Université de Lille, Faculté des sciences et technologies

Département de formation doctorale en informatique **École doctorale MADIS Lille**
UFR IEEA

Testing a Virtual Machine Developed in a Simulation-based Virtual Machine Generator

Tester une Machine Virtuelle Développée Dans un Générateur de Machine Virtuelle Basé sur une Simulation.

THÈSE

présentée et soutenue publiquement le 16/12/2022

pour l'obtention du

Doctorat de l'Université de Lille

(spécialité informatique)

par

Pierre Misse-Chanabier

Composition du jury

Président :	Dr Alain Plantec	Professeur – Université de Bretagne Occidentale
Rapporteurs :	Dr Jannik Laval	Maître de Conference – Université Lumière Lyon 2
	Dr Gordana Milosavljević	Professeur – Université de Novi Sad
Examinateur :	Dr Gordana Rakić	Associate Professor – University of Novi Sad
Directeurs de thèse :	Dr Stéphane Ducasse	Directeur de Recherche – Cristal, INRIA Lille, Nord Europe
	Dr Noury Bouraqadi	Professeur – IMT Lille Douai

Centre de Recherche en Informatique, Signal et Automatique de Lille — UMR 9189
INRIA Lille - Nord Europe

Région Hauts-de-France

Université de Lille

informatiques mathématiques

RMod

Abstract

Virtual Machines (VMs) are modern programming language implementations. Testing and debugging a VM is a laborious task without the proper tooling. This is particularly true for VMs implementing garbage collection, Just in Time (JIT) compilation and interpreter optimisations. This situation is worsen when the VM builds and runs on multiple target processor architectures.

To ease the development of VMs, simulation-based VM generator frameworks were studied. Such frameworks allow developers to work on the VM in a simulation environment, and to generate the VM once it's ready. Developers leverage the simulation environment to ease not only the development process but also tooling and debugging the VM. However, it creates abstraction gaps between the simulated and generated VM. Therefore the execution of the simulated and the generated VM are not functionally equivalent.

In this thesis, we investigate how to test VMs developed in simulation-based VM generator framework. First, we investigate a methodology based on using multiple kinds of VM execution mode: Unit Testing, full-system simulation and real hardware execution. The methodology leverages the best properties of each execution mode. We leverage unit testing for its feedback-cycle speed and real hardware execution for its precision. Second, we propose *Test Transmutation* to extend simulation-based VM generator frameworks to co-generate the test cases with the VM. Co-generating the test cases with the VM allows developers to reuse the simulation test cases to apply unit testing on the generated VM. Results of the generated test cases are compared against the simulation test case results using differential testing to validate them. Finally, test cases are automatically mutated with non-semantic-preserving mutations. The mutated test case results should be equivalent. If they are not, a bug has been found. This allows us to automatically create more test cases to stress the test case generator. This improves our confidence in the test case generator as well as exploring the different paths of execution in the VM.

Our evaluation shows that we detect 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 VM developed in simulation-based VM generator frameworks.

Keywords: Virtual Machine, Testing, Simulation-Based Testing, Code Mutations

Résumé

Les Machines Virtuelles (MV) sont les implémentations des langages de programmation modernes. Tester et débugger une MV est une tâche laborieuse sans les outils appropriés. C'est particulièrement le cas pour des MV qui implémentent un ramasse miettes (Garbage collector) une compilation JIT (Just in Time compiler) et des optimisations de l'interpréteur. Cette situation empire lorsque la MV est compilée et exécutée sur plusieurs architectures de processeurs.

Pour faciliter le développement de MVs, les générateurs de MV basés sur une simulation ont été étudiés. Ce genre de frameworks permet aux développeurs de développer une MV dans un environnement de simulation et de générée la MV quand elle est prête. Les développeurs exploitent l'environnement de simulation pour faciliter non seulement le développement, mais aussi l'outillage et le débug. En revanche, cela créé des fossés d'abstraction entre la MV simulée et la MV générée. Par conséquent, l'exécution de MV simulée et de la MV générée ne sont pas fonctionnellement équivalents.

Dans cette thèse, nous étudions comment tester des MVs développées dans des générateurs de MV basés sur une simulation. Premièrement, nous étudions une méthodologie basées sur l'utilisation de multiple modes d'exécution de la MV: les tests unitaires,la simulation complète du système et l'exécution sur le matériel. Cette méthodologie exploite les meilleures propriétés de chaque mode d'exécution. Par exemple, nous exploitons les tests unitaires pour la vitesse du cycle de feedback et l'exécution sur le matériel pour sa précision d'exécution. Deuxièmement, nous proposons *Test Transmutation* pour étendre les générateurs de MV basés sur une simulation pour co-générer les tests unitaires avec la MV. Co-génerer les tests unitaires avec la MV permet aux développeurs de réutiliser les tests unitaires de la simulation pour tester la MV générée. Les résultats des tests unitaires sur la MV générée sont comparés avec les résultats de leurs exécutions sur la MV simulée en utilisant le différentiel de résultats pour les valider. Enfin, les tests unitaires sont automatiquement mutés avec des mutations ne préservant pas la sémantique initiale. Les résultats des tests unitaires mutés doivent être équivalents. Si ils ne le sont pas, un bug a été trouvé. Ceci permet de créer plus de tests unitaires pour stresser le générateur des tests unitaires. Cela permet aussi de gagner en confiance dans le générateur des tests unitaires et d'explorer différente route d'exécution du code de la MV.

Notre évaluation montre que nous détectons des bugs qui sont représentatifs des modifications typiques des développeurs sur la MV. Nous montrons l'efficacité de notre méthode en l'appliquant sur les tests unitaires de la MV Pharo. Ceci nous as permis de trouver de multiples problèmes qui n'étaient pas connus de l'équipe de développement la MV Pharo. Notre approche montre des résultats prometteurs pour tester des MV développées dans des générateurs de MV basés sur une simu-

lation.

Mots-clés: Machine virtuelle, Test, Test Basé sur une Simulation, Mutation de Code

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

Contents

1.1 Virtual Machines

Surprisingly unknown to their users, virtual machines are everywhere: from phones to web browsers or even servers. Everyone uses at least several virtual machines every day without realizing it. We describe in this section what is a virtual machine.

1.1.1 What is a Virtual Machine ?

A program is ultimately executed on a computer. A program is either executed directly by the computer's hardware or it is executed by some other program. A popular kind of program executing programs is a Language Virtual Machine.

Virtual machines provide an abstraction layer over hardware to execute code. It enables several features, we describe here a few of them.

Portability. Whenever a new processor architecture comes out, programs must be compiled to this new processor architecture. Writing processor-independent code is hard because there are so many small things that are processor dependent, particularly on low-level code. In the case of programming languages executed on a VM, only the VM needs to be compiled to the new processor architecture. Programs working on this VM however are ready to use whenever the compiled VM is available for the new architecture. When distributing a software, only one version of the program has to be provided rather than one for each processor architecture and operating system. The phrase "*Compile once, run everywhere*" is often used to describe this feature.

Managed Memory. Manual memory management is commonly considered a difficult and error-prone developer task. Many if not most language VMs implement a memory manager removing the developer from that responsibility. The memory that is detected to not be used anymore is reclaimed by the memory manager and reused for further memory allocations.

Dynamic Recompilation. Most of the compilation steps are done statically. Therefore it is done before the execution of the program. However, a lot of information is unknown at static time, and many optimizations are not applicable. Just in Time (JIT) compiler is another kind of compiler that re-compiles parts of a program during its execution to optimize it. It leverages dynamically obtained information to guide the dynamic compilation to optimize the code further.

Different Primitives. Some methods that require direct access to VM internals are typically so-called primitives. These methods either do not fit into the processor features or support a specific language. They are routines provided as part of the runtime, implemented in the implementation language of the VM. They are generally faster and are sometimes implemented as extensions of a VM to speed up some operations [Behnel 2011].

1.1.2 Main VM Components

Most VMs are made of multiple components. We give a short overview of the main components and their responsibilities. The VM not only executes code with the interpreter but does other operations such as memory garbage collection and JIT compilation.

Interpreter. The interpreter is the execution engine of the VM. It defines the semantics of the language. It takes some code as input and executes it. In some

VMs, there is no interpreter at all. The code is directly compiled to binary rather than being interpreted.

The Object Representation. In memory, there are only bytes. The VM manages those bytes. It writes and reads them. Therefore the VM also defines how to interpret those bytes to be able to work with them. For example, many object-oriented VMs describe every object with a header. A header contains multiple pieces of information that allow the VM to deal with this object correctly. A header may contain the size of the object, if the object is immutable or if it can be moved safely in another part of the memory.

Memory Manager. Manual memory management is considered to be a difficult and error-prone task for developers. VMs most often relieve developers of the responsibility of managing their memory by implementing a memory manager. The memory manager is a component that manages the memory. Particularly it implements how the objects are allocated in memory and how memory is reclaimed.

