Differential testing of simulation-based VM generators

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

Differential Testing of Simulation-Based VM Generators

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

Pierre Misse-Chanabier
Inria, Univ. Lille, CNRS, Centrale Lille, UMR 9189 - CRIStAL
pierre.misse-chanabier@inria.fr

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

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

Noury Bouraqadi
IMT Lille Douai, Institut Mines-Télécom, Univ. Lille, Centre for Digital Systems, F-59000
noury.bouraqadi@imt-lille-douai.fr

Luc Fabresse
IMT Lille Douai, Institut Mines-Télécom, Univ. Lille, Centre for Digital Systems, F-59000
luc.fabresse@imt-lille-douai.fr

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

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. Simulation-based VM generator frameworks allow one to write test cases on the simulation, however they do not ensure the correctness of the generated artifact due to the semantic gap between the environments.

In this article we propose Test Transmutation. It extends simulation-based VM generator frameworks to also generate simulation test cases and execute them on the generated VMs. It extends such frameworks to translate test cases and applies differential testing and non-semantic-preserving mutations. Test Transmutation detects bugs that are representative of typical VM modifications. Moreover, we apply it to a set of real test cases of the Pharo VM and find several issues. Our approach shows promising results to test simulation-based VM generator frameworks.

CCS CONCEPTS

• **Software and its engineering** Compilers; Software testing and debugging;

KEYWORDS

Testing, Virtual Machine, Code Mutation, Simulation

ACM Reference Format:
Pierre Misse-Chanabier, Guillermo Polito, Stéphane Ducasse, Noury Bouraqadi, Luc Fabresse, and Pablo Tesone. 2022. Differential Testing of Simulation-Based VM Generators: Automatic Detection of VM Generator Semantic Gaps Between Simulation and Generated VMs. In *The 37th ACM/SIGAPP Symposium on Applied Computing (SAC '22), April 25–29, 2022, Virtual Event, .* ACM, New York, NY, USA, Article 4, 4 pages. https://doi.org/10.1145/3477314.3507171

SAC '22, April 25–29, 2022, Virtual Event,

ACM ISBN 978-1-4503-8713-2/22/04.
https://doi.org/10.1145/3477314.3507171

1 INTRODUCTION AND PROBLEM

Testing and debugging Virtual Machines (VMs) is a laborious task without the proper tooling, which is aggravated when the VM targets multiple architectures [5]. To cope with the elevated cost of building language VMs, approaches such as *VM generation frameworks* [9, 11, 13, 15, 20, 25, 33], *Metacircular VMs* [28, 31, 32] and *simulation-based VM generators* [15, 20, 24, 25] were created.

Simulation-based VM generators 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 code depicted in Listing 1. The test cases show that this primitive works in both the simulation and generated VM. However if we remove the type annotation, the generated VM does not work anymore: it casts a float to an int, producing unexpected results which are not detected by simulation test cases. Similar problems have been informally reported for other simulation-based VM generator frameworks such as RPython [26].

```
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 cCode: [ receiver log ]
8       inSmalltalk: [ receiver ln ])
```

Listing 1: Excerpt of VM code showing specific simulation code and type annotations ignored during simulation.

Problem Statement. Using simulation-based VM generators creates a *semantic gap* [6] due to the differences between the simulated and generated VMs *e.g.,* typing, memory management, integer semantics, etc. The simulation environments is therefore not functionally equivalent [23] to the target production environments. We have observed that in current state-of-the-art simulation-based VM generators, testing only the simulated VMs is not enough.

Solution. We propose to extend simulation-based VM generator frameworks to also generate simulation test cases and execute them on the generated VMs. We compare the test results using differential testing [19] to validate that the simulated VM and the generated VM are working properly. Moreover, to minimize the impact of semantic

gaps on test cases we mutate them [16]. Test cases with different test results, mutated or not, are evidence of bugs.

We performed an empirical analysis by applying our approach to a subset of the Pharo VM test cases. Although this VM has been stable and industrially-used for several years, we found bugs on the VM generation toolchain and the simulation. Our analysis found promising results to test simulation-based VM generators.

