On the Effectiveness of Interpreter-Guided Compiler Testing

Federico Lochbaum[1]
Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France

Guillermo Polito
Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France

Abstract

Guaranteeing that a compiler behaves correctly is a complex task often approached through test generation and fuzzing. Compiler test generation must not only ensure that a compiler generates code that does not break, but also that it implements the programming language semantics. Recently, *interpreter-guided test generation* has been proposed to test JIT compilers: Concolic-execution on the interpreter yields test cases for the language semantics which are then validated between differential testing of the interpreter and compiler. In previous work, this solution has been shown to find interpreter/compiler differences. However, little has been said about the effectiveness and the solution limits.

In this paper we study the behavior of this technique, to shed light on future improvements and research. We experiment with this technique on the JIT compiler for the Pharo programming language, on two different backends: ARMv7 and x86. We explore how effective the solution is in terms of compiler coverage and its limitations, and we discuss how future research can overcome them. Moreover, we investigate how this technique combined with random *constraint mutations* increases backend compiler coverage.

2012 ACM Subject Classification Software and its engineering → Compilers; Software and its engineering → Software testing and debugging

Keywords and phrases Virtual Machines, Concolic Testing, JIT compilers, interpreters, Differential Testing, Constraint Mutations, Compiler Coverage

Digital Object Identifier 10.4230/OASIcs.Programming.2025.20

Funding This work was funded by ANR JCJC Sapper.

1 Introduction

Testing language implementations is a challenging and important topic [5]. Compilers often have several layers, transformation phases, and error propagations, which make it difficult to explore by hand. Compiler bugs impact all applications compiled with it and appear often in corner cases that are hard to detect and reproduce. Research has been pursued to tackle this problem, with solutions such as fuzzing [34, 26, 18], simulation environments [30, 14, 23], Metacircular VMs [31] or multi-level debuggers [33, 17].

Recently, *interpreter-guided* test generation appeared as a promising technique to generate compiler tests [25]. Such a technique uses concolic testing [27] and differential testing [21] to find cases when the compiler or the interpreter does not correctly implement the programming language's semantics. On one side, concolic testing allows us to generate paths by a determined interpreter implementation. On the other side, differential testing provides us with a method to compare results with the expected oracle results. In the article, the authors found that this technique quickly finds behavioral divergences between an interpreter and different compilers.

[1] Corresponding author

However, besides the ability to find bugs, little has been said about such an approach, and questions remain to be asked: What kind of bugs does it find? What are its limitations? How can this technique be improved or complemented? Informally speaking, we can argue that execution paths guided by an interpreter do not entirely represent all paths found during compilation. For example, instructions manipulating integer constants may have a single execution path in the interpreter but different paths in the compiler, representing different instruction encodings for different constants.

In this paper, we present an in-depth analysis of the behavior of the technique when testing the Cogit JIT compiler [23, 22] for the Pharo programming language, on two different backends: ARMv7 and x86. We study the generated tests using both a quantitative and a qualitative approach based on code coverage.. Our analysis shows that most of the used frontend instructions are covered. Uncovered code is mostly represented by (a) features not used by the compiler frontend (*i.e.,* instructions never used by the language will thus never be compiled) and (b) by compiler infrastructure details that are not represented by interpreter-generated tests. Moreover, we identified several cases that can be improved by **constraint mutations**: mutating the concolic constraints to create altered versions of the same test. The contributions of this article are:

- **A quantitative analysis of the effectiveness of interpreter-guided testing.**
- **An identification of the limits of Interpreter-guided testing:** On the one hand, compiler frontend coverage saturates with low values because the compiler frontend implements several concerns (intermediate code generation, code patching, code garbage collection), while the technique tests a single one of them (code generation). On the other hand, compiler backend coverage misses instructions that are not used by the frontend, and encodes special values that have different semantics in the generated code while having a single semantics in the interpreter.
- **A first empirical analysis of constraint mutations applied.** We propose several mutations and evaluate their effectiveness to improve code coverage saturation.

2 Background

This section presents the concepts and terminology required to understand the rest of the paper. First, we briefly expose related work on language implementation testing, before going into interpreter-guided test generation and its implementation called *Ranger*. Finally, we explain our experimental platform, the Cogit JIT compiler, and the Pharo VM, and then present our research questions that guide the rest of this article.

2.1 Language Implementation Testing

Several solutions have been proposed to aid in testing and debugging programming language implementations. The main challenges of compiler testing are the automatic generation of test programs and oracles for their validation [5]. Existing work covers *black box* approaches to test generation using pseudo-random generation [34], grammar-based solutions [26], and program mutations [18]. The most commonly used oracle is differential testing [21], where two different implementations (or configurations) are compared side by side.

Lately, several works have explored the path of program fuzzing and differential testing between different Virtual Machines for the same programming language [7, 6, 13, 35, 24]. Similar approaches have been proposed to test virtual machine optimizations [1, 10, 2].

