zu minimieren, da der Java-TX-Compiler und somit auch das Interface eine geringe Laufzeitperformance haben, werden empfangene Daten im Server zwischengespeichert und nur beim Speichern aktualisiert. Der Language-Server versucht die vorhandenen Daten korrekt anzupassen.
Die Clients, tauschen über `System.in` und `System.out` Nachrichten mit dem Language Server aus. Die standardisierten LSP-Nachrichten, die Informationen wie Typhinweise, Diagnostiken oder Textänderungsvorschläge umfassen, werden über diesen Kanal übertragen.

5.1 Komponenten im Detail

Anschließend werden die einzelnen Komponenten im Detail beschrieben und Designentscheidungen erläutert.

5.1.1 Compiler-Interface. Im Java-TX-Compiler ist das Compiler-Interface integriert. Es erlaubt das Kompilieren einer Quellcodedatei basierend auf ihrer URI, normalerweise der Datei, die derzeit in der IDE geöffnet ist. Falls erforderlich, werden weitere abhängige Dateien zusätzlich mitkompiliert.
Ein Transferobjekt wird nach dem Kompilierungsvorgang an den Language Server übermittelt. Es umfasst den abstrakten Syntaxbaum, in dem spezielle Typplatzhalter anstelle konkreter Typen eingefügt wurden, sowie sogenannte ResultSets, die die entsprechenden Typauflösungen liefern. Generische Typen sind ebenfalls Bestandteil des Transferobjekts.
Die Typplatzhalter, die im AST enthalten sind, können durch die konkreten Typinformationen aus dem ResultSet ersetzt werden. Oft gibt es mehrere ResultSets, weil verschiedene gültige Typkombinationen existieren.

```
1 import java.lang.Integer;
2
3 class Test {
4     addOne(i){
5         return i+1;
```

```
    }
}
```

Listing 3. Ursprüngliche Klasse

```
class Test {

Test()({
  }):: TPH V
  TPH N addOne(TPH O i)({
    return (i):: TPH O | 1;
  }):: TPH R

  Test()({
    super(()) Signature: [TPH T];
  }):: TPH U

}
```

Listing 4. Abstrakte Syntax

Ein Beispiel für diese Typplatzhalter ist in Listing 3 und Listing 4 zu finden. Der Code in Listing 3 beinhaltet, wie es in Java-TX üblich ist, keine Typnotationen für die Methode `addOne` und ihren Parameter `i`. Dieser Code wird beim Parsen in den abstrakten Syntaxbaum überführt, wobei die Methode `addOne` den Platzhalter `TPH_N` und der Parameter `i` den Platzhalter `TPH_O` erhält (vgl. Listing 4).
Konkrete Typauflösungen für diese Platzhalter sind im zugehörigen ResultSet (Listing 5) zu finden. Auf diese Weise können die ursprünglichen Platzhalter durch abgeleitete Typen ersetzt werden.

```
[[(TPH P = java.lang.Integer),
  (TPH O = java.lang.Integer),
  (TPH N = java.lang.Integer),
  (TPH Q = java.lang.Integer)]]
```

Listing 5. ResultSet

Im Verlauf des Kompilierungsvorgangs wird auch Bytecode erstellt, der im Verzeichnis `out` abgelegt wird, relativ zum Pfad der kompilierten Datei.
Sollen jedoch nur Syntaxfehler aufgetan werden, muss der Compiler die gesamte Datei immer wieder verarbeiten, um eine vollständige Liste aller Fehler zu erstellen. Das Nachteilige daran ist, dass die Zeitspanne zwischen Anfrage und Antwort sich dadurch erhöht, was für eine Echtzeitverarbeitung in einer IDE problematisch ist. Eine Erweiterung, die dem Entwickler sofortige Rückmeldungen geben soll, kann unter diesen Umständen nur eingeschränkt performant arbeiten.
Das Compiler-Interface bietet deshalb die Möglichkeit, nur nach Parserfehlern zu suchen, ohne die gesamte Typinferenz zu durchlaufen. Hierfür kommt die ANTLR-Grammatik des Java-TX-Compilers zum Einsatz. In diesem Fall gibt das Parser-Interface syntaktische Fehler aus. Eine Übersicht der Interfaces können in Abbildung 11 gesehen werden.

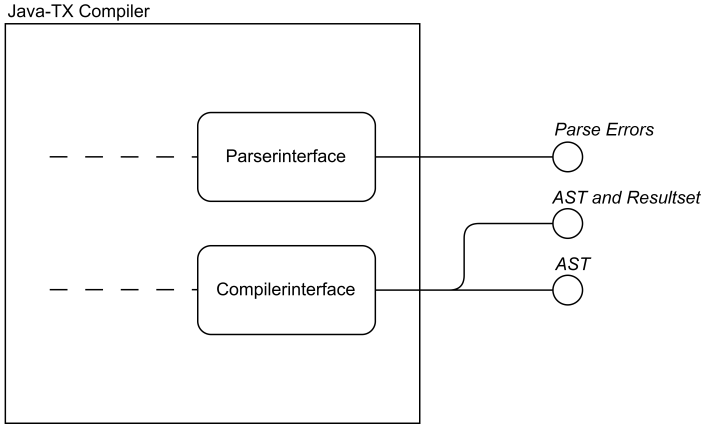


Fig. 11. Die Parserinterfaces

5.1.2 Language Server. Der Language Server nutzt das Compiler-Interface, speichert und aktualisiert interne Datenstrukturen und reagiert auf die Anfragen des Clients. Weil die Kommunikation mit dem Compiler-Interface eine vergleichsweise hohe Latenz hat, wird versucht, direkte Anfragen an dieses Interface weitestgehend zu umgehen.
Es wurden unterschiedliche interne Mechanismen eingeführt, die es ermöglichen, relevante Positionen und ResultSets direkt im Server anzupassen, wenn am Quelltext geändert oder ein bestimmter Typ ausgewählt wird. Dadurch können Compilerzugriffe, die viel Zeit in Anspruch nehmen, umgangen werden.
Der Language Server bereitet die Daten, die er vom Compiler-Interface erhält, auf und speichert sie zwischen. Typauflösungen für die einzelnen Typplatzhalter aus den ResultSets werden als Inlayhint an den Client gesendet. Außerdem werden Informationsdiagnosen erstellt. Quick-Fixes sind auch verfügbar und erlauben es dem Nutzer, einen Typ auszuwählen, um ihn direkt in den Quellcode einzufügen.

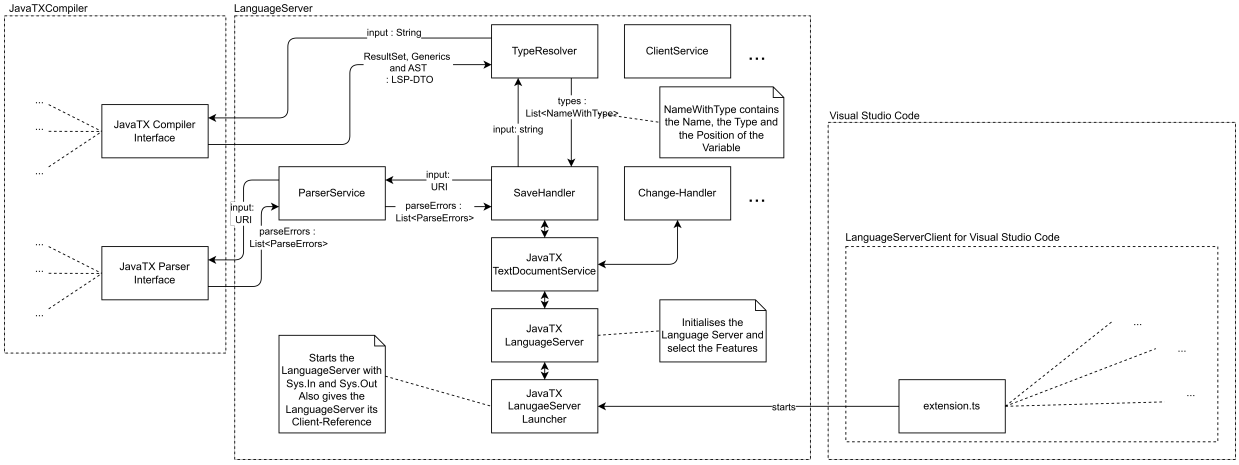


Fig. 12. Genaue Architektur des Langauge Servers. (Nur ein Handler mit Service wird gezeigt)

Die Informationen, die zur Berechnung der Position benötigt werden, stammen aus den Tokenpositionen, die ANTLR generiert und die im AST enthalten sind.
Parserfehler, welche über das Compiler-Interface bereitgestellt werden, sind Diagnosen, die an

den Client zurückgegeben werden. Ein genaues Abbild der Architektur wird in Abbildung 12 dargestellt.

Die Klasse `JavaTXTextDocumentService` ist der zentrale Einstiegspunkt für alle eingehenden Anfragen. Sie implementiert ein Interface, das von LSP4J bereitgestellt wird, und bietet für jede Methode, die in der LSP-Spezifikation definiert ist, eine entsprechende Java-Methode an.
Weil Java-TX nicht alle Funktionen im Protokoll benötigt, haben viele dieser Endpunkte nur leere Rückgaben. Für die Funktionalität von Java-TX sind die folgenden Methoden jedoch unerlässlich:

(1) `change`: Wird ausgelöst, wenn am aktuell geöffneten Dokument Änderungen vorgenommen werden. Parserfehler werden hierbei aktualisiert.
(2) `codeAction`: Ist dafür da, um Quick-Fixes, also Vorschläge für einzufügende Typen, bereitzustellen.
(3) `formatting`: Wird angefordert, wenn eine Dokumentenformatierung gewünscht ist. Dieses Merkmal hat jedoch eine geringere Priorität.
(4) `inlayHint`: Fordert Inlay-Hints an. Dieses Feature ist zusammen mit der Typauswahl das Herzstück der Extension. Sie zeigen inferierte Typen an ohne den Quellcode zu verändern.
(5) `save`: Wenn das Dokument gespeichert wird, wird das Compiler-Interface erneut aufgerufen. Die Cache-Daten werden überschrieben, um einen aktuellen und konsistenten Datenstand zu schaffen.

Handlerklassen, die speziell für die verschiedenen Methoden des Language Servers entwickelt wurden, werden eingesetzt, um Flexibilität zu gewährleisten. Diese Handler übernehmen die Fachlogik und besitzen `handle`-Methoden, die die vom Client gesendeten Parameter annehmen und passende Antworten zurück geben.

(1) `ChangeHandler`: Kümmert sich um alle Änderungen im Dokument. Er ruft das Parser-Interface des Compilers auf und sendet resultierende Diagnostiken direkt an den Client. Die internen Cache-Dateien werden angepasst, um dies zu berücksichtigen. Falls nötig werden die ResultSets eingeschränkt, und illegale ResultSets entfernt. Wenn ein Typ ausgewählt wird, wird einer Variable dann einen festen Wert zugeschrieben. Dadurch können vorher legale ResultSets illegal werden, da die Variable dort einen anderen Typ annimmt. Der entsprechende Typhint, Quickfix und die dazugehörige Diagnostik entfallen. Ohne einen erneuten Kompilierungsvorgang wird der AST abgefragt. Die Typplatzhalter bleiben dabei konsistent, was eine Positionsaktualisierung der Typvariablen ermöglicht. Es erfolgt keine automatische Erkennung neuer Typen. Dies geschieht nur durch eine vollständige Kompilierung, die beim Speichern ausgelöst wird. Alle aktualisierten Typhinweise und Diagnostiken werden erneut gecacht und veröffentlicht.
(2) `CodeActionHandler`: Verantwortlich für das Anbieten der Quick-Fixes. Die Auswahl der Typen erfolgt anhand der gecachten Diagnosen und Typhinweisen. Sobald man einen Typ auswählt, wird über `change` eine neue Anfrage gesendet, die zu einer Positionsaktualisierung und einer Einschränkung der Typauswahl führt.
(3) `FormattingHandler`: Diese Komponente kontrolliert das Dokument auf trailing whitespaces und beseitigt sie, wenn eine Formatierungsanfrage gestellt wird. Es wurden keine weiteren Funktionen umgesetzt.
(4) `SaveHandler`: Eine erneute Kompilierung wird durch das Compiler-Interface angestoßen. Der Cache aktualisiert sich, und Typhints sowie Diagnostiken werden

neu erstellt. Im Gegensatz zum `ChangeHandler` werden hier keine Einschränkungen des ResultSets vorgenommen, weil immer die neuesten Informationen vom Compiler abgerufen werden.

Diese Handler nutzen spezialisierte Services und Hilfsklassen, die nach dem Single Responsibility Principle entworfen sind. Die Anwendungslogik ist dort implementiert, während die Handler die Orchestrierung übernehmen. Die interne Datenstruktur `LSPVariable` wird genutzt, um die enge Beziehung zwischen Typdiagnostiken, Typhints und Quickfixes abzubilden. Sie beinhaltet die Variable, ihren Standort, alle möglichen Typen sowie den ursprünglichen Typplatzhalter. Die entsprechenden Typhinweise und Diagnostiken werden ebenfalls gecacht.

(1) `CacheService`: Zentrale Verwaltung aller temporären Cache-Daten. Setzt sich überwiegend aus Speicherfeldern und passenden Getter-/Setter-Methoden zusammen.
(2) `ClientService`: Service zur Kommunikation mit dem Client. Er verwaltet die Veröffentlichung von Diagnostiken und Benachrichtigungen.
(3) `LogService`: Zuständig für das Logging auf der Client- und Serverseite. Verwendet Log4J und die von LSP4J festgelegten Log-Level.
(4) `ParserService`: Dient zur Kommunikation mit dem Parser-Interface des Compilers. Parserfehler werden durch ihn in Diagnosen umgewandelt.
(5) `TextDocumentService`: Kümmert sich um geöffnete Dateien, speichert Pfadinformationen, erkennt Textänderungen und setzt Vergleichs- sowie Positionslogik für Strings um.
(6) `TypeResolver`: Die zentrale Komponente des Language Servers. Er kommuniziert über das Compiler-Interface und bereitet ResultSets auf und verwaltet diese. Erstellt Inlayhints, Diagnosen und Quick-Fixes. Identifiziert Positionen über Tokeninformationen im AST. Verwendet `GenericHelper` für die getrennte Verarbeitung generischer Typen.
(7) `TypeUtils`: Erstellt konkrete Type-Objekte aus den ResultSets. Gibt alle gültigen Typen für einen spezifischen Typplatzhalter zurück.
(8) `TextHelper`: Abstraktion für spezifische Textoperationen, wie zum Beispiel das Festlegen von Endpositionen in einer Textdatei.
(9) `GenericUtils`: Behandelt generischen Typen, die über ein separates ResultSet bereitgestellt werden.
(10) `DuplicationUtils`: Entfernt redundante Typinformationen damit keine doppelten Typhints und Typauswahllisten bei einer Variable verwendet werden.
(11) `ConversionHelper`: Wandelt interne Modellklassen in Inlayhints und Diagnosen um.
(12) `ASTTransformationHelper`: Transformiert DEN AST in interne Modellklassen, wie beispielsweise Methodenparameter oder Konstruktorvariablen.

Intern werden eigene Modellklassen verwendet, um die Daten strukturiert zu speichern. In der Klasse `LSPVariable` sind unter anderem der Name der Variable, mögliche Typen, Positionen und der ursprüngliche Platzhalter enthalten. Die Klasse `Type` beinhaltet den Namen des Typs und ein Flag, das generische Typen kennzeichnet.
Der gesamte Verarbeitungsfluss startet im `JavaTXTextDocumentService`, das LSP4J-Endpunkte bereitstellt und eingehende Anfragen an die zuständigen Handler weiterleitet. Die Kommunikation zwischen Services und Helfern wird von den Handlern orchestriert. Das Ergebnis wird von den Handler zurückgegeben und an den Client gesendet, möglicherweise begleitet von Fehlern oder Warnungen, etwa bei nicht importierten Typen.
Es gibt neben der fachlichen Logik auch Konfigurationsklassen, die dem Client sagen, welche

Features verfügbar sind. Die Klassen, die LSP4J-Interfaces umsetzen, bieten eine granulare Kontrolle über die Funktionsvielfalt.
Zum Start des Language-Servers existiert eine Klasse mit `main`-Methode, die den Language Server startet.

5.1.3 Editor-Clients. Clients, die mit dem Language Server kommunizieren, sind normalerweise sehr leichtgewichtig und lassen sich durch bereits vorhandene Frameworks umsetzen. Die grundlegende Client-Unterstützung ist in vielen Entwicklungsumgebungen bereits vorhanden, weil die Spezifikation der LSP berücksichtigt wurde. Die Hauptaufgabe eines Clients ist es, den Language Server zu starten, Anfragen zu formulieren und die erhaltenen Informationen richtig in der Benutzeroberfläche darzustellen.
Die Implementierung wird exemplarisch dargestellt, indem sie in Visual Studio Code integriert wird.
Die Erweiterung nutzt die Standardprojektstruktur für Extensions, die Microsoft für Visual Studio Code vorgesehen hat. Dabei wird Typescript verwendet. Node.js wird zum bauen und testen benutzt. Die Bibliothek `vscode-languageclient` wird verwendet, um die LSP spezifischen Funktionen zu bieten.
Die Datei `extension.ts` enthält die Hauptlogik der Extension. Diese Datei legt fest, wann und auf welche Weise der Language Server gestartet wird. Wie in Listing 6 dargestellt, wird der Language Server für alle Dateien mit der Endung `.jav` durch das Pattern `**/*.jav` aktiviert. Danach wird der Server mit festgelegten Optionen initialisiert und gestartet.

```
1  const clientOptions: LanguageClientOptions = {
2          documentSelector: [{ scheme: 'file', language: 'java' }],
3          synchronize: {
4                  fileEvents: vscode.workspace.createFileSystemWatcher(
                       '**/*.jav')
5          },
6          revealOutputChannelOn: 4
7  };
8
9
10
11 const client = new LanguageClient(
12         'javaTxLanguageServer',       // ID des Clients
13         'Java-TX␣Language␣Server',     // Name des Clients
14         serverOptions,
15         clientOptions
16 );
```

Listing 6. Initialisierung des Servers

Im Zentrum der Startlogik steht die Festlegung der `serverOptions` (siehe Listing 7). An dieser Stelle wird bestimmt, wie der Language Server gestartet werden soll. Hierbei wird ein Java-Prozess mit dem Befehl `java -jar ...` gestartet, weil der Language Server über LSP4J in Java umgesetzt ist. Es ist möglich, zwischen Debug- und Normalmodus zu unterscheiden. In diesem Fall gibt es keinen separaten Debug-Modus, weshalb beide Modi denselben Startbefehl nutzen.

```
1  const serverOptions: ServerOptions = {
2          run: {
3                  command: 'java',
```

```
4                   args: ['-jar', workspaceFolder + '/
                        JavaTXLanguageServer-1.0-SNAPSHOT-jar-with-
                        dependencies.jar'],
5           },
6           debug: {
7                   command: 'java',
8                   args: ['-jar', workspaceFolder + '/
                        JavaTXLanguageServer-1.0-SNAPSHOT-jar-with-
                        dependencies.jar'],
9           }
10  };
```

Listing 7. Initialisierung des Servers

Weitere Informationen wie der Name, die Beschreibung oder Symbole der Extension werden in der Datei `package.json` verwaltet. Das Tool `vsce`, welches von Visual Studio Code bereitgestellt wird, wird genutzt, um den finalen Build der Extension zu erstellen. Dies erstellt eine `.vsix`-Datei, die als installierbare Extension fungiert.
Es ist notwendig, dass Java bereits installiert ist, um die Extension auszuführen, da der Language Server auf einer Java-basierten Architektur aufbaut. Weil der Compiler, auf dem alles basiert, ebenfalls in Java geschrieben ist, entsteht dadurch keine zusätzliche Abhängigkeit.