2 TEST TRANSMUTATION

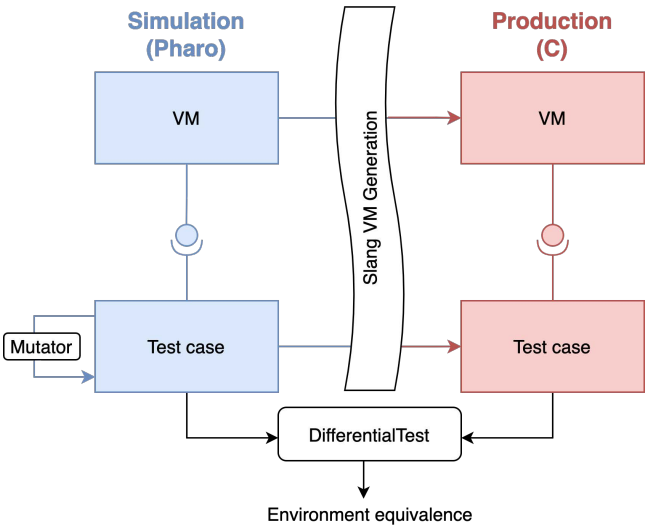

Figure 1: Test Transmutation overview. We generate test cases with the VM and compare their results through differential testing. Mutations increase the number and variety of test cases.

We propose Test Transmutation to generate the VM test cases with the VM (cf. Figure 1). We execute both simulated and generated test cases using differential testing [19] to detect execution differences. A core insight of the solution 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 are a success, 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 along with the VM 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 two challenges, we extend our approach with automatic non-semantic-preserving mutations [10]. Mutated test cases are expected to keep behaving similarly when executed in the simulated and the generated VMs. For example, if a mutation breaks a test case on the simulation, the mutated test case should also be broken when executed on the generated code. A non-equivalence of test results shows that there is 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 to compare test case results

2.1 Differential Testing of Test Cases

We check for functional equivalence of both simulated and generated VMs by applying differential testing [19] and validating whether a test case has *similar* results in both. We approached test result by implementing a heuristic-based oracle. 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: its execution, including the check for assertions, must complete without problems. We consider that a test case fails if anything prevents it from reaching the end of its execution, *i.e.,* in our case failures happen because of four different reasons: the VM generator rejects the program, the generated source code does not compile, an unexpected runtime error happens or an assertion check fails. Notice that we do not filter-out compilation errors, because they may indicate bugs in the VM generator itself.

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, we consider that the test case reveals a bug.

2.2 Test Case Variations with Non-Semantic-Preserving Mutations

Since we generate test cases from existing simulation test cases, the test cases suffer from semantic gaps and they all have poor variations in their results because they are maintained to pass. We tackle this issue by applying non-semantics-preserving mutations on the test cases. Mutations create more inputs allowing one to gain confidence in the correctness of the generation. 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. Also, if a test case is still passing in spite of a mutant, we expect the generated test case to be passing as well. 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.

3 PRELIMINARY RESULTS

3.1 Experimentation Platform

We applied Test Transmutation to the Virtual Machine of the Pharo language implementation. Pharo is an object-oriented dynamically-typed language from the Smalltalk tradition [7]. The Pharo VM is an industrial level Virtual Machine written in Pharo itself and generating a C version using a VM-specific generator called the Slang VM generator [15]. 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. Most

debugging and development happens in the simulation environment and is compiled to a machine executable VM [20, 25]. The Pharo VM has around 1000 unit test cases [24] written in the simulation environment, covering the bytecode interpreter, the language native functions, and the Just-In-Time compiler. These test cases are executed with different parameters configuring different word sizes (32/64 bits) and processor architectures (x86, x86-64, ARM32 and ARM64). Notice that since the VM is used in industry for several years, we expect that the VM is stable and that few bugs may be present in the source code.

To evaluate the capability of Test Transmutation to find bugs in existing test cases, we apply our approach on a subset of the existing VM test suite [1]. Since the full set of test cases are originally written in plain Pharo and not meant to be translated, their generation was not available out of the box: test cases use many idioms not supported by the Slang VM generator. In this article we focused on a subset of the available test cases, namely the memory management test cases. They cover the implementation of a generational scavenger collector and a mark-compact collector for older objects [30].