Recently, graybox approaches have been proposed to increase the efficiency of compiler testing, either guided by coverage [9] or interpreter concolic meta-interpretation[25].

```
1  Interpreter >> bytecodePrimAdd
2      | rcvr arg result |
3      rcvr := self internalStackValue: 1.
4      arg := self internalStackValue: 0.
5      (objectMemory areIntegers: rcvr and: arg) ifTrue: [
6          result := (objectMemory integerValueOf: rcvr) + (objectMemory integerValueOf: arg).
7          "Check for overflow"
8          (objectMemory isIntegerValue: result) ifTrue: [
9              self
10                 internalPop: 2
11                 thenPush: (objectMemory integerObjectOf: result).
12             ^ self fetchNextBytecode "success"]].
13     "Slow path, message send"
14     self normalSend
```

Figure 1 Excerpt of the bytecode handler implementing addition in the Pharo Virtual Machine.

2.2 Ranger: Interpreter-Guided Compiler Test Generation

A recent promising approach to compiler testing is guiding test generation with an interpreter [25]. This solution uses concolic execution on the interpreter instruction handlers (*i.e.,* the code implementing each instruction *e.g.,* a bytecode) to identify their different execution paths as sequences of symbolic constraints. Then, the set of concrete values that satisfies a path's constraints is used to execute the program along that path. The insight is that interpreter paths represent the language implementation semantics, and thus they can be exploited to test the compiler using differential testing.

To illustrate this solution, let us consider the instruction implementing addition in Pharo, a dynamically typed high-level language, illustrated in Figure 1. This operation works on different types and presents four execution paths, as illustrated in Table 1. These four different paths should be semantically equivalent in both the interpreter and compiler, otherwise they show a fault in one of the implementations.

Table 1 Path constraints from the byte-code instruction in Figure 1.

Case	Path
Normal	isInt(arg0), isInt(arg1), isInt(arg0+arg1)
Overflow	isInt(arg0), isInt(arg1), isOverflow(arg0+arg1)
Type Error 1	not(isInt(arg0))
Type Error 2	isInt(arg0), not(isInt(arg1))

2.3 Experimental Context: The Cogit JIT Compiler

Our experimentation platform is the Pharo Virtual Machine and its Cogit JIT compiler, an industrial-level Virtual Machine [14, 23]. The VM implements a threaded byte-code interpreter and a linear non-optimizing JIT compiler named Cogit [22] that includes polymorphic inline caches [12]. It implements 255 bytecode instructions and 340 native methods, several duplicated in both the interpreter and the JIT compiler.

Experimental Backends/Code Generators. We consider two different compiler configurations: an x86 and an ARMv7 code generation target. Both code generators present different features and constraints imposed by the target platform.

Compiler Architecture. The Cogit JIT compiler is a method compiler[2] that translates bytecode methods into machine code methods. It is organized into three main modules:

CogitRTL. The intermediate language used by the compiler to represent and manipulate programs. This language is an intel-inspired assembly-like 2 address code (2AC) language. Instruction operands are either constants or virtual registers from a fixed list of registers that are manually mapped ahead of time to machine registers by JIT-compiler developers.

Frontend. This module translates a Pharo method (bytecodes) into a CogitRTL program. Each bytecode is translated separately into a sequence of CogitRTL instructions.

Backend. This module implements an assembler, translating CogitRTL to machine code.

Notice that the frontend is strongly coupled to the programming language under translation, the Pharo language in this case. Also, the backend is designed to be independent of the source language, but dependent on the target (the instruction set architecture). Thus, the backend implements many instructions not necessarily generated by the frontend (See 5.5).

2.4 Research Questions

In their paper, Polito et al. explored a design using concolic testing and differential testing together, focusing on its ability to find interpreter/compiler differences [25]. However, little has been said about the effectiveness and limits of this solution, which would shed light on future improvements and research. In this paper, we explore the following research questions:

RQ1: Effectiveness. How effective is it to combine *Differential Testing* with *Concolic testing* for *Interpreter-guided testing*?

RQ2: Limits. What are the limits of using an interpreter as a source of compiler tests?

RQ3: Mutations. How much can *Constraint Mutations* improve the effectiveness of compiler testing? Can they be used to explore a major diversity of compiler paths?

3 General Methodology

This section presents the general methodology for our study. Subsequent sections present our results and analyses.

Test Subjects. We analyze separately the interpreter concolic exploration of *bytecodes* and *native methods*, which represent two separate concepts for the compiler and the Pharo programming language. **Bytecodes** are unsafe intermediate instructions of the virtual machine instruction set. We selected a subset of 175 bytecodes out of 255. **Native methods** are low-level functions defining essential functionality in a safe manner (*e.g.,* it includes type checks). We selected a subset of 132 native methods out of 340. Both can be compiled into machine code using the CogitJIT compiler. For each, we generate all test cases for compiler implementations for the `x86` and `ARMv7` architectures.

