HAL
open science

# Interpreter-guided Differential JIT Compiler Unit Testing

Guillermo Polito, Pablo Tesone, Stéphane Ducasse

## To cite this version:

Guillermo Polito, Pablo Tesone, Stéphane Ducasse. Interpreter-guided Differential JIT Compiler Unit Testing. Programming Language Design and Implementation - PLDI 2022, Jun 2022, San Diego, United States. 10.1145/3519939.3523457 . hal-03607939

**HAL Id: hal-03607939**

**https://hal.science/hal-03607939v1**

Submitted on 18 Mar 2022

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

# Interpreter-guided Differential JIT Compiler Unit Testing

Guillermo Polito
Univ. Lille, CNRS, Inria, Centrale Lille,
UMR 9189 CRIStAL, F-59000 Lille, France
guillermo.polito@univ-lille.fr

Pablo Tesone
Pharo Consortium
Univ. Lille, Inria, CNRS, Centrale Lille,
UMR 9189 CRIStAL
pablo.tesone@inria.fr

Stéphane Ducasse
Univ. Lille, Inria, CNRS, Centrale Lille,
UMR 9189 CRIStAL
stephane.ducasse@inria.fr

## Abstract

Modern language implementations using Virtual Machines feature diverse execution engines such as byte-code interpreters and machine-code dynamic translators, a.k.a. JIT compilers. Validating such engines requires not only validating each in isolation, but also that they are functionally equivalent. Tests should be duplicated for each execution engine exercising the same execution paths on each of them.

In this paper we present a novel automated testing approach for virtual machines featuring byte-code interpreters. Our solution uses concolic meta-interpretation: it applies concolic testing to a byte-code interpreter to explore all possible execution interpreter paths and obtain a list of concrete values that explore such paths. We then use such values to apply differential testing on the VM interpreter and JIT compiler. This solution is based on two insights: (1) both the interpreter and compiler implement the same language semantics and (2) interpreters are simple executable specifications of those semantics and thus promising targets to (meta-) interpretation using concolic testing. We validated it on 4 different compilers of the open-source Pharo Virtual Machine and found 468 differences between them, produced by 91 different causes, organized in 6 different categories.

***Keywords:*** virtual machine, concolic testing, JIT compilers, interpreters

## 1 Introduction

Modern Virtual Machines support code generation for JIT compilation and dynamic code patching for techniques such as inline caching. They are often structured around a byte-code interpreter, a baseline JIT compiler, and a speculative inliner. This complexity is aggravated when the VM builds and runs on multiple target architectures [1]. Validating the execution of interpreted code and its compiled counterpart is challenging.

Several solutions have been proposed to aid in VM testing tasks. Traditionally, VM simulation environments have appeared in Self [29], Smalltalk [14, 22] and Metacircular VMs such as Maxine [30]. Complementary to simulation environments, multi-level debuggers [16, 31] aid VM developers to switch views between the program-level and the implementation (VM)-level. These solutions are indeed beneficial to identify and track problems once an issue has been spotted and reproduced. However, reproducing bugs still remains an expensive and time-consuming task because millions of instructions may need to be executed before hitting the actual problem. For example, it has been reported that debugging memory corruption bugs in a simulation could take several hours of execution[1]. Recently, the team of Maxine reported a test-based infrastructure for cross-ISA debugging [15]. They reported that most debugging happens in gdb when bootstrapping a new architecture, in a different abstraction level than the original source code, and that they were not able to cover many parts of their codebase. Béra *et al.,* [3] and Flückiger *et al.,* [8] focus on the validation interaction between speculative compiler transformations and deoptimization.

Moreover, Virtual Machines often include several execution engines with different trade-offs: it is common for example to mix byte-code interpretation for *cold* code, with Just in Time compilers that optimize *hot* code. Since these different components are meant to be semantically equivalent, test scenarios need to be duplicated too for each of them.

In this paper we propose to guide the automatic unit testing of a JIT compiler by the interpreter definition. Our technique is based on two insights. First, we consider interpreters executable specifications of the programming language semantics and thus we propose to use them to *automatically generate test inputs*. Second, since both the interpreter and compilers for a language should implement the same language semantics, we apply *differential testing* on them [19], comparing their behavior as test oracles.

We first apply concolic testing on the interpreter to (1) discover all possible execution paths and (2) produce an abstract description of the input values. We then generate the compiled code for each case and exercise it each with values equivalent to those in the interpreter. Our path exploration differs from traditional concolic testing in that it does not stop as soon as it finds a concrete error. Instead, it tracks for each execution path an exit condition indicating how the instruction finished (*e.g.,* success, failure), or if it exited

[1] http://forum.world.st/OpenSmalltalk-opensmalltalk-vm-Reproduceable-Segmentation-fault-while-saving-images-444-td5106898i20.html

*PL'18, January 01–03, 2018, New York, NY, USA*
2018.

the main interpreter for some runtime service (*e.g.,* message send or method return). Tracking the exit condition allows us to test that the compiled code has the same observable behavior than the interpreted code.

We applied it to the byte-code interpreter and four different compilers of the Pharo Virtual Machine: both the native method template-based compiler and the stack-to-register byte-code compiler considered stable and in production since more than 10 years, plus two non-productive compilers. Our approach generated in less than 10 minutes more than 4.5K tests, and found 468 differences from 91 different causes.

The contributions of the article are:

- we show that interpreters are a valid resource when generating JIT compiler test inputs, unveiling many differences between them;
- we show that our approach finds differences ranging from clear bugs producing segmentation faults, to optimisation and behavioural differences;
- we are, to the best of our knowledge, the first to use concolic testing in the domain of Virtual Machines and JIT compilers;
- we are, to the best of our knowledge, the first to combine concolic testing with differential testing to test compilers;
- we show that the approach is practical and applicable on-line.

Section 2 presents the problems and the solution for validating differences between interpreter and compiler. Section 3 describes the execution model that we designed to be able to capture the test domains (stack, operand, or format shapes). Section 4 describes the setup of our experience, while Section 5 describes the results obtained. Further sections present related work and conclude.

## 2 Interpreter-Guided differential VM Testing

### 2.1 Problem: Testing Duplicated Semantics in Virtual Machines

Modern virtual machines include several execution engines with different trade-offs. It is common to have simple yet slow byte-code interpreters to execute code that is rarely found at run time, and one or more dynamic translation tiers that translate *hot* code to machine *Just in time* (JIT). Such a schema is designed to achieve a good balance between the time spent executing useful work and the time spent compiling to machine-code. A key challenge is to validate the correctness of these components that generally have few code in common and present very different architectures. For example, while a byte-code interpreter executes directly the byte-code, each compilation tier uses a different intermediate representation design.