3.2 Prototype Results

Table 2 reports on the status of the compilation of the test cases. Currently, the 256 initial test cases are passing in the simulation. 238 of them are passing in the generated VM for Ubuntu 20.0 and x64. The remaining 16 test cases use more complex features not yet supported by our prototype implementation.

From the existing 256 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 for Ubuntu x64.

3.3 Result Analysis

Note that since the VM has been used industrially for several years now, we expect that the VM is stable and that few bugs may be present in the source code. However, in the current prototype implementation we detects three bugs with 23 different mutations.

Stack Allocation Simulation Issues. The simulation does not take into account that stack allocations are freed as soon as the allocating function returns. The tests cases were failing, causing memory access violations because of an implicit, required inlining.

Division Differences. In Pharo the division is computed by the method #/, which returns an exact result and #//, which truncates the result. In C however, both selector are translated to the / operator. The mutants break the simulated VM but not in the generated VM.

Runtime Assertion Differences. Runtime assertions check for invariants to eagerly prevent complex scenarios such as memory corruptions. On the one hand, invariant assertions in the VM stops the execution and fails tests. On the other hand, invariant assertions log an error and continue the execution. 21 mutants produce test cases passing in the generated VM but not in the simulated VM

4 RELATED WORK

Equivalence Modulo Input proposes to create functionally equivalent test input programs. Those test input programs are created by applying mutations on dead code [17, 18] or semantic-preserving mutations on live code [29]. Our solution applies instead non-semantic-preserving mutation on live code.

Many works uses automated mutations to guide test case generation [12, 14, 22]. Papadakis et al. [21] 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.

Several work have applied mutations on simulation-based models outside of Virtual Machines. Rutherford et al. [27] present an automated approach on top of distributed system simulations using mutations. Aichernig et al. [2–4] 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 Virtual Machines, and particularly, Virtual Machine simulation environments.

PharoJS [8] validates generation from Pharo to Javascript with application test cases. The Pharo environment execute the test case in both Pharo and Javascript environments. It executes the test case in Javascript by sending messages to the Javascript environment rather than compiling the test cases. Moreover PharoJS does not apply mutations.

5 CONCLUSION

This paper describes a novel technique to test simulation-based VM generator 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 production VMs. We apply non-semantic-preserving mutations to increment test variability and minimize the impact of semantic gaps introducing during test generation. We apply our prototype implementation on the Pharo VM executing it on real test cases. Our preliminary evaluation shows that our approach detects bugs and translation differences on a an existing VM that is stable and has many users. In the future we plan to investigate how Test Transmutation can be extended with VM specific mutations, test suite reduction and support testing of JIT compilers.

ACKNOWLEDGMENTS

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] M. Abdi, H. Rocha, and S. Demeyer. Test Amplification in the Pharo Smalltalk Ecosystem. In *International Workshop on Smalltalk Technologies (IWST'19)*, Aug. 2019.

[2] B. K. Aichernig, J. Auer, E. Jöbstl, R. Korošec, W. Krenn, R. Schlick, and B. V. Schmidt. Model-based mutation testing of an industrial measurement device. In M. Seidl and N. Tillmann, editors, *Tests and Proofs*, pages 1–19, Cham, 2014. Springer International Publishing.

[3] B. K. Aichernig, H. Brandl, E. Jöbstl, and W. Krenn. Efficient mutation killers in action. In *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, pages 120–129, 2011.

Category	Sub-Category	# Passing test cases in the Simulation	# Passing generated test cases	# of mutants execution	# of mutants detecting a bug
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; Number of passing test cases in both the simulation environment and in the generated VM; Number of mutated programs generated; Number of mutants that were able to detect a bug.

[4] B. K. Aichernig, H. Brandl, E. Jöbstl, W. Krenn, R. Schlick, and S. Tiran. Killing strategies for model-based mutation testing. *Softw. Test. Verif. Reliab.*, 25(8):716–748, Dec. 2015.

[5] 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.