Test coverage metric as test quality. To measure the effectiveness of interpreter-guided test generation, we use test coverage as quality metric. We measure coverage at the basic block level: a basic block that started executing is considered executed. We discuss in Section 7 the tradeoffs of this decision. Moreover, we segmented the resulting coverage into *Compiler Backend* and *Compiler Frontend* methods for fine-grained analysis.

[2] compiles one method at a time

Table 2 Compiler classes covered running test cases guided by native methods and bytecodes. Each row shows the relative coverage percentage and the absolute number of covered basic blocks in between parentheses.

	Class Name	Bytecode coverage	Native method coverage	Union coverage
Backend	CogAbstractInstruction	16.45% (37)	28.48% (63)	29.74% (66)
	CogOutOfLineLiteralsARMCompiler	33.33% (74)	29.16% (96)	33.33% (108)
	CogIA32Compiler	14.69% (361)	36.49% (605)	37.91% (624)
	CogARMCompiler	21.84% (332)	54.20% (588)	55.46% (606)
Frontend	StackToRegisterMappingCogit	1.57% (6)	1.57% (6)	1.57% (6)
	SimpleStackBasedCogit	19.21% (50)	5.41% (16)	21.18% (58)
	Cogit	15.94% (183)	25.59% (279)	27.55% (306)
	Total	16.12% (1043)	25.55% (1653)	27.42% (1774)

Test Execution. For each interpreter instruction, we generate all test cases using the concolic executor. Each test knows its executed path, its input constraints, and its expected result. For each generated test, we execute it on the compiler for each of the target architectures and compare the result to the expectations set by the interpreter.

Interpreting Test Results. We remove from the study concolic cases containing unsatisfiable constraints. For each other test we execute the compiler to generate the associated machine code and execute the machine code on a simulator[3]. If the test case expects a fault result, we validate that the simulation executes until the predicted failure. If we expect a successful result, we assert that the simulation returns the same value as the interpreter.

4 Interpreter-Guided Testing Effectiveness

Running the testing technique generates 5692 bytecode test cases and 3508 native method test cases. These are reduced respectively to 2944 and 1672 after the cleaning step (See 6.1).

Table 2 presents the coverage results of both cases. Each line shows a different compiler component, organized in Backend and Frontend. Columns show code coverage as a percentage of the total code of covered blocks and in parentheses as the absolute number. We split coverage into **bytecode coverage**, **native method coverage**, and in the last column, the set union of them, **union coverage**. The table shows that bytecode tests cover 1043 basic blocks representing **16.12 %** of all compiler code while native methods cover 1653 basic blocks representing **25.53%**.

The table shows that backend classes exhibit higher native method coverage compared to bytecode coverage. For instance, `CogARMCompiler` achieves 54.20% native method coverage, whereas its bytecode coverage remains at 21.84% (32.36% less). We observe a similar pattern in the `CogIA32Compiler` backend for `x86`. Frontend classes, however, show lower overall differences between bytecodes and native methods. For example, `SimpleStackBasedCogit`

[3] `https://www.unicorn-engine.org/`

```
1 SimpleStackBasedCogit >> genPushReceiverBytecode
2   needsFrame
3     ifTrue:
4       [self MoveMw: FoxMFReceiver r: FPReg R: TempReg.
5        self PushR: TempReg]
6     ifFalse:
7       [ self PushR: ReceiverResultReg ].
```

Figure 2 Code example of a not needed frame uncovered case. The red code is not covered.

achieves 19.21% coverage through bytecodes but only 5.41% via native methods (13.8% less) but `Cogit` achieves 9.65% more in native methods than bytecodes. In contrast, the union coverage reveals that bytecode contributes only 1.87% unique nodes to the overall coverage.

Conclusion. Combining Differential Testing and Concolic Testing using Interpreter-Guided Testing is a good beginning for generating compiler tests. However, we find an upper bound on the amount of explored basic blocks. This upper bound appears lower in Frontend compiler components than in Backend components. And these last, show higher reachability by Interpreter-Guided Testing.

5 Limitations of Interpreter-Guided Testing

Although we explore all native methods and bytecodes, covering 100% of their interpreter implementation, the observed compiler coverage is lower. This section presents a qualitative analysis to understand the limitations of the approach. For this, we manually inspected the uncovered code and we grouped each instance of uncovered code into five different families, presented in each of the following subsections.

5.1 Compilation specific properties

The Pharo compiler makes specific optimization decisions that are independent of the interpreter. Since a compiler operates at a lower level, it must deal with particular decisions like the following:

Calling convention. The compiler defines a calling convention that maximizes the usage of registers depending on the method's arity. Meanwhile, the interpreter is oblivious to registers and uses a stack.