Garbage Collection. The idea of a Garbage collector is to reclaim memory that is not used by the application executed on the VM. This is achieved by defining at the VM level from where an object is accessible. For example, objects that are on the execution stack are accessible to the user, and therefore cannot be reclaimed.

A VM executing a program does not only execute the program. Garbage collection is an operation that may stop the execution of the program to collect unreachable objects. There exist multiple garbage collection algorithms. Some stop the execution of the program such as *Copy* garbage collection. Other such as *Reference Counting* garbage collection execute at the same time as the program.

Allocation Scheme. The Allocation Scheme defines where new objects are allocated in memory. It is heavily influenced by the garbage collector algorithm that is used. Particularly, in the case of a *copy garbage collector*, the next object is allocated at the end of the heap. This is because this kind of garbage collector compacts the memory every time it is triggered.

JIT Compiler. The JIT compiler is a compiler that is embedded in the VM and compiles code while executing a program. It is another component that may stop the execution of the program to execute like garbage collection. There are multiple levels of optimization for JITs. As in any compiler, a JIT is able to have multiple levels of optimization. For example, baseline JITs are the first level of JIT optimization, which compiles some code quickly and with little optimization. By keeping statistics on the executed code, it can be optimized further. This is usually applied to the code that is the most used, called **Hot code**. For example, given

some properties, speculative inliner [Béra 2016] are able to replace a message sent by the body of the method inside the sender to optimize a method further.

1.2 Virtual Machine Development

VMs are complex software. As they often require to do unsafe operations, they are developed in unsafe language [Blackburn 2004]. Therefore many industrial VM providers choose to develop in low-level languages such as C. The research community has investigated multiple approaches to ease VM development as we present below.

1.2.1 Challenges of Low-Level Development VMs

Ungar et al. [Ungar 2005] identify challenges of developing a VM in a low-level language by looking at a *Self* VM.

Behavior duplication. Some behavior is duplicated in multiple components: the runtime, the interpreter, and the JIT compiler [Holzle 1994]. They should all be semantically equivalent. Particularly, some works have focused on generating the JIT from an interpreter [Wimmer 2013, Bolz 2015] to reduce the multiplication of different implementations of the same behavior in VMs.

Debugging difficulties. When executing the VM in a low-level debugger such as GDB, developers rely on what is available, which is not much. Moreover, the representation of an object at the user level is usually easier to read than at the VM level. Works such as Maxine and Pharo [Kotselidis 2017, Polito 2021] have presented tools they used to ease debugging of their respective VMs. The Klein VM [Ungar 2005] uses its advanced reflective system to ease debugging.

Slow turnaround time. Ungar et al. [Ungar 2005] give the example that changing a single header results in at least 5 minutes for recompilation which is fairly similar to our experimental context. Simulation-based VM generator frameworks have a very quick turnaround time by developing the VM in a simulation and only recompiling after the main part of the development is achieved [Rigo 2006, Miranda 2018, Polito 2021]. The compilation cost is therefore significantly smaller overall.

1.2.2 How to Ease VM Development

Multiple approaches have attempted to ease the development cost. The three main approaches are:

- VM generation frameworks [Ingalls 1997, Miranda 2018, Ertl 2003b, Whaley 2005, Gregg 2004, Casey 2005, Rigo 2006, Würthinger 2013];
- Meta-Circular VMs [Alpern 2000, Wimmer 2012, Ungar 2005, Simon 2006];
- Simulation-based VM generator frameworks [Ingalls 1997, Miranda 2018, Rigo 2006].

Virtual Machine Generation Frameworks. To lower the cost of creating a VM, several works have focused on generating VMs components. Graal VM takes as input an AST interpreter and generates an environment suitable to run a hosted language. This allows for many languages to have an aggressive JIT at a significantly more reasonable cost. Similarly, RPython generates a JIT and garbage collector. The cost of implementing a new language is therefore only what RPython needs for that: an interpreter.

Meta-circular VMs. Meta-circular VMs are VMs that are able to interpret themselves. This implies that they are written in the same language they are executing, Such VMs allow developers to use a higher level of abstraction to develop the VM, as well as allow them to stay in the same environment. Such VMs exists for the Java language [Alpern 2000, Wimmer 2012, Simon 2006, Whaley 2005] and the Self language [Ungar 2005].

Simulation-based VM generator Frameworks. Similarly to Meta-Circular VMs, simulation-based VM generator frameworks allow developers to write VMs in a high-level language. Unlike Meta-Circular VMs the input language is restricted to allow for the generation of a VM code toward a lower level. The VM is executable in what is called a simulation environment. Once ready, the VM is generated and compiled to binary for efficient production execution.

The two main representatives of this approach in VM development are the OpenSmalltalk-VM [Ingalls 1997] and PyPy VM [Rigo 2006]. The OpenSmalltalk-VM is written in a sub-Smalltalk language called Slang, and the PyPy VM is written in a sub-Python language called Restricted Python (RPython). Both compile their input languages to C code and use the C compiler to produce the final VM binary.

Developing VMs in simulation-based VM generator frameworks has many advantages:

1. Execution in the simulation;
2. Abstraction over hardware;
3. Fast feedback;
4. Custom tooling;
5. Generation-specific optimization;
6. One code base defines multiple VM versions.

1.3 Problem Statements and Research Questions

Implementing an efficient execution engine for a new language is expensive. It requires consequent knowledge of low-level code and hardware. This thesis looks at the validation of VMs written in a simulation-based VM generator framework. simulation-based VM generator frameworks reduce the knowledge gap required to implement a new efficient execution engine.

Although simulation-based VM generator frameworks have many advantages, they also come with a few limitations. As with any kind of software, validation that this software is working is important. It is even more important given that VMs are the execution engine that executes many programs. In the case of simulation-based VM generator frameworks, a VM has multiple executable forms which should all work similarly. Developers work in the simulation environment and should be able to reliably count on the generator to generate code semantically equivalent to the one she wrote. State of the art testing for VM developed in simulation-based VM generator frameworks is to test the simulated VM [Béra 2016, Bolz-Tereick 2022, Polito 2021]. The generated artifact does not undergo unit testing but rather executes programs as a validation. However, as we show in this thesis, the generated VM sometimes does not have the same semantics as the simulated one. Divergences in the semantics of the VMs are called semantic gaps [Hein 2010, Besnard 2017].

Problem: Semantic Gaps. Most of the VM testing is done in the simulation. However multiple semantics gaps exist between the simulated and generated VMs, which may introduce bugs in the generated VM that are undetectable in the simulated VM.

We formulate the following research question.

RQ1 How can we leverage the simulation environment to test the VM?

RQ2 Can we reuse part of the simulation environment testing to test the generated VM?

1.4 Contributions

The main contribution of this thesis is an approach to test VMs developed in simulation-based VM generator frameworks. It includes:

1. A simulation-based testing methodology to test a VM developed in a simulation-based VM generator framework (Chapter 4).

2. Test Transmutation, an approach extending the simulation-based test methodology to apply unit testing on the generated VM based on test co-generated with the VM test case mutation (Chapters 5 and 6).

1.5 Publications

The list of papers published in the context of the thesis is listed below in chronological order:

1. Parts of this paper [Misse-Chanabier 2019] were used in this thesis.
   Pierre Misse-Chanabier, Vincent Aranega, Guillermo Polito and Stéphane Ducasse. *Illicium A modular transpilation toolchain from Pharo to C*. In International workshop of Smalltalk Technologies (IWST'19), Köln, Germany, August 2019.

2. Parts of this paper [Polito 2021] were used in this thesis.
   Guillermo Polito, Pablo Tesone, Stéphane Ducasse, Luc Fabresse, Théo Rogliano, Pierre Misse-Chanabier and Carolina Hernandez 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, September 2021.

3. Parts of this paper [Misse-Chanabier 2022b] were used in this thesis.
   Pierre Misse-Chanabier, Guillermo Polito, Stéphane Ducasse, Noury Bouraqadi, Luc Fabresse and Pablo Tesone. *Differential testing of simulation-based VM generators: automatic detection of VM generator semantic gaps between simulation and generated VMs*. In Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing (SAC'22), pages 1280–1283, 2022.

4. Parts of this paper [Misse-Chanabier 2022a] were used in this thesis.
   Pierre Misse-Chanabier, Guillermo Polito, Noury Bouraqadi, Stéphane Ducasse, Luc Fabresse and Pablo Tesone. *Differential Testing of Simulation-Based Virtual Machine Generators*. In International Conference on Software and Software Reuse (ICSR'22), pages 103–119. Springer, 2022.

5. This paper [Misse-Chanabier 2022c] was **not** used in this thesis. Pierre Misse-Chanabier and Théo Rogliano. *Ease Virtual Machine Level Tooling with Language Level Ordinary Object Pointers*. In Proceedings of the 14th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (VMIL'22), 2022.

1.6 Dissertation Outline

The thesis is organized as follows:

Chapter 2: Testing Language Implementation in the Literature describes the state of the art related to this thesis and how it helped us devise our solution.

Chapter 3: Experimental Context: The Pharo VM present the Pharo language and its VM.

