Differential Testing of Simulation-Based Virtual Machine Generators Automatic Detection of VM Generator Semantic Gaps Between Simulation and Generated VMs

Pierre Misse-Chanabier, Guillermo Polito, Noury Bouraqadi, Stéphane Ducasse, Luc Fabresse, Pablo Tesone

Differential Testing of Simulation-Based Virtual Machine Generators

Automatic Detection of VM Generator Semantic Gaps Between Simulation and Generated VMs

Pierre Misse-chanabier[1], Guillermo Polito[2], Noury Bouraqadi[3], Stéphane Ducasse[1], Luc Fabresse[3], and Pablo Tesone[1]

[1] Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France
[2] Univ. Lille, CNRS, Inria, Centrale Lille, UMR 9189 CRIStAL,F-59000 Lille, France
[3] IMT Nord Europe, Institut Mines-Télécom, Univ. Lille, Centre for Digital Systems, F-59000 Lille, France

Abstract. Testing and debugging language Virtual Machines (VMs) is a laborious task without the proper tooling. This complexity is aggravated when the VM targets multiple architectures. To solve this problem, simulation-based VM generator frameworks allow one to write test cases on the simulation, but those test cases do not ensure the correctness of the generated artifact due to the semantic gaps between the simulated VM and generated VMs.
We propose Test Transmutation to extend simulation-based VM generator frameworks to support test case generation. It validates the generated VM by also running test cases generated from existing simulation test cases. Results of the generated test cases are compared against the simulation test cases using differential testing. Moreover, test cases are automatically mutated with non-semantic-preserving mutations.
Test Transmutation detects bugs that are representative of typical VM modifications. We demonstrate its effectiveness by applying it to a set of real test cases of the Pharo VM. It allowed us to find several issues that were unknown to the VM development team. Our approach shows promising results to test simulation-based VM generator frameworks.

Keywords: Testing, Virtual Machine, Code Mutation, Simulation

1 Introduction

Modern language implementations are based on language Virtual Machines (VMs) enabling features such as automatic memory management and JIT compilation. To cope with the elevated cost of building language VMs, there are three main kinds of solutions. First, *VM generation frameworks* help with interpreter generation and JIT compiler generation [18,25,13,15,9,31,40]. Second, *Metacircular VMs* such as Maxine [39], Self [38] and Squawk [34] provide rich VMs development environments that allow live debugging. Third, *simulation-based VM generators* are VM generation frameworks with simulation environments easing VM development and debugging. Examples of this last approach are the OpenSmalltalk-VM for Pharo and Squeak [18,25], and PyPy

VM [4] which is supported by the RPython framework [31]. A main challenge of using such simulation-based VM generator frameworks is the *semantic gaps* [6] between the simulated VM and the generated VMs. Semantic gaps are differences in the semantic before and after compilation. Examples of the semantic gaps are differences in typing or abstraction level, which make simulation environments not functionally equivalent to the generation target environments [29].

Testing and debugging VMs is a laborious task without the proper tooling, which is aggravated when the VM targets multiple architectures [5]. Simulation-based VM environments allow one to write test cases on the simulation environment, but those test cases do not ensure the correctness of the generated VM. Recently, the team of Maxine and Pharo reported QEMU-based[5] test infrastructures for testing and debugging [20,30]. Maxine's Metacircular approach compiles directly to machine code, losing its rich debugging capabilities when bootstrapping in a new platform. Pharo's approach benefits from the simulation, but tests only the simulated VM. Finally, an abundance of work exists on compiler testing outside of the VM community [10], but these works are not directly applicable to the VM field.

In this article, we propose a novel approach to test simulation-based VM generator frameworks that we call Test Transmutation. We propose to extend such frameworks to also reuse simulation test cases. We generate the test cases along with the VM and execute them on the generated VM. We use differential testing [23] to compare the generated test results against the simulation test results for each test case. Moreover, generated test cases are also impacted by semantic gaps such as the typing differences. To minimize the test cases' semantic gaps, we mutate the test cases [19]. Test cases with different test results, mutated or not, are evidence of bugs in the VM or VM generator.

We validate that our approach detects errors by manually introducing bugs in the Pharo VM code. We perform an empirical analysis applying our approach to a subset of the Pharo VM test cases. Although this VM has been stable and industrially-used for decades, we find bugs on the VM generation and simulation. Our analysis shows that our approach is promising to test simulation-based VM generator frameworks.

The contributions of this paper are the following:
- A testing approach for simulation-based VM generators.
- Validation of a code generation framework using *non-semantic-preserving mutations* together with differential testing. To the best of our knowledge, we are the first to propose such an approach to validate a code generation framework.

We first explain why testing only the simulation in simulation-based VM generator frameworks is insufficient (Section 2). We then present Test Transmutation (Section 3) and our experimental context (Section 4). We validate that Test Transmutation detects manually introduced bugs (Section 5). We also show empirical evidence of Test Transmutation on an industry level VM (Section 6). Finally, we present the relevant related work (Section 7) before concluding the paper (Section 8).