Frameless methods. The compiler optimizes code generation when a method does not need a method stack frame (see Figure 2). The interpreter, on the other hand, uses a single strategy and therefore, the concolic runner generates a single test case to execute.

Block closures. Compiling in the context of a block closure or a method has different behavior because the code layout differs, while the behavior is the same in the interpreter.

Integer encoding. The interpreter manages integer representations in the same manner. The compiler, on the other hand, covers different types of integer representations that can be encoded differently in machine code. For example, consider the case of shiftable or rotatable immediate values (see Figure 3).

```
1  CogARMCompiler >> rotateable8bitBitwiseImmediate: constant ifTrue: trueAlternativeBlock
       ifFalse: falseAlternativeBlock
2      | value |
3      value := constant.
4      [ ...
5      value = constant] whileTrue:
6        [value := constant < 0
7          ifTrue:[ -1 - constant ]
8          ifFalse:[constant bitInvert32]].
9      ^ falseAlternativeBlock value
```

Figure 3 Code example of rotatable integer encoding uncovered case. The red code is not covered.

```
1  Cogit >> SignExtend16R: reg1 R: reg2
2      | first |
3      reg1 = reg2
4        ifTrue: [first := self LogicalShiftLeftCq: BytesPerWord * 8 - 16 R: reg1]
5        ifFalse: [
6            first := self MoveR: reg1 R: reg2.
7            self LogicalShiftLeftCq: BytesPerWord * 8 - 16 R: reg1 ].
8      self ArithmeticShiftRightCq: BytesPerWord * 8 - 16 R: reg2.
9      ^first
```

Figure 4 Code example of an uncovered case requiring a specific register allocation. The red code is not covered.

These properties hint to us that simple fuzzing of compiler parameters/configurations could improve over the first three cases. Compiler settings are independent of the tested instruction, and a generic input generation based on the interpreter can't reach them. However, to cover the different integer encoders, we need to augment the generated tests with domain (compiler)-specific integer variations.

5.2 Missing integration test cases

One of the main weaknesses of guiding test generation with single interpreter instructions is that generated tests do not cover programs with many instructions. Thus, code-generation decisions that are guided by a sequence of instructions, such as register allocation (see Figure 4) or literal encoding, are not reachable by single instruction programs. Overcoming this limitation requires a different fuzzing approach generating random instruction sequences.

5.3 Runtime interactions

The classes implementing the Pharo JIT compiler contain responsibilities other than compilation and code generation: *e.g.,* runtime interactions,*code garbage collection*, *trampolines*, and *polymorphic inline caches*. Since these features are activated by compiler-specific behavior, their code is inherently unreachable from the interpreter. For example, generated tests are not capable of covering different machine code GC configurations (see Figure 5). This suggests, on the one hand, that refactoring the compiler code to separate these responsibilities could simplify further analyses. On the other hand, testing these features requires separate tests and fuzzing strategies.

```
Cogit >> cogitPostGCAction: gcMode
    (gcMode = GCModeFull and: [ objectRepresentation allYoungObjectsAgeInFullGC ])
        ifTrue: [ methodZone voidYoungReferrersPostTenureAll ].
    gcMode = GCModeBecome ifTrue: [ methodZone followForwardedLiteralsInOpenPICList ].
    ...
```

Figure 5 Code example of Garbage Collection. The red code is not covered.

```
Cogit >> cFramePointerAddress
    ^ (backEnd wantsNearAddressFor: #CFramePointer)
        ifTrue: [ self simulatedReadWriteVariableAddress: #getCFramePointer in: self ]
        ifFalse: [coInterpreter inMemoryCFramePointerAddress]
```

Figure 6 Code example of VM Simulation. The red code is not covered.

5.4 Debugging and simulation

The PharoVM is written in Pharo itself, which allows its simulation as a normal Pharo program. In this mode, the VM supports debugging features that are unavailable in a production environment (see Figure 6). To make this behavior reachable, test execution should be augmented to run in a debugging environment. This raises the question of whether it is meaningful to test the VM's debugger features.

5.5 Unused compiler features

The compiler backend is engineered as a generic compiler instruction supporting a complete assembly-like intermediate language (CogRTL), which is reusable in different scenarios and compiler frontends. However, not all features from this intermediate language are exercised by our evaluated frontend. This is due to the backend's language-independent design, which supports functionalities beyond the front end's requirements. A common example of unused frontend features is the code generation for instructions that are not used by the frontend (see Figure 7).

```
CogARMCompiler >> dispatchConcretize
    ...
    opcode caseOf: {
        "Noops & Pseudo Ops"
        [Label] -> [^self concretizeLabel].
        ...
        [JumpOverflow]    -> [^ self concretizeConditionalJump: VS ].
        [JumpCarry]    -> [^ self concretizeConditionalJump: CS ].
        ...
        [ MovePatcheableC32R ]    -> [ ^ self concretizeMovePatcheableC32R ]}
```

Figure 7 Code example of unreachable instruction. The red code is not covered.