Die Einbindung in andere Entwicklungsumgebungen wie IntelliJ IDEA oder Emacs geschieht auf ähnliche Weise. Für IntelliJ IDEA wird unter anderem das von Red Hat entwickelte Framework `LSP4IJ` eingesetzt. Es ermöglicht die einfache Anbindung eines Language Servers, indem man die Startbefehle und die unterstützten Dateitypen konfiguriert.
In Emacs kann man ebenfalls über die `lsp-mode` eine Integration vornehmen, indem man dort ein Startkommando für den Language Server festlegt. Dies ist zum jetzigen Zeitpunkt jedoch noch nicht umgesetzt. Die `.jar`-Datei des Servers ist unabhängig nutzbar und funktioniert auch ohne eine spezifische IDE-Integration.
Generische LSP-Clients, die über ein Benutzerinterface konfiguriert werden können, existieren neben individuellen Integrationen. Sie erlauben es, beliebige Language Server zu nutzen, indem man ein Startkommando und die zugehörigen Dateimuster angibt. Dies wird zum Beispiel für Intellij IDEA verwendet.

5.2 Kommunikationswege und Datenfluss

Ein vereinfachtes Beispiel zeigt den grundlegenden Datenfluss des Systems. Der Kommunikationsfluss zwischen Client, Language Server und Compilerinterface kann man grundsätzlich in zwei Richtungen unterteilen: Entweder der Client stellt gezielte Anfragen an den Language Server, oder dieser sendet proaktiv Informationen an den Client.
Ein typischer Ablauf startet damit, dass der Nutzer eine Datei speichert. Hierbei schickt der Client eine `save`-Nachricht an den Language Server. Daraufhin ruft der Server das Compilerinterface auf, welches die modifizierte Datei untersucht und entsprechende Ergebnisse (AST, ResultSets, Parserfehler usw.) zurückliefert.
Der Language Server kümmert sich dann um die Verarbeitung der empfangenen Daten. Parserfehler werden als Diagnostiken unmittelbar an den Client gesendet. Währenddessen werden Typhinweise und Quick-Fix-Vorschläge basierend auf den ResultSets erstellt und im internen Cache abgelegt.

Der Client fordert dann weitere Informationen vom Server an, zum Beispiel durch gezielte Abfragen nach InlayHints oder CodeActions. Diese Informationen werden nicht erneut berechnet, sondern direkt aus dem Cache zurückgegeben, weil die Berechnung bereits im Rahmen des vorherigen `save`-Events stattfand.
Darüber hinaus informiert der Language Server den Client über neue Typhinweise und Diagnostiken, was es ermöglicht, dass die Benutzeroberfläche sofort angepasst wird.

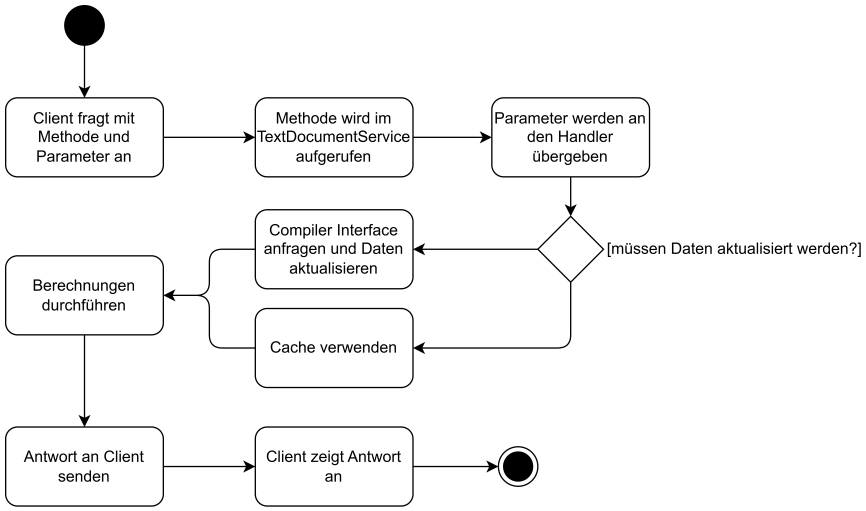


Fig. 13. Basiskommunikationsfluss

Das Kommunikationsmuster, das hier beschrieben und in Abbildung 13 aufgezeigt wird, ist ein typisches Beispiel für den Ablauf zwischen Client und Server, der bei allen Interaktionen ähnlich ist. Egal, welche Methode (z.B. `change`, `hover`, `codeAction`) genutzt wird, die Interaktion läuft immer nach dem gleichen Prinzip ab: Eine Anfrage, eventuell eine Kommunikation mit dem Compilerinterface, die Verarbeitung der Daten und schließlich die Rückgabe an den Client oder ein Push von Informationen durch den Server.

6 Implementierung mit LSP4J und Vorraussetzungen

6.1 Technische Hintergründe

Mindestens Java 23 muss installiert sein, um die Java-TX Language Server Extension auszuführen. Es gibt keine weiteren spezifischen Systemvoraussetzungen. Um den Java-TX-Compiler auszuführen, wird ebenfalls die gleiche Java-Version benötigt, sodass keine weiteren Abhängigkeiten erforderlich sind.
`Maven` wird Intern verwendet, um benötigte Bibliotheken und externe Abhängigkeiten automatisch zu finden und bereitzustellen. Die Struktur des Projekts ist die eines klassischen Maven-Projekts, was eine einfache Integration und Wartung ermöglicht.

Weil während der Typinferenz und Kompilierung viele Berechnungen angestellt werden, kann es bei Programmen, die sehr groß oder komplex strukturiert sind, zu einer spürbaren Verzögerung der Antwortzeit des Language Servers kommen. Eine Caching-Strategie wurde implementiert, um diesem Umstand entgegenzuwirken, indem sie wiederholte Anfragen möglichst performant bedient und die Anzahl der vollständigen Kompilierungsvorgänge minimiert.

7 Integration in Entwicklungsumgebungen

7.1 Visual Studio Code

Die `.vsix`-Datei kann manuell in Visual Studio Code installiert werden.

Die Installation wird in diesen Schritten durchgeführt:

(1) Visual Studio Code starten und zum Erweiterungen-Tab navigieren. Dies ist durch die Suchleiste, die linke Menüleiste oder `Strg + Shift + X` möglich.
(2) Das Drei-Punkte-Menü (...) in der oberen rechten Ecke öffnen.
(3) "Install from VSIX..." aus auswählen.
(4) Zum Ordner, in dem die `.vsix`-Datei, gespeichert ist navigieren und die Datei auswählen. Die Installation geschieht danach automatisch.

Nach der Installation kann die Erweiterung für alle Dateien mit der Endung `.jav` verwendet werden. Bei etwaigen Problemen kann ein Neustart von Visual Studio Code helfen.

7.2 IntelliJ IDEA

Die Integration der Java-TX Language Server Erweiterung in IntelliJ IDEA erfolgt, indem die `.jar`-Datei des Language Servers heruntergeladen und sie dann manuell registriert wird. Anhand diesen Schritten kann die Erweiterung installiert werden.

Zu den Einstellungen unter File > Settings > Plugins navigieren und das Plugin LSP Support (LSP4IJ) von Red Hat suchen und installieren. Mit diesem Plugin können beliebige Language Server über das LSP-Protokoll eingebunden werden.

(1) Nach der Installation kann ein neuer Eintrag mit dem Namen Language Servers in der linken Seitenleiste geöffnet werden.
(2) Einen neuen Eintrag hinzufügen, indem mit der rechten Maustaste auf die Liste geklickt wird und New Language Server ausgewählt wird.
(3) Einen beliebigen Namen verwenden, zum Beispiel `Java-TX`. Im Command-Feld, welches den Befehl enthalten muss um den Language Server zu starten, kann folgender Befehl verwendet werden: `java -jar "<Pfad/zur/LanguageServer.jar>"`
(4) Zum Tab *Maps* wechseln.
(5) Den Punkt *File name* pattern auswählen.
(6) Die Konfiguration um ein neues Mapping erweitern, um den Language Server für Dateien mit der Endung .jav zu aktivieren. Dazu kann auf das + gedrückt und Folgendes hinzugefügt werden: File name pattern: *.jav, Language ID: java_tx
(7) Um die Erweiterung zu verwenden, kann eine `.jav`-Datei geöffnet werden.

Sobald eine `.jav` Datei geöffnet wird, wird unten rechts in der Statusleiste der aktuelle Verbindungsstatus des Language Servers angezeigt.

8 Fazit und Ausblick

8.1 Zusammenfassung der Ergebnisse

Insgesamt wurde ein Language Server erstellt, der die Hauptvorteile von Java-TX aufgreift und gezielt unterstützt. Die Erweiterung, die entwickelt wurde, erlaubt es Entwicklerinnen und Entwicklern, die typlose Syntax von Java-TX zu verwenden, ohne die gewohnte Unterstützung durch moderne Entwicklungsumgebungen aufgeben zu müssen.
Obwohl es noch Einschränkungen gibt, vor allem bezüglich der Laufzeit, umfasst der Language Server bereits eine Reihe von nützlichen Funktionen. Die aufgrund der langsamen Kompilierungsvorgänge des Java-TX-Compilers erforderliche Caching-Strategie kann in einigen Fällen zu inkonsistenten Anzeigen führen. Hier sollten Verbesserungen erfolgen.

Dennoch ist es ein großer Vorteil, dass wichtige Funktionen wie die direkte Kompilierung, die visuelle Darstellung von inferierten Typen und die Auswahl möglicher Typoptionen integriert wurden. Dies ermöglicht es, Feedback zur Typinferenz zu erhalten und mit den bereits berechneten Typen zu arbeiten. Auch Syntaxfehler der Sprache Java-TX werden erkannt und visualisiert, was die Entwicklung vereinfacht und beschleunigt.
Die Erweiterung vereinfacht den Einstieg mit Java-TX: Sie ermöglicht die Entwicklung, ohne einen Konsolen-Workflow nutzen zu müssen, und macht Java-TX dadurch auch für weniger technikaffine Menschen zugänglich.
Es ist besonders bemerkenswert, dass zwei scheinbar gegensätzliche Paradigmen miteinander verbunden werden: Einerseits bewahrt die Java-TX den typlosen Ansatz, während andererseits durch das Berechnen und Anzeigen der Typen die Typsicherheit von Java gewahrt bleibt. So bringt der Language Server die Vorzüge von dynamisch typisierten Sprachen und die Sicherheit von statischen Typisierungen zusammen.

8.2 Nutzen des Language Servers

Die entwickelte Erweiterung hat den entscheidenden Vorteil, dass sie technische Komplexität abstrahiert und den Entwicklungsprozess für Nutzerinnen und Nutzer von Java-TX erheblich vereinfacht. Der Entwicklungsworkflow wird durch den Wegfall der manuellen Kompilierungsschritte über die Konsole und der fehlenden Transparenz über inferierte Typen erheblich verbessert. Java-TX als Programmiersprache wird dadurch insgesamt praktikabler und leichter zugänglich. Vor allem für Entwicklerinnen und Entwickler, die moderne IDEs nutzen, ist die Arbeit mit Java-TX deutlich einfacher, das ist ein wichtiger Faktor für die Nutzbarkeit und die Akzeptanz der Sprache.

8.3 Geplante Erweiterungen

Zukünftige Erweiterungen könnten insbesondere die Unterstützung weiterer Entwicklungsumgebungen umfassen. Die Zielumgebung Emacs, die bereits in den Anforderungen festgelegt wurde, ist noch nicht implementiert.
Durch zukünftige Optimierungen der Compiler-Performance könnte die momentan verwendete cache-basierte Verarbeitung im Language Server überarbeitet und auf eine direkte Kommunikation mit dem Compiler gesetzt werden. Auf diese Weise würden bestehende Ungenauigkeiten oder falsche Darstellungen minimiert.
Hauptfokus liegt jedoch auf den Fehlerkorrekturen und allgemeine Verbesserungen des Language Servers, um die Stabilität und Funktionalität der Erweiterung weiter zu verbessern und bestehende Bugs zu beheben.

References

[1] Eclipse Foundation AISBL. 2016. Eclipse LSP4J | projects.eclipse.org. https://projects.eclipse.org/projects/technology.lsp4j.

[2] JetBrains s.r.o. [n. d.]. IntelliJ IDEA – the IDE for Pro Java and Kotlin Development. https://lp.jetbrains.com/intellij-idea-promo.

[3] Microsoft Corporation. 2025. Official page for Language Server Protocol. https://microsoft.github.io/language-server-protocol.

[4] Martin Plümicke. 2024. Completing the Functional Approach in Object-Oriented Languages. In *A Second Soul: Celebrating the Many Languages of Programming - Festschrift in Honor of Peter Thiemann's Sixtieth Birthday,* Freiburg, Germany, 30th August 2024, Annette Bieniusa, Markus Degen, and Stefan Wehr (Eds.). Electronic Proceedings in Theoretical Computer Science, Vol. 413. Open Publishing Association, 43–56. doi:10.4204/EPTCS.413.4

Umsetzbare Abbildung von Untersorten und partiellen Operationen auf Many-Sorted-Algebra mit Konstruktoren

EDWARD SABINUS, Martin-Luther-Universität Halle-Wittenberg, Deutschland
WOLF ZIMMERMANN, Martin-Luther-Universität Halle-Wittenberg, Deutschland

Untersorten und partielle Operationen sind grundlegende Konzepte der Programmierung und doch werden sie in Sprachen wie Haskell, Coq oder Isabelle/HOL nicht unterstützt. Die Lösung dafür ist eine Abbildung von Order-Sorted-Algebra (OSA) auf die Many-Sorted-Algebra (MSA) unter Berücksichtigung von Konstruktoren. Sowohl Common-Algebraic-Specification-Langauge (CASL) als auch OSA geben eine Abbildung von Untersorten auf die MSA vor. OSA diskutiert auch Ansätze zur Darstellung von partiellen Operationen. Jedoch berücksichtigt die Spezifikation dieser Abbildung die Konstruktoren nicht, sodass die Umsetzung dieser Abbildung in eine der oben genannten Sprachen nicht möglich ist. In diesem Papier werden die Grenzen der Abbildung aus CASL und OSA auf MSA in Bezug auf die Umsetzbarkeit mit Konstruktoren gezeigt und eine umsetzbare Abbildung vorgestellt, die die Konstruktoren berücksichtigt.

CCS Concepts: • **Mathematics of computing** → **Coding theory**; • **Theory of computation** → *Proof theory*; • **Computing methodologies** → **Special-purpose algebraic systems**.

Zusätzliche Schlagwörter und Phrasen: Untersorten, Partielle Operationen, Many Sorted Algebra, Order Sorted Algebra, Konstruktoren, Abbildung

1 Einleitung

Untersorten und partielle Operationen sind grundlegende Konzepte der Programmierung, doch funktionale Programmiersprachen wie Haskell, Coq [2] oder Isabelle/HOL [8] unterstützen diese nicht: In den genannten Sprachen gibt es zwar Typklassen und Monaden, aber das reicht nicht für die Umsetzung von Untersorten im objektorientierten Sinne und weit übergreifenden partiellen Operationen. Die Lösung dafür ist eine Abbildung von Untersorten und partiellen Operationen auf eine einfachere Struktur: Many-Sorted-Algebra (MSA) [10]. Dabei ist zu beachten, dass die genannten Sprachen Operationen streng aufteilen in Konstruktoren, zur Darstellung von Sorten, und Funktionen, zur Berechnung von Ergebnissen. In Order-Sorted-Algebra (OSA) [5] sowie in Common-Algebraic-Specification-Language (CASL) [7] werden die Sorten nach der Untersortenrelation geordnet und eine Abbildung auf MSA vorgestellt.

Jedoch werden bei dieser Abbildung die Konstruktoren nicht berücksichtigt. So soll laut dieser Abbildung eine Untersorte s an ihre Obersorte s' durch eine Operation $embed_{s,s'}$ angepasst werden. Benutzt man beispielsweise zur Darstellung partieller Operationen eine Sorte $Undef$ als Untersorte von s', so würde bei der Abbildung auf eine funktionale Programmiersprache die Funktion $embed_{Undef,s'}$ einen Term der Sorte s' zurückgeben, der dann ein Standardwert für s' ist, wodurch $embed(undef)$ nicht darstellbar ist. Grund für das Problem ist, dass die Anpassungsoperation kein Konstruktor ist.

Daraus ergibt sich die Forschungsfrage: Wie kann eine Abbildung von Untersorten und partiellen Operationen auf MSA aussehen, sodass diese in funktionalen Programmiersprachen umgesetzt werden kann?

Die Idee ist die Anpassungsoperationen als Konstruktoren der Obersorte darzustellen, um das beschriebene Problem zu umgehen.

Authors' Contact Information: Edward Sabinus, edward.sabinus@informatik.uni-halle.de, Martin-Luther-Universität Halle-Wittenberg, Halle, Deutschland; Wolf Zimmermann, Martin-Luther-Universität Halle-Wittenberg, Halle, Deutschland, wolf.zimmermann@informatik.uni-halle.de.

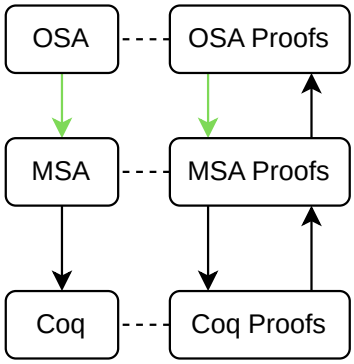


Fig. 1. Anwendung: Order-Sorted-Proof-Assistant (OPA)

Das Ergebnis ist eine Abbildung von Untersorten und partiellen Operationen auf MSA, die direkt in MSA-Unterstützende Sprachen wie beispielsweise funktionale Sprachen wie Haskell oder Coq implementiert werden können.

Eingesetzt werden soll das in den Order-Sorted-Proof-Assistant (OPA) (vgl. Abbildung 1), der das Ziel meiner Dissertation ist, in dem Spezifikationen und Beweise in OSA formuliert werden und in den funktionalen Proof-Checker Coq übersetzt werden.