[4] www.pypy.org
[5] https://www.qemu.org

2 Problem: Testing simulation-based VM generator Frameworks

Simulation-based VM generator frameworks propose two modes of execution: the simulated execution and the generated code execution. Most debugging and development happens in the simulation environment, and once the VM is ready, it is compiled to a machine executable VM [25,31]. However, the simulated VM presents **semantic gaps** [6] with the generated VMs because they trade-off precision for other software qualities such as fast feedback loops. Moreover the test cases are only executed on the simulation environment, but those test cases do not ensure the correctness of the generated VM. This Section presents the semantic gaps problem of simulation-based VM generators and the limits of simulation-based testing with Slang VM generator examples, the simulation-based VM generator framework used by Pharo.

2.1 Example Context: The Slang VM generator

Pharo is an object-oriented dynamically-typed language from the Smalltalk tradition [7]. The Pharo VM is an industrial level VM written in Pharo itself and generating a C VM using a VM-specific generator called Slang VM generator [18]. The VM implements at the core of its execution engine a threaded bytecode interpreter, a linear non-optimising JIT compiler named Cogit [24] that includes polymorphic inline caches [17] and a generational scavenger garbage collector that uses a copy collector for young objects and a mark-compact collector for older objects [37]. The VM generator is in charge of generating from high-level object-oriented Pharo code, low-level functional/imperative C code and applying VM specific optimizations to it.

2.2 Pharo VM Semantic Gaps by Example

The Slang VM generator takes as input a VM definition written in Pharo itself and generates C code [26]. Code generation does not happen without loss: the Slang VM generator generates an efficient VM by restricting the input language and its generation. Pharo's object-oriented features are either mapped to C code [36] or rejected.

```
1 primitiveNaturalLogarithm
2   <var: receiver type: #double>
3   | receiver |
4   receiver := self stackFloatValue: 0.
5   self successful ifFalse: [ ↑ self primitiveFail ].
6   self putOnStackTopFloatObjectOf:
7     (self generationEnvironment: [ receiver log ]
8           simulationEnvironment: [ receiver ln ])
```

Listing 1.1. Excerpt of VM code for the primitiveNaturalLogarithm. It is showing environment specific code and type annotations that are ignored during simulation.

Let us consider the example in Listing 1.1 that presents some of Slang VM generator features. The example defines a native function (*i.e.,* primitives in Pharo's terminology) computing the natural logarithm from a positive floating point number. The Slang VM

generator provides a framework to define stack-based bytecode machines and access the execution stack *e.g.,* the call to stackFloatValue: (line 4 and 6). It also provides VM developers with a way to specify environment specific code (line 7 and 8). For example, the natural logarithm is implemented in the C standard library using the **log** function, while in Pharo, it is implemented using the **ln** message. Indeed, the **log** method identifier (*i.e.,* selector in Pharo's terminology) exists in Pharo but computes a base 10 logarithm.

Moreover, Pharo is a dynamically-typed language that does not use explicit type annotations. However, the Slang VM generator needs them to produce correct C code although the simulation ignores them (line 2). This means that wrong type annotations only impact the generation introduces issues specific to the generated VM. Notice that the semantic gaps are aggravated by additional Pharo features *e.g.,* inheritance, polymorphism, method redefinition, super message sends, block closures, managed memory, absence of stack allocation which require more complex code transformations.

2.3 Limits of Simulation-Based Testing

Simulation-based VM generator frameworks allow VM developers to debug and develop in a simulation environment. Simulation environments allow developers to apply automated unit testing on the VM. Let's consider for example the VM test case depicted in Listing 1.2. This test case is executed as plain Pharo code during simulation and passes, regardless of bugs in type annotations. If we modify or remove the type annotation, the generated VM does not work anymore: it casts a float to an int, producing unexpected results. Similar problems have been informally reported for simulation-based VM generator frameworks such as RPython [32].

```
1 testPrimitiveNaturalLogarithmShouldFailForNegativeNumber
2   | aFloat |
3   aFloat := self newFloatFromInt: −1.
4   interpreter push: aFloat.
5   interpreter primitiveNaturalLogarithm.
6   self assert: interpreter failed.
```

Listing 1.2. Excerpt of a test case validating that the logarithm of a negative number sets the interpreter in a failed state. This test cases passes in simulation but does not capture potential typing failures of the generated code.

2.4 Problem Statement

Problem. Simulation-based VM generators introduces semantic gaps due to the differences between the simulated and generated VM *e.g.,* semantic typing difference, memory management, integer semantics, etc. We have observed that in current state-of-the-art simulation-based VM generators, testing only the simulated VM is not enough.

Proposal We propose to extend simulation-based VM generator frameworks to reuse simulation test cases by generating and executing them on generated VMs. We validate the outputs of both simulated and generated test cases using differential testing: a

difference in the execution indicates a VM bug or a VM generator bug. However, test cases generated in such a way may also fall into semantic gaps similar to the ones found in the VM. To validate that our test case generation is correct, we propose to combine differential testing with non-semantic-preserving mutations.

We formulate the following research questions: **RQ1.** Does Test Transmutation detect semantic gaps between simulated and generated VMs? **RQ2.** Does Test Transmutation detect bugs in existing VM test cases?