Conclusion. Overall, several compiler features are unreachable using this approach, related to the grey-box nature of the technique, which uses white-box testing from the interpreter's point of view but black-box testing from the compiler's. Indeed, generating tests from an interpreter does not traverse all compiler responsibilities. Moreover, the constraint solver does not have any consideration over the target and generates simple values that do not cover machine code encoding edge cases *e.g.,* integer variations such as rotatable, shiftable, positive, and negative.

6 Concolic Path Mutations

In this section, we investigate how to cover the limitations above using constraint mutations over generated tests, in particular, the integer encoding issues. For instance, let us consider the constraint $int(x)$ *and* $x > 10$. A constraint solver for this constraint typically generates simple values such as $x = 11$. However, let us suppose that a value of $x = 32548$ triggers a potential compiler bug: such a bug is potentially unreachable. The idea behind constraint mutations is to force the constraint solver to generate different values for symbolic variables.

6.1 Methodology

Recall that each generated test is represented as a collection of logic constraints, defining a single conjunction logic constraint. We then apply a set of mutations generating, for each possible mutation match, a new test mutant. Moreover, we introduce two steps before constraint mutations to limit the explosion of generated cases. Firstly, we perform a cleaning step, keeping only satisfiable paths. Secondly, we perform *constraint minimization*. Finally, we test the individual mutants and compare them with the original cases. Each mutation is applied independently of other mutants departing from the baseline of cases commented in the previous section.

6.2 Constraints minimization

Generated logic constraints contain redundant constraints and therefore increase the amount of redundant mutants. We first transform each constraint to its *Conjunctive normal form* (CNF), which allows us to reduce the number of mutation criteria matches. Then, we reduce well-known equivalences to minimize the constraint's size. Finally, knowing that the logic expression is in *CNF*, we reduce the size of these constraints by removing redundant ones under the expression's root, which by definition is an *and* expression.

6.3 Mutations Design

Following our third research question, we focus our mutation design on mutations that preserve the semantics of the interpreter while improving compiler code coverage. The key idea is to segment the set of valid inputs into subsets that are more restrictive than the initial one. This allows us to segment the universe of possible values, forcing the constraint solver to generate diverse values. For example, consider Figure 8 illustrates an input set segmented into four different subsets. From this idea, we designed four different mutation operators:

- **Subset Segmentation:** Mutates each path containing the condition $isInt(x)$ by adding bounds to it. For example, $isInt(x) \wedge x > n$ and $isInt(x) \wedge x \leq n$.

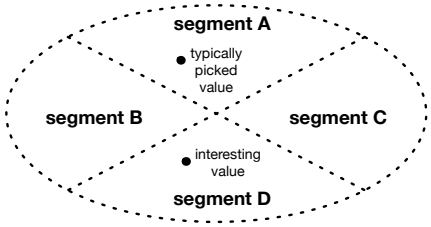

Figure 8 Segmenting inputs in four subsets.

- **Shiftable Integers:** Mutates each path containing the condition $isInt(x)$ by forcing a random value that can be represented as shiftable 16bit integer. For example, $0x00010000$ (65536) and $0x12340000$ (1195376640) can be represented as $0x0001 \ll 16$ and $0x1234 \ll 16$ respectively, both with a left shift magnitude of 16.
- **Commutative:** Mutates each path containing commutative operators by reversing the operands.
- **Randomness:** Mutates each path containing binary operators by adding a same random number to both sides of the operator. For example, form $var(A) \otimes n$ becomes $var(A) + X \otimes n + X$, where X is a random number between the minimum and maximum representable numbers.

6.4 Constraint Mutation Effectiveness

Constraint mutations generate a total of **45233** new test cases. Table 3 presents the reached new nodes obtained from these new test cases.

The first column lists the mutators used. The second column shows the number of mutants generated by each mutator, along with a factor indicating how many times that number multiplies the number of base cases. The third column displays the number of newly covered nodes reached by the mutation. Despite the substantial number of mutants generated, the total number of newly reached nodes remains limited, with a maximum of **4** unique nodes discovered across all mutations.

Segmentation and *Shiftable* produced the highest number of mutants (15,115 and 11,150, respectively), followed by *Commutative* (10,595). These mutations increased the number of generated test cases by factors of **3.27x**, **2.41x**, and **2.29x**, respectively. However, they all reach the same 4 new method nodes, indicating that their effects may be redundant in terms of reachability. Conversely, *BinArithOpRand* generated only 210 mutants (+**0.45x**), and *IsNotIntNotEqRand* produced 8,163 mutants (+**1.76x**), yet they reached the same or fewer new nodes (with *IsNotIntNotEqRand* covering only 2 additional nodes).