Chapter 4: Simulation-Based VM Testing presents a methodology we devised to test VM developed in simulation-based VM generator frameworks.

Chapter 5: Test Transmutation presents an approach to test the generated VM for VM developed in simulation-based VM generator frameworks.

Chapter 6: Test Transmutation Validation presents the validation of Test Transmutation.

Chapter 7: Conclusion and Future Works summarize each chapter of this thesis before concluding and presenting future works.

CHAPTER 2

Testing Language Implementations in the Literature

Contents

This chapter describes the problems and their solutions in the current state of the art. We highlight two well-known problems: the **oracle** problem and the **test input** problem. (1) The oracle problem, or how to automatically determine whether a given input matches the expected semantics, is solved with two main approaches: **differential** or **metamorphic** testing. (2) The test input problem, or how to automatically generate relevant test inputs, is solved with three main approaches: **hand-written programs**, **test program generation**, and **test program mutations**. (3) Moreover, we also investigate simulation-based testing in other domains.

We conclude from other simulation-based approaches that testing only the simulation is not enough. We started by writing handwritten test cases on the simulated VM. We have decided that the best approach for our research was differential testing with a cross-environment strategy and reusing handwritten test inputs and generating them to C. Moreover, the handwritten test inputs are also mutated to test the test case generator.

2.1 Introduction

The holy grail of the validation field is to use mathematical proofs that the program matches its specification. This was achieved for the C– language, a sub-set of the C language [Leroy 2006]. However, this requires a significant investment which most language implementations cannot provide. This is worsened by the constant evolution of programming languages.

Testing VMs is an active research topic [Béra 2016, Polito 2022a]. In this dissertation, we focus on the testing of VM developed in simulation-based VM generator [Rigo 2006, Miranda 2018, Polito 2021, Simon 2006]. Testing language implementations have several challenges [Chen 2020]. We present the two that are relevant to this state of the art: the **oracle** problem (Section 2.2.1) and the **test input** problem (Section 2.2.2).

VM testing research [Béra 2016,Polito 2022a,Chen 2016,Chen 2019a,Lima 2020, Ye 2021] uses **differential testing**. Differential testing was initially devised to solve the oracle problem for compilers [McKeeman 1998] (Section 2.3.1). Another technique used to test compilers is called **metamorphic testing** [Chen 1998](Section 2.3.2). Moreover, simulation-based VM generator frameworks leverage *simulation environment* to develop VMs. Therefore we also investigate and present research works that leverage simulation environments for development purposes outside of VM development (Section 2.6).

The main takeaways are:

- VM testing research has mostly focused on the test input generation problems which require an oracle. However, not every language has multiple implementations available to create an oracle;
- VM testing mostly uses black box approaches and test programs as test inputs;
- Simulation-based testing approaches use native execution when possible.

2.2 Challenges of Testing Language Implementations

Testing programming language implementations is an active research topic. It has multiple important research focuses [Chen 2020]. We present in the following the two challenges that are relevant to this state of the art: the **oracle** problem and the **test input** problem.

2.2.1 The Oracle Problem

Let's take an example to drive this explanation.

$$1 + 1 * 2$$

A language may define the semantics of this mathematical expression in different ways.

For example, evaluating this expression in Python should evaluate $1 + (1 * 2)$ and return 3. Python chooses to implement the mathematical priority of operators. However, evaluating this expression in Pharo should evaluate $(1 + 1) * 2$ and return 4. Pharo uses message-sending to perform mathematical operations. It, therefore, uses message-sending priorities instead of implementing a particular rule for mathematical operations.

Language developers have to define the language semantics *e.g.* how the language behaves for a given input. Then they have to match these semantics in their implementation so their users know how to write code and have the expected behavior.

The **oracle** problem is the problem of automatically determining whether a given input matches the expected semantics. The two main solutions to the **oracle** problem are **differential** testing (Section 2.3.1) and **metamorphic** testing (Section 2.3.2)

2.2.2 The Test Input Problem

Multiple academic works have been able to find bugs in compilers that have been used in production for decades. This is due to the enormous space of possible inputs for a programming language implementation. Therefore many researchers have investigated how to generate millions of test inputs to check whether a language implementation behaves as expected. A test input is simply some code that is fed to the language implementation. However, it requires an **oracle**, a way to know if a given code has the expected behavior.

Chen et al. [Chen 2020] identifies three challenges for test input creation:

Validity of test programs. Language implementations go through several layers to execute a program (parsing, compilation, execution). Invalid programs

are often discarded in the early layers. Their usage therefore fairly limited. Creating a valid program that exercises multiple layers of the compiler is difficult.

Diversity of inputs. Creating test programs aims at exercising different parts of the language implementation. Two test programs exercising the same part of the language implementation do not give more pieces of information on whether the compiler behaves properly.

Specificity requirements. The requirements of the test input depend on which part of the language implementation is tested. Particularly, when testing whether a language implementation rejects invalid syntactic inputs, the program is not required to be valid. Therefore the requirements for test input are lower.

The main approaches in the literature are of three kinds:

- Handwriting test programs (Section 2.4.1);
- Test program generation (Section 2.4.2);
- Test program mutation (Section 2.4.3).

Now that we have seen the two challenges, we present the current solutions.

2.3 Solutions to the Oracle Problem

The **oracle** problem is the problem of automatically determining whether a given input matches the expected semantics according to the language developer.

The two main approaches solving the oracle challenge for language implementation are:

Differential testing (Section 2.3.1) The key insight of **differential testing** is to feed two or more language implementations an input and compare the results.

Metamorphic testing (Section 2.3.2) The key insight of **metamorphic testing** is to feed one language implementation two semantically equivalent test inputs and compare the results.

2.3.1 Differential Testing

Differential testing is a software testing technique introduced by McKeeman [McKeeman 1998] that is widely used to test VMs and compilers. The key insight of differential testing is that two implementations of the same specification should be observably functionally equivalent (Figure 2.1). Basically, it is comparing multiple implementations of a program. It feeds the same input to each implementation and then compares the results. If there is a reference implementation that is considered to be always right, any implementation that disagrees with the reference implementation contains a bug. If no reference implementation is available, the bug may be in any of the implementations.

Since its initial description, multiple strategies have been applied. We describe the main strategies that have been applied: **Cross-Compiler** strategy, **Cross-Optimization** strategy, and **Cross-Version** strategy.

Cross-Implementation Strategy. Cross-implementation strategy is also called Cross-compiler strategy in Compiler literature [Chen 2020]. The cross-implementation strategy leverages multiple implementations of the same specification. It consists in feeding multiple language implementations with the same input and comparing their results (Figure 2.1).

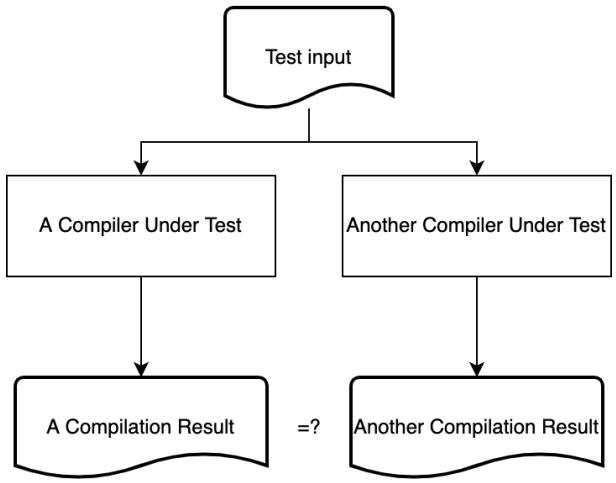

Figure 2.1: **Differential Testing**
The key insight is that the same test input is fed to two different language implementations. The results of the compilation or the results of the execution of the compiled test input are then compared.

This strategy is the one initially described by McKeeman [McKeeman 1998] and has been many times since [Yang 2011, Sheridan 2007, Ofenbeck 2016]. That is also primarily the strategy that is used by the VM testing research. Differential

testing has been applied to validate Java VMs [Sirer 1999,Chen 2016,Chen 2019a], Javascript VMs [Ye 2021, Lima 2020] and a Python VM [Bolz-Tereick 2022].

Cross-Optimization Strategy. Cross-Optimization Strategy (Figure 2.2) uses the hypothesis that one compiler compiling a given program with different optimization should output functionally equivalent code.

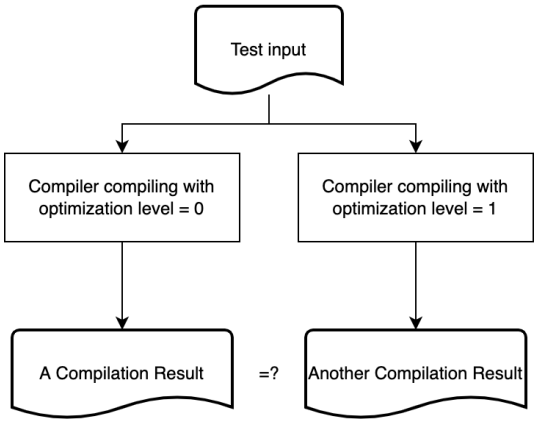

Figure 2.2: **Cross-Optimization Strategy**
Cross-Optimization Strategy uses the hypothesis that one compiler compiling a given program with different levels of optimization should output functionally equivalent code.