3 Test Transmutation

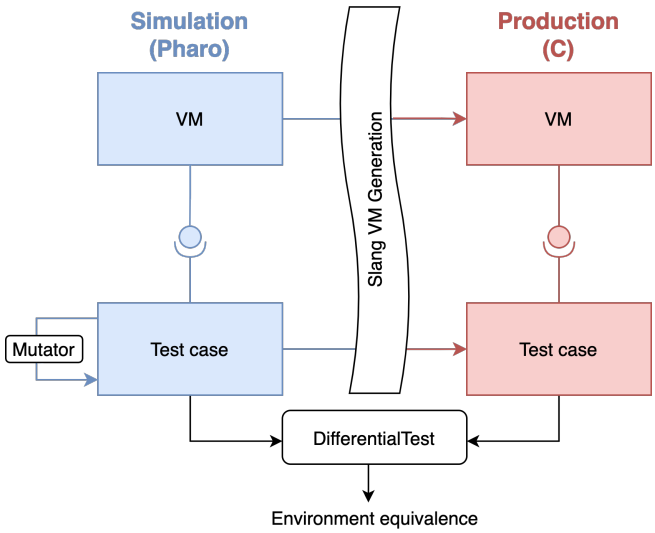

Fig. 1. Test Transmutation overview. We generate test cases with the VM and compare their results with differential testing. Mutations increase the number and variety of test cases.

We propose Test Transmutation a solution to address semantic gaps by generating the simulated VM's test cases (cf. Figure 1). Existing simulation test cases are therefore reused to also test the generated VM. We execute both simulated and generated test cases and compare their results with differential testing [23] to detect execution differences. A core insight is that regardless of the differences between simulated and generated code, test cases are always self-validating and have a deterministic and discrete result: they either succeed, or they fail in an assertion check or a runtime error. Our differential testing process relies on this self-validating property to build the test oracle: two test cases have equivalent behavior if they both pass or if they both fail.

Generating the test cases presents two main challenges. First, generated test cases suffer from semantic gaps similar to the ones found in VMs. Second, the existing and

maintained VM test cases will in general successfully execute, making them poor inputs for the differential testing process.

To address these challenges, we extend our approach with automatic non-semantic-preserving mutations [12]. Mutated test cases are expected to keep behaving similarly when executed in the simulated and generated VM. For example, if a mutated test case breaks in the simulated VM, it should also fails in the generated VM. A non-equivalence of test results shows that there is a bug in the VM or in the VM generator.

3.1 Differential Testing of Test Cases

We check for functional equivalence of both simulated and generated VMs by applying differential testing [23] and validating whether a test case has *similar* results in both. Our heuristic differentiates test results in a binary way: a test case either passes or fails.

We consider that a test case passes in the well-known usage of the word: It runs all its instructions without error. An error can be either: a failed assertion, a run-time exception or a compilation failure. In case of any error, the test fails. In our case, failures happen because of four different reasons: (i) either the VM generator rejects the program, or (ii) the generated source code does not compile, (iii) or an assertion check fails, or (iv) an unexpected runtime error happens.

Table 1 summarizes the behavior of our oracle. When the test results are the same (*i.e.,* both success or both failure), we consider that the differential test case passes and no bug was found. When the test case results differ, the test case reveals a bug.

Simulated code	Generated code	Differential
✓ Passing	✓ Passing	✓ Equivalent
✗ Failing	✗ Failing	✓ Equivalent
✓ Passing	✗ Failing	✗ Non-Equivalent (bug!)
✗ Failing	✓ Passing	✗ Non-Equivalent (bug!)

Table 1. Truth table for the interpretation of the differential testing function.

3.2 Test Case Variations with Non-Semantic-Preserving Mutations

Since we generate test cases from existing simulation test cases, the test cases suffer from similar semantic gaps as the VMs *e.g.,* typing semantics differences, as well as a few others *e.g.,* the testing framework. Moreover all have poor variations in their results because the VM is maintained to pass all test cases. We tackle this issue by applying non-semantics-preserving mutations on the available test cases. Mutations automate the creation of more inputs allowing one to gain confidence in the correctness of the generation process and the generated test cases. Our main observation here is that applying mutations to the original simulated test cases should keep the generated counterpart functionally equivalent. In other words, if a mutant breaks a simulated test case, we expect it to break the generated test case too. Similarly, if a mutant keeps a simulation test case passing, we expect the generated test case to be passing. Therefore we validate mutants with the same process of differential testing explained in the previous section.

We perform non-semantics-preserving mutations of the test cases passing in both environments [1]. For example, Listing 1.3 shows the application of the mutation operator *removeStatementMutation* which removes a statement from the method.

```
1 testScavengeNonSurvivorObjectShouldLeaveMemoryEmpty
2   self newNewSpaceObject.
3   "memory collectNewSpace. <— removed statement"
4   self assert: memory isEmpty
```

Listing 1.3. Example of an automatically mutated test case. The original test case allocates an object, executes a garbage collection and expects the memory to be freed. A mutation operator removes the garbage collection, making the assertion (line 4) fail.