Conclusion. The results in Table 3 highlight two key insights. First, there is a need to generate fewer mutations while maintaining the same reachability. The current approach produces a huge number of mutants, but many of them do not contribute to coverage. Second, exhaustive mutation does not scale well, thus a mutation stop criteria based on reachability should be considered. If an operator does not cover new code, further mutations of the same kind should be less frequent. This would help prioritize the most effective mutations, improving both scalability and effectiveness.

Table 3 New reached compiler method nodes using constraint mutations.

Mutation	Generated mutants	New reached nodes
Segmentation	15115 (+3.27x)	4
Shiftable	11150 (+2.41x)	4
Commutative	10595 (+2.29x)	4
BinArithOpRand	210 (+0.45x)	4
IsNotIntNotEqRand	8163 (+1.76x)	2
Total	45233 (+9.8x)	4

7 Discussion

- **Black-box and grey-box mutations.** The mutation results indicate that the proposed mutations do not significantly increase the number of newly covered compiler method nodes. This suggests that a purely black-box approach may be insufficient. To improve coverage, it would be necessary to introduce custom transformations and design targeted mutations that specifically aim at unexplored node paths, following a grey-box approach.
- **Generalization.** Focused on the *A32* and *ARM* code generators, our results may not represent other backends or compilers.
- **Coverage measurement.** Another limitation stems from our coverage measurement, which is currently based on block-level granularity. While this approach provides useful insights, it may avoid other important aspects of execution, such as the number of line-based, control flows or the number of method calls. Expanding the coverage criteria to include alternative metrics would provide a more comprehensive understanding of constraint mutation effectiveness.
  Other testing techniques are required to reach the remaining compiler functionalities, like *Simulation*, *Gargage collection* or *Unused compiler features*.

8 Related Work

As far as we know, there has not been much work done on Compiler testing combining Concolic testing and Differential testing. We already outlined related work on compiler testing in Section 2.1. In this section, we present related work in concolic testing, and in the study of the effectiveness of compiler testing techniques.

8.1 Studying Compiler Fuzzing Effectiveness

Several works evaluate the impact and validity of compiler testing approaches. Most compiler testing techniques use *the number of detected compiler bugs* [4, 19]. In this work, we decided to approach this problem in a white-box fashion, using code coverage. A similar approach is taken by [11] where they generate random valid compiler inputs using grammars.

When comparing compiler testing effectiveness among different techniques, the baseline convention is to use already-known bugs to prove the fuzzing accuracy. *Mettoc* [29] proposes to use *metamorphic relations* to generate equivalent source programs. Similarly, *equivalence module inputs* compares compilers by removing dead code modulo program inputs from source programs [19]. They establish a set of target configurations finding common compiler bugs

using *OpenCL benchmarks*. Another technique compares *Randomized Differential Testing*, *Different Optimization Levels* and *Equivalence Modulo Input* proposing a new measurement: *Correcting Commits* [4]. They perform their experiments using *CSSmith* to generate unknown programs and compare three different techniques. Finally, other work compares submitted bug entries to GCC and LLVM [28].

8.2 General Fuzzing Effectiveness

In this work, we used code coverage as a proxy for fuzzing effectiveness: the insight is that the larger the coverage, the more effective the fuzzer. Several works study in depth the relation between these two properties, although not in compilers.

Böhme et al. show the relation between coverage and bugs mentioning "the best fuzzer at achieving coverage, may not be best at finding bugs" [3]. They study the correlation between coverage and bug finding using *branch coverage*.

Wolff et al. suggest that fuzzer performance evaluation is usually driven by a specific benchmark and proposes a methodology to quantify the impact of benchmark properties on the performance evaluation [32]. They found that the measured performance depends on the properties of the programs used for benchmarking. Also, they explain how specific evaluation setup leads to a superior ranking of fuzzers in practice.

Klooster et al. study how to adapt fuzzing to be fitted in CI/CD pipelines and what is a reasonable fuzzing duration that is compatible with CI/CD pipelines [16]. They explore the effectiveness of fuzzing testing in CI/CD pipelines, suggesting that continuous fuzzing is beneficial for secure software development processes.

8.3 Variations of Concolic Testing

Several limitations have been identified in the past on concolic testing [27]. For example, they list issues managing 64-bit numbers, bitwise operations, and solution generation overhead. Our focus is on understanding its limitations in compiler testing, to improve coverage.

Several solutions propose variants to overcome the limits of concolic testing. Some work proposes the usage of heuristics to reduce false alarms [15]. Other approaches combine concolic testing with *random testing* until saturation with a fixed bound [20]. When random testing saturates, they switch to *Concolic Execution* keeping the *current program state*. Finally, other work propose the use of *Symbolic Backward Execution* [8]. In particular, they propose *Symetric Execution* which demonstrates to be efficient in finding targeted inputs.