An approach to validate the deoptimizer of an optimizing JIT compiler was proposed by Bera et al [Béra 2016]. Rather than comparing different implementations of the full execution engine, they compare the code generated at two levels of optimizations of a single JIT compiler. They use an abstract interpretation of both the optimized and deoptimized code. They then compare the stacks of execution from the abstract interpretation. This allows them to turn any executable method into a test case.

Polito et al. propose a concolic approach to test the Pharo VM primitive functions [Polito 2022a]. Primitive functions are interpreter-level functions that are available to the code that is executed on the VM. Because of how the execution engine works, each primitive has two versions. One for the interpreter and another one for the JIT compiler. By leveraging these different implementations of the primitives, they are able to use differential testing between the interpreter and JIT versions of the primitive.

Cross-Version Strategy. Cross-Version Strategy (Figure 2.3) compares the compilation results across multiple versions of the same compiler [Sun 2016a].

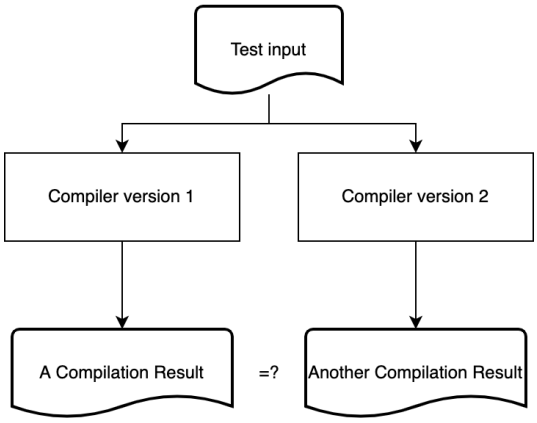

Figure 2.3: **Cross-Version Strategy**
Cross-version strategy compares the result of the compilation by two versions of a given compiler.

2.3.2 Metamorphic Testing

Metamorphic testing is the other popular basis to solve the oracle problem [Chen 1998]. The key insight of metamorphic testing relies on constructing a metamorphic relation. A metamorphic relation specifies how a particular change affects the output (Figure 2.4). A common example of a metamorphic relation is $sin(x) = sin(\pi - x)$. By definition, evaluating $sin(x)$ and $sin(\pi - x)$ should yield the same result. In execution engine testing, metamorphic relations are either characterized as **equivalent relations** or **equivalent modulo input relations** (EMI) (Section 2.4.3). Compiling both the initial and mutated test input should yield similar results. If the output of the compiler provides different results, a bug has been uncovered.

```
1 void initialOrionFunction(){
2   if ( false ) { return 1 + 1; }
3   return 2 + 1;
4 }
5
6 void mutatedOrionFunction(){
7   /* Orion removed dead code here */
8   return 2 + 1;
9 }
```

Listing 2.1: Example of a mutated code using Orion. It removes dead code parts from the test input.

Particularly, the most important work on metamorphic testing in compiler testing is called Equivalence Modulo Input (EMI) [Le 2014]. The first instance of EMI, named Orion, creates test inputs by removing dead code, portions of code

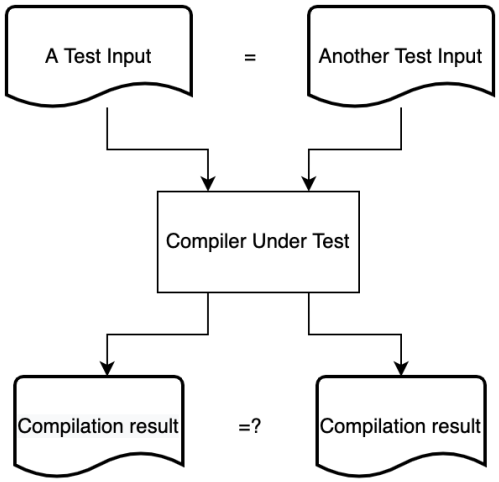

Figure 2.4: **Metamorphic Testing.**
An example of metamorphic relation is to apply a semantics-preserving mutation on the test input.t It feeds two semantically equivalent test inputs to one compiler. Both the initial and semantically equivalent results should be functionally equivalent. If the output of the compiler provides different results, a bug has been uncovered.

that are not executed (Listing 2.1).

2.4 Solutions to the Test Input Problem

The test input problem is how to construct a test program to exercise the language implementation features.

The main approaches in the literature are of three kinds:

- Handwriting test programs (Section 2.4.1);
- Test program generation (Section 2.4.2);
- Test program mutation (Section 2.4.3).

2.4.1 Hand Writing test programs

Handwriting test program has been around since there has been language implementation to test. Multiple language implementations have public test suites available. For example, this is the case of the LLVM [LLVM] and GCC [GCC] compilers with the infamous torture tests. In VMs, the OpenJDK provides a test suite as well [OpenJDK]. There are also external conformance test suites that have been

developed independently to a language implementation such as The Plum Hall Validation Suite [PlumHall] for C and Test262 [ECMAScript] for Javascript.

Particularly, PyPy describes how they test their VM written in a simulation-based VM generator [Rigo 2006]. PyPy (Python in Python) is a Python VM written in Restricted Python (RPython). RPython is both the name of the simulation-based VM generator framework and the name of the language that it uses. RPython is a subset of the Python language. It is also valid Python code that is executable by a Python VM.

PyPy describes two kinds of VM testing: **interpreter level test cases** and application **level test cases** [Bolz-Tereick 2022].

Interpreter level test cases are unit test cases written in Python and executed on the simulated VM. The simulated VM is executed by a Python VM. These unit test cases call methods of the simulated VM written in RPython (Listing 2.2). This allows developers to test the simulated VM before the generation step. This is particularly important because the generation step in RPython is time-consuming [Rigo 2006].

```
1 def testIntValue(vm):
2   integerObject = vm.IntObjectOf(42)
3   integerObjectHash = integerObject.hash()
4   assert integerObjectHash.intvalue == 42
```

Listing 2.2: Example of an interpreter-level test case. This test case is executed in the simulation on the PyPy VM. It is creating a VM integer object. It is then checking that the value of this VM integer object is the expected one.

Application level test cases. Application-level test cases are equivalent to test programs in the literature. The test program is fed to the VM as regular input. The VM interprets the test input. Application-level test cases are executed on both the simulated and generated VM. The PyPy VM uses *self-checking programs test program* (Listing 2.3). These are test programs that contain assertions. Their execution results are therefore whether the test passed or not. It also uses test programs from the CPython VM test suite. Those test cases are fed to both CPython and PyPy implementations. They validate that the test program behaves as expected with cross-implementation differential testing as an oracle.

```
1 def testIntValue():
2   assert hash(42) == 42
```

Listing 2.3: Example of self-checking application level test case. The PyPy VM under test takes this test program as input. It is checking that the value of the hash of an integer equals the initial integer.

RPython informally describes that developers may have to fix a few typing issues after the interpreter generation to C [RPythonCommunity 2016]. The unit

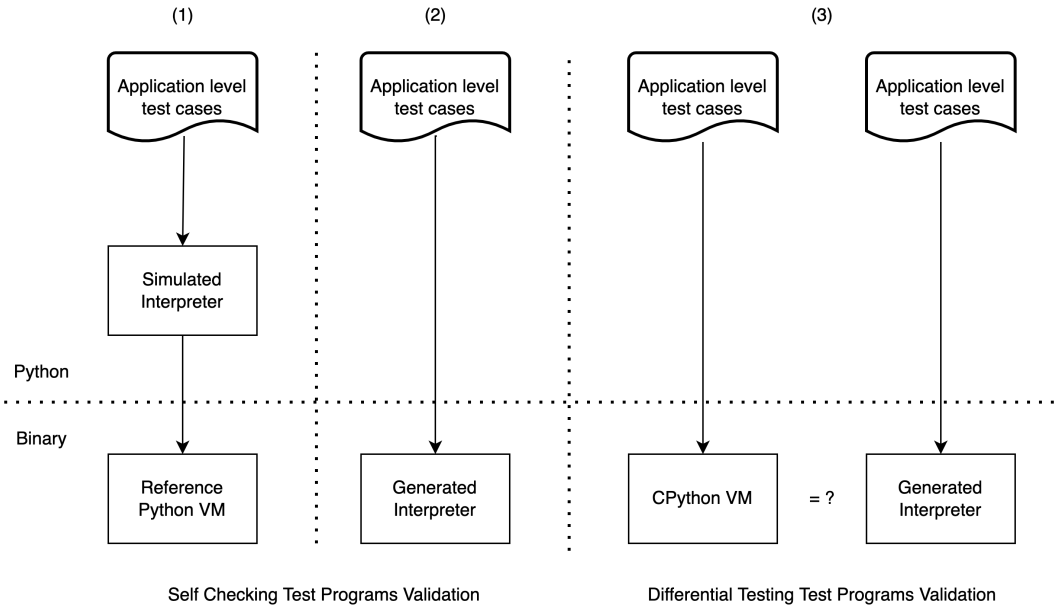

Figure 2.5: Application level test cases on the PyPy VM. The test programs are either self-checking or regular programs. If the test input is self-checking, it is fed to either the simulated or generated interpreter (cases 1 and 2) and validated by the assertion it contains. If the test input behavior is unknown, it is compared between the CPython VM and the PyPy-generated VM (case 3).

test cases are written in pure Python, and cannot be generated to C. They are not executed on the generated interpreter. Therefore the only feedback of whether the interpreter is working comes from the application-level test cases which are more difficult to debug.