4 Experimental Context of the Validation and Threats to Validity

We implemented Test Transmutation on top of the Pharo VM [7], an industrial-level VM used by the Pharo programming language, a continuation of the work performed in the OpenSmalltalk-VM. At the time of writing, the Pharo VM has around 1000 unit test cases [30] written in the simulation environment covering the memory manager, the bytecode interpreter, the language primitives, and the Just-In-Time compiler. These test cases are executed for simulations ofdifferent for simulations ofword sizes (32/64 bits) and processor architectures (x86, x86-64, ARM32 and ARM64).

Selected Test Case Subset. Test cases are written in plain Pharo, thus their generation was not initially possible: test cases use many idioms not supported by the Slang VM generator such as the testing framework generation. Therefore, we focused on a subset of the available test cases, namely the memory management test cases. Such test cases cover the implementation of a generational scavenger garbage collector and a mark-compact collector for older objects [37] their structure and their allocation schemes.

The selected subset may be considered not representative of the entire VM process. To address this issue, Section 5 shows that our approach covers many semantic gaps that are independent of the tested component.

Testing Requires Disabling Optimizations The Slang VM generator is tailored for performance. In addition, many optimizations are required for a correct generation. For example, method inlining is required for stack-allocation to work as expected. Indeed, methods allocating stack-memory (*i.e.,* using alloca(), Section 5.1) need to be inlined in their caller, otherwise the allocated memory is freed right-away. Such optimizations affect the observability of the program, and need to be disabled to allow their testing. Thus, they pose a potential threat to validity. To minimize this threat to validity, we only disable inlines required by test cases.

5 Bug Detection Assessment

To evaluate the ability of Test Transmutation to detect bugs (**RQ1**), we manually introduce bugs in the VM based on bugs experimented by VM developers in the past.

We then evaluate those bugs on 238 test cases covering the modified code. This section presents the introduced bugs, their rationale, relevance and reports the observed the outputs. This section shows that Test Transmutation detects semantic gaps between simulated and generated VMs.

Our results show that our approach is able to unveil several differences between the simulated and generated VMs. We consider as a correct behavior that the Slang VM generator rejects problematic code during generation time. However, our evaluation shows that the Slang VM generator presents the following flaws that are presented in the remainder of this section:

Memory Management. Careless memory management produces runtime failures.
Type annotations. Errors in type annotations may produce runtime failures.
Name Conflicts. Name conflicts creates compile-time errors.
Undefined Behavior. The Slang VM generator is unaware of the target language undefined behavior. Therefore the generated code may exhibit undefined behavior.
Unsupported Simulation. Some programs are invalid at simulation time, although they can be generated and executed correctly.

5.1 Memory Management Differences

Background. The simulation running in Pharo presents deep differences with the target C runtime environment in terms of memory management. Indeed, Pharo is a managed language with garbage collection and no explicit stack allocation, while C allows developers to perform stack allocations by using standard library functions such as alloca(). This means that the developers should take special care when simulating manual memory management while developing the VM memory manager. An example of such a simulation is illustrated in Listing 1.4.

```
1 alloca: desiredSize
2   memory := self generatedCode: [ self alloca: desiredSize ]. " stack allocation "
3                 simulationCode: [ ByteArray new: desiredSize ]. " heap allocation "
```

Listing 1.4. Example of simulation specific code simulating stack-allocation with heap allocation.

Bug Description. Stack allocations are simulated as simple heap allocations. This means that simulated stack allocations out-live their defining stack-frames, allowing programs to freely access and modify such a piece of memory. The bug introduced memory accesses to stack-allocated regions returned to their caller. **Results.** All simulated test cases passed, whereas all generated test cases failed with segmentation faults.

5.2 Type Annotation Errors

Background. Simulated and generated VMs present differences in typing. On the one hand, the simulated VM executes on top of Pharo and uses its dynamically-typed features. On the other hand, generating the VM to C requires the VM source code to have some type annotation. Moreover, even though type annotations are available in the source code, they are ignored by the simulation environment. An example of such type annotations is illustrated in Listing 1.5.

```
addGCRoot: varLoc
  "<var: #varLoc type: #'long int *'> ### Original annotation"
  <var: #varLoc type: #'char *'> "### Buggy replacement"
  extraRootCount >= ExtraRootsSize ifTrue: [ ↑ false ]. "out of space"
  extraRootCount := extraRootCount + 1.
  extraRoots at: extraRootCount put: varLoc.
  ↑true
```

Listing 1.5. Example of type Annotation Error

Bug Description. Wrong or missing type annotations break the generation process or make the executable VMs produce runtime errors. The introduced bug is a change of a type annotation from a long int pointer to a char pointer.
Results. All simulated test cases passed, whereas all generated test cases failed with segmentation faults. We consider this a VM generator bug.

5.3 Literal Type Errors

Background. Another simulated vs generated VMs typing error happens during type checking and type inference at generation-time. The VM generation process performs type inference to guide generation.
Bug Description. We introduced a bug by changing the type of a literal from integer to float with the same value as illustrated in Listing 1.6.

```
weakArrayFormat
  "↑4 ### Original code"
  ↑4.0 "### Buggy replacement"
```

Listing 1.6. Bug introduced by changing the type of a literal from integer to float.