9 Conclusion

This paper presents an in-depth analysis of interpreter-guided test generation in the case of the JIT compiler present in the Pharo virtual machine. This analysis is motivated by the importance of compiler testing: indeed, understanding its behavior will help us in the future to better guide it and improve its efficiency. Before this article, little has been said besides the ability of the technique to find bugs.

In this article, we explore the effectiveness of this fuzzer using code coverage. Our analysis shows that interpreter-guided compiler testing is a good approach for generating compiler tests with high reachability in the compiler's backend. However, this technique achieves a seemingly low upper bound in its overall coverage (27%). Manual inspection of uncovered code shows us that uncovered code is related to compilation-specific properties and orthogonal features existing in the compiler infrastructure that are non-existent in the

interpreter. Thus, while the unit-testing nature of the approach allows us to quickly generate tests for the compiler code generation component, it fails to cover the integration with other components. We propose to language implementors to complement this kind of fuzzing technique with separate fuzzers of other components (*e.g.,* garbage collection) and to augment it with random fuzzing mutations.

References

1 Aurèle Barrière, Sandrine Blazy, Olivier Flückiger, David Pichardie, and Jan Vitek. Formally verified speculation and deoptimization in a jit compiler. In *Conference on Principles of Programming Languages (POPL)*, January 2021. doi:10.1145/3434327.

2 Clément Béra, Eliot Miranda, Marcus Denker, and Stéphane Ducasse. Practical validation of bytecode to bytecode jit compiler dynamic deoptimization. *Journal of Object Technology*, 15(2):1:1–26, 2016. doi:10.5381/jot.2016.15.2.a1.

3 Marcel Böhme, László Szekeres, and Jonathan Metzman. On the reliability of coverage-based fuzzer benchmarking. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, pages 1621–1633, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3510003.3510230.

4 Junjie Chen, Wenxiang Hu, Dan Hao, Yingfei Xiong, Hongyu Zhang, Lu Zhang, and Bing Xie. An empirical comparison of compiler testing techniques. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 180–190, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2884781.2884878.

5 Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. A Survey of Compiler Testing. *ACM Computing Surveys*, pages 1–36, May 2020. URL: https://dl.acm.org/doi/10.1145/3363562.

6 Yuting Chen, Ting Su, and Zhendong Su. Deep Differential Testing of JVM Implementations. In *International Conference on Software Engineering (ICSE'19)*, pages 1257–1268. IEEE Press, 2019. doi:10.1109/ICSE.2019.00127.

7 Yuting Chen, Ting Su, Chengnian Sun, Zhendong Su, and Jianjun Zhao. Coverage-directed differential testing of JVM implementations. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 85–99. Association for Computing Machinery, June 2016. doi:10.1145/2908080.2908095.

8 Peter Dinges and Gul Agha. Targeted test input generation using symbolic-concrete backward execution. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 31–36, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2642937.2642951.

9 Karine Even-Mendoza, Arindam Sharma, Alastair F. Donaldson, and Cristian Cadar. Grayc: Greybox fuzzing of compilers and analysers for c. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2023, pages 1219–1231, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3597926.3598130.

10 Olivier Flückiger, Gabriel Scherer, Ming-Ho Yee, Aviral Goel, Amal Ahmed, and Jan Vitek. Correctness of speculative optimizations with dynamic deoptimization. In *Principles of programming languages (POPL'17)*, 2017. doi:10.1145/3158137.

11 Christian Holler, Kim Herzig, and Andreas Zeller. Fuzzing with code fragments. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 445–458, Bellevue, WA, August 2012. USENIX Association. URL: https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/holler.

12 Urs Hölzle, Craig Chambers, and David Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *European Conference on Object-Oriented Programming (ECOOP'91)*, 1991. doi:10.1007/BFb0057013.

13 Sungjae Hwang, Sungho Lee, Jihoon Kim, and Sukyoung Ryu. JUSTGen: Effective Test Generation for Unspecified JNI Behaviors on JVMs. In *International Conference on Software Engineering (ICSE'21)*, pages 1708–1718, 2021. doi:10.1109/ICSE43902.2021.00151.

14 Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. Back to the future: The story of Squeak, a practical Smalltalk written in itself. In *Proceedings of Object-Oriented Programming, Systems, Languages, and Applications conference (OOPSLA'97)*, pages 318–326. ACM Press, November 1997. doi:10.1145/263700.263754.

15 Yunho Kim, Youil Kim, Taeksu Kim, Gunwoo Lee, Yoonkyu Jang, and Moonzoo Kim. Automated unit testing of large industrial embedded software using concolic testing. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 519–528, 2013. doi:10.1109/ASE.2013.6693109.