Im Folgenden wird zuerst die Order-Sorted-Algebra beschrieben, davon insbesondere die Konzepte von Untersorten und partiellen Operationen. Dann wird die Abbildung von partiellen Operationen diskutiert. Danach wird die Abbildung von Untersorten auf MSA nach CASL sowie ihre Grenzen dargestellt. Anschließend wird die umsetzbare Abbildung von Untersorten auf MSA vorgestellt und an einem Beispiel erläutert. Zum Schluss werden verwandte Arbeiten diskutiert.

2 Order-Sorted-Algebra

In diesem Abschnitt wird die Order-Sorted-Algebra (OSA), entsprechend [5] eingeführt, sowie anhand eines Beispiels demonstriert. Ebenso wird vorgestellt wie ein Beweis in der OSA aussehen kann.

Als Grundlage für die Algebren verwenden wir Abstrakte Datentypen gemäß der Definition von Martin Wirsing [11].

Definition 1. *Order-Sorted-Signature [5]*
Eine Order-Sorted-Signature ist ein Tripel $\Sigma = (S, \sqsubset, F)$ *mit*

(1) S *ist die Menge von Sorten*
(2) $(S, \sqsubset^*)$ *ist eine Halbordnung (Untersortenrelation)*
(3) F *ist eine Familie von Mengen* $(F_{w,s})_{w \in S^*, s \in S}$ *von Operationssymbolen mit folgender Monotonie-Eigenschaft:* $f \in F_{w,s} \cap F_{w',s'} \wedge w \sqsubset^* w' \rightarrow s \sqsubset^* s'$

Eine Sorte s_1 *gilt als Untersorte von* s_2*, gdw.* $s_1 \sqsubset^* s_2$*.* $\sqsubset$ *ist die direkte Untersortenrelation.*

Definition 2. *Order-Sorted-Algebra [5]*
Sei $(S, \sqsubset, F)$ *eine Order-Sorted-Signature. Eine* $(S, \sqsubset, F)$*-Algebra* $A = (A_S, A_F)$ *ist eine Order-Sorted-Algebra, falls*

(1) A_S *ist eine Familie von Mengen* $\{A_s | s \in S\}$*, genannt Trägermengen*
(2) *für jedes Operationssymbol* $f_{w,s} \in F$ *ist die Operation* $A_f : A_w \rightarrow A_s \in A_F$
(3) $\forall s, s' \in S : s \sqsubset s' \rightarrow A_s \subseteq A_{s'}$
(4) $f \in F_{w,s} \cap F_{w',s'} \wedge w \sqsubset^* w' \rightarrow A_f : A_w \rightarrow A_s$ *ist äquivalent zu* $A_f : A_{w'} \rightarrow A_{s'}$ *für* A_w*.*

Definition 3. *Konstruktoren [11]*
Sei (S, F) eine Many-Sorted-Signature. Die Menge $F_C \subseteq F$ der Konstruktoren ist eine Menge von Operationssymbolen, so dass für jeden Σ-Grundterm t der Sorte $s \in S$ ein Σ_C-Grundterm t' der Sorte s mit $t = t'$ existiert, wobei $\Sigma_C = (S, F_C)$ ist.
D.h. in der Quotiententermalgebra enthält jede Äquivalenzklasse einen Konstruktorterm.
Analog dazu sind Konstruktoren in einer Order-Sorted-Signature mit $\Sigma_C = (S, \sqsubset, F_C)$.

Definition 4. *Abstrakte Sorte*
Sei $(S, \sqsubset, F)$ eine Order-Sorted-Signature. Eine Sorte $s \in S$ heißt abstrakt gdw. es keine Konstruktoren von s gibt mit Ausnahme von Konstruktoren von s' mit $s' \sqsubset^+ s$.

Definition 5. *Globale Untersorte „Undefiniert“*
Sei $(S, \sqsubset, F)$ eine Order-Sorted-Signature. Weiterhin sei $\bot$ die globale Error-Untersorte mit:

(1) $\bot = \{c_\bot\}$
(2) $c_\bot$ *ist Konstruktor von* $\bot$
(3) $\bot \sqsubset^* s, \forall s \in S$

Notation: $\bot = \{c_\bot\}$ wird auch als Undef={undef} *formuliert.*

Definition 6. *Abbildung partieller Operationen auf totale Operationen*
Sei $(S, \sqsubset, F)$ eine Order-Sorted-Signature und $pf_{w,s} \in PF_{w,s} \subseteq F$ mit $w = (s_1, ..., s_n) \in S^*, s \in S$ eine partielle Operation, dann wird $pf_{w,s}$ wie folgt auf eine totale Operation $tf_{w,s}$ abgebildet:

(1) $(x_1, ..., x_n, x) \in pf_{w,s} \rightarrow (x_1, ..., x_n, x) \in tf_{w,s}$
(2) $\forall (x_1, ..., x_n) \in w : \left(\nexists x \in s : (x_1, ..., x_n, x) \in pf_{w,s}\right) \rightarrow (x_1, ..., x_n, c_\bot) \in tf_{w,s}$

Bemerkung: Im Folgenden werden Operationen $tf_{w,s}$ mit $(x_1, ..., x_n, c_\bot) \in tf_{w,s}$ als partiell bezeichnet.

Definition 7. *Striktheit partieller Operationen*
Sei $(S, \sqsubset, F)$ eine Order-Sorted-Signature und $PF \subseteq F$ partielle Operationen.

$$\forall n \in \mathbb{N} : \forall s_1, ..., s_n, s \in S : \forall pf_{s_1,...,s_n,s} \in PF_{s_1,...,s_n,s} :$$
$$(x_1, ..., x_n, x) \in pf_{s_1,...,s_n,s} \rightarrow ((\exists i \in \{1, ..., n\} : x_i = c_\bot) \rightarrow x = c_\bot)$$ *(Striktheitsannahme)*

Die Striktheitsannahme gilt, außer es werden explizit Ausnahmen dafür spezifiziert.

Beispiel 1. *Order-Sorted-Algebra mit Beweis mittels Struktureller Induktion*

```
sorts Prog, Decls, Stats, Node, OCC, Nat, Undef
subsorts Prog, Decls, Stats < Node ; Undef < Prog, Decls, Stats, OCC, Nat
constructors prog: Decls >< Stats -> Prog
             noDecls: Decls
             noStats: Stats
             empty:OCC
             dot: OCC >< Nat -> OCC
             null: Nat
             succ: Nat -> Nat
             undef: Undef
operations occ: OCC >< Node -> Node
variables o:OCC, x:Node, d:Decls, s:Stats, n:Nat
axioms O1: occ(o,x) = prog(d,s) => occ(dot(o,null),x) = d
       O2: occ(o,x) = prog(d,s) => occ(dot(o,succ(null)),x) = s
       O3: occ(o,x) = prog(d,s) => occ(dot(o,succ(succ(n))),x) = undef
       O4: occ(o,x) = noDecls => occ(dot(o,n),x) = undef
       O5: occ(o,x) = noStats => occ(dot(o,n),x) = undef
```

```
Lemma O0p: occ(empty,prog(d,s)) = prog(d,s)
Lemma O0d: occ(empty,noDecls) = noDecls
Lemma O0s: occ(empty,noStats) = noStats

Theorem occEmpty: forall x:Node : occ(empty,x) = x
proof by induction x
IB: show occ(empty,prog(d,s)) = prog(d,s) by lemma O0p
IB: show occ(empty,noDecls) = noDecls by lemma O0d
IB: show occ(empty,noStats) = noStats by lemma O0s
qed
```

Das Beispiel 1 modelliert einen kleinen Teil des abstrakten Syntaxbaums einer Programmiersprache und führt einen einfachen Beweis mittels struktureller Induktion.

Dabei sind zwei Sorten besonders: `Node` und `Undef`. `Node` wird als abstrakte Sorte von allen Baumknoten des abstrakten Syntaxbaums eingeführt und hat deswegen keine eigenen Konstruktoren. `Undef` mit einem einzigen Konstruktor `undef` wird als Untersorte von allen anderen Sorten eingeführt und wird für die Darstellung partieller Operationen verwendet, wann immer diese nicht definiert sind. Durch das `Undef` wird die partielle Operationen `occ` total.

Das Besondere an dem Beweis über der abstrakten Sorte `Node` ist, dass die strukturelle Induktion über alle Konstruktoren der direkten Untersorten von `Node` iteriert.

Bemerkung 1.
Neben $\bot$ als globale Error-Untersorte gibt es auch alternative Abbildungen partieller Operationen auf totale Operationen:

(1) *Domainspezifische Untersorte: Dabei hat eine Sorte s eine Untersorte $\bot_s \sqsubset s$, sodass eine partielle Operation mit der Sorte s abgebildet werden kann auf eine totale Operation mit der Sorte $\bot_s$ [5, 11].*
Beispiel dafür ist der `Stack` *mit der Untersorte* `NonEmptyStack`*, sodass die* `pop` *Operation einen nicht leeren Stack bekommt und damit total wird [5].*
Das Problem bei dieser Abbildung ist, dass für jede Sorte eine spezielle Untersorte erforderlich ist und es nicht klar ist ob eine solche Abbildung immer automatisiert gefunden werden kann.
(2) *Error-Obersorte $\top$: Eine Sorte $\top$ mit $s \sqsubset^* \top, \forall s \in S$ die einen Fehler darstellt [5].*

Der Vorteil der Error-Untersorte $\bot$ im Vergleich zur Error-Obersorte $\top$ besteht bei den Sortenanpassungen: Tritt ein Fehler ein, so kann dieser als $\bot$ an jede andere Sorte angepasst werden, sodass keine Folgefehler ausgelöst werden. Als $\top$ können nur andere Sorten an den Fehler angepasst werden, sodass viele Folgefehler entstehen können. Liegt beispielsweise ein Fehler im Else-Teil einer bedingten Anweisung vor, so müsste bei der Error-Obersorte $\top$ auch der Then-Teil an $\top$ angepasst werden, sodass der Then-Teil fälschlicherweise einen Fehler darstellt, während mit der Error-Untersorte $\bot$ der Fehler an die Sorte des Then-Teils angepasst wird und keine Folgefehler entstehen.

Weiterhin unterstützt die Error-Untersorte $\bot$ auch domainspezifische Untersorten $\bot_s$, da $\bot \sqsubset^+ \bot_s, \forall s \in S$, während es bei der Error-Obersorte $\top$ dabei ein Problem mit der Fehlerdarstellung gäbe, da $\bot_s \sqsubset s \sqsubset^+ \top$, sodass $\bot_s$ an $\top$ angepasst werden müsste, wobei s dazwischen liegt.

3 Abbildung von Untersorten

Eine Abbildung von Untersorten wurde bereits durch CASL [7] vorgestellt. In diesem Abschnitt wird zuerst die Abbildung von Untersorten nach CASL vorgestellt und dann die Grenzen dieser

Abbildung gezeigt. Dann wird eine Lösung für das Problem der Abbildung nach CASL vorgestellt sowie Bedingungen für die Erfüllbarkeit dieser Abbildung geschlussfolgert.

Die Abbildung von Untersorten nach CASL ist wie folgt [7]

(1) Für jede Untersorte $s \sqsubset^* s'$ wird hinzugefügt
   - totale Einbettungsoperation $embed_{s,s'}$
   - partielle Projektionsoperation $proj_{s',s}$
   - Zugehörigkeitsprädikat $isMember_{s'}$, welches bestimmt ob ein Term t von der Sorte s ist.

(2) In Axiomen werden in atomaren Formeln, genauer in Gleichungen, die Einbettungsoperationen eingesetzt.

Die Grenzen der Abbildung nach CASL bestehen darin, dass Konstruktoren nicht berücksichtigt werden: CASL gibt die Operationen an, jedoch nicht ob welche davon Konstruktoren sein sollen. Nimmt man an, dass keine der Einbettungs- oder Projektionsoperationen Konstruktoren sind, treten folgende Probleme ein:

- Die Error-Untersorte $\bot$ kann nicht als Untersorte dargestellt werden, denn die totale Einbettungsoperation $embed_{\bot,s'}$ muss einen Wert aus s' als Standardwert auswählen. Dann wird durch Einsetzen der Einbettungsoperation in den Axiomen $\bot$ durch diesen Standardwert ersetzt und damit wird $\bot$ nicht umgesetzt.
- Abstrakte Sorten wie `Node` aus Beispiel 1 können nicht dargestellt werden, da sie keine Konstruktoren haben.

Die Schlussfolgerung ist, dass Konstruktoren notwendig sind. Definition 8 stellt die Abbildung von Untersorten auf MSA mit Konstruktoren vor.

Definition 8. *Abbildung von Untersorten auf MSA mit Konstruktoren*
Sei A_{OSA} eine Order-Sorted-Algebra mit der Signatur $(S, \sqsubset, F)$, Prädikaten P, Gleichungen E und A_{MSA} eine Many-Sorted-Algebra mit der Signatur (S', F'), Prädikaten P' und Gleichungen E'. Weiterhin sei $F = F_C \uplus F_O$ wobei F_C Konstruktoren und F_O keine Konstruktoren sind, dann wird A_{MSA} wie folgt aus A_{OSA} erzeugt:

(1) *Sorten: $S' = S$*
(2) *Operationen: $F' = F'_C \uplus F'_O$ mit*
   (a) *Hinzufügen von Einbettungskonstruktoren:*
   $F'_C = F_C \cup Embed$ *mit* $\forall s, s' \in S : s \sqsubset s' \rightarrow embed_{s,s'} \in Embed$
   (b) *Hinzufügen partieller Projektionsoperationen: [7]*
   $F'_O = F_O \cup Proj$ *mit* $\forall s, s' \in S : s \sqsubset s' \rightarrow proj_{s',s} \in Proj$
(3) *Hinzufügen von isMember-Prädikaten: [7]*
   $P' = P \cup IsMember$ *mit* $\forall s, s' \in S : s \sqsubset s' \rightarrow isMember^s_{s'} \in IsMember$, *wobei* $isMember^s_{s'}$ *bestimmt ob ein Term t von der Sorte s ist.*
(4) $embed_{s,w,s'}$ *mit* $w = s_1, ..., s_n \in S^*, s, s' \in S$ *ist eine Komposition von Einbettungskonstruktoren* $embed_{s_n,s'} \circ ... \circ embed_{s,s_1}$, *genannt Anpassungsfolge (engl. coercion sequence)*
(5) *Hinzufügen von Gleichungen: $E' = E_{Adapt} \cup E_{Proj} \cup E_{IsMember} \cup E_{CoerceSeqAquiv}$ mit*
   (a) E_{Adapt} *sind mit Einbettungskonstruktoren angepasste Gleichungen aus E, sodass [5]*
      (i) *Sei* $t^{OSA}_s := f_{s_{1'},...,s_{n'},s}(t^{OSA}_{s_1}, ..., t^{OSA}_{s_n})$ *ein Term aus $(S, \sqsubset, F)$, dann ist induktiv definiert*
      $t^{MSA}_s := f_{s_{1'},...,s_{n'},s}(embed_{s_1,w_1,s_{1'}}(t^{MSA}_{s_1}), ..., embed_{s_n,w_n,s_{n'}}(t^{MSA}_{s_n}))$ *ein Term aus (S', F').*
      (ii) $\forall t^{OSA}_s = t'^{OSA}_{s'} \in E$ *gilt:* $\exists s'' : s'' = s \sqcup s' \wedge embed_{s,w,s''}(t^{MSA}_s) = embed_{s',w,s''}(t'^{MSA}_{s'}) \in E_{Adapt}$
   (b) $\forall proj_{s',s} \in Proj : \left(proj_{s',s} \circ embed_{s,s'} = id\right) \in E_{Proj}$

(c) $\forall proj_{s',s} \in Proj, t \in s, c \in F'_{C_{s'}} : c(t) \neq embed_{s,s'}(t)$
$\rightarrow \exists w \in S^* : \left(proj_{s',s}(c(t)) = embed_{\perp,w,s}(c_\perp)\right) \in E_{Proj}$

(d) $\forall isMember^s_{s'} \in IsMember, t \in s' : \left(isMember^s_{s'}(t) = (proj_{s',s}(t) \neq embed_{\perp,w,s}(c_\perp))\right) \in E_{IsMember}$

(e) $\forall w_1, w_2 \in S^*, s, s' \in S : \left(embed_{s,w_1,s'} = embed_{s,w_2,s'}\right) \in E_{CoerceSeqAquiv}$
(Äquivalenz von Anpassungsfolgen)

Definition 9. *Projektionsfolge*
Sei A eine Many-Sorted-Algebra konstruiert entsprechend Definition 8. Dann ist $proj_{s,w,s'}$ *mit* $w = s_1, ..., s_n \in S^*, s, s' \in S$ *eine Komposition von Projektionsoperationen* $proj_{s_n,s'} \circ ... \circ proj_{s,s_1}$*, genannt Projektionsfolge.*

Bemerkung 2. *Das Konzept der Anpassungsfolgen (coercion sequence) kommt aus dem Übersetzerbau [9]. Für Anpassungsfolgen mit* $w \in S^0$ *ist* $embed_{s,w,s'} = embed_{s,s'}$ *und* $embed_{s,w,s} = id$*. Analog bei Projektionsfolgen.*

Bemerkung 3. *Die Einbettungskonstruktoren und Projektionsoperationen basieren auf der Theorie der Retrakte [3, 4]. Dabei ist der Einbettungskonstruktor die Inklusion und die Projektionsoperation die Retraktion. Entsprechend der Theorie der Retrakte gilt die Eigenschaft aus Satz 1, sowie gelten die Eigenschaften aus Definition 8.5b und Satz 1 auch bei Folgen von Einbettungen bzw. Projektionen, vgl. Satz 2.*

Satz 1. *Einbettung als bedingte Linksinverse der Projektion [4]*

(1) $t = embed_{s,s'} \rightarrow embed_{s,s'} \circ proj_{s',s}(t) = id(t)$
(2) $t \neq embed_{s,s'} \rightarrow embed_{s,s'} \circ proj_{s',s} = embed_{\perp,w,s'}$

Satz 2. *Axiome der Projektion erweitert auf Projektionsfolgen [3]:*

(1) $proj_{s',w^{-1},s} \circ embed_{s,w,s'} = id$
(2) $t \neq embed_{s,w,s'}(t) \rightarrow proj_{s',w^{-1},s}(t) = embed_{\perp,w',s}(c_\perp)$

Satz 3. *Äquivalenz von Projektionsfolgen:* $\forall w_1, w_2 \in S^*, s, s' \in S : proj_{s,w_1,s'} = proj_{s,w_2,s'}$

Beweis 1. *Beweis von Satz 3.*
Zu zeigen: $\forall w_1, w_2 \in S^*, s, s' \in S : proj_{s,w_1,s'}(t) = proj_{s,w_2,s'}(t)$
Fall 1. $t = embed_{s',w_1^{-1},s}(t')$

$proj_{s,w_1,s'}(t) = proj_{s,w_2,s'}(t)$	*Annahme*
$\leftrightarrow proj_{s,w_1,s'} \circ embed_{s',w_1^{-1},s}(t') = proj_{s,w_2,s'} \circ embed_{s',w_1^{-1},s}(t')$	*Satz 2.1*
$\leftrightarrow t' = proj_{s,w_2,s'} \circ embed_{s',w_1^{-1},s}(t')$	*Definition 8.5e*
$\leftrightarrow t' = proj_{s,w_2,s'} \circ embed_{s',w_2^{-1},s}(t')$	*Satz 2.1*
$\leftrightarrow t' = t'$	*Reflexivität* (=)

Fall 2. $t \neq embed_{s',w_1^{-1},s}(t')$
Umformung Annahme:

$t \neq embed_{s',w_1^{-1},s}(t')$	*Definition 8.5e*
$\leftrightarrow t \neq embed_{s',w_2^{-1},s}(t')$	

Beweis:

$proj_{s,w_1,s'}(t) = proj_{s,w_2,s'}(t)$	*Satz 2.2, Annahme*
$\leftrightarrow embed_{\perp,w',s'} = embed_{\perp,w'',s'}$	*Definition 8.5e*
$\leftrightarrow embed_{\perp,w',s'} = embed_{\perp,w',s'}$	*Reflexivität* (=)

q.e.d.