Results

Modifying a literal's type had an effect both in the simulated and generated test cases. 10 out of 238 test cases failed on the simulation because the Float class does not implement the message #bitAnd:. At the same time, none of the test cases could be generated because VM generation eagerly rejected them with a type error.

Notice that we consider this the correct behavior: generated test cases detect the difference, in this case at VM-generation time.

5.4 Name Conflicts and Name Mangling

Background. The simulated and the generation target environment do not have the same rules for name management. For example, it is perfectly valid in the simulation (written in Pharo) to have a method and a class' attribute with the same name. However, in C a function and a global variable with the same name produce name conflicts. This difference is aggravated by the fact that the Pharo convention dictates that getters and setters have the same name as the attributes they access.
Bug Description. We introduced a bug by explicitly preventing the inlining of a getter and setter pair as shown in Listing 1.7.

```
bogon
  <inlineInC: #false>
  ↑ bogon
```

Listing 1.7. Example of non-inlined getter, creating a name conflict on generation.

Results. All these being valid Pharo idioms, simulated test cases remain passing. However, the VM generation process accepts this idiom but generates wrong C code, causing an ulterior compilation error. We consider this a VM generator bug: the VM-generator should manage the name-mangling of methods and generate a working VM.

5.5 Undefined Behavior

Background. The C programming language targeted by the Slang VM generator contains 193 undefined behavior in C99 [11]. Upon encountering an undefined behavior, C compilers are able to apply aggressive optimization. Thus, C developers must avoid writing them in their program. However, since VM developers write mainly code in the simulation environment, both the developer and the Slang VM generator should make sure to not produce code with undefined behavior.
Bug Description. We introduced a C undefined behavior by introducing an arithmetic overflow (Listing 1.8).

```
howManyFreeObjects
  | counter |
  counter := 0.
  self allFreeObjectsDo: [ :freeObject | counter := counter + 1 ].
  "↑ counter ### Original code"
  ↑ counter + self maxCInteger + 1 " ### Buggy replacement"
```

Listing 1.8. Introducing a potential cause of undefined behavior .

Results. Since in Pharo, integer arithmetics is potentially unbound, the expression self maxCInteger + 1 is automatically coerced to a large precision integer, making 5 simulation test cases fail. At the same time, all 238 generated test cases pass in the generation target environments. We consider this a Slang VM generator bug: either the program should be rejected by the Slang VM generator, or the simulation environment should be modified to have the same integer arithmetics as the target.

5.6 External functions and Name-mangling

Background. The Slang VM generator allows any function or method to call any external C function by just mangling method-invocation selectors to C function names at generation time. However, the Slang VM generator's name mangling can produce the same function name for different method selectors. Moreover, to enable simulation of external functions, a simulation specific implementation requires to be provided. Thus, VM developers must use the correct method invocation so a function call works on both simulated and generated code.
Bug Description. We introduced a bug by replacing an external function call with an equivalent one from the name mangling point of view (Listing 1.9).

```
"Original code"
self me: destAddress mc: sourceAddress py: bytes

"Buggy replacement"
self memc: destAddress p: sourceAddress y: bytes
```

Listing 1.9. Example changing a call to the C function memcpy by a version that generates correct code but cannot be properly simulated.

Results. This modification produced 50 failing simulated test cases, while all 238 generated test cases passed. We consider this a VM-generator bug: the simulation should have been allowed to run or the VM-generator should have rejected the program.

6 Empirical results on the Pharo VM

To evaluate the capability of Test Transmutation to find bugs in existing test cases (**RQ2**), we apply our approach on a subset of the existing VM test cases. This section shows that Test Transmutation is able to detect bugs in existing VM test cases.

6.1 Test Case Characterization

Our validation executes Test Transmutation on a subset of 256 tests, out of which 238 are correctly generated by our prototype implementation. We categorize the test cases as follow:

Memory structure. Test cases checking that the memory structural properties. For example, they check that Garbage Collection specific variables are initialized.

Memory allocation. Test cases checking the multiple object allocation heuristics in the VM heap. For example, they check that creating a new object reduces the space available in the new space.

New space garbage collection. Test cases checking the copy garbage collector used in the new space. For example, they check that allocating two objects in a row, should allocate both objects next to one another.

Old space garbage collection. Test cases checking the mark compact algorithm used for old generation garbage collection. For example, they check that objects referenced are marked, and not reclaimed nor moved by the garbage collection.

6.2 Prototype Results

Table 2 reports on the test cases generation status. The test cases are executed on Pharo 9 to test the Pharo 9 VM. Currently all 256 initial test cases are passing in the simulation. Of those 256, 238 are passing in the generated VM for Ubuntu 20.0 and x64.

From the existing 238 test cases we generate a total 494 mutants. The mutants are generated by using coverage directed mutant creations to only create mutants that are in the path of at least one test case. The execution of the test cases also uses coverage information to only execute necessary test cases for each mutant. Every executed test case for each mutant is executed in both the simulated and generated VM. There was 1890 test case executions to validate the 494 generated mutants.