However, few academic works have taken interest in it, as it requires significant effort. The main scientific contributions have investigated specific Fortran compiler features such as loop vectorization [Callahan 1988] and detection of loop parallelisation opportunities [Dongarra 1991]. Finally, Wolf et al. [Wolf 2016] experiment with manually translating a language specification in test programs.

2.4.2 Test program generation

Test program generation is mainly approached by grammar-based techniques. First, grammar-based approaches take a grammar as input and generate test programs. However, only generating a program without taking care of the context of the program only tests the first steps of the compilation. To generate programs that test the steps after the parsing, multiple grammar-based approaches were studied (Section 2.4.2.1). Second, a few other approaches have been applied but do not fit in either Grammar-based nor Mutation (Section 2.4.2.2).

2.4.2.1 Grammar-Based Approaches

Grammar-based approaches take as input a grammar and generate test programs [Purdom 1972]. However, only generating a program without taking care of the context of the program only tests the first steps of the compilation, such as the parsing. More complex approaches were studied to generate programs that test the following steps.

Grammar-based approaches are of two main categories: grammar-directed approaches and grammar-aided approaches. Also, Attribute Grammar-directed approaches generates test programs solely with the grammar as input. Grammar-aided approaches combine a grammar as input with other heuristics, such as starting from a code template and filling it up with the help of a grammar.

Grammar-directed approaches. Grammar-directed approaches generate test programs solely with the grammar as input. **Affixes-Grammars** and **Attribute-Grammars** use similar concepts [Koster 1991]. They both increase context-free grammar with context-sensitive information. For example, when an identifier is found by the parser, a rule for this identifier is added to the grammar. Introduced by Knuth [Knuth 1968] and first described to help test compiler later [Duncan 1981], it has since then been applied on Pascal [Burgess 1986] and Fortran [Burgess 1996]. Particularly, [Burgess 1996] improves upon [Burgess 1986] by adding the possibility of defining weights on grammar rule usage, and the generation of self-checking test programs, *e.g.* test programs containing assertion checks. Even more recently such a grammar-based solution was approached by training a recurrent neural network [Amodio 2017].

W-Grammars rely on a grammar and a meta grammar. The role of the meta grammar is to generate the terminal symbols for the grammar, which are context-sensitive symbols, first used in the context of ALGOL68 [vanWijngaarden 1965]. For example, the meta grammar generates variables [Bazzichi 1982]. Rather than using directly a grammar or a subset of a grammar, studies [Zelenov 2007, Zelenov 2003] use a model of a grammar and generate a restricted grammar with model transformation. Another approach is to use small generators [Lindig 2005a, Lindig 2005b] with the initial grammar, which generates only a small part of the test program.

Grammar-aided approaches. Grammar-aided approaches combine a grammar as input with other heuristics, such as starting from a code template and filling it up with the help of a grammar.

The oldest academic work we found on VM testing uses this approach to generate test programs [Sirer 1999]. They use a template that they fill with parts of a grammar. Moreover, they generate self-checking test programs. The most prominent work in grammar-aided approaches is CSmith [Yang 2011]. CSmith generates

diverse valid inputs. It uses complex heuristics to avoid undefined behaviors. The code fragment is generated only if a number of safety checks all pass. CSmith has inspired many works that have used it as a seed test program for mutations (Section 2.4.3.

It has also been adapted to generate concurrent C/C++ programs [Morisset 2013] and for OpenCL [Lidbury 2015]. Other approaches generate configurations for CSmith rather than modifying it [Groce 2012, Alipour 2016, Chen 2019b]. CSmith configurations select which features to use in the generated program. Swarm testing [Groce 2012] creates many configurations of CSmith. It has been utilized further to generate configurations based on historical data [Chen 2019b] and directed by previous results [Alipour 2016].

Rather than starting from a placeholder and filling it up with code snipper, Boujarwah et al. [Boujarwah 1999] start by generating small code snippets and then generate the necessary code to make it semantically correct *e.g.* variable declarations. Finally, rather than using hardcoded test programs as an initial template, Holler et al. [Holler 2012] take real-world programs and replace parts of them with grammar-driven generated code snippets.

2.4.2.2 Other Approaches

Other approaches have been investigated by academia to generate test programs.

Berry et al. [Berry 1983] take a statistical approach to create test programs that use features that are used by developers. They generate code by using hardcoded code snippets.

Test program generation has also been approached through the use of orthogonal lattice square which has the advantage of not generating duplicated test programs [Mandl 1985].

The origin of CSmith, RandProg, was created to test the *volatile* and *const* C keywords [Eide 2008]. It was achieved by having a predefined program, and randomly adding the keywords in front of the variable declarations.

Another approach tests C arithmetic by using an existing program and constructing random arithmetic expressions [Nagai 2012] with random operators, global variables, and local variables. They generate self-checking test programs that compare the runtime value of the expression with the statically computed value.

A type-centric approach was applied to Haskell [Pałka 2011]. It expands nonterminal terms into well-typed Haskell expressions. Another type-centric approach expresses type requirements using constraint logic programming [Dewey 2015]. The code generation is spread between multiple small generators. A refinement of this approach [Midtgaard 2017] was used to specifically test the evaluation order of function parameters. The evaluation order of function parameter is unspecified behavior in C for example [Committee 2007]. This approach also uses heuristics

to avoid undefined behaviors.

Some generators using templates and input use configurations to test Ada compilers [Austin 1991] and Java classes [Yoshikawa 2003]. It has also been used to test compiler optimizations specified as temporal logic formulas [Zhao 2009].

To test the APL compiler, Ching et al. [Ching 1993] use real-world applications. During generation, some symbols from a template are replaced with built-in APL functions. Kalinov et al. [Kalinov 2003a, Kalinov 2003b] test an mpC, a parallel compiler. They take as input an expression that is used as a seed to fill up a template.

Another approach uses skeletal program enumeration [Zhang 2017]. Given a program with holes, the approach exhaustively fills the holes with unique combinations of parameters.

Recent advances in machine learning allowed other test program generation techniques. *Treefuzz* [Patra 2016] is a recent machine learning approach that is testing Javascript compilers. This work trains on abstract syntax trees by learning which kinds of nodes are typically inside a particular kind of node. Lastly, an approach creates a grammar by learning which kind of inputs a black box language implementation accepts. The grammar is generated to support an incremental subset of the language.

Rather than generating whole test programs, Polito et al. [Polito 2022a] generates only the inputs to predefined programs. They approach testing of their implementations of the interpreter primitive and JIT primitives with concolic testing [Godefroid 2005, Sen 2005, Qu 2011]. They then compare the results of the execution of each version. This is a cross-optimisation differential testing approach.

2.4.3 Test Program Mutation

Rather than generating test programs ex-nihilo, some approaches take existing programs and modify them automatically. This approach is called **mutation** or **fuzzing**. The input to the mutator is called the **seed**. The mutator may create zero, one, or multiple inputs from a given seed. The seed is any program that is valid. Particularly the seeds may come from real programs, or test programs generated from tools such as the ones presented in the previous Section (Section 2.4.2).

The mutations are either preserving the semantics of the seed, called **semantics-preserving mutations**, or modifying the semantics of the seed called **non-semantics-preserving mutations**.

Semantics-preserving mutation approaches. The most important work on sem antics-preserving mutations in compiler testing are Equivalence Modulo Input (EMI) [Le 2014, Le 2015, Sun 2016b]. EMI proposes to create functionally equivalent test input programs modulo a given set of inputs to this program. This is achieved by

modifying test inputs without changing their semantics. A simple example of mutated test input would be to add a space at the start of a C file. Both the initial and mutated test inputs should be functionally equivalent. If the output of the compiler provides different results, a bug has been uncovered. This works under the assumption that the initial program does not contain undefined behavior. CSmith generates test inputs free of undefined behaviors [Yang 2011], it is therefore used as initial test inputs for EMI mutations. The test programs are validated with metamorphic testing (Section 2.3.2).