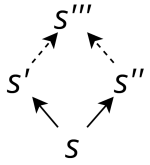


Fig. 2. Untersortenverband

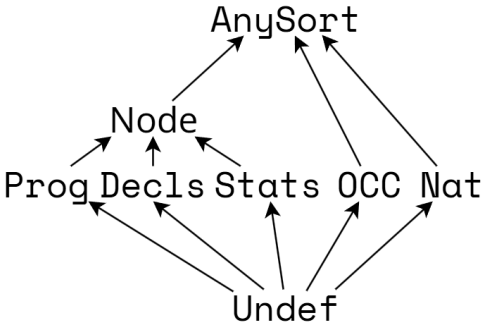


Fig. 3. Untersortenverband für Beispiel 1

Bemerkung 4. *Untersortenhalbverband*
Mit Untersorten können Gleichungen $t_s = t'_{s'}$ mit Termen $t_s, t'_{s'}$ und $s \sqsubset s'$ mit $s, s' \in S$ existieren [5]. Entsprechend kann $t_s = t''_{s''}$ mit $s \sqsubset s''$ existieren. Aus der Transitivität für $(=)$ folgt $t'_{s'} = t''_{s''}$, wobei weder $s' \sqsubset s''$ noch $s'' \sqsubset s'$ definiert wurde. Mit $s' \not\sqsubset s'' \wedge s'' \not\sqsubset s'$ wäre $t'_{s'} = t''_{s''}$ laut OSA [5] erlaubt, jedoch funktioniert die Abbildungsvorschrift auf MSA nicht, da $embed : s' \longrightarrow s''$ bzw. $embed : s'' \longrightarrow s'$ nicht existiert. Damit die Abbildung auf MSA funktioniert, muss also ein s''' existieren mit $s' \sqsubset s'''$ und $s'' \sqsubset s'''$, vgl. Abbildung 2. Allgemein betrachtet muss die Untersortenrelation $\sqsubset$ ein nach oben abgeschlossener Halbverband sein [5].

Satz 4. *Untersortenverband*
Aus dem nach oben abgeschlossenen Halbverband für die Untersortenrelation, und der Abbildung partieller Operationen mit der Error-Untersorte $\perp$, die $(\sqsubset)$ nach unten abschließt, folgt die Verbandseigenschaft für die Untersortenrelation $(\sqsubset)$.

Die Verbandseigenschaft für die Untersortenrelation $(\sqsubset)$ ist weiterhin für die Eindeutigkeit der Projektionsoperationen notwendig.

Beispiel 2. *Untersortenverband für Beispiel 1*
Abbildung 3 zeigt den Untersortenverband für Beispiel 1 als Hassediagramm.

Im Folgenden wird die Abbildung von Untersorten inklusive der durch die Untersorten umgesetzten partiellen Operationen an einem Beispiel dargestellt. Dazu wird das Beispiel 1 zur Order-Sorted-Algebra mit der beschriebenen Abbildung auf MSA abgebildet. Die dazugehörige Modellierung als MSA wird in Beispiel 3 dargestellt.

Beispiel 3. *Abbildung von Beispiel 1 als MSA*

```
sorts Prog, Decls, Stats, Node, OCC, Nat, Undef, AnySort
constructors
  embedUndefToProg: Undef -> Prog
  embedUndefToDecls: Undef -> Decls
  embedUndefToStats: Undef -> Stats
```

```
  embedProgToNode: Prog -> Node
  embedDeclsToNode: Decls -> Node
  embedStatsToNode: Stats -> Node
  embedUndefToOcc: Undef -> OCC
  embedUndefToNat: Undef -> Nat
  embedNodeToAnySort: Node -> AnySort
  embedOCCToAnySort: OCC -> AnySort
  embedNatToAnySort: Nat -> AnySort
operations
  projAnySortToNode: AnySort -> Node
  projAnySortToOCC: AnySort -> OCC
  projAnySortToNat: AnySort -> Nat
  projNodeToProg: Node -> Prog
  projNodeToDecls: Node -> Decls
  projNodeToStats: Node -> Stats
  projNatToUndef: Nat -> Undef
  projOCCToUndef: OCC -> Undef
  projProgToUndef: Prog -> Undef
  projDeclsToUndef: Decls -> Undef
  projStatsToUndef: Stats -> Undef
variables o:OCC, x:Node, d:Decls, s:Stats, p:Prog, n: Nat
axioms
O1: occ(o,x) = embedProgToNode(prog(d,s))
      => occ(dot(o,null),x) = embedDeclsToNode(d)
O2: occ(o,x) = embedProgToNode(prog(d,s))
      => occ(dot(o,succ(null)),x) = embedStatsToNode(s)
O3: occ(o,x) = embedProgToNode(prog(d,s))
      => occ(dot(o,succ(succ(n)),x) = embedProgToNode(embedUndefToProg(undef))
O4: occ(o,x) = embedDeclsToNode(noDecls) => occ(dot(o,n),x)
      = embedProgToNode(embedUndefToProg(undef))
O5: occ(o,x) = embedStatsToNode(noStats) => occ(dot(o,n),x)
      = embedProgToNode(embedUndefToProg(undef))
O6: occ(embedUndefToOcc(undef),x) = embedProgToNode(embedUndefToProg(undef))
O7: occ(o,embedProgToNode(embedUndefToProg(undef)))
      = embedProgToNode(embedUndefToProg(undef))
P1: projNodeToProg(embedProgToNode(p)) = p
P2: projNodeToProg(embedDeclsToNode(d)) = embedUndefToProg(undef)
P3: projNodeToProg(embedStatsToNode(s)) = embedUndefToProg(undef)
U0: embedProgToNode(embedUndefToProg(undef))
      = embedDeclsToNode(embedUndefToDecls(undef))
U1: embedDeclsToNode(embedUndefToDecls(undef))
      = embedStatsToNode(embedUndefToStats(undef))
```

In Beispiel 3 kommen durch die Abbildung bis zu zwei neue Sorten dazu: `Undef` (falls noch nicht in der OSA spezifiziert) und `AnySort`, die den Untersortenverband abschließen, entsprechend Beispiel 2.

Neben den bisherigen Konstruktoren aus Beispiel 1 (hier übersichtlichkeitshalber weggelassen) kommen Anpassungskonstruktoren für jede direkte Untersortenbeziehung hinzu. Analog dazu die Projektionsoperationen.

Die Axiome `O1` bis `O5` aus Beispiel 1 werden direkt übersetzt. Da die Operation `occ` partiell ist, kommen durch die Striktheitsannahme weitere Axiome (`O6` und `O7`) hinzu. Weiterhin kommen neue Axiome für die Projektionsoperationen hinzu. In Beispiel 3 wurden diese exemplarisch für `projNodeToProg` dargestellt.

Bemerkbar ist hier auch, dass die Darstellung der Error-Untersorte `Undef` für `Node` nicht eindeutig ist. Jede dieser Darstellungen ist dann gleich mit jeder anderen. Das wird mit den Axiomen `U0` und `U1` umgesetzt.

Beispiel 4. *Beweis aus Beispiel 1 in MSA*

```
Lemma O0p: occ(empty,embedProgToNode(prog(d,s))) = embedProgToNode(prog(d,s))
Lemma O0d: occ(empty,embedDeclsToNode(noDecls)) = embedDeclsToNode(noDecls)
Lemma O0s: occ(empty,embedStatsToNode(noStats)) = embedStatsToNode(noStats)

Theorem occEmpty: forall x:Node : occ(empty,x) = x
proof by induction x
  IB: show occ(empty,embedProgToNode(p)) = embedProgToNode(p) by induction p
    IB: show occ(empty,embedProgToNode(prog(d,s))) = embedProgToNode(prog(d,s))
      by lemma O0p
  IB: show occ(empty,embedDeclsToNode(d)) = embedDeclsToNode(d) by induction d
    IB: show occ(empty,embedDeclsToNode(noDecls)) = embedDeclsToNode(noDecls)
      by lemma O0d
  IB: show occ(empty,embedStatsToNode(s)) = embedStatsToNode(s) by induction s
    IB: show occ(empty,embedStatsToNode(noStats)) = embedStatsToNode(noStats)
      by lemma O0s
qed
```

Beispiel 4 zeigt den Beweis aus Beispiel 1 in MSA. Bei der Induktion über `x` wird somit über alle Einbettungskonstruktoren der Sorte `Node` iteriert. Um das selbe zu erreichen wie in Beispiel 1 muss eine verschachtelte Induktion durchgeführt werden, sodass die Konstruktoren innerhalb der Einbettungskonstruktorterme sichtbar werden, um so die selben Lemmata anwenden zu können.

4 Verwandte Arbeiten

Verwandte Arbeiten sind CASL [7] und Order-Sorted-Algebra (OSA) [5]. Beide stellen eine Abbildung von Untersorten auf Many-Sorted-Algebra (MSA) vor. OSA diskutiert auch Abbildungen von partiellen Operationen.

CASL stellt eine Abbildung von Untersorten auf MSA entsprechend Abschnitt **??** vor. Dabei ist das Problem, dass durch die Anpassungsoperationen Terme von Untersorten eliminiert werden, sodass die Abbildung nicht umsetzbar ist.

OSA stellt zwei Möglichkeiten für partielle Operationen dar: eine Domainspezifische Untersorte wie beispielsweise nicht-leerer-Stack oder eine einheitliche Error-Obersorte. Die Möglichkeit einer einheitlichen Error-Untersorte wird nicht erwähnt. OSA stellt ebenso wie CASL Anpassungsoperationen vor und schlagen zusätzlich vor, dass die Sorten einer Gleichung in der selben Zusammenhangskomponente bezüglich der Untersortenrelation liegen. Das reicht jedoch bei Mehrfachvererbung nicht aus. Wir zeigen, dass ein Verband erforderlich ist.

Es gibt weitere verwandte Arbeiten, die Erweiterungen mit Untersorten durchführen:

In [6] wird CASL um Logik höherer Ordnung (HOL) erweitert, wobei auch Untersorten und partielle Operationen berücksichtigt werden. Jedoch bleibt das bisherige Problem von CASL mit Untersorten bestehen.

In [1] wird der Calculus-of-Inductive-Constructions (CIC), welcher eine Basis für das Typsystem von Coq [2] darstellt, um Untersorten auf Konstruktoren erweitert. Damit wird also das Überladen von Konstruktoren ermöglicht, jedoch wird keine Abbildung von Untersorten auf MSA beschrieben. Weiterhin ist diese Erweiterung für CIC nicht für Coq verfügbar.

5 Zusammenfassung

Abbildung von Untersorten auf Many-Sorted-Algebra (MSA) ist eine Umsetzungsmöglichkeit von Untersorten.

Jedoch waren die bisherigen Abbildungen, die von Common-Algebraic-Specification-Language (CASL) und Order-Sorted-Algebra (OSA) vorgestellt wurden nicht ausreichend für eine praktische Umsetzung, denn durch die Anpassungsoperationen verschwinden die Terme von Untersorten.

Durch Spezialisierung der Anpassungsoperationen zu Konstruktoren werden Terme von Untersorten auch zu Termen Ihrer Obersorte, bei Verwendung des Anpassungskonstruktors, sodass die Umsetzung der Abbildung von Untersorten auf MSA realisierbar wird.

Damit können Untersorten auch mit Sprachen umgesetzt werden, die nur eine MSA unterstützen, aber keine OSA, wie beispielsweise funktionale Programmiersprachen wie Haskell oder Coq.

Literatur

[1] Gilles Barthe and Femke van Raamsdonk. 2000. Constructor Subtyping in the Calculus of Inductive Constructions. In *Foundations of Software Science and Computation Structures*, Jerzy Tiuryn (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 17–34.

[2] Yves Bertot and Pierre Castéran. 2013. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media.

[3] K. Borsuk. 1967. *Theory of Retracts*. Number Bd. 77 in Monografie matematyczne. Państwowe Wydawn. Naukowe. https://books.google.de/books?id=GIxsAAAAMAAJ

[4] Elfriede Fehr. 1989. *Semantik von Programmiersprachen* (1st ed.). Springer Berlin, Heidelberg. doi:10.1007/978-3-642-70271-6

[5] Joseph A. Goguen and José Meseguer. 1992. Order-sorted algebra I: equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science* 105, 2 (1992), 217–273. doi:10.1016/0304-3975(92)90302-V

[6] Till Mossakowski, Anne Haxthausen, and Bernd Krieg-Brückner. 2000. Subsorted Partial Higher-Order Logic as an Extension of CASL. In *Recent Trends in Algebraic Development Techniques*, Didier Bert, Christine Choppy, and Peter D. Mosses (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 126–145.

[7] Peter D. Mosses. 2004. *Casl Reference Manual: The Complete Documentation Of The Common Algebraic Specification Language (LECTURE NOTES IN COMPUTER SCIENCE)*. SpringerVerlag.

[8] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. 2002. *Isabelle/HOL. A proof assistant for higher-order logic*. Vol. 2283. Berlin: Springer. xiii + 218 pages.

[9] W. M. Waite and G. Goos. 1985. *Compiler Construction*. Springer-Verlag, Berlin, Heidelberg.

[10] Hao Wang. 1952. Logic of many-sorted theories. *Journal of Symbolic Logic* 17, 2 (1952), 105–116. doi:10.2307/2266241

[11] Martin Wirsing. 1991. *Algebraic Specification*. MIT Press, Cambridge, MA, USA, 675–788.

Enhancing Security and Robustness of Cyber-physical Systems using the Lemming Runtime System

NILS SCHEIDWEILER, Friedrich Schiller University Jena, Germany
CLEMENS GRELCK, Friedrich Schiller University Jena, Germany

Cyber-physical systems (CPSs) integrate computational and physical capabilities. Such systems do not only require functional correctness, but must also adhere to non-functional properties such as energy, time, security, and robustness (ETSR). Deploying CPSs on parallel and heterogeneous computing platforms while meeting the ETSR constraints and objectives introduces complex challenges, such as solving scheduling and mapping problems. We present our novel runtime system Lemming designed to simplify the deployment of CPS applications on parallel and heterogeneous hardware. Lemming adopts the exogenous coordination model TeamPlay and radically separates coordination concerns from application logic: Applications are organized as directed acyclic graphs (DAGs) of tasks that implement application-specific computations. Edges connect tasks and define timing dependencies and stream-based data exchange. Lemming serves as a toolbox from which the user can choose and configure features. Therefore, Lemming can be used for a wide range of applications. In this paper, we highlight Lemming's features for enhancing security and robustness using process-level and Linux user-based task isolation.

1 Introduction

Cyber-physical systems (CPSs) are computer systems that integrate computational and physical capabilities [3]. The system receives input from the physical world through sensors. It performs computations and controls the actuators which serve as interfaces to the physical world. CPSs are prevalent with applications ranging from medical devices to aerospace systems [13, 10].

For CPSs non-functional properties such as energy, time, security, and robustness (ETSR properties), shown in Figure 1, are as important as functional correctness [2]. A battery-powered CPS must operate in an energy efficient manner [8, 6]. Furthermore, a CPS must react to sensor input with actuator output within a certain time frame and therefore has real-time constraints [14]. The security of a CPS is important as a successful cyber attack can have severe consequences [7, 9]. Likewise, robustness against hardware and software faults is crucial [12, 11].

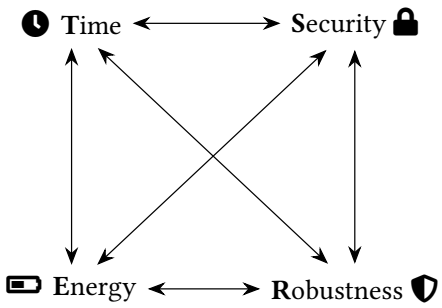


Fig. 1. Non-functional properties and their interdependence

The non-functional properties influence each other, as emphasized by the arrows in Figure 1. For example, when using dynamic voltage and frequency scaling (DVFS) [16, 4], the CPS can significantly reduce energy consumption. This often comes at the cost of increased computation time. Another example is the usage of n-modular redundancy to increase the fault-tolerance of a CPS at the cost of higher energy consumption as the same computation is performed multiple times. If enough computing units are available at that time, replication can be done in parallel. Otherwise, redundant execution requires more computation time.

Modern computing systems are parallel and heterogeneous and integrate multiple CPUs, GPUs, and specialized computing units. Besides meeting ETSR constraints, CPSs must also achieve various

Authors' Contact Information: Nils Scheidweiler, Friedrich Schiller University Jena, Germany, nils.scheidweiler@uni-jena.de; Clemens Grelck, Friedrich Schiller University Jena, Germany, clemens.grelck@uni-jena.de.

objectives, such as minimizing energy consumption [14]. Deciding which computing unit to use and when to execute a job in order to meet these objectives presents a challenging problem.

The TeamPlay coordination model [14] aims to address these challenges by separating coordination concerns from application logic. This separation simplifies the development process, as developers can focus on writing application-specific code while TeamPlay's compiler and runtime system handle coordination, mapping, and scheduling.

The TeamPlay's existing runtime system YASMIN [5] focuses on meeting deadlines and on optimizing for energy consumption, whereas security and robustness are of less importance. For instance YASMIN runs the complete CPS in a single process and models multiple cores via kernel threads.