[6] V. Besnard, M. Brun, P. Dhaussy, F. Jouault, D. Olivier, and C. Teodorov. Towards one Model Interpreter for Both Design and Deployment. In *Third International Workshop on Executable Modeling (EXE 2017)*, Sept. 2017.

[7] A. P. Black, S. Ducasse, O. Nierstrasz, D. Pollet, D. Cassou, and M. Denker. *Pharo by Example*. Square Bracket Associates, Kehrsatz, Switzerland, 2009.

[8] N. Bouraqadi and D. Mason. Mocks, Proxies, and Transpilation as Development Strategies for Web Development. In *Proceedings of the 11th edition of the International Workshop on Smalltalk Technologies*, IWST'16, pages 1–6. Association for Computing Machinery, Aug. 2016.

[9] K. Casey, D. Gregg, and M. A. Ertl. Tiger – an interpreter generation tool. In R. Bodik, editor, *Compiler Construction*, pages 246–249, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[10] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Program mutation: A new approach to program testing. *Infotech State of the Art Report, Software Testing*, 2(1979):107–126, 1979.

[11] M. A. Ertl and D. Gregg. Optimizing indirect branch prediction accuracy in virtual machine interpreters. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 278–288, 2003.

[12] G. Fraser and A. Zeller. Mutation-driven generation of unit tests and oracles. *IEEE Transactions on Software Engineering*, 38(2):278–292, 2012.

[13] D. Gregg and M. A. Ertl. A language and tool for generating efficient virtual machine interpreters. In *Domain-Specific Program Generation*, pages 196–215. Springer, 2004.

[14] M. Harman, Y. Jia, and W. B. Langdon. 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*, ESEC/FSE '11, page 212–222, New York, NY, USA, 2011. Association for Computing Machinery.

[15] 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. ACM Press, Nov. 1997.

[16] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser. 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*, pages 654–665, 2014.

[17] V. Le, M. Afshari, and Z. Su. Compiler validation via equivalence modulo inputs. In *Programming Language Design and Implementation*, PLDI '14, 2014.

[18] V. Le, C. Sun, and Z. Su. Finding deep compiler bugs via guided stochastic program mutation. *ACM SIGPLAN Notices*, 50:386–399, Oct. 2015.

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

[20] 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.

[21] M. Papadakis, C. Henard, M. Harman, Y. Jia, and Y. Le Traon. Threats to the validity of mutation-based test assessment. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, page 354–365, New York, NY, USA, 2016. Association for Computing Machinery.

[22] M. Papadakis and N. Malevris. Automatic mutation test case generation via dynamic symbolic execution. In *2010 IEEE 21st International Symposium on Software Reliability Engineering*, pages 121–130, 2010.

[23] S. Person, M. B. Dwyer, S. Elbaum, and C. S. Păsăreanu. Differential symbolic execution. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, SIGSOFT '08/FSE-16, pages 226–237. Association for Computing Machinery, Nov. 2008.

[24] 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 *MPLR '21, Germany*, Münster, Germany, Sept. 2021.

[25] A. Rigo and S. Pedroni. PyPy's approach to virtual machine construction. In *Proceedings of the 2006 conference on Dynamic languages symposium*, pages 944–953, New York, NY, USA, 2006. ACM.

[26] RPythonCommunity. Rpython documentation on test translation, 2016.

[27] M. Rutherford, A. Carzaniga, and A. Wolf. Evaluating test suites and adequacy criteria using simulation-based models of distributed systems. *Software Engineering, IEEE Transactions on*, 34:452–470, aug 2008.

[28] D. Simon, C. Cifuentes, D. Cleal, J. Daniels, and D. White. 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*, pages 78–88, New York, NY, USA, 2006. ACM Press.

[29] C. Sun, V. Le, and Z. Su. Finding compiler bugs via live code mutation. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, pages 849–863. Association for Computing Machinery, Oct. 2016.

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

[31] 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.

[32] C. Wimmer, M. Haupt, M. L. V. D. Vanter, M. Jordan, L. Daynes, and D. Simon. Maxine: An approachable virtual machine for, and in, java. Technical Report 2012-0098, Oracle Labs, 2012.

[33] T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. Wolczko. One vm to rule them all. In *International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward'13)*, 2013.