Let us illustrate this difference with the interpreter implementation of the addition byte-code in the Pharo Virtual Machine shown in Listing 1. The addition byte-code is implemented in Pharo's interpreter using static type predictions, inlining the common case for integer arithmetics, and defaulting to user-defined methods if not suitable [4, 11]. Such an instruction pops two elements from the operand stack, checks if they are both small integers and, if they are, it adds them up. If the result does not overflow, arguments are popped, the result is pushed to the operand stack, and execution continues with the next byte-code. If none of the conditions above hold, the instruction takes a slow path and performs a normal message send.

```
Interpreter >> bytecodePrimAdd
  | rcvr arg result |
  rcvr := self internalStackValue: 1.
  arg := self internalStackValue: 0.
  (objectMemory areIntegers: rcvr and: arg) ifTrue: [
    result := (objectMemory integerValueOf: rcvr) + (
      objectMemory integerValueOf: arg).
    "Check for overflow"
    (objectMemory isIntegerValue: result) ifTrue: [
      self
        internalPop: 2
        thenPush: (objectMemory integerObjectOf: result).
      ^ self fetchNextBytecode "success"]].
  "Slow path, message send"
  self normalSend
```

**Listing 1.** Excerpt of the byte-code interpretation implementing addition in the Pharo Virtual Machine.

At the same time, when the Pharo VM first-tier JIT compiler parses that byte-code it generates the sequence of intermediate representation (IR) instructions illustrated in Listing 5. Although those IR instructions are meant to be compiled to machine code, they conceptually represent the same behavior as in the interpreter: they perform type checks on the arguments and result, and fall back to a slower message send if any of those conditions do not hold.

```
  ... # previous bytecode IR
        checkSmallInteger t0
        jumpzero notsmi
        checkSmallInteger t1
        jumpzero notsmi
        t2 := t0 + t1
        jumpIfNotOverflow continue
notsmi: #slow case first send
        t2 := send #+ t0 t1
continue:
  ... # following bytecode IR
```

**Listing 2.** Illustration of the Intermediate Representation instructions created when compiling the byte-code instruction in Listing 1.

It is indeed possible to manually write tests for these components, but such manual specification is labor intensive and

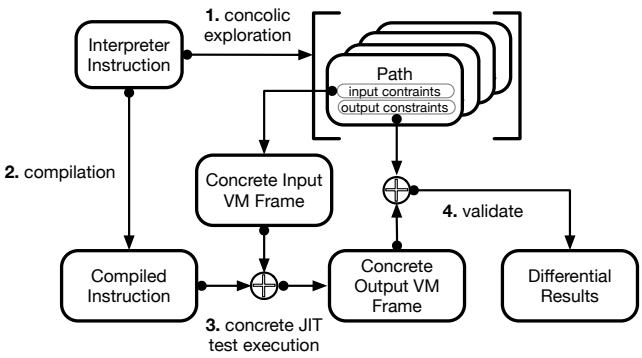


**Figure 1. Solution Overview.** We use a concolic exploration on the interpreter to obtain the different execution paths to test on the JIT compiled code.

error prone. However, since these different components are meant to be semantically equivalent, test scenarios need to be duplicated too for each of them. Instead, we propose to use the VM interpreter to drive automatic VM testing.

### 2.2 Interpreter-Guided VM Testing

In this paper we use *interpreter-guided differential testing between interpreter and JIT compiler* to discover differences between them. Our approach is interpreter-guided in the sense that we use the interpreter implementation to direct automatic test case generation for the JIT compiler. This technique is based on the following insights. First, we consider Virtual Machine interpreters executable specifications of a language semantics [9]. Second, interpreters often implement complete semantics since they are the execution engine that a VM falls back into when the more complex tiers do not support a feature. Finally, since both interpreter and compilers should implement the same language semantics, comparing their behavior is useful as test oracles.

Figure 1 illustrates our solution for one VM instruction. We first (step 1) perform a concolic execution [10, 27] of the interpreter instruction to discover the input values that exercise all of its execution paths. During our concolic execution, we record *input and output constraints* on both the VM state before and after the instruction. We then compile the instruction with the JIT compiler (step 2), use the input constraints to build concrete VM stack frames and execute the compiled code on it (step 3). Finally, we use the output constraints to validate that the compiled code had the same observable side effects on the stack frame (step 4).

### 2.3 Interpreter Concolic Testing

Concolic testing [10, 27] is an automated testing technique that combines **conc**rete and symb**olic** execution of a program to generate input values that explore all of the program's possible execution paths. In a nutshell, concolic testing executes the program under test many times, each time with different concrete values. Each execution is instrumented to produce symbolic constraints and track all control flow conditions in so called *path conditions*. When the execution finishes, it negates the last path condition not already negated to explore a new path, and its constraints are fed to an automatic constraint solver. The values obtained from the automatic constraint solver are the inputs for the next iteration. The search finishes when all possible paths are explored.

In our example of bytecodePrimAdd above, applying concolic testing yields the results shown in Table 1. The first time, our VM concolic tester will execute the instruction push integers as arguments, and record that the execution checked that both are integers, and their sum is in range too. It then negates the last condition, and the constraint solver generates two integers that summed up generate an overflow. The code is re-executed with those values and no new constraints are found. It then negates the previous non-negated condition and generates one integer argument and one non-integer argument. It continues in such a way until all paths are exercised.

### 2.4 Interpreter-Compiler Differential Testing

After compiling the code corresponding to an interpreter instruction using the JIT compiler, we perform a concrete execution using compiled code and compare its result with the interpreter result. The concrete execution requires to set up a concrete VM stack frame, created from the constraints recorded on the interpreter input frame.

Our approach does not require that interpreter and compiler have stack frames with the same shape. This is the case of our guiding example, where our interpreter implements a stack-based machine while our compiler is a register-based machine. In this case, it is the differential tester that interprets the input frame constraints and sets up a VM frame that suits the compiler structure and calling convention *e.g.*, arguments should be pushed to the stack in the interpreter, while they need to be put in registers in the compiled version.

## 3 Execution Model

In this section we present our concolic execution model extended with Virtual Machine semantics.

### 3.1 Vocabulary: Byte-code and Native Methods

For clarity of the presentation, this subsection presents some vocabulary points. Our solution works with the compilation of two kind of instructions:

**Byte-code instructions.** Instructions used as a VM intermediate language. The programming language source code is compiled to a sequence of byte-code instructions *e.g.*, push instance variable, duplicate the top of the stack. Byte-code instructions in our implementation are by design unsafe for performance reasons. For

| Argument 0 (type) | Argument 1(type) | Path |
|---|---|---|
| 0 (integer) | 0 (integer) | isInteger(arg0), isInteger(arg1), isInteger(arg0+arg1) |
| 0xFFFFFFFF (integer) | 1 (integer) | isInteger(arg0), isInteger(arg1), isNotInteger(arg0+arg1) |
| 0 (integer) | object1 (object) | isInteger(arg0), isNotInteger(arg1) |
| object1 (object) | 0 (integer) | isNotInteger(arg0), isInteger(arg1) |
| object1 (object) | object2 (object) | isNotInteger(arg0), isNotInteger(arg1) |

**Table 1.** Example of concolic execution paths obtained on the byte-code instruction in Listing 1. Each line in the table shows the concrete values fed as arguments, and the constraint path obtained for that exploration case.