In contrast, our novel runtime system *Lemming* [15] additionally focuses on security and robustness. Lemming improves the security of applications by building trust zones and therefore separating confidential from non-confidential computations. Lemming uses process isolation of computations to enhance the robustness of the system. A crash in one computation does not crash the entire CPS application. Lemming is implemented as a library for Linux and acts as a middleware between the application and the operating system. Also, Lemming serves as a toolbox from which the user can select and configure the runtime system's features. This enables users, among other things, to perform application-specific weighting of ETSR constraints and objectives. In a nutshell, Lemming facilitates the deployment of CPS applications on parallel and heterogeneous computing platforms and considers both ETSR constraints and application objectives.

The rest of the paper is organized as follows: In Section 2, we present Lemming's used task model. In Section 3, we take a look at Lemming's architecture and especially how our runtime system improves the security and robustness of CPSs. Finally, in Section 4 we draw conclusions.

2 Task model

TeamPlay [14] is an exogenous coordination approach because it separates coordination concerns from computation code. TeamPlay organizes applications as directed acyclic graphs (DAGs). Vertices implement application-specific code and are called tasks. Edges define timing dependencies and stream-based data-exchange between tasks and are called channels. Tasks have typed inports and outports. In more detail, a channel enables task communication by connecting a task's outport with another task's inport. A single data item transmitted through channels is called token.

A source task is a special task that has no inports and can be used as an interface to one or multiple sensors. In the same way, a sink task is a task with no outports and can be used as an interface to one or multiple actuators. A task is stateless, but a state can be modeled using short-circuit channels. These are channels which connect an outport with an inport of the same task. The execution of a DAG is periodic and must be completed before a deadline [14].

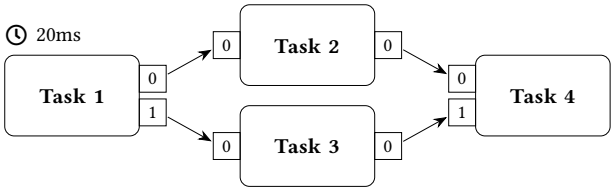


Fig. 2. A TeamPlay DAG with four tasks

Each task has a worst-case execution time and a worst-case energy consumption on a specific platform. These can be measured using static analysis or profiling [1, 17]. TeamPlay supports multi-version tasks, which permit multiple implementations per task with different ETSR properties. For example, a task that performs asymmetric RSA encryption can have multiple versions with different security levels by using different key sizes. The higher the security, the more computation time is needed, and the higher the energy consumption. If the objective is to reduce the energy consumption of the CPS, Lemming will prefer task versions with a lower security level.

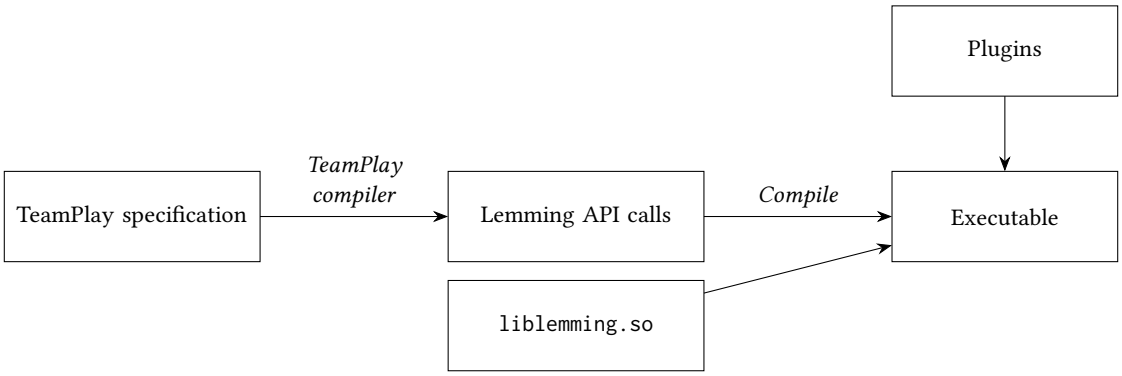


Fig. 3. Application development using TeamPlay

Figure 2 shows an example TeamPlay DAG. Task 1 is a source task and acts as a sensor. After execution of Task 1, Task 2, and Task 3 are executed. As soon as Task 2 and Task 3 finish execution, Task 4 executes and acts as an actuator. As indicated by the clock symbol, the DAG is executed periodically every 20 ms.

3 Lemming

Lemming is designed as a toolbox for CPS applications from which the user can choose and configure features. In this section, we first describe Lemming's architecture. After that, we focus on how Lemming enhances security and robustness through isolation mechanisms. Finally, we show how an application is implemented using the Lemming API.

Figure 3 illustrates the workflow for developing CPS applications using TeamPlay and Lemming. First, the user specifies the application using the TeamPlay coordination language. It defines the application's DAGs and the ETSR properties. The specification is then compiled into a C program containing a sequence of Lemming API calls to configure the runtime system. To produce the final executable, the C program is compiled and linked with the Lemming library. The task implementation is done in so-called plugins. These are loaded during the startup of the runtime system.

3.1 Architecture

Lemming has several components that we illustrate in Figure 4.

- **Controller**: The controller is the central component of Lemming. The controller creates a new process for each subcontroller. Furthermore, it creates a UNIX socket and shared memory for communication between the runtime system's components.
- **Subcontrollers**: The subcontrollers serve as intermediaries between the controller and the tasks. The subcontrollers each control a disjoint group of tasks and a disjoint subset of the system's computing units and orchestrate the task execution on these computing units. Each subcontroller spawns a worker for each computing unit. Additionally, the subcontroller spawns a process or thread for each task.
- **Workers**: The workers each control a single computing unit and orchestrate the task execution on it.
- **Tasks**: Tasks serve as interface between user code and the runtime system. Tasks perform the task communication and execute the plugins.

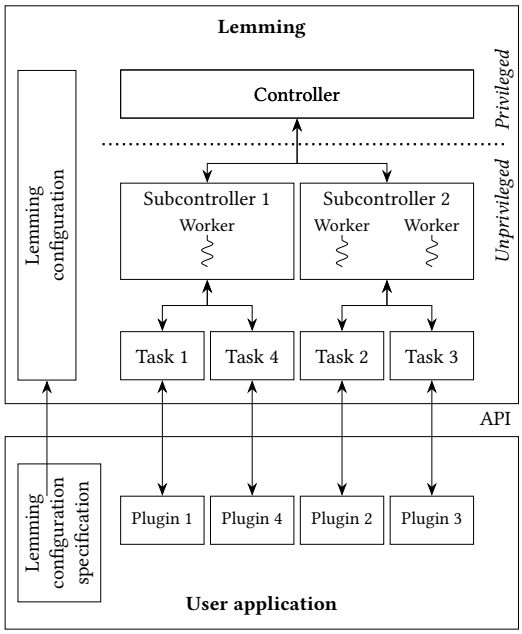


Fig. 4. Lemming architecture

- **Plugins**: The user assignes a plugin for each task. The plugins implements the user code. The user code reads from its task's inports, performs computations, and writes the result to its task's outport.
- **Channels**: Each channel connects a task's outport with another task's inport.

Figure 4 shows Lemming's configuration for the DAG in Figure 2 where Task 1 and Task 4 are controlled by Subcontroller 1 and Task 2 and Task 3 by Subcontroller 2.

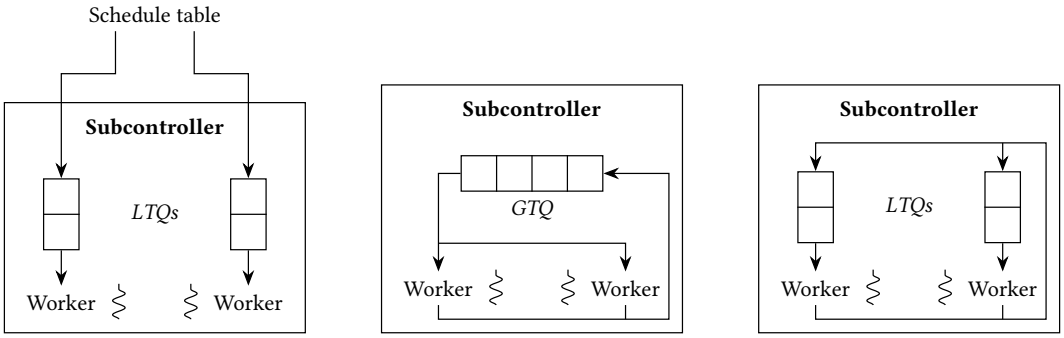
3.2 Task scheduling

In addition to periodically executed DAGs, Lemming also supports periodic, aperiodic, and sporadic tasks. Furthermore, Lemming offers three task scheduling classes: offline scheduling, global online scheduling, and partitioned online scheduling. Task scheduling is done non-preemptively. Task migration is allowed when offline or global online scheduling is used.

Offline scheduling is shown in Figure 5a. The user supplies a precomputed schedule table that defines the task release times and the target computing unit. Each worker and therefore computing unit has one LTQ (Local task queue) in which the runtime system inserts the tasks from the schedule table. When the user chooses online scheduling, scheduling decisions are performed during runtime. In the case of global online scheduling, Lemming uses one GTQ (Global task queue) for each subcontroller, as shown in Figure 5b. Scheduling is done using global earliest deadline first (EDF), which means that every task can be scheduled on every computing unit of its subcontroller. When partitioned online scheduling is used, each task is pinned to a computing unit. Lemming uses one LTQ for each worker, as shown in Figure 5c. Scheduling is done using EDF. For global and partitioned online scheduling, a schedulability analysis must be performed beforehand to ensure that the tasks are schedulable on the target system.

3.3 Task isolation

Lemming supports two task isolation mechanisms: The first mechanism aims to increase the robustness of the CPS by allowing the user to isolate tasks by running them in separate processes.



(a) Offline scheduling (b) Global online scheduling (c) Part. online scheduling

Fig. 5. Task scheduling classes

The second mechanism aims to increase the security of the CPS by placing tasks in different trust zones.

3.3.1 Robustness isolation. YASMIN runs all tasks in one single process [5]. The tasks therefore share the same address space. A crash of one task leads to the crash of the entire runtime system. Lemming improves robustness by offering a feature that isolates tasks in separate processes. It can be activated for each individual task. Therefore, the isolated tasks run in its own address space and a fault in one task does not affect the others. Communication between tasks is done via shared memory.

3.3.2 Security isolation. Lemming offers security isolation by building trust zones of tasks. Our security model offers isolation between tasks of different trust zones. Lemming achieves this by starting one subcontroller for each trust zone under different Linux users. If an attack successfully exploits a security vulnerability in one task, any potential harm is limited to the Linux user of the task's subcontroller. In the worst case, the attacker can read and manipulate data from all tasks in this trust zone, but the attacker does not have access to data from other trust zones. The security feature can be used, for example, to run tasks exposed to the physical environment in one trust zone. Confidential tasks, such as those that perform encryption and, therefore, store secret keys, can be run in another trust zone. To enable tasks from different trust zones to communicate with each other, the controller sets up shared memory for each so-called inter-subcontroller connection. The shared memory contains a ring buffer, which stores the tokens with additional information such as source task and port, and destination task and port. The shared memory must be accessible from both subcontrollers that run under different Linux users. Therefore, both Linux users must be assigned the same Linux group in advance. Lemming gives the Linux group read and write permissions for the shared memory. Figure 6 shows such inter-subcontroller connections for the application from Figure 4.

3.4 Lemming API usage

Figure 7a shows an example of how a task is configured using the Lemming API. The code specifies a single task by setting the task's attributes. This includes the task name, the path to the plugin binary, periodic scheduling, the period, the worst-case execution time (WCET), and the number of inports and outports. Furthermore, with the function `lem_task_attr_set_process` the robustness isolation is activated.

Figure 7b shows the implementation of a plugin which performs a string reverse. The compiled plugin binary must define a function called `plugin_run`, which is called by the runtime system for

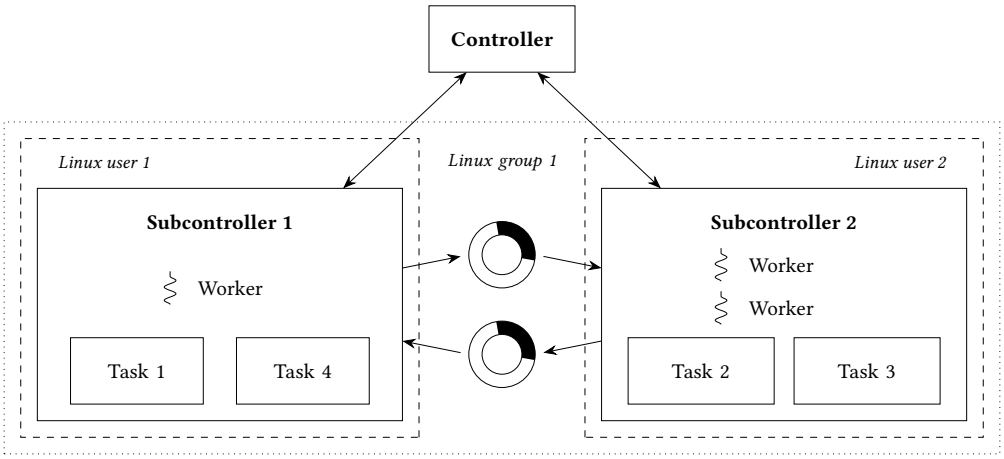


Fig. 6. Inter-subcontroller communication

```
lem_task_attr_init(&task_attr);
lem_task_attr_set_name(&task_attr, "Task 1");
lem_task_attr_set_plugin_path(&task_attr,
  ↪ "libplugin1.so");
lem_task_attr_set_rule(&task_attr,
  ↪ LEM_TASK_RULE_PERIODIC);
lem_task_attr_set_period(&task_attr, ms(20));
lem_task_attr_set_wcet(&task_attr, ms(1));
lem_task_attr_set_inport_count(&task_attr, 0);
lem_task_attr_set_outport_count(&task_attr, 2);
lem_task_attr_set_process(&task_attr, true);
```

(a) Task configuration using the Lemming API

```
void plugin_run(struct lem_plugin_context *pctx) {
    struct type *in =
        (struct type *)lem_task_get(pctx->task, 0);
    struct type *out =
        (struct type *)lem_task_put(pctx->task, 0);

    char *in_str = in->str;
    char *out_str = out->str;

    size_t length = strlen(in_str);
    for (size_t i = 0; i < length; i++) {
        out_str[i] = in_str[length - 1 - i];
    }
    out_str[length] = '\0';
}
```

(b) Plugin performing a string reverse

each task execution. The plugin reads from a task's inport by calling the function `lem_task_get` and specifying the ports index. In the same way, the plugin writes to an outport using `lem_task_put`.

4 Conclusion

In this paper, we introduced Lemming, our runtime system designed for cyber-physical systems on parallel and heterogeneous computing platforms. Lemming addresses the challenges of deploying CPS applications, including real-time scheduling, task mapping, inter-task communication, and meeting energy, time, security, and robustness (ETSR) constraints and objectives. By separating coordination concerns from application programming, Lemming simplifies the development process: Users only need to specify the application and provide plugins, while Lemming handles scheduling, mapping, and communication. Therefore, Lemming increases confidence in system correctness and lowers the bar for system engineers, enabling them to focus on application logic rather than low-level coordination. As a highly configurable toolbox, Lemming allows users to select and combine features tailored to their specific needs, making it suitable for a wide range of CPS applications. In this paper, we highlighted Lemming's features to enhance security and robustness using process-level and Linux user-based task isolation.

For future work, we plan to improve the security and robustness mechanisms. For example, we plan to implement security isolation through containerization of tasks. This has the advantage of requiring less setup, such as configuring Linux users and groups.

References

[1] Jaume Abella et al. “WCET analysis methods: Pitfalls and challenges on their trustworthiness”. In: *10th IEEE International Symposium on Industrial Embedded Systems (SIES)*. 2015, pp. 1–10. DOI: 10.1109/SIES.2015.7185039.

[2] Benny Akesson et al. “An Empirical Survey-based Study into Industry Practice in Real-time Systems”. In: *2020 IEEE Real-Time Systems Symposium (RTSS)*. 2020, pp. 3–11. DOI: 10.1109/RTSS49844.2020.00012.

[3] Radhakisan Baheti and Helen Gill. “Cyber-physical systems”. In: *The impact of control technology* 12.1 (2011), pp. 161–166.

[4] Mario Bambagini et al. “Energy-Aware Scheduling for Real-Time Systems: A Survey”. In: *ACM Trans. Embed. Comput. Syst.* 15.1 (Jan. 2016). ISSN: 1539-9087. DOI: 10.1145/2808231. URL: https://doi.org/10.1145/2808231.

[5] Rouxel Benjamin, Sebastian Altmeyer, and Clemens Grelck. *YASMIN: a Real-time Middleware for COTS Heterogeneous Platforms*. 2021. arXiv: 2108.00730 [cs.DC]. URL: https://arxiv.org/abs/2108.00730.

[6] Marco E. Gerards, Johann L. Hurink, and Philip K. Hölzenspies. “A survey of offline algorithms for energy minimization under deadline constraints”. In: *J. of Scheduling* 19.1 (Feb. 2016), pp. 3–19. ISSN: 1094-6136. DOI: 10.1007/s10951-015-0463-8. URL: https://doi.org/10.1007/s10951-015-0463-8.

[7] Houda Harkat et al. “Cyber-physical systems security: A systematic review”. In: *Computers & Industrial Engineering* 188 (2024), p. 109891. ISSN: 0360-8352. DOI: 10.1016/j.cie.2024.109891. URL: https://www.sciencedirect.com/science/article/pii/S0360835224000123.

[8] Houssam Kanso, Adel Noureddine, and Ernesto Exposito. “A Review of Energy Aware Cyber-Physical Systems”. In: *Cyber-Physical Systems* 10.1 (2024), pp. 1–42. DOI: 10.1080/23335777.2022.2163298. eprint: https://doi.org/10.1080/23335777.2022.2163298. URL: https://doi.org/10.1080/23335777.2022.2163298.

[9] Charalambos Konstantinou et al. “Cyber-physical systems: A security perspective”. In: *Proceedings - 2015 20th IEEE European Test Symposium, ETS 2014* (June 2015). DOI: 10.1109/ETS.2015.7138763.

[10] Edward Ashford Lee and Sanjit Arunkumar Seshia. *Introduction to Embedded Systems: A Cyber-Physical Systems Approach*. 2nd. The MIT Press, 2016. ISBN: 0262533812.

[11] Wouter Loeve and Clemens Grelck. “Fault-tolerance at your Finger Tips with the TeamPlay Coordination Language”. In: *Proceedings of the 35th Symposium on Implementation and Application of Functional Languages*. IFL ’23. Braga, Portugal: Association for Computing Machinery, 2024. ISBN: 9798400716317. DOI: 10.1145/3652561.3652571. URL: https://doi.org/10.1145/3652561.3652571.

[12] Risat Mahmud Pathan. “Fault-tolerant and real-time scheduling for mixed-criticality systems”. In: *Real-Time Syst.* 50.4 (July 2014), pp. 509–547. ISSN: 0922-6443. DOI: 10.1007/s11241-014-9202-z. URL: https://doi.org/10.1007/s11241-014-9202-z.