EMI is declined in three strategies:

- Orion [Le 2014] creates test inputs by removing dead code, portions of code that are not executed (Listing 2.4).
- Athena [Le 2015] creates test inputs by inserting code in the dead code portions of the test inputs (Listing 2.5).
- Hermes [Sun 2016b] creates test inputs by inserting semantics-preserving code in both dead and live code (Listing 2.6).

```
1  void initialOrionFunction(){
2    if ( false ) { return 1 + 1; }
3    return 2 + 1;
4  }
5
6  void mutatedOrionFunction(){
7    /* Orion removed dead code here */
8    return 2 + 1;
9  }
```

Listing 2.4: Example of a mutated code using Orion. It removes dead code parts from the test input.

```
1  void initialAthenaFunction(){
2    if ( false ) { return 1 + 1; }
3    return 2 + 1;
4  }
5
6  void mutatedAthenaFunction(){
7    if ( false ) {
8      if ( true ) { 1.0 + 1; } /* Athena inserted code here, in a dead code portion */
9      return 1 + 1;
10     }
11   return 2 + 1;
12 }
```

Listing 2.5: Example of a mutated code using Athena. It inserts code in the dead code portion of the test input.

```
1  void initialHermesFunction(){
2    if ( false ) { return 1 + 1; }
3    return 2 + 1;
4  }
5
6  void mutatedHermesFunction(){
7    if ( false ) { return 1 + 1; }
8    if ( true ) { 1.0 + 1; } /* Hermes inserted code here, in a live code portion */
9    return 2 + 1;
10 }
```

Listing 2.6: Example of a mutated code using Hermes. It inserts semantic-preserving code in both the dead and live portions of the test input.

Diverse strategies of EMI have been successfully applied in other cases such as testing of OpenGL compilers [Donaldson 2016] as well as graphics shader compilers [Donaldson 2017].

Non-semantics-preserving mutations. A non-semantics-preserving mutation could change a literal in an if's condition (Listing 2.7). The path of execution is altered, and the program has a different semantic.

```
1  void initialFunction(){
2    if ( true )
3      { return 1; }
4    else
5      { return 2; }
6  }
7
8  void mutatedFunction(){
9    // Change the true literal to the literal false
10   if ( false )
11     { return 1; }
12   else
13     { return 2; }
14 }
```

Listing 2.7: Example of non-semantic-preserving code. The mutation replaces the literal true with the literal false in the condition of the if statement, changing the semantics of the program.

First, Nagai et al. propose an extension of their grammar-directed approach to generate complex arithmetic expressions [Nagai 2012] with non-semantics-preserving mutations [Nagai 2014]. They take care of not generating expressions containing undefined behavior.

Classfuzz [Chen 2016] proposes to use coverage information to direct the fuzzing of test programs to test bytecode verifiers. The fuzzer is directed by using the

Markov Chain Monte Carlo (MCMC) sampling algorithm. Moreover, the mutated test input is first executed on a reference JVM. Only mutated test inputs exhibiting unique execution paths are executed in other JVMs to apply differential testing.

Classming [Chen 2019c] focuses on detecting deep compiler bugs. They consider that a test program reaches the "deep compiler" when they pass the bytecode verifier. This purpose required many executable mutants that were able to pass the bytecode verifier to test the execution engine. For that purpose, they start from existing Java projects which they mutate using and also apply MCMC sampling.

Javascript VMs testing has also been approached with mutations. Holler et al. [Holler 2012] describe *LangFuzz*. They create a code fragments pool by parsing a sample of programs. Test programs are then mutated by replacing parts of them with those code fragments of similar types.

Javascript has been studied with a fuzzer approach as well with Comfort [Ye 2021]. The Javascript execution engine implementations are directed by the ECMAScript standard. Implementing the whole standard is a complex task. Rather than looking for edge-cases, they search for standard conformance bugs. This is achieved by using differential testing on ten major Javascript execution environments. The test inputs are created with a deep learning approach. Then they apply fuzzing on it.

Rather than generating test inputs ex nihilo, test transplantation [Lima 2020] gathers test inputs from each Javascript implementation to execute them on all of the implementations available. This includes not only VMs implementation but also interpreters such as Duktape [Vaarala 2013] or test cases mined from issue trackers. They take test inputs passing on every Javascript execution engine and apply fuzzing on the test inputs.

2.5 Other Simulation-Based Testing

We also take a look at how simulation-based solutions approach testing outside of language implementations. Particularly we investigate how they deal with the fact that they are in fact in a simulation rather than native execution in their testing.

A **simulation** imitates a system. The imitation of the system is implemented to a degree varying from being identical to very loose. Using a simulation usually offers different properties from the initial implementation. For example, it may offer more abstraction and control over the flow of execution or debugging properties. It may also have some costs such as a loss of precision.

Process and system VMs. *Process* VMs simulate the CPU and *System* VMs simulate operating systems rather than languages. Both of these kinds of VM simulate the CPU. However, to improve performance, they execute code natively on the CPU as much as possible and use the simulation as little as possible. Martignoni

et al. [Martignoni 2010] test multiples existing Process and System VMs. In their case, the ideal reference implementation is obvious *i.e.* the physical CPU is always right. However, using the CPU directly is not possible. For example, if the CPU goes into an infinite loop, there is no way to reclaim control. They have to use a hardware-assisted virtual machine. They wrote 65 test cases and then applied fuzzing on them to generate many more inputs. They validate these inputs by using a cross-version differential testing approach. Moreover, they execute the test cases for each architecture, which enables the use of cross-architecture differential testing.

Phone application. Another field that massively uses simulation-based testing, is phone application development. The hardware is extremely diverse, as well as the kind of sensors making it difficult to test everything properly at the simulation level. Multiple services now offer cloud-based real hardware testing because simulation testing is considered to be insufficient [Prathibhan 2014, Starov 2015, Vilkomir 2015]. This allows phone application developers to automatically test their projects in parallel on multiple real hardware devices.

Network protocols. Network becoming increasingly complicated, simulation-based testing has been a logical solution to simplify testing. Particularly, research works [Zarrad 2017, Alsmadi 2020] have attempted to improve the confidence of developers in network simulators with mutation testing.

Cross-language development. PharoJS [Bouraqadi 2016] is a tool allowing developers to write Javascript for web and mobile applications. Developers express the code in Pharo, transpile it to Javascript and execute it in browsers. To validate applications written in PharoJS, it uses unit test cases on both the Pharo and Javascript environments. This is achieved by sending the Javascript environment commands of what to do and queries to check the final state. The application developer is expected to have both the Pharo and Javascript test case passing.

This section showed that simulation-based testing has been used successfully in several different domains. However, when possible native execution is preferred on top of simulation testing. Simulation-based solutions approach to testing is summarized by Table 2.1.

2.6 Conclusion

This state of the art gives many hints on how to test a VM written in a simulation-based VM generator framework. Particularly to address automatically the Oracle problem, and to test the generated VM.

Table 2.1: Summary of how simulation-based solutions approach testing. We show here whether they apply native testing, mutation testing and if the test cases are handcrafted.

	Native Testing	Mutation Testing	Hand Crafted Test Cases
Process and system VMs	✓	✓	✓
Phone application	✓		✓
Network protocol		✓	
Cross-language development	✓		✓

Addressing the Oracle Problem in VM developments. All of the works on VM testing presented in this state of the art only address the test input generation problem. This is because they are able to leverage multiple implementations of a language. Rather than leveraging multiple implementations, some research uses cross-optimization in VM testing [Béra 2016, Polito 2022a]. Many languages only have one implementation and therefore can not use cross-implementation differential testing or cross-optimization strategy to solve the oracle problem. We show in this thesis that using a simulation-based VM generator framework to develop a VM enables differential testing by default, with a cross-environment testing strategy.

Testing Simulation-Based VM Generators VMs. The usage of simulation-based VM generator creates multiple executables. Particularly they use different environments, with different runtimes. PyPy's approach validates each of these environments differently. The simulated interpreter is validated with both interpreter and application-level test cases. The generated interpreter is validated only with application-level test cases. We propose in this dissertation to apply interpreter-level test cases on both interpreters. We show in this dissertation that the simulated and generated VM catch different bugs, that are in either or only in one of the environments.

In the following Chapter, we present our experimental context, which drove our research toward these problems.

CHAPTER 3

Experimental Context: The Pharo VM

Contents

This chapter presents our experimental context: Pharo. Particularly we have presented Pharo itself, its syntax, and its relevance for this thesis. More importantly, we have presented the Pharo VM. The Pharo VM is written in Slang, a subset of the Pharo language. The Pharo VM is generated to C and compiled by a C compiler for production purposes.

3.1 Introduction

Contributions of this thesis target virtual machines developed in a simulation-based VM generator framework. We'll be referring to them as VM for short.

To answer our research questions, we experimented on the Pharo Virtual Machine which is developed in a simulation-based VM generator [Miranda 2018, Polito 2021]. Other simulation-based VM generator frameworks are used to develop VMs. The Squawk VM is a Java VM for bare metal execution on sensor devices [Simon 2006]. The simulation-based VM generator framework RPython

has been used to develop multiple VMs. Particularly a Python VM named the PyPy VM [Rigo 2006, Ancona 2007], a Squeak VM named Spy [Bolz 2008] and another named RSqueak [Felgentreff 2016].