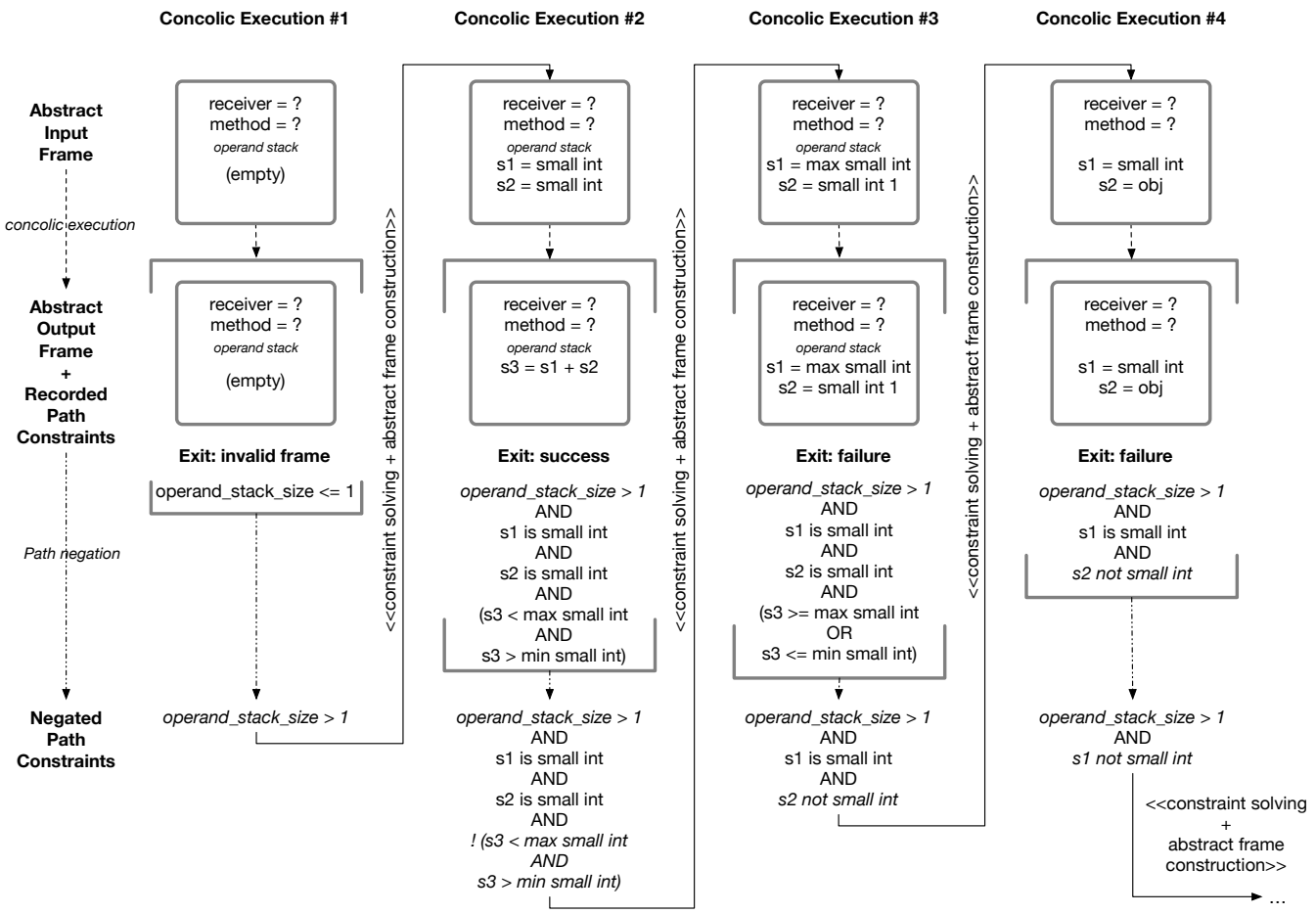


**Figure 2. Example of constraint tracking on the add byte-code.** Each column illustrates one concolic path execution. Each concolic execution starts with an abstract input frame, and produces an abstract output frame, an exit condition, and the recorded constraint path. Already negated conditions are in italics. The last not-already-negated constraint path is negated and leads to the next concolic path execution.

example, a pop instruction does not validate the number of elements in the operand stack, assuming that values were pushed to the stack before its execution.

**Native Methods.** Primitive operations exposed by the Virtual Machine as methods. Native methods are used implement basic functionality and natively optimised versions of some functionality *e.g.,* computing trigonometric functions, object allocation. Native methods in our implementation are by design safe. They check the types and shapes of all their operands and fail with a failure code in case an operand is incorrect.

From the point of view of this paper, we will consider both byte-code and native methods as VM *instructions*. Some functionality in our VM implementation is provided as both byte-code instructions *and* native methods, duplicated for performance reasons *e.g.,* integer addition. Moreover, native

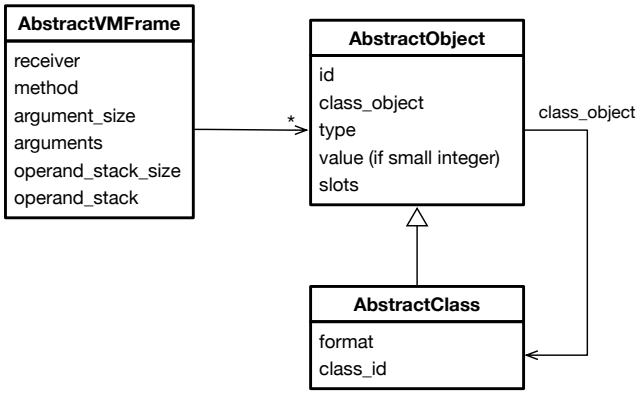


**Figure 3. Constraint model.** Constraint variables are grouped in abstract frames, objects and classes. Abstract objects model concrete objects and are interpreted to build concrete objects.

methods tend to be larger and more complex, as we present in Section 5.4.

### 3.2 Abstract Objects and Abstract Frames

Our constraint model represents path constraints together with VM object constraint, as depicted in Figure 3. We use VM object constraints to model dynamic VM data structures and keep a link to the flat constraint variables fed to the constraint solver. In this sense, re-creating a VM input implies interpreting the results of the constraint solver using the structural information in the VM object constraints. The core of our contraint model includes the abstract modeling of:

- VM stack frames with receiver object, method, locals, operand stack;
- VM objects with their class, memory format, slots referencing other objects;
- VM classes with their class table id.