Category	Sub-Category 1	Passing test case in simulated VM	Passing test case in generated VM	Mutant Executions	Mutant detecting bugs
Memory structure	New space	8	7	7	0
	Old space	7	1	1	0
Memory allocation	Allocation in new space	9	8	13	0
	Allocation in old space	21	21	71	0
	Allocation strategies	127	127	1464	2
New space garbage collection	GC of regular objects	38	29	62	11
	GC of weak objects	9	9	34	0
	GC of ephemeron objects	19	19	101	10
Old space garbage collection	GC of regular objects	9	8	33	0
	GC of unmovable object	9	9	104	0
Total		256	238	1890	23

Table 2. Detailed categorization of the test cases considered and relevant statistics.

6.3 Result Analysis

Since the VM has been used industrially for several years now, the VM is stable and we expect to find few bugs in the code. However, in the current prototype implementation we detect three kinds of bugs captured using 23 mutated test cases.

Stack Allocation Issues. At first, Test Transmutation uncovered the alloca simulation issue described in Section 5.1. Non-inlined stack allocations are freed as soon as the allocating function returns, causing memory access violations in all tests. We had to fix this bug in order to obtain these results and the following bugs.

Division Differences. Mutations unveil a difference in the behavior between simulated and generated VMs regarding integer division. In Pharo both the selector #// and #/ exists. The first is returning an exact integer whereas the second is returning a fraction. Moreover, #// truncates the result of the division, and returns an integer. Both selectors are generating the / operator in C. Mutants break an implicit invariant of the VM, creating a runtime error in the simulated VM.

Runtime Assertion Differences. Mutations unveil a difference in the behavior between simulated and generated VMs regarding runtime assertions. Runtime assertions check for invariants to eagerly prevent complex scenarios such as memory corruptions. On the one hand, invariant assertions in the simulated VM stop the execution and fail test cases. On the other hand, invariant assertions in the generated VM log an error and continue the execution. Because of these differences, 21 mutated test cases are passing in the simulated VM but not in the generated VM.

7 Related work

Simulation-based VM generator frameworks are VM specific compilers: they generate VMs from a specification to an executable form, applying VM specific transformations and optimizations. In this sense, this section also presents compiler testing approaches that are related to ours.

Differential Compiler testing. Differential testing was introduced by McKeeman [23]. The author shows the effectiveness of differential testing on C compilers to find bugs. They generate random test cases ex-nihilo in different ways, depending on the level

they test (e.g. sequence syntactically correct C program or Type-correct C program). On the one hand, we also apply differential testing to uncover differences between two execution environments, in our case a simulation and a generation target environment. On the other hand, we do not use random test cases: we start from existing test cases and apply non-semantic-preserving mutations on them.

CSmith [41] expands on previous work to generate random C programs free of undefined behavior. They describe their generated programs as having no need for an oracle. They apply the generated program with a cross-compiler differential testing strategy. Our work does not require to generate programs free of undefined behavior, since the simulation environment has no undefined behavior. It allows us to detect when the VM-generator introduces undefined behavior in the generated VM.

Equivalence Modulo Input. Equivalence Modulo Input (EMI) proposes to create functionally equivalent test input programs. Those test input programs are created by applying mutations on dead code [21,22] or semantic-preserving mutations on live code [35]. The mutated programs are expected to be functionally equivalent to the initial program. Therefore, they check that the observable behavior is equivalent between the original program and the mutated program. The mutated programs are designed to not introduce new undefined behavior. This works under the assumption that the initial program does not contain undefined behavior. Our solution applies non-semantic-preserving mutation on live code instead.

Mutation-based Test Assessments. Many works use mutation-based approaches to measure the effectiveness and efficiency of testing techniques [27]. Similar to us, many works use automated mutations to guide test case generation [14,16,28]. Papadakis et al. [27] identify that many approaches present type I errors because of the subsumed mutant threat, that creates irrelevant mutants that inflate mutant scores and skew results, altering conclusions. Such threat does not apply to our work, because we only use mutants to generate new test cases and we do not compare two testing approaches nor measure mutation scores.

Mutation-based Simulation Testing. Several work have applied mutations on simulation-based models outside of VMs. Rutherford et al. [33] present an automated approach on top of distributed system simulations using mutations. Aichernig et al. [4,2,3] present a mutation-testing approach for UML models. To the best of our knowledge, Test Transmutation is the first approach applying mutation-based testing for VMs code, and VM simulation environments.

Dealing With Semantic Gaps PharoJS [8] validates application generation from Pharo to Javascript with the application test cases. It uses cross compiler differential testing, comparing the Pharo and Javascript compiler. It also uses a different implementation in which the Pharo environment executes the test case and is sending messages to the Javascript environment rather than compiling the test cases to Javascript and then executing them. Moreover PharoJS test cases do not rely on mutations.

Also, rather than creating semantic gaps, some prefer removing it [6]. Besnard et al. unify the semantics of a model for execution, deployment and tooling. They implement a VM to run the executable model. This VM allows them to control remotely the execution, allowing tooling such as debuggers.

8 Conclusion

This paper describes a novel technique to test simulation-based VM generator frameworks we call Test Transmutation. It extends simulation-based VM generator frameworks to support the generation of test cases, to then apply differential testing between simulated and generated VMs. It reuses existing simulation test cases, if any are available. We apply non-semantic-preserving mutations to increment test variability and minimize the impact of semantic gaps introduced during test generation. We apply our prototype implementation on the Pharo VM and validate it by manually introducing bugs inspired from VM developers and by executing it on real test cases. Our evaluation shows that our approach detects typical bugs and detects bugs and translation differences on a an industrially-used VM that is stable and has many users.