We present in this chapter the Pharo language (Section 3.2.1) and its VM (Section 3.2.3). We also give an indication of how to read Pharo code (Section 3.2.2). We then show that Pharo is a relevant environment for this research (Section 3.2.4). Finally, we show how to execute the multiple version of the Pharo Virtual Machine (Section 3.2.4).

3.2 The Pharo Language

Pharo [1] is a general-purpose dynamically-typed language. We present in the following the language itself (Section 3.2.1), its syntax (Section 3.2.2), and its VM 3.2.3. We conclude this section by highlighting the reasons the Pharo language is relevant for this kind of research (Section 3.2.4).

3.2.1 Pharo in a Nutshell

Pharo is a pure object-oriented dynamically-typed general-purpose programming language [Black 2009] derived from Smalltalk [Goldberg 1983]. Pharo code is compiled to bytecode by the Opal Compiler [Béra 2013]. The bytecode is fed to the Pharo Virtual Machine for execution by an interpreter and a baseline JIT. Pharo supports many industrial applications as well as research projects[2].

Here is a selection of the features Pharo offers to developers:

Pure object-oriented programming. Everything interaction in the system is achieved through the use of message sends. Even primitive types such as integers are objects and they are able to receive messages.

Fully reflexive. Pharo supports not only introspection but also intercession. It allows one to modify any object in the system programmatically.

Simple syntax. The two previous items allow one to remove many concepts from the programming language syntax. For example, classes are objects. Therefore they do not need special syntax to instantiate it. Instead, they only need to be able to receive a message that asks for an instantiations. The full Pharo syntax fits on half a postcard 3.1.

[1] https://pharo.org
[2] https://consortium.pharo.org

Metacircular environment. The Pharo system is written in itself. This allows developers to explore the system, redefine it, and hack it. Developers are able to increase the core by adding messages on primitive types.

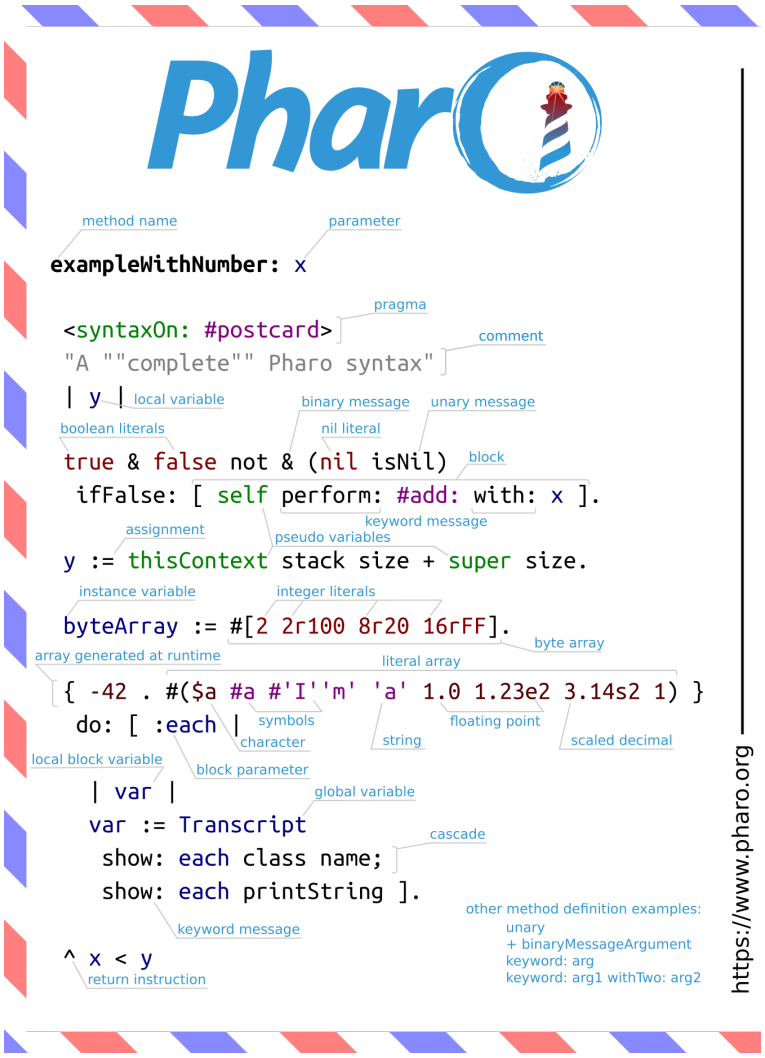

Figure 3.1: The full Pharo syntax on half a postcard. The syntax is described further in Section 3.2.2.

3.2.2 Basic syntax

To allow the reader to follow the code snippets in this dissertation, here are a few language syntax explanations. We also give equivalent syntax in java.

Comments. Comments in Pharo are expressed between double quotes (Listing 3.1).

```
" This is a Pharo comment "

// This is a Java comment
/* This is a Java comment */
```

Listing 3.1: Comments syntax in Pharo.

Keywords. Pharo uses only 6 keywords. We present the 4 that are relevant to this work.

- true, false are the boolean constants.
- nil represents the undefined object. It is similar to NULL in java.
- self is the receiver in the context of the method. It is expressed as the this keyword in Java.

Particularly, the new java keyword that instantiates classes does not have a keyword equivalent. Instantiation is achieved by sending the message new to a class object.

Assignments. Assignments in Pharo are expressed with a column and an equal sign (Listing 3.2).

```
" Pharo assignment "
aVariable := aValue.
// Java assignment
aVariable = aValue;
```

Listing 3.2: Assignments syntax in Pharo.

Returns. Returns in Pharo are expressed with caret sign (Listing 3.3).

```
" Pharo return "
^ aValue

// Java return
return aValue;
```

Listing 3.3: Return syntax in Pharo.

Message Sends. Every message is sent to a receiver. The receiver of a message is an expression. In Java, message sends have a unified syntax. the receiver is followed by a dot, the method name, and the arguments between parenthesis.

Pharo gives developers more flexibility by defining three kinds of message sends. Listing 3.4 presents an example of each of these message sends kinds.

Unary Messages. A unary message takes a single argument: the receiver (Line 2). It is written in plain text.

Binary Messages. A binary message takes an argument as well as the receiver (Line 3). It is represented by a symbol (*e.g.* +, -, =) or a string of symbols (*e.g.* ->, ==). It does not have a Java equivalent.

Keyword Messages. A keyword message takes one or more arguments as well as the receiver (Lines 4 and 5). It's written in all letters, with a column character at the end of each part of the method name. For example, the name of the method in Line 5 is aKeywordMessage:withMultipleArguments.

```
1  "Pharo message sends"
2  aReceiverExpression anUnaryMessage.
3  aReceiverExpression <|-- anArgument. "Binary message, no java equivalent"
4  aReceiverExpression aKeywordMessage: anArgument.
5  aReceiverExpression aKeywordMessage: firstArgument withMultipleArguments: secondArgument.
6  /*Java message sends*/
7  aReceiver.anUnaryMessage();
8  aReceiver.aKeywordMessage(anArgument);
9  aReceiver.aKeywordMessagewithMultipleArguments(firstArgument, secondArgument);
```

Listing 3.4: Message sends syntax in Pharo.

Blocks. Pharo blocks are lexical closures. Although they have more properties, we limit their usage as blocks of code with no delayed execution in this dissertation. Particularly, as there are only 6 keywords in Pharo, the control flow is achieved with message sends. if statements take Pharo blocks as arguments (Listing 3.5). Blocks are written with square brackets characters.

```
1  " Pharo blocks "
2  true ifTrue: [ aBlockBody ].
3
4  // Java blocks
5  if(aCondition) then { aBlockBody; }
```

Listing 3.5: Block syntax in Pharo. The Java version is a semantic subset of the Pharo blocks. However, in this dissertation, we only use this simplified version.

Method Definition. Method definitions are generally written a special tab in the Pharo IDE which defines in which class the method is installed. We use in this dissertation a slightly different syntax which allows the reader to know in which class the method is installed, for additional contextual information (Listing 3.6). We start with the capitalized name of a class followed by the definition of the name of the method and of the arguments. This is followed by the method body.

```
1   " Pharo method definition "
2   ClassName << aMethodName: aMethodArgument
3     aMethodBody.
4
5   // Java method definition
6  class ClassName{
7    aReturnType aMethodName( anArgumentType aMethodArgument){
8    aMethodBody;
9    }
```

Listing 3.6: Method definition syntax in Pharo.

Method Annotation. The method may contain meta-data that may be useful for various tools. For example, they direct compilation, unit testing, debugging, scripting and UI. Pharo calls them *pragma*s, Java calls them *annotation*s. Pharo pragma uses an identifier similar to message sends. They use either a unary message identifier or keyword message identifier 3.7.

```
1   " Pharo method pragma example"
2   ClassName << aMethodName
3     <aPragma>
4     <aPragma: aPragmaArgument>
5     aMethodBody.
```

Listing 3.7: Pragmas syntax in Pharo.