One key aspect of our solution is that we store copies of both the input and output constraints created during the concolic execution. Input constraints serve to re-create a concrete input frame to execute the compiled code. We use the stored output constraints to perform the differential validation and check that the compiled code has the same observable behavior as the interpreted code. Moreover, recording both input and output constraints for each concolic path execution requires storing a copy of each of them because VM instructions have side effects: if an instruction pushes a value to the operand stack or modifies a variable that is local to the frame, such modification must not alter the input frame. Figure 2 illustrates this with our guiding example *i.e.,* integer addition. Each column in the figure illustrates one concolic path execution, for which it depicts: input frame, output frame, exit condition (*e.g.,* whether the instruction finished with error or success), recorded constraint path and finally the negated constraint path that leads to the next concolic path execution.

### 3.3 Virtual Machine Constraints

The VM does ultimately treat objects as raw unstructured data through pointer arithmetics. This means that recording constraints at that low-level of abstraction would complicate object re-creation and misses semantic information that is important for condition negation during the concolic execution. For example, the VM concrete execution checks if a value is not an integer by checking if it is not a tagged value, which we could represent with a constraint (v && 1) == 1. However, negating that constraint obtains the constraint (v && 1) != 1, which does not correctly represent tagged integers: tagged integers need also to be within certain bounds.

To deal with this issue, our execution model models the VM semantics instead of the concrete memory manipulations done by the VM. In our example above, we record if an object is a tagged small integer or not using semantic conditions such as isSmallInteger(v) and isNotSmallInteger(v). Our semantic conditions includes conditions such as *class index of*, *integer to float conversions*, *object-to-native data conversions*, and so on.

Such a decoupling does help when negating conditions during the concolic execution, but also allows conditions to be address independent. This decoupling also makes our solution work on constraint solvers that do not support bitwise manipulations used not only for pointer tagging but also to extract object header meta-data.

### 3.4 Instruction Exit Conditions

Alongside the input and output constraints and the constraint paths, our concolic execution model tracks also the *exit status* of an instruction, as shown in Figure 2. An instruction exit status models how the instruction execution finished, allowing us to validate the behavioral equivalence between the interpreted and compiled versions. Moreover, it allows us to detect certain conditions from which the execution leaves the main interpreter to execute slower execution paths. Our concolic execution model tracks the following exit conditions:

**Success.** Represents the correct execution of an instruction until its end. It is the main exit condition of byte-code instructions manipulating the operand stack (*e.g.,* push, pop, dup) and of native methods. Successful executions should execute until the end in compiled byte-code, or return to the caller in compiled native methods.

**Failure.** Represents the invalid attempt to execute a native method. As specified above, native methods in our model are safe: they check their operands' types and shape and fail if they are not as expected. Failing executions do fall-back to user defined code instead of returning to the caller.

**Message Send.** Represents the attempt to activate a message-send. Several byte-code instructions perform method activations, either in their main execution path or as a slow path for optimized byte-code instructions. Message send executions should perform a call to a trampoline or to a method linked through mono-, poly- or mega-morphic inline caches [12].

**Method Return.** Represents the attempt to return to the caller. Returned executions should return to the caller both in interpreted and compiled methods.

**Invalid Frame.** Represents the attempt to access a non-existing value in the stack frame. Our concolic execution generates VM frames based on constraints, and thus it does only generate values in a frame if a constraint required so. An invalid frame exit indicates our concolic execution engine that subsequent executions need extra elements in the stack. We consider invalid frame exits as expected failures in our test runner.

**Invalid Memory Access.** Represents the attempt to perform an out-of-bounds access on an object. Our concolic execution validates that object accesses are within bounds, and fails if not. An invalid memory access indicates our concolic execution engine that subsequent executions need more slots in an object. In our test runner we consider invalid memory access executions as expected failures for byte-code instructions because they are unsafe by design. However, we consider them as errors for native methods because native methods are supposed to validate accesses and fail instead of performing the invalid reads/writes.

Although not yet covered by our implementation, our model remains extensible with new exit conditions such as activating a garbage collection.

# 4 Realization

## 4.1 Experimentation Platform: Pharo VM

Our experimentation platform is the Pharo Virtual Machine. The Pharo Virtual Machine is an industrial level Virtual Machine written in Pharo itself and transpiled to C using a VM-specific translator called Slang [14]. The VM implements at the core of its execution engine a threaded byte-code interpreter, a linear non-optimising JIT compiler named Cogit [21] that includes polymorphic inline caches [12] and a generational scavenger garbage collector that uses a copy collector for young objects and a mark-compact collector for older objects [28]. The following numbers illustrate the complexity of this Virtual Machine:

- It implements 255 byte-codes, organized in a total of 77 different families [2].
- It implements about 340 native methods, several duplicated in both the interpreter and in the JIT compiler.

***Experimental Compilers.*** The Pharo VM JIT compiler we test in our evaluation has different backends in charge of translating IR to Machine-Code specialized per target ISA, and three different front-ends: three byte-code compiler frontends and a native method compiler frontend. The byte-code compilers parse byte-code into IR through abstract interpretation. The byte-code compiler used in production (StackToRegisterCogit) performs a stack-to-register mapping using a parse-time stack, to avoid unnecessary stack accesses in the generated machine-code. We included in our evaluation two additional byte-code compilers not used in production: (a) the SimpleStackBasedCogit is a simpler version of the compiler that maps push and pop byte-code instructions to their equivalent push and pop machine-code instructions, and (b) the experimental RegisterAllocatingCogit extends the StackToRegisterCogit with a linear register allocator. Finally, native methods implementing primitive operations are translated to IR using a hand-written template-based approach.

***Pharo Testing Infrastructure.*** The Pharo VM presents a high-level simulation environment that is very handy to simulate full executions including JITt'ed code and live-program the VM [22], illustrated in Figure 4. The simulation environment is extended with a testing infrastructure [25] that allows VM developers to write fine-grained testing scenarios, making tests small, fast, reproducible, and cross-ISA.

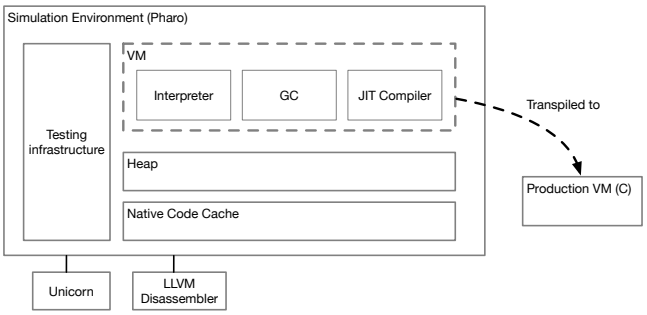


**Figure 4. Development environment of the Pharo VM.** The VM is executed as Pharo code in the simulation environment and transpiled to C to produce the production artefact. The testing infrastructure extends the simulation environment.

## 4.2 Compiling Instructions

The interpreted and compiled code present many semantic differences that our differential tester needs to deal with. First, interpreted byte-code is stack-based, while compiled code is mainly register-based. Second, the granularity of compiled code is the method, meaning that single instructions cannot be compiled in isolation. Last, but not least, our three experimental compilers generate code that has different behavior and expectations.