16 Thijs Klooster, Fatih Turkmen, Gerben Broenink, Ruben Ten Hove, and Marcel Böhme. Continuous fuzzing: A study of the effectiveness and scalability of fuzzing in ci/cd pipelines. In *2023 IEEE/ACM International Workshop on Search-Based and Fuzz Testing (SBFT)*, pages 25–32, May 2023. doi:10.1109/SBFT59156.2023.00015.

17 Bastian Kruck, Stefan Lehmann, Christoph Keßler, Jakob Reschke, Tim Felgentreff, Jens Lincke, and Robert Hirschfeld. Multi-level debugging for interpreter developers. In *Companion to Proceedings of the international conference on Modularity*, pages 91–93. ACM, 2016. doi:10.1145/2892664.2892679.

18 Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. In *Conference on Programming Language Design and Implementation (PLDI'14)*, June 2014. URL: https://dl.acm.org/doi/10.1145/2594291.2594334.

19 Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F. Donaldson. Many-core compiler fuzzing. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 65–76, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2737924.2737986.

20 Rupak Majumdar and Koushik Sen. Hybrid concolic testing. In *29th International Conference on Software Engineering (ICSE'07)*, pages 416–426, 2007. doi:10.1109/ICSE.2007.41.

21 William M McKeeman. Differential Testing for Software. *DIGITAL TECHNICAL JOURNAL*, 1998.

22 Eliot Miranda. The cog smalltalk virtual machine. In *Proceedings of VMIL 2011*, 2011.

23 Eliot Miranda, Clément Béra, Elisa Gonzalez Boix, and Dan Ingalls. Two decades of Smalltalk VM development: live VM development through simulation tools. In *Proceedings of International Workshop on Virtual Machines and Intermediate Languages (VMIL'18)*, pages 57–66. ACM, 2018. doi:10.1145/3281287.3281295.

24 Jihyeok Park, Seungmin An, Dongjun Youn, Gyeongwon Kim, and Sukyoung Ryu. JEST: N+1-Version Differential Testing of Both JavaScript Engines and Specification. In *International Conference on Software Engineering (ICSE'21)*, pages 13–24. IEEE Press, 2021. doi:10.1109/ICSE43902.2021.00015.

25 Guillermo Polito, Pablo Tesone, and Stéphane Ducasse. Interpreter-guided Differential JIT Compiler Unit Testing. In *Programming Language Design and Implementation (PLDI'22)*, 2022.

26 Paul Purdom. A sentence generator for testing parsers. *BIT Numerical Mathematics*, 12(3):366–375, 1972. doi:10.1007/BF01932308.

27 Xiao Qu and Brian Robinson. A case study of concolic testing tools and their limitations. In *2011 International Symposium on Empirical Software Engineering and Measurement*, pages 117–126, 2011. doi:10.1109/ESEM.2011.20.

28 Chengnian Sun, Vu Le, Qirun Zhang, and Zhendong Su. Toward understanding compiler bugs in gcc and llvm. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, pages 294–305, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2931037.2931074.

29 Qiuming Tao, Wei Wu, Chen Zhao, and Wuwei Shen. An automatic testing approach for compiler based on metamorphic testing technique. In *2010 Asia Pacific Software Engineering Conference*, pages 270–279, 2010. doi:10.1109/APSEC.2010.39.

30 David Ungar, Adam Spitz, and Alex Ausch. Constructing a metacircular Virtual machine in an exploratory programming environment. In *Companion to Object-Oriented Programming, Systems, Languages, and Applications conference (OOPSLA'05)*, pages 11–20, New York, NY, USA, 2005. ACM. doi:10.1145/1094855.1094865.

31 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 Transaction Architecture Code Optimization*, 9(4), January 2013. doi:10.1145/2400682.2400689.

32 Dylan Wolff, Marcel Böhme, and Abhik Roychoudhury. Explainable fuzzer evaluation, 2022. doi:10.48550/arXiv.2212.09519.

33 Thomas Würthinger, Michael L. Van De Vanter, and Doug Simon. Multi-level virtual machine debugging using the java platform debugger architecture. In *Perspectives of Systems Informatics*, pages 401–412. Springer, 2010. doi:10.1007/978-3-642-11486-1_34.

34 Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and Understanding Bugs in C Compilers. In *Programming Language Design and Implementation*, PLDI '11, 2011. doi:10.1145/1993498.1993532.

35 Guixin Ye, Zhanyong Tang, Shin Hwei Tan, Songfang Huang, Dingyi Fang, Xiaoyang Sun, Lizhong Bian, Haibo Wang, and Zheng Wang. Automated conformance testing for javascript engines via deep compiler fuzzing. In *Proceedings of Conference on Programming Language Design and Implementation (PLDI'21)*, pages 435–450, New York, NY, USA, 2021. ACM. doi:10.1145/3453483.3454054.