[13] Ragunathan Rajkumar et al. “Cyber-physical systems: The next computing revolution”. In: *Design Automation Conference*. 2010, pp. 731–736. DOI: 10.1145/1837274.1837461.

[14] Julius Roeder et al. “Towards Energy-, Time- and Security-Aware Multi-core Coordination”. In: *Lecture Notes in Computer Science*. Ed. by Simon Bliudze and Laura Bocchi. Vol. LNCS-12134. Coordination Models and Languages. Part 2: Coordination Languages. Valletta, Malta: Springer International Publishing, June 2020, pp. 57–74. DOI: 10.1007/978-3-030-50029-0_4. URL: https://inria.hal.science/hal-03273984.

[15] Nils Scheidweiler. "Lemming: Design and Implementation of a Novel Runtime System for Cyber-physical Systems". Friedrich Schiller University Jena. Master’s thesis. Aug. 2025.

[16] Dawei Sun et al. “Re-Stream: Real-time and energy-efficient resource scheduling in big data stream computing environments”. In: *Information Sciences* 319 (2015), pp. 92–112. ISSN: 0020-0255. DOI: 10.1016/j.ins.2015.03.027. URL: https://www.sciencedirect.com/science/article/pii/S0020025515001929.

[17] Peter Wägemann et al. “Worst-Case Energy Consumption Analysis for Energy-Constrained Embedded Systems”. In: *2015 27th Euromicro Conference on Real-Time Systems*. 2015, pp. 105–114. DOI: 10.1109/ECRTS.2015.17.

A Brief Comparison of Module Systems in SML and Java

JULIAN SCHMIDT, Baden-Wuerttemberg Cooperative State University, Germany

Since its release, SML has inspired many functional programming languages—such as Haskell, OCaml or F#—with features like type inference, algebraic data types (ADTs) and higher order functions. Over the past decades, many of these features have also been—at least partially—adopted to object-oriented languages, such as Java. Another powerful feature of SML is its module system. However, this aspect of SML has not been fully embraced in OOP languages, which typically rely on classes and interfaces to group related code. Java has introduced a module system in Version 9, called the JPMS. This document gives a small introduction to the Java and SML module systems and briefly compares the two languages with regards to their abstraction capabilities.

1 SML's Module System

Standard ML's (SML's) module system is infamous among functional programming languages. It encourages modularization of cohesive code into modules and therefore promotes code reuse. The module system of SML is distinct from the core language [3] and consists of the following three concepts:

- Structures
- Signatures
- Functors

In the following sections, an overview of these concepts is provided.

1.1 Structures

A SML structure packages related elements such as functions and values into a module. That means multiple related elements are combined into a single container.

```
structure Stack =
  struct
    exception E;
    val empty = [];
    fun push(q,x) = ...
    fun peek(x::q) = ...
    fun pop(x::q) = ...
    fun size(x::xs) = ...
  end;
```

Listing 1. An example stack structure in SML

Listing 1 shows an example of a Stack structure implemented using a list. As the implementations of the functions are mostly straigtforward and add little to the discussion, they are omitted for brevity in this and the following examples. In SML, the `struct - end` block is used to define a structure [6][p. 60]. In between this block is the actual definition of exceptions, values and functions contained in the structure. As the stack is implemented based on a list, we denote the empty stack to be the empty list and implement the common stack methods by pattern matching on the builtin list data type. Notice that we don't need to provide any type information, as SML can infer all types during compile time. This block is then assigned to a structure named `Stack`.
The structure can then be used in the REPL as follows:

Author's Contact Information: Julian Schmidt, Baden-Wuerttemberg Cooperative State University, Horb, Germany, j.schmidt@hb.dhbw-stuttgart.de.

```
- val s = Stack.empty;;
val s = [] : 'a list
- val s1 = Stack.push(s, 42);;
val s1 = [42] : int list
- Stack.peek(s1);;
val it = 42 : int
```

First `Stack.empty` is assigned to a new variable s. This assigns an empty list of arbitrary elements to the variable, as this is how we defined the empty value in Listing 1. Next the value 42 is pushed onto the empty Stack. At this point it is evident, that the list's elements need to be of type `int`, which is also returned by the Read–eval–print loop (REPL). Lastly we use the peek function to determine the top element of the stack, which the REPL correctly evaluates to 42. As we didn't assign the result to a new variable, the REPL automatically assigned the value to the variable `it`. One caveat of this approach is, that the internal data type of the stack is not opaque. That is, the list type is exposed to the outside and can be used with the stack functions directly, as the following example shows:

```
- Stack.push([2,3], 1);;
val it = [1,2,3] : int list
```

We will come back to this in the next section, to show how we can hide the internal data type of a structure.

1.2 Signatures

Signatures define a contract, which each structure that matches the signature, must satisfy. That is, a signature abstractly defines specifications, a structure must implement.
Listing 2 shows an example signature for the stack module.

```
signature STACK =
  sig
    type 'a t;
    exception E;
    val empty : 'a t;
    val push : 'a t * 'a -> 'a t;
    val peek : 'a t -> 'a;
    val pop : 'a t -> 'a t;
    val size : 'a t -> int;
  end;
```

Listing 2. An example stack signature in SML

At the beginning, we denote a new abstract type `'a t`, which we use throughout the function signatures. In SML `'a` is a type variable, i.e. it stands for an arbitrary type [6][p. 65]. For now `t` is an abstract type, which means we leave it to the structure to define the type. This also means multiple structures which match the signature, can have different implementations of the type. For the stack functions, we omitted all implementation details of the functions and instead provided the types. In SML, the asterisk (*) denotes the Cartesian product, which represents tuples. So `A * B` is the type of the tuple `(a: A, b: B)`
Since the structure in Listing 1 matches the signature in Listing 2, we can explicitly ascribe the structure to the signature.

```
structure Stack1 : STACK = Stack;
```

This constrains the structure to satisfy the signature [6][p. 269]. That is, if the structure does not implement every function and specify each abstract type, the code won't compile.
However with this approach, the internal data type of the Stack (i.e. 'a list) is still exposed. We can prevent this from using the opaque ascription :> instead of the transparent ascription : [6][p. 269-270].

```
structure Stack1 :> STACK = Stack;
```

With this change, the internal types of the structure are now hidden and only the abstract types of the signature, are present to the user of the module. The example from above will result in an error:

```
- Stack1.push([2,3], 1);;
Error: operator and operand do not agree
```

1.3 Functors

Functors are functions on modules and are not to be confused with Haskell functors. Functors in SML are completely separate from the core language and take modules as input to return a new module. This comes in handy in a number of scenarios, but let's look at a simple example to clarify this concept.
Suppose we want to create a module which represents an ordered set. We could yet again use the builtin list data type to represent a list, but we still run into a problem: We cannot allow the set to hold any data type. Since we want to create ordered set, we need to make sure, that an order is defined on the items in the set. Basically we want to constraint the types of elements which the set can hold to data types that can be ordered.
To achieve this, we start by generating a new signature which defines function, that compares elements of an arbitrary type `t`.

```
datatype order = LT | EQ | GT
signature ORDERED =
  sig
    type t;
    val compare t * t -> order;
  end;
```

With that, we can now write a functor that takes a structure which matches our `ORDERED` signature and generates a new structure that represents a sorted set. For the comparison of the elements, we use the compare function from the `ORDERED` signature.

```
functor GenOrderedSet (O : ORDERED) =
  struct
    exception E
    type elem = O.t
    type set = elem list

    val empty = []
    fun add([], x) = ...
    fun remove([], x) = ...
    fun contains([], x) = ...
  end;
```

Yet again, this use of a functor can be thought of as a constraint, i.e. the functor will provide an ordered set of elements type `t`, iff one explicitly provides an order for type `t`. This is similar

to Haskell type classes, with the key difference, that Haskell automatically passes the correct implementation for a type while the module needs to be passed explicitly in SML.

2 Java's module system

With version 9, the Java Platform Module System (JPMS) was introduced with two primary goals in mind [5].

- Reliable configuration, to replace the brittle, error-prone class-path mechanism with a means for program components to declare explicit dependences upon one another.
- Strong encapsulation, to allow a component to declare which of its public types are accessible to other components, and which are not.

This is done by extending the typical package-based modularization approach—which is primarily used for namespacing—through so-called modules. Similar to how multiple classes or interfaces can be bundled into packages, Java 9 provides the possibility to bundle multiple packages—and other data—into modules [2]. However, we will see that the definition of modules in Java is quite different from modules in SML.
The big new addition is a `module-info.java` file, which is located at the root of the projects package hierarchy and contains a module declaration [2, Section 1.1]. With it, we specify the name of our module, along with additional information to ensure **reliable configuration** and **strong encapsulation**.
Furthermore the module path replaces the class path for modules, i.e. module locations are provided in the module path as opposed to the class path [2, Section 2.1]. To ensure backwards compatibility, however, the class path is still available.
A basic example of a module declaration is given in Listing 3. In this case, we only define the name of the module. In the next two sections, the content will be extended with additional information.

```
module mymodule {}
```

Listing 3. "Basic declaration of a module-info.java file"

2.1 Reliable Configuration

Before Java 9 there was no way to declare explicit dependencies between code, directly in the Java language. Java would just search the classpath and try to find all the imports. One would therefore place all their dependencies onto the classpath and compile the project. This is typically done by build assistants like gradle or maven.
With the addition of the module system however, it is not only possible, but necessary to declare explicit dependencies [2]. This works only at the module level, i.e. a module declares which other modules it is required to compile and run. With the release of the module system, the JDK itself was also modularized [2, Section 1.4]. For example `java.base` contains the base code for the Java ecosystem, such as the packages `java.lang` and `java.util`. It is implicitly required by every other module, because it is mandatory for the JVM. Other JDK modules include `java.sql` for SQL related functionality and `java.desktop` for UI related tasks.
Suppose we want to access a SQL database in our code, so let's add the `java.sql` module to the dependencies in our module declaration file. We declare our dependency at both compile time and run time with the `requires` keyword.

```
module mymodule {
   requires java.sql;
}
```

This will ensure that during both compile time and run time, the module is available on the module path. In other words, the module system is tightly integrated into the java ecosystem with both the compiler and the JVM being aware of it.

2.2 Strong Encapsulation

Strong encapsulation provides a control mechanism to define which packages are for internal use and which packages should be exposed to other modules. That is, we can hide our internal code from outside use at package level.
Let's suppose we have the following structure in our example module:

```
├── module-info.java
└── org/
    └── example/
        ├── base/
        │   └── Converter.java
        └── internal/
            └── Utils.java
```

The `org` package is the root of our package hierarchy. We want to provide the `org.example.base` package for external use but keep the `org.example.internal` package hidden, as this is the location of the internal code. This code might change in the future and should not be used by external code. To denote which packages we want to expose for external use, we explicitly declare them in our `module-info.java` file. This is done with the `exports` keyword. Every package we do not explicitly declare here, can't be used from outside the module [2, Section 1.1]. This is not only ensured by the compiler during compile time, but also by the JVM at run time.
It is also possible to denote compile-time-only dependencies with the `requires static` option, respectively [1, §7.7.1]. Furthermore the module system does provide the possibility to restrict reflection access at runtime and an approach to load services. There are many other keywords and features of the JPMS, which can be read about in the Java Language Specification [1, §7].

```
module mymodule {
    requires java.sql;
    exports org.example.base;
}
```

3 Comparison of Abstraction Capabilities

It is clear that the JPMS and the module system of SML solve different problems.
The JPMS focuses on declaring dependencies and access control, while the SML module system enables modular programming by grouping related definitions into structures, specifying interfaces via signatures, and parameterizing modules through functors, which allows for encapsulation and abstraction [6, p. 313]. In Java, similar functionality is typically achieved using `classes` to group related code, `interfaces` to abstract over implementations and `packages` for namespace management.
These OOP concepts map vaguely to the concepts of the SML module system: For example, one might write the Stack example from Listing 2 in Java as shown in Listing 4. However, there is no direct mapping to SML functors in Java and abstract types seems to correspond only approximately to parametric polymorphism.

```java
import java.util.LinkedList;
interface IStack<T>{
    void push(T x);
    T pop();
    boolean empty();
    int size();
    T peek();
}

class Stack<T> implements IStack<T>{
    private final LinkedList<T> elements =
        new LinkedList<>();

    public void push(T x) {...}
    public T peek() {...}
    public T pop() {...}
    public int size() {...}
}
```

Listing 4. Stack implementation in Java

Compared to the SML implementation, we use an internal state in the `Stack` class, representing the stack as a list. Since the lists are mutable in Java, we do not need to return the data structure after each modification. Instead we mutate the internal list, following a more OOP-oriented approach. Similar to the opaque type abstraction in SML, we can also prohibit access on the internal data structure by declaring the field as `private`. Furthermore, we use generic type parameters, (i.e., `<T>`) to emulate SML's abstract types. Compared to the SML approach, the selection of the type is done at the time of instantiating the `Stack` class.

4 Conclusion

In this document, a brief comparison between Java's and SML's module system was provided. It is evident that, although they share the name "module system", they solve different problems. It seems more accurate to compare Java's OOP concepts like `classes`, `interfaces` and `packages` to SML's module system features like `signatures`, `structures` and `functors`. Nonetheless, these are still distinct abstraction mechanisms and provide different features in some aspects, i.e. there is no direct comparison to functors in Java.
Another interesting aspect would be to evaluate how abstract types and parametric polymorphism (i.e. generics) correlate. For further comparison, it would be valuable to look at Scala, another JVM language. Compared to Java, Scala supports and emphasizes the use of abstract types in addition to generics, seeming to offer a SML-like abstraction [4, Section 5.2].

References

[1] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, Daniel Smith, and Gavin Bierman. *The Java Language Specification, Java SE 24 Edition.* Oracle, 2025.

[2] Mark Reinhold. The state of the module system. https://openjdk.org/projects/jigsaw/spec/sotms/, 2016. Accessed: 2025-08-28.

[3] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised).* MIT Press, 1997.

[4] Martin Odersky and al. An overview of the scala programming language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004.

[5] OpenJDK. Project jigsaw. https://openjdk.org/projects/jigsaw/, 2017. Accessed: 2025-08-28.
[6] Lawrence C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 2nd edition, 1996.

Ein Typsystem für eine deklarative Sprache über ausführbare Zufallsexperimente

BALTASAR TRANCÓN WIDEMANN, TH Brandenburg / semantics gGmbH, Deutschland

Alea ist eine domänenspezifische deklarative Programmiersprache für Zufallsexperimente. Programme können auf zweierlei Arten interpretiert werden; einerseits statisch als Wahrscheinlichkeitsverteilung aller möglichen Ergebnisse, andererseits dynamisch als pseudozufällige Stichprobe. Alea soll als didaktisches Werkzeug in der Stochastik-Lehre dienen, als Simulationsumgebung, sowie in Entwurf, Analyse und Implementierung von Zufallselementen in Spielen. Die Benutzung der Sprache soll zwar elementare Mathematikkentnisse, aber möglichst wenig Programmiererfahrung erfordern. Im Vortrag wird der Entwurf eines Typsystems vorgestellt, dass diesen Anforderungen genügt.

Author's address: Baltasar Trancón Widemann, TH Brandenburg / semantics gGmbH, Brandenburg / Berlin, Deutschland, trancon@th-brandenburg.de.

Revising the TeamPlay Coordination Language: JenPlay

ERIC WINTZLER, Friedrich Schiller University Jena, Germany
CLEMENS GRELCK, Friedrich Schiller University Jena, Germany

A relevant branch of today's computing takes place on cyber-physical systems. On such systems non-functional properties are as important as functional properties. For lowering the challenge of writing software for those systems, one can use programming languages which introduce non-functional properties as first-class citizens. Such a language is TeamPlay. Following the concept of exogenous coordination, TeamPlay models an application as a directed acyclic graph (DAG) of components communicating via channels.

In this paper, we present the JenPlay coordination language. JenPlay is both a revision and an extension of the TeamPlay language with the general direction of making the language more expressive and to scale to larger systems. Focus areas of this work-in-progress presentation are the separation between component definition and instantiation, the seamless composition of larger DAGs out of previously defined ones and the structuring of channel lists.

1 INTRODUCTION

It is estimated that 98% of the world's computing devices operate in embedded or cyber-physical systems, where non-functional properties of program execution, such as energy (budgets), time (budgets), security or fault-tolerance can be as crucial as functional correctness. The rise of the internet-of-things with the edge-fog-cloud computing continuum and the increasing heterogeneity and parallelism of computing platforms rather solidify than mitigate the need for resource-aware software. These developments create new challenges for software engineering.

To address this challenge, we proposed the TeamPlay coordination language [11]. TeamPlay introduces energy, time, security and fault-tolerance as first-class citizens into the software design and engineering process. Following the concept of exogenous coordination [3], TeamPlay enforces a stringent software architecture with strict separation of concerns between operational detail and application-level design.

TeamPlay models an application as directed acyclic graph (DAG) of components. These components are connected through channels which servers as communication abstractions between components. These channels are defined through channel lists. However, this method does not allow for reusing a component multiple times in the DAG. Furthermore, it is hard to express the channels in a structured way. To address these uissues, we propose a revision and extension of the TeamPlay language. To distinguish between TeamPlay and our extension, we call our extension JenPlay.

We present the basic concepts of TeamPlay alongside with an example program in Section 2. Based on this, we introduce templates on component and graph level to mitigate the binding of a component to its position in the DAG in Section 3. In Section 4 we introduce functions as graph definition entities to allow for a more scalable and structured expression of the DAG. In Section 5 we discuss the related work, while summarising our work in Section 6. After all, we present the complete JenPlay syntax Appendix A.

2 THE TEAMPLAY COORDINATION LANGUAGE

A TeamPlay application specifies a set of components and how these components are connected in a directed acyclic graph (DAG). A TeamPlay program consists of three parts. In the first part application-wide properties – like the overall deadline of the specified program – are specified.

Authors' addresses: Eric Wintzler, Friedrich Schiller University Jena, Jena, Germany, eric.wintzler@uni-jena.de; Clemens Grelck, Friedrich Schiller University Jena, Jena, Germany, clemens.grelck@uni-jena.de.

After that, the abstractions of the computation entities are defined as components in the second part. In the last part, the dependencies between these components are expressed through communication channels.

An example of a TeamPlay application can be found in Listing 1. It describes the DAG illustrated in Figure 1. This program defines a simplified train computer of a high-speed railway train. Since the current position and speed of a train is are important information for a correct operating train service, our system calculates this information by two independent but exchangeable systems. For this, we model two source nodes `Position` and `Position2`, which determine the current position of the train. With those positions, the current velocity of the train is calculated by the `Speed` or `Speed2` components respectively. However, speed is not a value which can be calculated from only one position as input data. For this, we model our speed components to additionally have a state port, providing the last position as second input data.