***Compiling byte-code instructions.*** Pharo methods are compiled to stack-based byte-code instructions. The byte-code JIT compiler performs an abstract interpretation on the byte-code to generate the corresponding IR, and later perform code generation. However, as explained above in Section 4.1, our byte-code compilers behave differently regarding optimisation of stack access. Indeed the StackToRegisterCogit will use a parse-time stack to simulate pushes, and only generate stack accesses if a corresponding pop instruction consumes the operands in the stack. As a corollary, when testing the compilation of instructions pushing values to the stack, we must ensure that those instructions are followed by instructions consuming those stack values.

We solved the challenges above by implementing the following compilation schema. First, when testing a byte-code instruction, our compilation unit is a method. The method will have as many arguments or locals as required by the instruction (*e.g.,* to support a pushLocalVariable byte-code) and the compiler will ensure the method has a correct preamble. Second, we prepend in the method IR instructions that push literals to guarantee the shape of the operand stack.

```
ConcolicBytecodeTester >> compileBytecode: bytecode
  ^ self compile: [ | descriptor result |

    "Instantiate method and compile code
    pushing elements to the operand stack"
    cogit methodObj: self instantiateMethod.
    solution inputConstraints
      operandStack
      reversed
      do: [ :aStackValue |
        cogit genPushLiteral: (self instantiate: aStackValue)
      ].

    "Generate the instruction IR"
    descriptor := cogit generatorAt: bytecode.
    result := cogit perform: descriptor generator.

    "Generate a return instruction if necessary,
    returning the result of the instruction"
    solution exitCondition returnResultInto: self ]
```

**Listing 3.** Intermediate Representation created when compiling the byte-code instruction in Listing 1

***Compiling native method instructions.*** Native methods implement non-inlined versions of arithmetics, object and array accesses, reflection, and so on. The main difference between a native method and a byte-code instruction is that they are late-bound: what method is to be executed is defined from the type of the receiver object. This schema allows one to exploit polymorphism and operator redefinitions.

Pharo native methods are hybrid, they have a native behavior component and a byte-code component. On the one hand, native methods are interpreted by calling a VM function that manipulates the operand stack and returns to the caller if successful. If the native behavior fails, interpretation continues with byte-code execution of the user redefined method. On the other hand, JIT compiled native methods are a linear version of the above, where the machine-code method starts with a machine-code version of the native behavior, and falls through a compiled version of the byte-code in case of failure.

```
ConcolicNativeMethodTester >> compileNativeMethod:
    nativeMethodId
  ^ self compile: [ | descriptor result |

    "Generate the instruction IR"
    descriptor := cogit generatorAt: nativeMethodId.
    compilationResult := cogit
      objectRepresentation
      perform: generator.

    "Generate a break instruction to detect fall-through
      cases"
    cogit Stop ]
```

**Listing 4.** Intermediate Representation created when compiling the byte-code instruction in Listing 1

Second, we made our differential tester to only compile the native behavior and introduce a breakpoint/stop instruction after the last instruction to detect fall-through cases. Then, for each test path, we assert that the compiled code returns to the caller if we had detected no error condition, or to hit the breakpoint instruction otherwise.

### 4.3 Current Prototype Limitations

Our concolic tester prototype does not currently support several features of the Pharo interpreter, namely stack-frame reifications and byte-code look-aheads. Moreover, the constraint solver we are using for the concolic exploration limits integers to 56 bit precision and does not support bit-wise operations. None of these limitations are essential to the approach, but they rather require additional engineering effort.

***Stack-frame Reifications.*** Stack-frame reifications are implemented in the Pharo interpreter to support reflection on stack-frames, and the implementation of exceptions as a library rather than a language feature. The current implementation of stack-frame reifications implements lazy context-to-stack mapping [20, 21]: context objects are heap allocated only on demand and work as proxies to a stack-frame for some of their life time. This means that the interpreter handles such mapping in several execution paths, particularly during explicit stack reification (*i.e.,* the pushThisContext byte-code instruction) and during instance variable access instructions that handle proxified accesses to the stack.

***Byte-code look-aheads.*** Several byte-code instructions in our interpreter perform a look-ahead to avoid unnecessary instruction dispatches. This is the case for example of comparison instructions, that generate a boolean value by default, but skip modifying the operand stack and advances two instructions at a time if the following instruction is a branch instruction. Our implementation partially supports tracking constraints on the current method's byte-code, allowing one to generate byte-code sequences for the concrete execution, but requires extending it to symbolically look-ahead byte-code in the instruction stream.

***Constraint Solver Limitations.*** Doing a concolic execution of the VM requires a constraint solver that handles numbers with the same amount of precision as the VM. Indeed, the VM performs integer bound checks to apply safe arithmetics, detect potential overflow cases and the internal change integer representation. The constraint solver we are using supports (as for the time of writing this article) 56bit large numbers, and thus we constrained our usage for now to 32bit compilations. Other limitations of the constraint solver include the absence of bit-wise operations, which we overcome so far by abstracting the constraint model from the exact memory representation of objects (cf. Section 3.3).

# 5 Evaluation

In this section we evaluate our solution by the means of automatically testing the Pharo Virtual Machine JIT compiler. We first explain our evaluation methodology, and then follows a two-fold evaluation. On the one hand, we present and analyze our results applying interpreter-guided testing to a large set of the Virtual Machine byte-code and native method instructions. On the other hand, we evaluate the practicality of the approach by presenting time measurements of the concolic exploration and test execution time.

## 5.1 Evaluation Methodology

We evaluate our testing approach by applying it to the Pharo Virtual Machine JIT compiler. Our evaluation consists in four main experiments: (1) testing the IR-template compiler of native methods and (2-4) testing the three different byte-code to machine-code compilers, explained before in Section 4.1. Each test-case scenario found by the concolic exploration is executed on the JIT compiler using two different architectures: x86 and ARM32(v5-v7).

Time measurements for the second part of the evaluation were taken from machine with the following specs: 2015 MacBook Pro, 2,9 Ghz Intel Core i5, 16GB 1867 MHz DDR3. We report as time measurements averages with dispersion and totals for running all our tests. We did not run a thorough performance evaluation because we consider that the measurements are low-enough for practical usage, and performance is not at the core of this paper.

## 5.2 Results

Table 2 reports our results on top of the Pharo VM JIT compilers. Each line in the table reports the results for each of our compilers (column 1). The table shows the number of tested instructions (column 2), and how many execution paths were discovered by applying concolic testing on them (column 3). We semi-automatically curated the list of explored paths keeping only those paths that do work in our prototype implementation (column 4). The paths we removed paths are those not handled because of limitations of our prototype implementation: they either make our concolic execution to fail, they produce errors on the constraint solver, or they require special initializations on the JIT compiler we have not implemented in our testing infrastructure. Finally, we report how many of those paths present differences between interpreter and the JIT compilation, and the percentage they represent.

## 5.3 Analysis of Results

In this subsection we analyze the 468 path differences to identify its causing *defect*. We performed *defect* identification by manually inspecting and debugging the source code of the interpreter and the tested compiler. Because many paths do fail because of a same defect, we count a defect only once regardless of how many execution paths it lead to a failure.