In the future we plan to investigate Test Transmutation on JIT compiler test cases, to use with VM specific mutations derived from commits the history, and to compare the different generated VM.

Acknowledgements This work was supported by Ministry of Higher Education and Research, Hauts de France Regional Council and the AlaMVic Action Exploratoire INRIA - Lille Nord Europe.

References

1. Abdi, M., Rocha, H., Demeyer, S.: Test Amplification in the Pharo Smalltalk Ecosystem. In: International Workshop on Smalltalk Technologies (IWST) (Aug 2019), shorturl.at/floF4
2. Aichernig, B.K., Auer, J., Jöbstl, E., Korošec, R., Krenn, W., Schlick, R., Schmidt, B.V.: Model-based mutation testing of an industrial measurement device. In: Seidl, M., Tillmann, N. (eds.) Tests and Proofs. pp. 1–19. Springer International Publishing, Cham (2014)
3. Aichernig, B.K., Brandl, H., Jöbstl, E., Krenn, W.: Efficient mutation killers in action. In: 2011 Fourth IEEE International Conference on Software Testing, Verification and Validation. pp. 120–129 (2011). https://doi.org/10.1109/ICST.2011.57
4. Aichernig, B.K., Brandl, H., Jöbstl, E., Krenn, W., Schlick, R., Tiran, S.: Killing strategies for model-based mutation testing. Softw. Test. Verif. Reliab. **25**(8), 716–748 (Dec 2015). https://doi.org/10.1002/stvr.1522, https://doi.org/10.1002/stvr.1522
5. Alpern, B., Butrico, M.A., Cocchi, A., Dolby, J., Fink, S.J., Grove, D., Ngo, T.: Experiences porting the jikes rvm to linux/ia32. In: Java Virtual Machine Research and Technology Symposium. pp. 51–64 (2002)
6. Besnard, V., Brun, M., Dhaussy, P., Jouault, F., Olivier, D., Teodorov, C.: Towards one Model Interpreter for Both Design and Deployment. In: Third International Workshop on Executable Modeling (EXE 2017) (Sep 2017), https://hal.archives-ouvertes.fr/hal-01585318
7. Black, A.P., Ducasse, S., Nierstrasz, O., Pollet, D., Cassou, D., Denker, M.: Pharo by Example. Square Bracket Associates, Kehrsatz, Switzerland (2009), http://books.pharo.org
8. Bouraqadi, N., Mason, D.: Mocks, Proxies, and Transpilation as Development Strategies for Web Development. In: Proceedings of the 11th edition of the International Workshop on Smalltalk Technologies. pp. 1–6. IWST'16, Association for Computing Machinery (Aug 2016), http://www.esug.org/data/ESUG2016/IWST/Papers/IWST_2016_paper_23.pdf
9. Casey, K., Gregg, D., Ertl, M.A.: Tiger – an interpreter generation tool. In: Bodik, R. (ed.) Compiler Construction. pp. 246–249. Springer Berlin Heidelberg, Berlin, Heidelberg (2005)