In addition, we model a signal on the track with the `Signal` component. This signal is also a source node connected to the `TrainComputer` component. The output data of the `Signal` component dictates the maximum velocity at which the train is permitted to travel.

The `TrainComputer` component takes three velocity values as input, interpreting ones from `Speed` and `Speed2` as the current speed and the one from `Signal` as target speed of the train. The `TrainComputer` sends out status information to the `DriverDisplay` component and the `Acceleration` component through a broadcast link.

The `DriverDisplay` component is a sink node in the application. It collects the information coming from the `TrainComputer` and `Acceleration` components, presenting them to the train driver.

The `Acceleration` component calculates the amount of acceleration the driver should apply to the train. To inform the driver about that, it sends out the calculated value to the `DriverDisplay`. Since the acceleration amount can also be negative, the `Acceleration` component also sends its calculated value to the `Brake` component. Thereby, the `Brake` component serves as another sink node of the application.

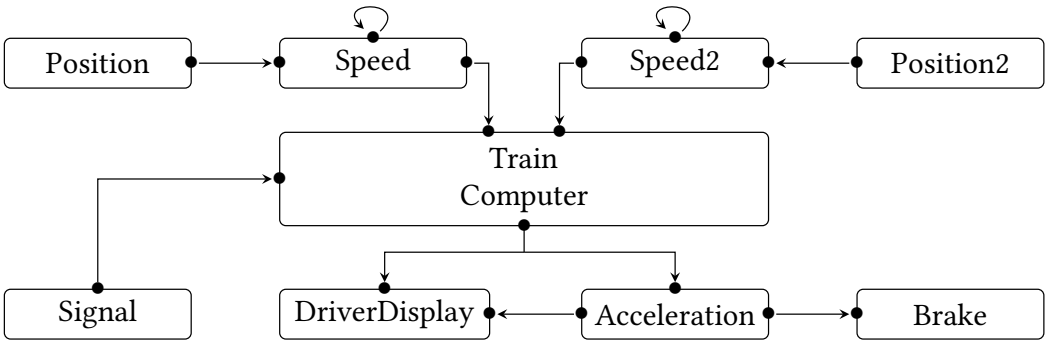


Fig. 1. The DAG specified by the code presented in Listing 1.

TeamPlay provides the *component* abstraction to represent computation steps. In Listing 1 components are defined at lines 5 – 33. The interface for input and output data of a component is defined in terms of named *ports* (for instance at line 8 of Listing 1). Additionally, ports define the type of data they expect or provide.

In TeamPlay components are defined as stateless computation abstractions. However, there might be applications which require some sort of state. To allow for expressing such applications in TeamPlay, a third set of ports can be defined – so-called *state ports*. These ports are defined in

```
app TrainController {
  deadline 50Hz
  period 50Hz
  components {
    Position { outports { position_t hectometer } }
    Position2 { outports { position_t position } }
    Speed {
      inports { position_t position }
      outports { int velocity }
      state { position_t old_position }
    }
    Speed2 {
      inports { position_t position }
      outports { int velocity }
      state { position_t old_position }
    }
    Signal {
      outports { int target_velocity }
    }
    DriverDisplay {
      inports { status_t current_speed; float acceleration }
    }
    Acceleration {
      inports { status_t target_speed }
      outports { float acceleration; int braking_power }
    }
    TrainComputer {
      inports { int current_speed_1; int current_speed_2; int target_speed }
      outports { status_t status }
    }
    Brake {
      inports { int power }
    }
  }
  channels {
    TrainComputer.status -> DriverDisplay.current_speed & Acceleration.target_speed
    Position -> Speed.position
    Speed.velocity -> TrainComputer.current_speed_1
    Acceleration.acceleration -> DriverDisplay.acceleration
    Signal.target_velocity -> TrainComputer.target_speed
    Position2.hectometer -> Speed2
    Acceleration.breaking_power -> Brake.power
    Speed2 -> TrainComputer.current_speed_2
  }
}
```

Listing 1. A simplified train control system in TeamPlay

Listing 1 at line 15. State ports are input and output ports at the same time, short-circuited from the output to input.

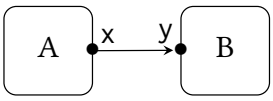
TeamPlay is designed for cyber-physical systems. In such systems the non-functional properties of an application are as important as its functional correctness. To express non-functional contracts concerning time, energy and security, one can also define *settings* for a component.

However, sometimes a functional contract can be fulfilled in different ways. To allow for selecting the best fitting variant of a component under given optimisation objectives and constraints for non-functional properties, TeamPlay introduces a multi-version feature. With this, multiple *versions* of a component can be defined, each of them having different non-functional properties. Thereby each version fulfils the same functional contract. With this, the versions of a component are semantically exchangeable.

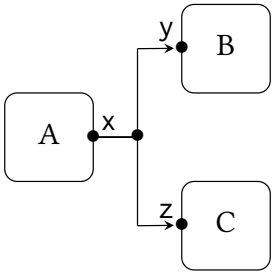
To express exchange of data between components, TeamPlay provides the channel abstraction. Channels are defined through channel lists. Thereby each channel is described in the form `<source> -> <sinks>`, connecting the specified output port of one component to the specified input ports of other components. Examples of channel definitions can be seen in Listing 1 at lines 36 – 42.

In TeamPlay multiple types of connections can be expressed, as illustrated in Figure 2. The most elementary connection type is a point-to-point connection (pipeline) from an output port of one component to an input port of another component, as illustrated in Figure 2a.

A pipeline connection can be extended to a broadcast connection, as illustrated in Figure 2b. In a broadcast connection, one component simultaneously sends the same data items to all its connected components. An example of a broadcast connection can be seen in Listing 1 at line 36.



(a) A pipeline connection.



(b) A broadcast connection.

Fig. 2. The communication paradigms natively supported by TeamPlay.

3 COMPONENT TEMPLATES

One limitation of TeamPlay is that one can not express that a component defined once can be reused. To illustrate this limitation and its background, we refer to the example of Listing 1 and Figure 1. We further assume that the components `Speed` and `Speed2` perform the same computation and therefore are equivalent. Because of this, we would like to delete `Speed2` and simplify the edges of our DAG through (implicitly) reusing the component `Speed` as in Listing 2.

One may assume, that the code of Listing 2 would lead to the DAG depicted in Figure 3. However, in Listing 1 we connect the output port of the `Signal` component to the second input port of `TrainComputer` at line 41 through a pipeline connection. Additionally, we connect the output port of `Position2` to the `Signal` component we want to omit.

In Listing 2, the objective is to connect the output of our copied `Speed` component to the second input port of the `TrainComputer` component. However, there is only one component named `Speed` defined. Since we use the component names for expressing the connections between them, the compiler can not identify that we want to create a copy of `Speed` and would therefore connect the outgoing edge of `Position` with the input port of the existing `Speed` component. Additionally,

```
app TrainController {
  deadline 50Hz
  period 50Hz
  components {
    Position {...}
    Position2 {...}
    Speed {
      inports { position_t position }
      outports { int velocity }
      state { position_t old_position}
    }
    Signal {...}
    DriverDisplay {...}
    Acceleration {...}
    TrainComputer {...}
    Brake {...}
  }
  channels {
    ...
    Speed.velocity -> TrainComputer.current_speed_1
    Speed.velocity -> TrainComputer.current_speed_2
    Position2.hectometer -> Speed
  }
}
```

Listing 2. A first approach on reusing the Speed component

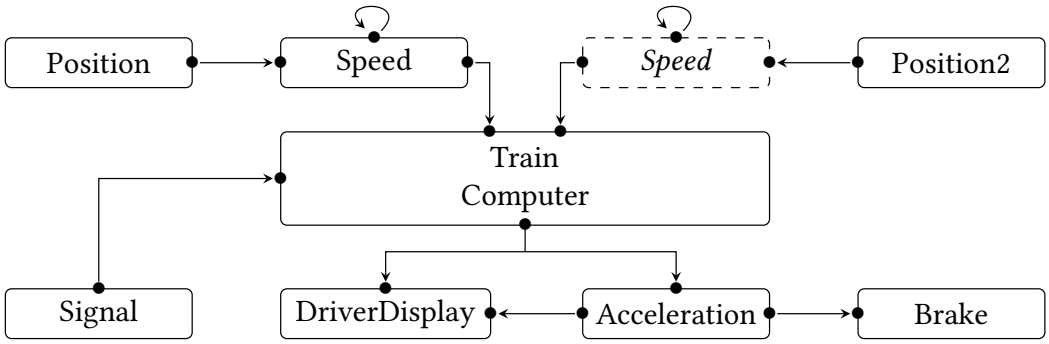


Fig. 3. What Listing 2 should express. The implicitly copied Speed component is highlighted.

we would get a second connection from the output port of Speed going to the second input port of TrainComputer. With this we obtain the DAG illustrated in Figure 4. However, this is a contradiction to what we want to express. Because of this, we say that components are bound to their position in the DAG.

In many general-purpose programming languages, the problem of reusing defined structures is solved through some kind of template mechanism. With this, a separation between definition and instantiation of a structure can be achieved. In combination with inheritance mechanisms, templating also allows programmers to express relationships between such definitions.

For defining templates, JenPlay introduces a new section in the JenPlay DSL, next to the sections for defining components and the channels. Within this part, we distinguish between two types of

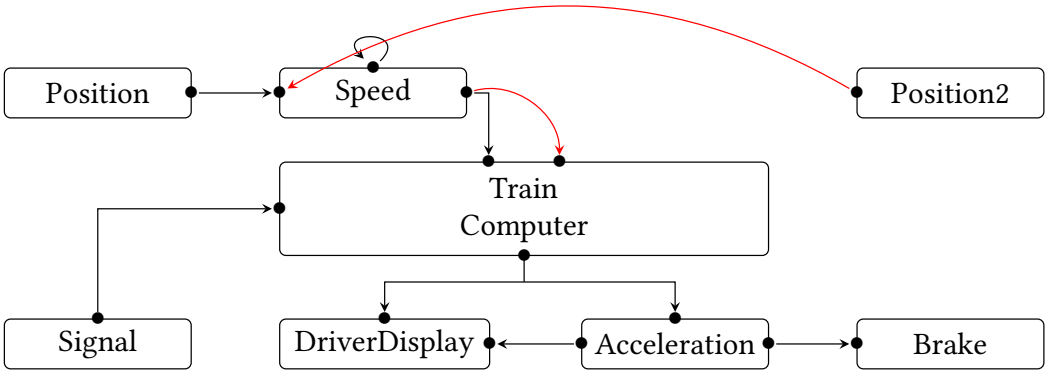


Fig. 4. What Listing 2 actually does express.

templates: *component templates* and *network templates*. Component templates are abstract definitions of a single component while network templates are abstractions of a DAG of components.

3.1 Component Templates

A *component template* is a set of *properties* (ports, settings and versions) that can be used as (partial) definition of a component. On the syntactic level, defining a template is similar to defining a component, but within the new `templates` section. With this, a template can specify ports as well as settings or versions of a component.

To allow for a flexible definition of components, we allow templates to be parametrised. For this, the parameters of a template are written in angular brackets after the name of the template. These parameters can then be used at any position in the template definition, where not a keyword is expected.

To apply a template to a component, the keyword `extends`, followed by the name of the template, has to be written after the components name. The effect of applying a template is, that everything defined in the template is now defined for the component instance. For parametrised templates, additionally the actual values of all parameters must be defined within angular brackets after the name of the template. With this, every occurrence of the parameter in the template definition is replaced by the specified value through textual substitution.

As an example of using templates we refer again to Listing 1 with the aim of replacing the `Signal` component through reusing the `Speed` component. For this, we define a template named `Velocity` and apply it to the components `Speed` and `Signal`. The resulting code is presented in Listing 3.

By applying a template to a component, templates can also be extended – as illustrated by the `Speed` and `Speed2` components in Listing 3. For this, the component, to which the template is applied to, also defines ports, settings or versions for itself. The effect of extending a template is in general, that the ports, settings and versions defined in the template are merged together with the ones defined in component.

If for a component and its instantiated template defining different ports, settings and versions are defined, we can simply merge them together. However, if – for instance – for a component an output port is defined and for the template, the component instantiates, another output port is defined, we merge the lists of output ports. However, if both ports are of equal names, we are merging the list of output ports by only using the port defined for the component. This strategy is applied to the merging of all ports and settings.

```
1  app TrainController {
2    deadline 50Hz
3    period 50Hz
4    templates {
5      Velocity<in_type, out_type> {
6        inports { in_type position }
7        outports { out_type velocity }
8      }
9    }
10   components {
...    ...
13     Speed extends Velocity<position_t, int> {
14       state { position_t old_position }
15     }
16     Speed2 extends Velocity<position_t, int> {
17       state { position_t old_position }
18     }
...      ...
36   }
37   channels {
...    ...
46   }
47 }
```

Listing 3. The simplified train control system with the usage of templates

Components can define multiple versions of themselves. However, one version can define several settings. Because of this, we have to adopt our merging strategy to this. For this, we are also merging versions by merging the settings of equal named versions together.

3.2 Network Templates

With introducing component templates one can specify multiple instances of a component without specifying it multiple times. However, for networks with repetitive elements one still must define the component instances and the connections between them for every repetition. To reduce the necessary amount of code duplication, JenPlay introduces templates also on graph level, which are called *network templates*.

As an example we refer to Listing 1 again. As one can see in Figure 5, parts of the graph are equivalent (boxed in grey) and can therefore be replaced by a network template application. For this we define a new network template called `Velocity` in Listing 4. This takes place within the `templates` section of a JenPlay application. To distinguish between component and network templates also on syntactic level, network templates start with the `network` keyword, followed by the name of the template.

To define a network template, one must define at least one component within the network template definition. Thereby, the components can be completely defined in place (lines 10 – 13) or by extending a component template (line 9). For a template to be a network template also the connections between the internally defined components must be specified. This is shown at lines 14 – 16 of Listing 4. For specifying the internal connections, a network template contains a `channels` section for specify the connections within this network. This section is syntactical equivalent to channel specification of the application outside the templates.

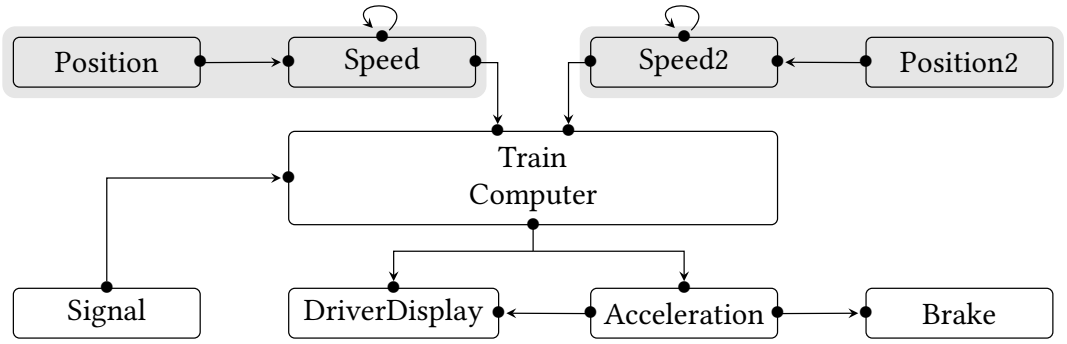


Fig. 5. The example DAG with highlighting where network templates will be used.

```
app TrainController {
  deadline 50Hz
  period 50Hz
  templates {
    Location {
      outports { position_t hectometer }
    }
    network Velocity {
      Position extends Location { }
      Speed {
        inports { position_t position }
        outports { int velocity }
        state { position_t old_position }
      }
      channels {
        Position.hectometer -> Speed.position
      }
    }
  }
  components {
    NewSpeed1 implements Velocity { }
    NewSpeed2 implements Velocity { }
    ...
  }
  channels {
    NewSpeed1.velocity -> TrainComputer.curr_speed
    NewSpeed2.velocity -> TrainComputer.target_speed
    ...
  }
}
```

Listing 4. The simplified train control system using network templates

A network template is instantiated as a component. This is done within the `components` section of a JenPlay application. Our aim is to keep the syntax of JenPlay as simple as possible. Therefore, we do not allow to extend network templates.

To instantiate a network template, one must specify the name of the component to create. This is followed by the `implements` keyword and the name of the template to instantiate. In Listing 4 this is done at lines 20 and 21.

Instantiating a network template creates a component which offers the unbound input and output ports of the templates inner components as interface to the components outside the template. In the example of Listing 4, this means, that the components `CurrentSpeed` and `TargetSpeed` are both offering an output port named `velocity`, as this is the only unbound input or output port in the `Velocity` network.

An instantiated network template can be used like a normal component in the definition of the applications channels. In the example of Listing 4 this is done at lines 45 – 46. Here, the output ports of the instantiated templates are connected to the input ports of the `TrainComputer` component. This leads to the DAG illustrated in Figure 6.

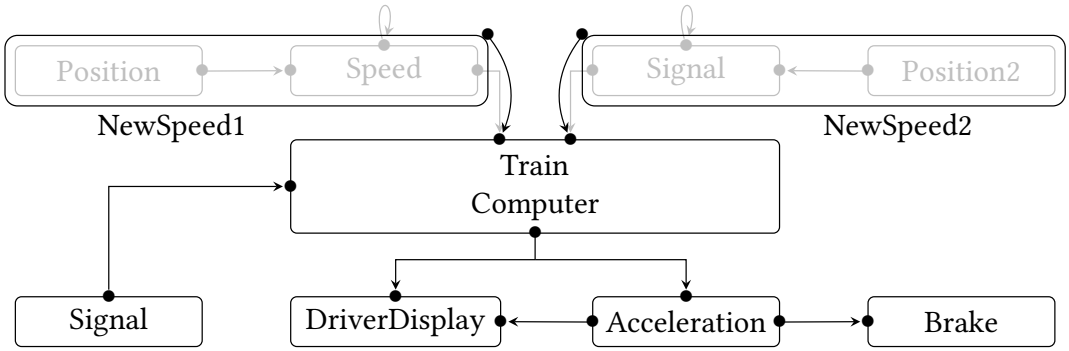


Fig. 6. The DAG expressed by Listing 4. The internal components of each subnet are shown in grey.

4 STRUCTURING GRAPH DEFINITION

To express the DAG, TeamPlay uses channel lists as described in Section 2. While this is a highly expressive but simple way of describing edges of a graph, it is an unstructured approach of defining a graph. With this, it is error-prone to structure the code describing the DAG. To introduce a stronger internal structure of the channel lists, we define a function-like notation for describing the channels.

TeamPlay allows for expressing the two communication paradigms illustrated in Figure 2: pipeline and broadcast. In addition, one can build two other communication types out of a pipeline connection, as illustrated in Figure 7. First, a pipeline connection can be extended to a fork connection. This is a connection where one component sends different data to each of the connected receiver components, as illustrated in Figure 7a. Second, with specifying multiple components each sending data to a different input port of another component, one can also define a join connection. This is a connection where multiple source sender components are sending data to one receiver component, as illustrated in Figure 7b.