Our testing approach detects six different categories of defects. We interpreted some differences as being caused by bugs in the interpreter or compiler (*e.g.,* respective missing type-checks), while others are arguably correct in both and thus we interpreted them in a more neutral manner (*e.g.,* behavioral differences). Table 3 summarizes all the defects causing the differences we found.

Most of the bugs found are in the byte-code front-end, and thus failed in both back-ends: ARM32 and x86. The first thing to notice is that native methods present more execution paths than byte-code instructions, shown in Figure 5, explaining why much more differences are found in native methods. Indeed, byte-code instructions present in average few more than 2 paths, while native method instructions approach 10 paths in average.

***Missing interpreter type check.*** Type checks are missing in the interpreter, allowing some paths to execute on wrong conditions. Missing type checks, regardless of being in the interpreter or compiled code, produce unpredictable results and ultimately crashes at run time. This is the case of the `primitiveAsFloat` native method shown below. This native method checks the receiver type using an assertion that is removed at compile-time instead of explicitly failing the method execution. If the receiver is a pointer, its value will be coerced as an integer through pointer untagging, and then coerced to a double precision float, producing random numbers.

| Compiler | # Tested Instructions | # Interpreter Paths | # Curated Paths | # Differences (%) |
|---|---|---|---|---|
| Native Methods (primitives) | 112 | 2024 | 1520 | 440 (28,95%) |
| Simple Stack BC Compiler | 175 | 1308 | 1136 | 18 (1,59%) |
| Stack-to-Register BC Compiler | 175 | 1308 | 1136 | 10 (0,88%) |
| Linear-Scan Allocator BC Compiler | 175 | 1308 | 1136 | 10 (0,88%) |
| **Total** | 462 | 4640 | 4582 | 468 (32,29%) |

**Table 2. Results running our approach on four different compilers.** Number tested instructions indicate the explored interpreter instructions. Interpreter paths indicate the number of paths founds during the concolic execution. Curated paths indicate the number of paths supported by our implementation. Differences indicate how many of those paths differ between interpreter and compiler.

| Family | # Cases |
|---|---|
| Missing interpreter type check | 1 |
| Missing compiled type check | 13 |
| Optimisation difference | 10 |
| Behavioral difference | 5 |
| Missing Functionality | 60 |
| Simulation Error | 2 |

**Table 3. Summary of found defects.** Interpreter-guided testing finds differences including behavioural differences, missing type-checks leading to runtime errors and different optimized paths.

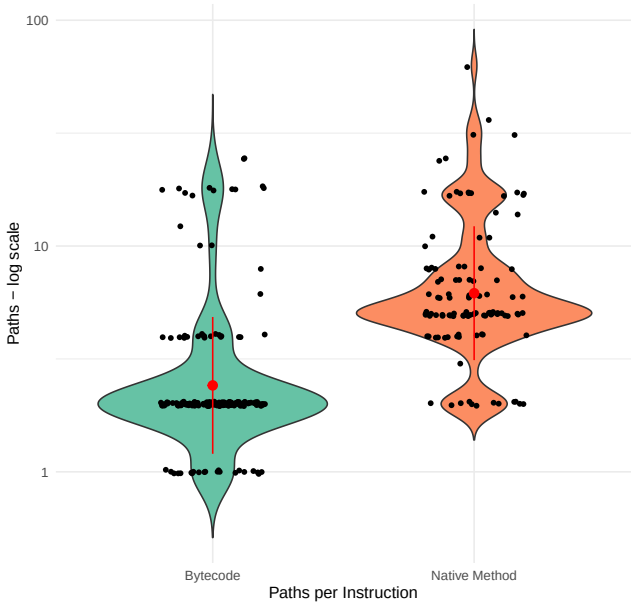


**Figure 5. Paths per instruction.** Byte-code instructions present in average few more than 2 paths, while native method instructions approach 10 paths in average.

```
primitiveAsFloat
  | rcvr |
  rcvr := self stackTop.
  self assert: (objectMemory isIntegerObject: rcvr).
  self pop: 1 thenPushFloat: (objectMemory integerValueOf:
      rcvr) asFloat
```

**Listing 5. Missing type-check in the interpreter primitiveAsFloat**

***Missing compiled type check.*** Type checks are missing in the compiled code, allowing some paths to execute on wrong conditions. As above, missing type checks produce unpredictable results and ultimately crashes at run time. For example, we have found that all floating-point related native methods (*i.e.,* all arithmetics and comparisons) do not perform a type check on the receiver. The compiled code proceeds to unbox a double from the receiver's body producing a segmentation fault on the wrong receiver type.

***Optimization difference.*** Optimizations exist on the compiler but not on the interpreter instruction, or vice-versa. Optimization differences do not produce incorrect executions as in the examples above, but raise the awareness of potential performance improvements. For example, our interpreter performs static type predictions on arithmetic byte-code instructions, inlining integer and float arithmetics [4, 11] and performing a slower message-send only if types do not match the expectations. However, not all of our byte-code compilers implement the same: the simpler SimpleStackCogit implements no static type predictions, while the productive StackToRegisterMappingCogit and the experimental RegisterAllocatingCogit inline only integer arithmetics but not floating point arithmetics.

***Behavioral difference.*** The behavior differs in some way between compiled and interpreted code. For example, we have found that bit-wise operations on the interpreter fail with negative integers and fall-back to slower library code, while compiled code works both with positive and negative integers by treating both as unsigned integers.

***Missing Functionality.*** Behavior is missing in the compiler or interpreter, not implemented and failing at run time with *e.g.,* a not yet implemented exception. For example, we have found that several native methods introduced to accelerate FFI (Foreign Function Interface) memory and structure

accesses were never implemented in the 32 bit compiler version.

***Simulation Error.*** Error in our testing/simulation environment. Since our interpreter-guided approach is more exhaustive at testing that the pre-existing hand-written tests, it has found two different non-implemented paths in the simulation run time. These non-implemented paths are related to the simulation of invalid memory accesses: the simulation disassembles the failing instruction and performs a read-/write operation using reflection to call the corresponding register setter/getters. Our dynamic approach has found that some setter/getter methods reflectively called were missing, and thus difficult to detect by static analyses and lint quality rules.

### 5.4 Execution Time Analysis

To analyze the practicality of the approach, in this section we evaluate the different components that make the execution time. We identify two different kinds of run times: concolic execution run time, and test run time.

On the one hand, concolic execution run time, shown in Figure 6, is the time taken to concolically explore all execution paths of an instruction. Our measurements show that a single byte-code instruction takes in average ~600 ms to explore, while native methods take in average ~1700 ms. Total run time aggregates to 3 and 4.5 minutes respectively. It is worth noticing that most of this run time is taken by the constraint solver and a non-optimized AST-interpreter implementation, and that the results of the concolic exploration can be cached and reused multiple times.