10. Chen, J., Patra, J., Pradel, M., Xiong, Y., Zhang, H., Hao, D., Zhang, L.: A Survey of Compiler Testing. ACM Computing Surveys (May 2020), https://dl.acm.org/doi/10.1145/3363562
11. Commitee, C.S.: C99 specification (2007), shorturl.at/goyJQ
12. DeMillo, R.A., Lipton, R.J., Sayward, F.G.: Program mutation: A new approach to program testing. Infotech State of the Art Report, Software Testing **2**(1979), 107–126 (1979)
13. Ertl, M.A., Gregg, D.: Optimizing indirect branch prediction accuracy in virtual machine interpreters. In: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation. pp. 278–288 (2003)
14. Fraser, G., Zeller, A.: Mutation-driven generation of unit tests and oracles. IEEE Transactions on Software Engineering **38**(2), 278–292 (2012). https://doi.org/10.1109/TSE.2011.93
15. Gregg, D., Ertl, M.A.: A language and tool for generating efficient virtual machine interpreters. In: Domain-Specific Program Generation, pp. 196–215. Springer (2004)
16. Harman, M., Jia, Y., Langdon, W.B.: Strong higher order mutation-based test data generation. In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering. p. 212–222. ESEC/FSE '11, Association for Computing Machinery, New York, NY, USA (2011). https://doi.org/10.1145/2025113.2025144
17. Hölzle, U., Chambers, C., Ungar, D.: Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In: European Conference on Object-Oriented Programming (ECOOP'91) (1991). https://doi.org/10.1007/BFb0057013
18. Ingalls, D., Kaehler, T., Maloney, J., Wallace, S., Kay, A.: 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). pp. 318–326. ACM Press (Nov 1997). https://doi.org/10.1145/263700.263754
19. Just, R., Jalali, D., Inozemtseva, L., Ernst, M.D., Holmes, R., Fraser, G.: Are mutants a valid substitute for real faults in software testing? In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 654–665 (2014)
20. Kotselidis, C., Nisbet, A., Zakkak, F.S., Foutris, N.: Cross-isa debugging in meta-circular vms. In: Proceedings of International Workshop on Virtual Machines and Intermediate Languages (VMIL'17). pp. 1–9 (2017). https://doi.org/10.1145/3141871.3141872
21. Le, V., Afshari, M., Su, Z.: Compiler validation via equivalence modulo inputs. In: Programming Language Design and Implementation. PLDI '14 (2014). https://doi.org/10.1145/2594291.2594334
22. Le, V., Sun, C., Su, Z.: Finding deep compiler bugs via guided stochastic program mutation. ACM SIGPLAN Notices **50**, 386–399 (Oct 2015), https://doi.org/10.1145/2858965.2814319
23. McKeeman, W.M.: Differential Testing for Software. Digital Technical Journal (1998)
24. Miranda, E.: The cog smalltalk virtual machine. In: Proceedings of VMIL 2011 (2011)
25. Miranda, E., Béra, C., Boix, E.G., Ingalls, D.: 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). pp. 57–66. ACM (2018). https://doi.org/10.1145/3281287.3281295
26. Misse-Chanabier, P., Aranega, V., Polito, G., Ducasse, S.: Illicium a modular transpilation toolchain from pharo to c. In: International workshop of Smalltalk Technologies. Köln, Germany (Aug 2019)
27. Papadakis, M., Henard, C., Harman, M., Jia, Y., Le Traon, Y.: Threats to the validity of mutation-based test assessment. In: Proceedings of the 25th International Symposium on Software Testing and Analysis. p. 354–365. ISSTA 2016, Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2931037.2931040
28. Papadakis, M., Malevris, N.: Automatic mutation test case generation via dynamic symbolic execution. In: 2010 IEEE 21st International Symposium on Software Reliability Engineering. pp. 121–130 (2010). https://doi.org/10.1109/ISSRE.2010.38

29. Person, S., Dwyer, M.B., Elbaum, S., Păsăreanu, C.S.: Differential symbolic execution. In: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering. pp. 226–237. SIGSOFT '08/FSE-16, Association for Computing Machinery (Nov 2008), https://doi.org/10.1145/1453101.1453131
30. Polito, G., Tesone, P., Ducasse, S., Fabresse, L., Rogliano, T., Misse-Chanabier, P., Phillips, C.H.: Cross-ISA Testing of the Pharo VM: Lessons Learned While Porting to ARMv8. In: MPLR,Germany. Münster, Germany (Sep 2021). https://doi.org/10.1145/3475738.3480715
31. Rigo, A., Pedroni, S.: PyPy's approach to virtual machine construction. In: Proceedings of the 2006 conference on Dynamic languages symposium. ACM, New York, NY, USA (2006)
32. RPythonCommunity: Rpython documentation on test translation (2016), shorturl.at/gBDGT
33. Rutherford, M., Carzaniga, A., Wolf, A.: Evaluating test suites and adequacy criteria using simulation-based models of distributed systems. Software Engineering, IEEE Transactions on **34**, 452–470 (aug 2008). https://doi.org/10.1109/TSE.2008.33
34. Simon, D., Cifuentes, C., Cleal, D., Daniels, J., White, D.: Java on the bare metal of wireless sensor devices: the Squawk Java virtual machine. In: VEE '06: Proceedings of the 2nd international conference on Virtual execution environments. pp. 78–88. ACM Press, New York, NY, USA (2006). https://doi.org/10.1145/1134760.1134773
35. Sun, C., Le, V., Su, Z.: Finding compiler bugs via live code mutation. In: Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications. pp. 849–863. OOPSLA 2016, Association for Computing Machinery (Oct 2016), https://doi.org/10.1145/2983990.2984038
36. Terekhov, A.A., Verhoef, C.: The realities of language conversions. IEEE Software **17**(6), 111–124 (Nov 2000). https://doi.org/10.1109/52.895180
37. Ungar, D.: Generation scavenging: A non-disruptive high performance storage reclamation algorithm. ACM SIGPLAN Notices **19**(5) (1984). https://doi.org/10.1145/390011.808261
38. Ungar, D., Spitz, A., Ausch, A.: Constructing a metacircular virtual machine in an exploratory programming environment. In: Companion to Object-Oriented Programming, Systems, Languages, and Applications conference (OOPSLA'05). ACM (2005)
39. Wimmer, C., Haupt, M., Vanter, M.L.V.D., Jordan, M., Daynes, L., Simon, D.: Maxine: An approachable virtual machine for, and in, java. Tech. Rep. 2012-0098, Oracle Labs (2012)
40. Würthinger, T., Wimmer, C., Wöß, A., Stadler, L., Duboscq, G., Humer, C., Richards, G., Simon, D., Wolczko, M.: One vm to rule them all. In: International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (ONWARD'13) (2013)
41. Yang, X., Chen, Y., Eide, E., Regehr, J.: Finding and Understanding Bugs in C Compilers. In: Programming Language Design and Implementation. PLDI '11 (2011). https://doi.org/10.1145/1993498.1993532