To allow for expressing all of these communication paradigms, we introduce four instructions, each describing one communication paradigm. All of these instructions have in common, that they accept two parameters. These parameters will be called channel ends. Thereby a channel end can be the name of a component, the ID of a port or another communication instruction. However, the concrete syntax and semantics differs between the paradigms.

- **Point-to-Point Connections.** A point-to-point connection – or pipeline connection – (see Figure 2a) is expressed via the `pipe` function. This function builds up a simple point-to-point connection between the two specified channel ends.

(a) A fork connection. (b) A join connection.

Fig. 7. The communication paradigms that can be expressed through multiple pipelines.

- **Broadcast Connections.** In a broadcast connection (see Figure 2b) the sender sends the same message to all participants. In JenPlay this is expressed through the `broadcast` function. This function requires the first parameter (the sender) to be a single channel end while the second parameter (the receiver) must be a list of channel ends with at least one element.
- **Fork Connections.** A fork connection splits up the data stream from one component to multiple receivers (see Figure 7a). In JenPlay this is expressed through the `fork` function. This function requires the first parameter to be a component identifier and the second parameters a list of channel ends. Since this function splits up the output stream of a single component, it is required that the component specified as first parameter specifies as many output ports as channel ends specified in the second parameter.
- **Join Connections.** A join connection merges the data streams from multiple components in a single component (see Figure 7b). In JenPlay this is expressed through the `join` function. This function requires the first parameter to be a list of channel ends and the second one to be an identifier of a component. Since this function merges data streams into a single component, it is required that the input component specified by the second parameter has as many input ports as channel ends are specified by the first parameter.

Two examples of how to use these instructions can be found in Listing 5 and in Listing 6. These examples are modelling the same DAG as Listing 4. Therefore, we assume that the definition of the components and templates are equal throughout these examples. In addition, both listings are expressing the same DAG. The only difference between them is, that Listing 5 is only using pipeline and broadcast communications. In contrast, Listing 6 makes also use of fork and join connections. As a result we have replaced four pipeline connections by two complex connections.

By allowing a channel end to be also a communication instruction, all the communication functions can be nested. With this, the channel ends of a function can be replaced by another function call. Using this feature leads to a more graphical structure of the DAG in the source code.

In the definition above, the `fork` and `join` functions are mapping a list of ports to a component or vice versa. This introduces the problem of how such a mapping can be defined unambiguously, since the functions do not provide any knowledge of that. To solve this problem, we use the ordering of the port definitions for the component and the list of channel endpoints. With this, we map the first port defined for the component to the channel endpoint defined first and so on. However, this adds the implicit requirement, that the types of the ports mapped with this method must be compatible.

```
app primes {
  deadline 50Hz
  period 50Hz
  templates {
      ...
  }
  components {
    ...
  }
  channels {
    pipe(Signal.target_velocity, TrainComputer.target_speed)
    pipe(NewSpeed1.velocity, TrainComputer.current_speed_1)
    pipe(NewSpeed2.velocity, TrainComputer.current_speed_2)
    broadcast(TrainComputer, [DriverDisplay.curr_speed, Acceleration])
    pipe(Acceleration.acceleration, DriverDisplay.acceleration)
    pipe(Acceleration.braking_power, Brake.power)
  }
}
```

Listing 5. The exemplified train control system using the new channel notation

```
app primes {
  deadline 50Hz
  period 50Hz
  templates {
      ...
  }
  components {
    ...
  }
  channels {
    join(
      [
        NewSpeed1.velocity,
        NewSpeed2.velocity,
        Signal.target_velocity
      ],
      TrainComputer
    )
    broadcast(TrainComputer, [DriverDisplay.curr_speed, Acceleration])
    fork(Acceleration, [DriverDisplay.acceleration, Brake])
  }
}
```

Listing 6. The exemplified train control system using the new channel notation

5 RELATED WORK

Reo [2, 3] is a coordination model focussed on how components can be connected. Reo mainly focusses on various types of connecting channels through connectors. For this, Reo defines a channel as a named connection between exactly two components and thus the atomic form of a

connector. Thereby each channel end can be connected to at most one component. With knowledge of the names of the channel ends one can perform several operations on this channel. Especially, one can add or remove components from the channel, building up n-way connectors between components. Since those operations can be spread over the whole program code, the channel definition is not inherently structured. In contrast to JenPlay, Reo channels (and thus connectors) are not directed. However, the channel ends assigned to the components are directed.

S-Net [7] is a coordination language for asynchronous stream processing. In S-Net the components are composed to networks through operators. For this S-Net provides operators for serial compositions (pipeline) and parallel compositions (fork and join together) as well as for serial and indexed parallel replication. In contrast to JenPlay, all components are of Single-In-Single-Out (SISO) type. With this, components must have only one input and only one output port. This allows for defining components only by their input and output type. Like JenPlay, S-Net enforces a separation of the definitions of the components from the definitions of the connections between the components.

CAPIO-CL [12] is a coordination language for file-based I/O-streams on HPC workflows. Instead of defining a new programming language, CAPIO-CL uses JSON for expressing the components and channels. In contrast to JenPlay, channels in CAPIO-CL are not explicitly defined. Instead, channels are defined implicitly through the files a component reads as input or writes as output. Thereby, multiple files can be grouped together under a common name. The names of these groups are then used to define the inputs and outputs of the components. Having the same file or group name in an input and output of a component implicitly creates a channel.

Curracurrong [8] is both, a stream programming environment for wireless sensor networks and a query language for such systems. An application written in Curracurrong is also defined through specifying a graph. In contrast to JenPlay, the components and channels are specified through so-called queries. These queries are defined through operators, determining the type of stream transformation. Thereby five operators are presented: one for selecting source nodes and one for selecting sink nodes as well as operators for describing filters, splits (forks) and joins. With this, a stream graph in Curracurrong is not described by defining the components and their connections, but through combining the described operators to a query. Thereby, the operators are annotated with attributes to allow for a fine-grained description of the graph.

FastFlow [1] and WindFlow [9] are stream processing languages for multicore architectures. They describe a stream processing application as a DAG, too. In contrast to JenPlay and the approaches discussed before, those languages are no stand-alone languages but highly integrated into the C++ programming language: components and channels are defined as C++ classes. While each component is defined as its own class, the channels are predefined in the FastFlow or WindFlow library. In particular there exists classes for pipelines and broadcasts as well as for forks and joins. The components are then added to the channels through invocation of the channels member functions. With this, one could achieve a separation between component and channel definitions like in JenPlay. However, this separation is not as strictly enforced as in JenPlay.

CAPH [14, 15] is a dataflow programming language for configurable hardware. It is built from two layers: an actor description language (ADL) and a network description language (NDL). In the ADL the components are defined through ports for input and output data as well as a description of the components internal computations. The channels in CAPH are defined in the NDL. This is a small, higher-order, purely functional language, which describes the channels through defining and applying so-called wiring functions. These functions are describing how the components are connected to channels. This approach of using a functional notation for connecting the components is similar to JenPlay.

StreamIt [16] is a programming language designed to provide high-level abstractions for streaming applications and to serve as common machine language for grid-based processors. StreamIt programs are expressed as a DAG in a Java-like syntax. StreamIt uses the object-oriented approach as described for FastFlow and WindFlow to describe the DAG. With this, components and channels are defined in terms of classes. Thereby, the classes for the channels are derived from a set of predefined classes. These predefined classes allow for expressing pipeline connections (with multiple stages), SplitJoin connections (a fork followed by a join at the end) and FeedbackLoop connections. Thereby the FeedbackLoop can be compared to JenPlays stateful components. In contrast to JenPlay, StreamIt does not allow for expressing a stand-alone fork or join connection. Fork and join must always occur in combination through a SplitJoin. Furthermore in contrast to JenPlay, StreamIt does not provide a strong separation of describing the computation of an application and its structure in terms of a graph. In StreamIt components are added to the channels through the invocation of member functions within the channels' initialisation function.

Rust-SSP [10] and its successor SPar-Rust [5] are implementing high-level domain-specific languages for stream parallelism. For this, they are using Rusts built-in macro system. In both languages, the components are defined as Rust functions or closures. With this, those languages did not provide a strong separation of computation and structure description as in JenPlay. To connect the components to a DAG, Rust-SSP and SPar-Rust provide a set of macros. In contrast to JenPlay, Rust-SSP and SPar-Rust only provide pipeline connectors as well as so-called parallel connectors which are a fork with an implicit join to the next stage of the computation.

CircuitFlow [4] is a declarative language that describes how data flows through a workflow. CircuitFlow is a Haskell library describing dataflow applications as a DAG. For this, components are defined through type classes and thus have no explicit notation of ports. With implementing a Haskell library, CircuitFlow did not provide a strong separation between computation and description of the application as JenPlay. In CircuitFlow, the components are connected through so-called circuit constructors. With those constructors one can express pipeline and broadcast as well as fork and join connections.

AADL [6] is a modelling language supporting early and repeated analyses of a system's architecture with respect to performance-critical properties. AADL comes with huge set of abstractions for components, allowing for modularisation of components. However, like in JenPlay components can define input and output ports. Additionally, as in JenPlay, the connections between the components are defined in an special syntax. Thereby an output port of one component is connected to an input port of another component. In contrast to JenPlay, this simple form of creating channels in AADL only allows for creating a connection between exactly one output and exactly one input port. To create a connection involving multiple input ports or output ports or mixed directions, one has to involve another type of communication abstraction.

6 SUMMARY

A relevant number of todays computing is performed on cyber-physical systems. On such systems non-functional properties are as important as functional properties. To lower the challenge of writing software for such system, a domain-specific programming language like TeamPlay can be used. However, TeamPlay has some limitations in terms of programming productivity. In particular these are the equality of component definition and instantiation as well as the unstructured approach of defining the connections between those components.

In this paper we present the JenPlay extension to the TeamPlay coordination language. To overcome the limitation of equality of component definition and instantiation in TeamPlay, JenPlay introduces component templates as abstract definition of component instances which can be

extended by the components definitions. To reduce the amount of code duplication, JenPlay also provides network templates, which can utilised to reuse parts of the DAG.

For allowing a more structured graph definition, JenPlay changes the way of describing the DAG. Instead of channel lists, JenPlay uses a more graphical description through a function-like syntax. With this, all participants of a connection are located very close to each other.

Our work will continue in multiple directions. At one side, we work on modelling a number of use cases with our DSL to validate and possibly refine the design of JenPlay. On the implementation side, we will develop a compiler for the JenPlay coordination language. This compiler should generate code for the Lemming runtime system [13].

REFERENCES

[1] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. 2017. *FastFlow: High-Level and Efficient Streaming on Multicore.* John Wiley & Sons, Ltd, Chapter 13, 261–280. https://doi.org/10.1002/9781119332015.ch13

[2] Farhad Arbab. 2004. Reo: a channel-based coordination model for component composition. *Mathematical Structures in Computer Science* 14, 3 (May 2004), 329–366. https://doi.org/10.1017/s0960129504004153

[3] Farhad Arbab. 2006. *Composition of Interacting Computations.* Springer Berlin Heidelberg, 277–321. https://doi.org/10.1007/3-540-34874-3_12

[4] Riley Evans, Samantha Frohlich, and Meng Wang. 2022. CircuitFlow: A Domain Specific Language for Dataflow Programming. In *Practical Aspects of Declarative Languages (Lecture Notes in Computer Science, Vol. 13165)*, James Cheney and Simona Perri (Eds.). Springer International Publishing, Cham, 79–98. https://doi.org/10.1007/978-3-030-94479-7_6

[5] Leonardo G. Faé, Renato B. Hoffman, and Dalvan Griebler. 2023. Source-to-Source Code Transformation on Rust for High-Level Stream Parallelism. In *Proceedings of the XXVII Brazilian Symposium on Programming Languages (SBLP '23).* ACM, 41–49. https://doi.org/10.1145/3624309.3624320

[6] Peter Feiler, David Gluch, and John Hudak. 2006. *The Architecture Analysis & Design Language (AADL): An Introduction.* Technical Report CMU/SEI-2006-TN-011. https://doi.org/10.1184/R1/6584909.v1 Accessed: 2026-Jan-7.

[7] Clemens Grelck, Sven-Bodo Scholz, and Alex Shafarenko. 2010. Asynchronous Stream Processing with S-Net. *International Journal of Parallel Programming* 38, 1 (Jan. 2010), 38–67. https://doi.org/10.1007/s10766-009-0121-x

[8] Vasvi Kakkad, Saeed Attar, Andrew E. Santosa, Alan Fekete, and Bernhard Scholz. 2014. Curracurrong: a stream programming environment for wireless sensor networks. *Software: Practice and Experience* 44, 2 (Nov. 2014), 175–199. https://doi.org/10.1002/spe.2165

[9] Gabriele Mencagli, Massimo Torquati, Andrea Cardaci, Alessandra Fais, Luca Rinaldi, and Marco Danelutto. 2021. WindFlow: High-Speed Continuous Stream Processing With Parallel Building Blocks. *IEEE Transactions on Parallel and Distributed Systems* 32, 11 (Nov. 2021), 2748–2763. https://doi.org/10.1109/tpds.2021.3073970

[10] Ricardo Pieper, Dalvan Griebler, and Luiz Gustavo Fernandes. 2019. Structured Stream Parallelism for Rust. In *Proceedings of the XXIII Brazilian Symposium on Programming Languages (SBLP 2019).* ACM, 54–61. https://doi.org/10.1145/3355378.3355384

[11] Julius Roeder, Benjamin Rouxel, and Clemens Grelck. 2021. *Report on Environment-Adaptive Energy-, Time- and Security-aware Coordination.* Deliverable D2.3. Institut national de recherche en sciences et technologies du numérique. https://www.teamplay-h2020.eu/index2898.html?page=deliverables

[12] Marco Edoardo Santimaria, Alberto Riccardo Martinelli, Iacopo Colonnelli, Barbara Cantalupo, Massimo Torquati, and Marco Aldinucci. 2025. CAPIO-CL: The CAPIO Coordination Language. *International Journal of Parallel Programming* 53, 2 (Feb. 2025). https://doi.org/10.1007/s10766-025-00789-0

[13] Nils Scheidweiler and Clemens Grelck. 2025. Enhancing Security and Robustness of Cyber-physical Systems using the Lemming Runtime System. In *23. Kolloquium Programmiersprachen und Grundlagender Programmierung.* To appear.

[14] J. Sérot and F. Berry. 2014. High-Level Dataflow Programming for Reconfigurable Computing. In *2014 International Symposium on Computer Architecture and High Performance Computing Workshop.* IEEE, 72–77. https://doi.org/10.1109/sbac-padw.2014.18

[15] Jocelyn Sérot and Francois Berry. 2019. The CAPH Language, Ten Years After. In *Embedded Computer Systems: Architectures, Modeling, and Simulation*, Dionisios N. Pnevmatikatos, Maxime Pelcat, and Matthias Jung (Eds.). Springer International Publishing, 336–347. https://doi.org/10.1007/978-3-030-27562-4_24

[16] William Thies, Michal Karczmarek, and Saman Amarasinghe. 2002. StreamIt: A Language for Streaming Applications. In *Compiler Construction (Lecture Notes in Computer Science, Vol. 2304)*, R. Nigel Horspool (Ed.). Springer Berlin Heidelberg, 179–196. https://doi.org/10.1007/3-540-45937-5_14

A THE JENPLAY SYNTAX

In this appendix, we present the syntax of JenPlay as EBNF rules. For this we use the following notation:

- **Non-Terminal Symbols.** Names that are enclosed in angle brackets denote non-terminal symbols.
- **Terminal Symbols.** Names that are enclosed in quotation marks denote terminal symbols.
- **Grouping.** A list of symbols that is placed in brackets denotes a group.
- **Options.** A list of symbols enclosed in square brackets denote an option.
- **Alternative.** A | between two symbols or groups denotes that one can choose between the two alternatives.
- **Repetition.** A + after an (optional) group or symbol denote, that this group or symbol can appear one or more times.
- **Optional Repetition.** A * after an (optional) group or symbol denote, that this group or symbol can appear zero or more times.
- **Differences to TeamPlay.** Changes from the original TeamPlay syntax, as presented in [11] are highlighted in colour.
  - Changes coming from the introduction of component templates are highlighted in red.
  - Changes coming from the introducing of network templates are highlighted in green.
  - Changes coming from the modification of the channel syntax are highlighted in blue.

```
<App>           =>  "app" <Id> "{" <AppBody> "}"

<AppBody>       =>  ["period" <FrequencyConst>]
                    ["deadline" <FrequencyConst>]
                    [<Templates>]
                    <Components>
                    <Channels>

<Templates>     =>  "templates" "{" (<Template> | <Subnet>)+ "}"

<Template>      =>  <Id> ["<" <ParamList> ">"] "{"
                      ["inports" <PortList>]
                      ["outports" <PortList>]
                      ["state" <PortList>]
                      [<Settings>]
                      <Version>*
                    "}"

<Subnet>        =>  "network" <Id> ["<" <ParamList> ">"] "{"
                       <Component>+
                       <Channels>
                    "}"

<ParamList>     =>  <Id> ("," <Id>)*

<Components>    =>  "components"  "{"
                      (<Component> | <NetworkInst>)*
                    "}"
```

```
<Component>       =>  <Id> ["extends" <Id> ["<" <ParamList> ">"]] "{"
                        ["inports" <PortList>]
                        ["outports" <PortList>]
                        ["state" <PortList>]
                        [<Settings>]
                        <Version>*
                      "}"

<Setting>         =>  "period" <FrequencyConst>
                    |  "deadline" <FrequencyConst>
                    |  "arch" <StringConst>
                    |  "security" <IntConst>
                    |  "cname" <StringConst>

<Settings>        =>  "{" <Setting> (";" <Setting>)* [";"] "}"

<Version>         =>  "version" <Id> [<Settings>]

<PortList>        =>  "{" <Port> (";" <Port>)* [";"] "}"

<Port>            =>  <Type> <Id> ["[" IntConst "]"]

<NetworkInst>     =>  <Id> "implements" <Id> ["<" <ParamList> ">"] "{" "}"

<Channels>        =>  "channels" "{" <Channel>* "}"

<Channel>         =>  <OneToOne> | <Broadcast> | <Fork> | <Join>

<OneToOne>        =>  "pipe" "(" <ChannelEnd> "," <ChannelEnd> ")"

<Broadcast>       =>  "broadcast" "(" <ChannelEnd> "," <ChannelEndList> ")"

<Fork>            =>  "fork" "(" <Id> "," <ChannelEndList> ")"

<Join>            =>  "join" "(" <ChannelEndList> "," <Id> ")"

<ChannelEndList>  =>  "[" <ChannelEnd> ("," <ChannelEnd>)* "]"

<ChannelEnd>      =>  <Channel> | <PortId>

<PortId>          =>  <Id> ["." <Id>]

<FrequencyConst>  =>  <IntConst> "Hz"
```