On the other hand, test run time, shown in Figure 7, is the time taken to run all generated tests of a single instruction. Our measurements show that all the byte-code compiler tests take in average ~little above 30 ms, while native methods take in average ~little less than 100 ms. Total run times aggregates to ~10 seconds in total per set of tests.

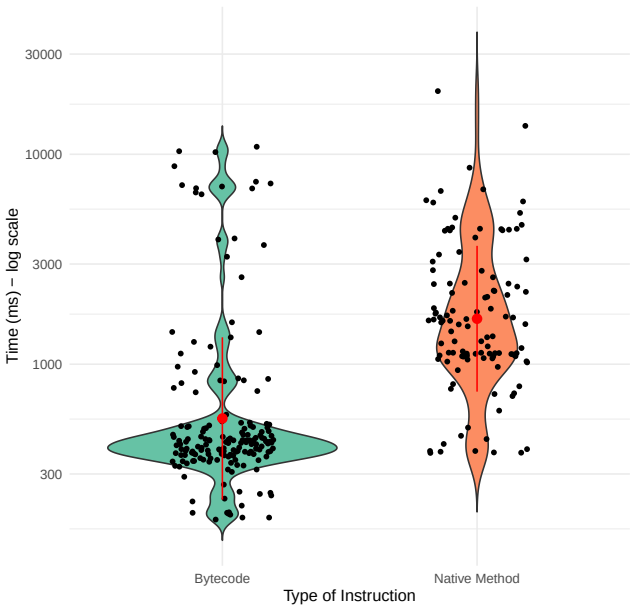


**Figure 6. Concolic execution time per kind of instruction.** Concolic execution per instruction, grouped by kind of instruction. Each data point represents the total execution time to concolically execute all paths of the instruction. Bytecode instructions have less execution paths and thus less execution time. Exploring instructions remains in the other of milliseconds for most instructions making them practical of on-line execution.

## 6 Related Work

Several solutions have been proposed in the past to aid in testing and debugging programming language implementation validation, let them be Virtual Machines or compilers.

***VM Simulation Environments and Meta-circular VMs.*** Meta-circular VMs and VM frameworks have offered for a long time simulation environments that helped in testing and debugging virtual machines. Such is the case of Self [29], Smalltalk [14, 22] and Maxine [30]. Our solution complements simulation environments with unit testing generation, generating tests that are unitary, fast-to-execute and exhaustive.

***VM Testing.*** Maxine and Pharo reported recently QEMU based unit testing infrastructures for cross-ISA testing and debugging [15, 25]. They reported that this infrastructures helped them in porting their VMs to ARMv7 and ARMv8 64bits respectively. Although their approaches are based on unit testing, they rely on manually written tests, while our approach performs automatic test generation.

Lately, several work has explored the path of program fuzzing and differential testing between different Virtual Machines for a single language. Several work on the Java Virtual Machine (JVM) expose bugs via differential testing and bytecode fuzzing [6, 7]. Moreover, test generation has been explored in the case of native extensions for the JVM (*i.e.,* JNI) [13]. Similar work appeared recently for JavaScript engines, applying differential testing with test transplantation [18] and compiler fuzzing [23, 33]. Although we share with these approaches the goal of automatic test generation, these explore a more coarse test generation using and usually think of the VM as a black-box. Our approach differs from these in several points: using the interpreter to guide the compiler testing helps us generating an exhaustive set of tests that are *unitary*. Indeed, our tests are fast to run and easy to debug. Additionally, one key aspect of our solution is that it is applicable in VMs that have a single implementation but many execution engines within it. Finally, our approach produces reproducible tests that exercise both the interpreter

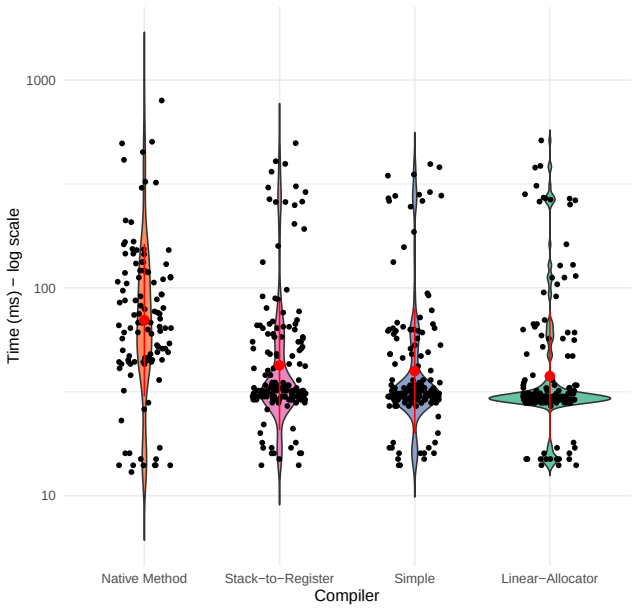


**Figure 7. Test execution time per kind of instruction.** Test execution per instruction, grouped by kind of instruction. Each data point represents the total execution time to execute all generated tests for the instruction. Although native method compiler tests seem to have a higher average than byte-code compiler tests, they all remain below the 100ms bar.

and JIT compilers, while the other approaches are subject to VM non-determinisms.

Finally, some work report efforts to validate optimising compilers in an automatic and semi-automatic way [3, 8]. Although this is not the focus of this paper, we plan to extend our infrastructure in the future to test our optimising JIT compiler.

***Compiler Testing.*** More broadly than VM testing, several work exist on the area of compiler testing and particularly on the automatic generation of test programs and oracles for their validation [5]. Remarkable existing work cover random test generation solutions such as CSmith [32], grammar-based solutions such as the work originated by Purdom [26] and mutation-based test generation such as equivalence-modulo-input (EMI) [17]. Our approach differs from existing work in that most existing work treats compilers as black boxes, while our approach benefits from considering the interpreter as the language specification. In such manner, our approach quickly generates fast and relevant tests that exercise different parts on the JIT compiler and interpreter.

***Differential Compiler testing.*** Differential testing was introduced by McKeeman [19], where he showed how comparing different C compilers to find bugs in their differences. They generate random test cases, depending on the level they test *e.g.,* sequence syntactically correct C programs or type-correct C programs. Our approach extends traditional differential testing by comparing two essentially different execution engines: an interpreter and a compiler.

***Concolic Testing.*** Concolic testing [10, 27] is an automated testing technique that combines **conc**rete and symb**olic** execution of a program to explore all of the program's possible execution paths. Traditionally concolic execution engines have been interpreter based, although recently they have been accelerated through compilation [24]. Our approach is so far implemented as a tree-walking AST interpreter, although, as shown in our evaluation, the execution times remain practical for its on-line usage. Instead, our current performance bottle-necks are in the constraint solver.

## 7 Conclusion

In this article we propose to guide the automatic unit testing of a JIT compiler by an interpreter definition based on two insights: first, we consider interpreters executable specifications of the programming language and second, both the interpreter and compilers for a language should implement the same language semantics. Based on these observations, our approach We first apply concolic testing on the interpreter to discover all possible execution paths and then validate the compiler using a differential testing approach.

We applied it to the byte-code interpreter and four different compilers of the Pharo Virtual Machine: both the native method template-based compiler and the stack-to-register byte-code compiler considered stable and in production since more than 10 years, plus two non-productive compilers. Our approach generated in less than 10 minutes more than 4.5K tests, and found 468 differences from 91 different causes. We show that interpreters are a valid resource when generating JIT compiler test inputs, unveiling many differences between them, ranging from clear bugs producing segmentation faults, to optimization and behavioral differences. Moreover, we show that the approach is practical and applicable on-line. In the future we plan to extend this work to generate minimal and relevant byte-code sequences for unit testing the JIT compiler.

## Acknowledgments

This work was funded by Inria's Action Exploratoire AlaMVic.

## References

[1] B. Alpern, M. A. Butrico, A. Cocchi, J. Dolby, S. J. Fink, D. Grove, and T. Ngo. Experiences porting the jikes rvm to linux/ia32. In *Java Virtual Machine Research and Technology Symposium*, pages 51–64, 2002.

[2] C. Béra and E. Miranda. A bytecode set for adaptive optimizations. In *International Workshop on Smalltalk Technologies (IWST 14)*, Aug. 2014.

[3] C. Béra, E. Miranda, M. Denker, and S. Ducasse. Practical validation of bytecode to bytecode jit compiler dynamic deoptimization. *Journal of Object Technology*, 15(2):1:1–26, 2016.

[4] C. Chambers and D. Ungar. Customization: Optimizing compiler technology for self, a dynamically-typed object-oriented programming language. In *Programming Language Design and Implementation (PLDI)*, PLDI '89, pages 146–160, New York, NY, USA, 1989.

[5] J. Chen, J. Patra, M. Pradel, Y. Xiong, H. Zhang, D. Hao, and L. Zhang. A Survey of Compiler Testing. *ACM Computing Surveys*, pages 1–36, May 2020.

[6] Y. Chen, T. Su, and Z. Su. Deep differential testing of jvm implementations. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, pages 1257–1268. IEEE Press, 2019.

[7] Y. Chen, T. Su, C. Sun, Z. Su, and J. 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, June 2016.

[8] O. Flückiger, G. Scherer, M.-H. Yee, A. Goel, A. Ahmed, and J. Vitek. Correctness of speculative optimizations with dynamic deoptimization. In *Principles of programming languages (POPL'17)*, 2017.

[9] Y. Futamura. Partial evaluation of computation process: An approach to a compiler-compiler. *Higher Order Symbol. Comput.*, 12(4):381–391, 1999.

[10] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *Proceedings of Programming Language Design and Implementation (PLDI'05)*, pages 213–223. ACM, 2005.

[11] U. Holzle. *Adaptive Optimization for Self: Reconciling High Performance with Exploratory Programming*. PhD thesis, Stanford University, Stanford, CA, USA, 1994.

[12] U. Hölzle, C. Chambers, and D. Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *European Conference on Object-Oriented Programming*, ECOOP '91, 1991.

[13] S. Hwang, S. Lee, J. Kim, and S. Ryu. Justgen: Effective test generation for unspecified jni behaviors on jvms. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1708–1718, 2021.

[14] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. 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, Nov. 1997.

[15] C. Kotselidis, A. Nisbet, F. S. Zakkak, and N. Foutris. Cross-isa debugging in meta-circular vms. In *Proceedings of International Workshop on Virtual Machines and Intermediate Languages (VMIL'17)*, pages 1–9, 2017.

[16] B. Kruck, S. Lehmann, C. Keßler, J. Reschke, T. Felgentreff, J. Lincke, and R. Hirschfeld. Multi-level debugging for interpreter developers. In *Companion to Proceedings of the international conference on Modularity*, pages 91–93. ACM, 2016.

[17] V. Le, M. Afshari, and Z. Su. Compiler validation via equivalence modulo inputs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2014.

[18] I. Lima, J. Silva, B. Miranda, G. Pinto, and M. d'Amorim. Exposing Bugs in JavaScript Engines through Test Transplantation and Differential Testing. *arXiv:2012.03759 [cs]*, 2020.

[19] W. M. McKeeman. Differential Testing for Software. *DIGITAL TECHNICAL JOURNAL*, 1998.

[20] E. Miranda. Context management in visualworks 5i, 1999.

[21] E. Miranda. The cog smalltalk virtual machine. In *Proceedings of VMIL 2011*, 2011.

[22] E. Miranda, C. Béra, E. G. Boix, and D. 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.

[23] J. Park, S. An, D. Youn, G. Kim, and S. Ryu. Jest: N+1-version differential testing of both javascript engines and specification. In *Proceedings of the 43rd International Conference on Software Engineering*, ICSE '21, pages 13–24. IEEE Press, 2021.

[24] S. Poeplau and A. Francillon. Symbolic execution with SymCC: Don't interpret, compile! In *29th USENIX Security Symposium*, pages 181–198, Boston, MA, Aug. 2020. USENIX Association.

[25] G. Polito, P. Tesone, S. Ducasse, L. Fabresse, T. Rogliano, P. Misse-Chanabier, and C. H. Phillips. Cross-ISA Testing of the Pharo VM: Lessons Learned While Porting to ARMv8. In *Proceedings of the 18th international conference on Managed Programming Languages and Runtimes (MPLR '21)*, Münster, Germany, Sept. 2021.

[26] P. Purdom. A sentence generator for testing parsers. *BIT Numerical Mathematics*, 12(3):366–375, 1972.

[27] K. Sen, D. Marinov, and G. Agha. Cute: A concolic unit testing engine for c. In *Proceedings of the European Software Engineering Conference (ESEC)*, pages 263–272, New York, NY, USA, 2005.

[28] D. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *ACM SIGPLAN Notices*, 19(5):157–167, 1984.

[29] D. Ungar, A. Spitz, and A. 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.

[30] C. Wimmer, M. Haupt, M. L. Van De Vanter, M. Jordan, L. Daynès, and D. Simon. Maxine: An approachable virtual machine for, and in, java. *ACM Transaction Architecture Code Optimization*, 9(4), Jan. 2013.

[31] T. Würthinger, M. L. Van De Vanter, and D. Simon. Multi-level virtual machine debugging using the java platform debugger architecture. In *Perspectives of Systems Informatics*, pages 401–412. Springer, 2010.

[32] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and Understanding Bugs in C Compilers. In *Programming Language Design and Implementation*, PLDI '11, 2011.

[33] G. Ye, Z. Tang, S. H. Tan, S. Huang, D. Fang, X. Sun, L. Bian, H. Wang, and Z. Wang. Automated conformance testing for javascript engines via deep compiler fuzzing. In *PLDI '21: Proceedings of the 2021 ACM SIGPLAN conference on Programming language design and implementation*, PLDI 2021, pages 435–450, New York, NY, USA, 2021.