3.2.3 The Pharo Virtual Machine

The Pharo virtual machine is an industrial-level Virtual Machine written in Pharo itself and generated to C using a VM-specific translator called the Slang VM generator [Ingalls 1997, Miranda 2018]. The VM implements at the core of its execution engine a threaded bytecode interpreter, a linear non-optimizing JIT compiler named Cogit [Miranda 2011] that includes polymorphic inline caches [Hölzle 1991] and a generational scavenger garbage collector that uses a copy collector for young objects and a mark-compact collector for older objects [Ungar 1984].

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.

The following numbers illustrate the complexity of the JIT Virtual Machine, one of the versions of the Pharo VM:

- It has 33 classes;
- They have 2656 methods and 252 variables;
- It generates 125 000 lines of C code from 33 000 Slang code lines;
- It implements 255 bytecodes, organized in 77 different families [Béra 2014];
- It implements about 340 primitive methods, with a number of them both duplicated in the interpreter and in the JIT compiler;
- The JIT compiler defines about 150 different IR instructions [Polito 2021].

3.2.4 Pharo's Relevance

Pharo is relevant for this research because of several features.

Simulation-based VM generator. We present in this dissertation how to test VM written in simulation-based VM generator. Reusing the Slang VM generator allowed us to have a validation on a real-world system.

Open source. Pharo is fully open source. Particularly, the Virtual Machine and its simulation-based VM generator are both available.

Industrial context. The Pharo Virtual Machine is an industrial-level Virtual Machine. Pharo supports many industrial applications as well as research projects[3].

3.3 Executing the Pharo Virtual Machine

Using a simulation-based VM generator framework to develop the Pharo VM means that the VM has two execution environments. The Pharo VM is executed either in a simulation environment (Section 3.3.1) or the generated environment (Section 3.3.2).

3.3.1 Simulating the Pharo VM

The Pharo VM is compiled to bytecode to execute the simulated VM. The simulated VM is used both for development and debugging purpose. Particularly,

[3]https://consortium.pharo.org

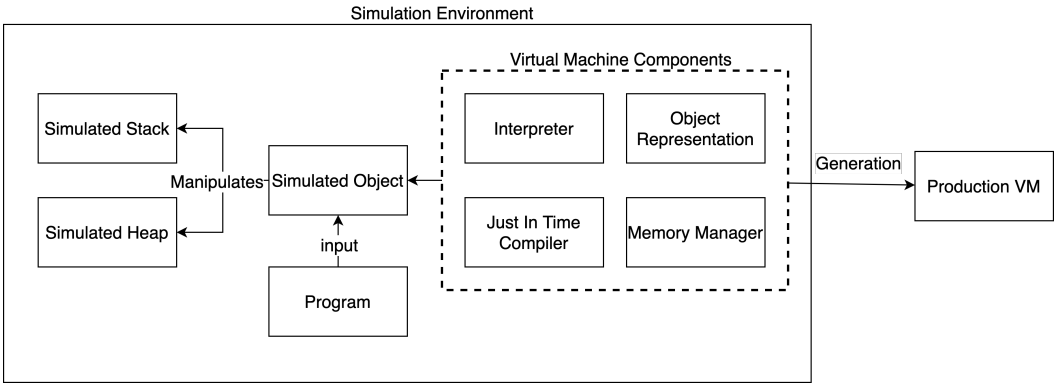

Figure 3.2: **Simulation overview.** The simulation environment contains the implementation of the VM components. A layer on top of the simulation defines a few simulation-specific features. Particularly, it allows the VM objects to use a simulated *stack* and *heap*. The VM components are processed by the Slang VM generator to generate the production VM.

the Pharo simulation has been used to debug multiple parts of the VM such as the GC [Miranda 2018], the JIT compiler [Miranda 2011] and test the JIT compiler [Polito 2022a]. This simulation is significantly slower than the version compiled by the Slang VM generator. However, we benefit from the full power of Pharo and its IDE for testing and debugging.

The simulation environment contains the implementation of the VM components (Figure 3.2). A layer on top of the simulation defines a few simulation-specific features. Particularly, it allows the VM objects to use a simulated *stack* and *heap*.

The VM components are processed by the Slang VM generator to generate the production VM.

3.3.2 Generating the Pharo VM

As Pharo is a Smalltalk-inspired language, the Pharo to C translation is done using the Slang VM generator, a Smalltalk-to-C VM generator [Ingalls 1997]. Slang operates by translating a group of classes into a single C file. Methods are translated into functions, and message-sends are translated as function calls. While the Pharo source program presents dynamic behaviour such as exceptions or runtime reflection, the Slang VM generator does not allow many of those. It either forbids them at generation time or generates an invalid C code.

Using Slang to develop the Pharo VM has two key advantages. First, Slang automatically introduces interpreter optimisations such as (a) the localisation of critical variables (frame pointer, instruction pointer) [Miranda 1987, Polito 2022b],

(b) the inlining of bytecode cases inside the interpretation loop, or (c) threaded code [Ertl 2003a]. Second, it allows us to simulate the Pharo VM just by executing it as normal Pharo code, avoiding expensive change-compile-test development cycles [Miranda 2018, Polito 2021].

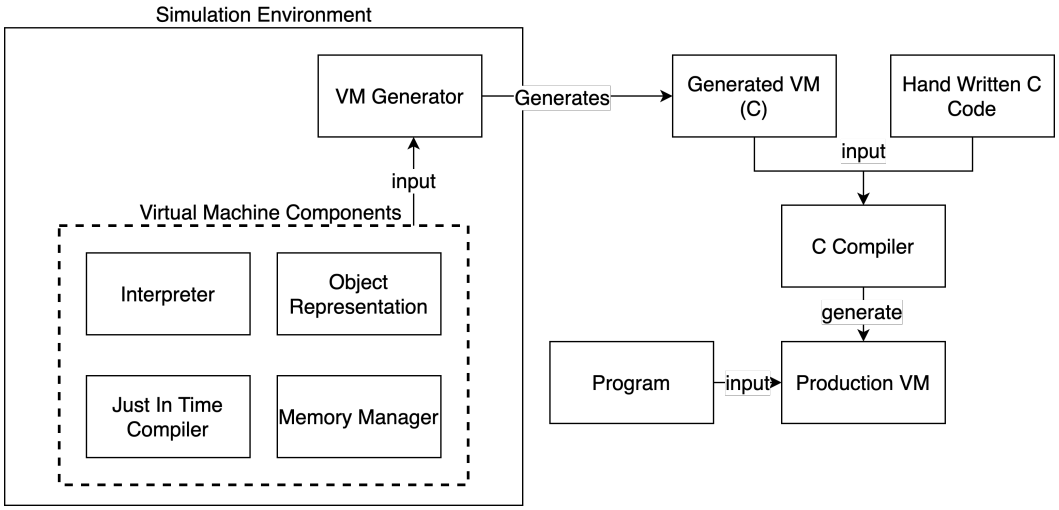

Figure 3.3: **Generation overview.** The VM components are processed by the Slang VM generator to generate a C source file. The generated VM is fed to a C compiler with a few handwritten files to generate the production VM. The production VM is then ready to execute code.

A graphical overview of the generation process is provided in Figure 3.3. As Slang is developed in a Pharo environment and is a subset of Pharo, it first goes through the same steps as regular Pharo code and is therefore also compiled to executable bytecode. During this first stage, the Pharo compiler first parses the code and creates the method's AST. This compilation process detects some errors and mistakes such as syntactic errors or undeclared variables.

Next, the Slang VM generator computes a set of classes to be translated and processes them in the second stage of the generation. During the generation process, the Slang VM generator handles the AST of input classes and methods, prepares them to be translated, and emits the corresponding C code. The Slang VM generator checks whether some unsupported features are used, such as dynamic arrays, and interrupts the translation process with an error if it encounters any of them. It then applies optimizations such as inlining and dead code elimination. It generates and flushes the emitted C code into a file. Finally, the generated C code is compiled using a traditional C compiler. The C compiler catches some more statically detectable errors such as invalid C code.

3.4 Conclusion

In this dissertation, all the experiments were applied to the Pharo Virtual Machine. We present in this chapter the Pharo language and its VM. The Pharo VM is developed with the help of a simulation-based VM generator framework. We have also presented the Slang language. In the remainder of this dissertation, we use the term virtual machine to describe a VM developed in a simulation-based VM generator framework. In the next Chapter, we explain how we use the simulation environment to test the Pharo VM.

CHAPTER 4

Simulation-Based VM Testing

Contents

This chapter presents the first step in our research: how to test the simulated VM. We present the methodology we devised to test the VM, as well as the different execution modes we leverage: unit testing, full system simulation, and real hardware execution. Basically, the aim is to use each execution mode where it performs best. Particularly, we leverage the execution speed and the feedback-cycle speed of unit test cases to execute them as much as possible. Similarly, we leverage the precision of the execution of the real hardware execution. However, as it is significantly slower, it is executed less often.

4.1 Introduction

The Pharo VM is developed with a high-level simulation environment that is very handy to simulate full executions and live-program the VM [Miranda 2018]. The VM uses a Just In Time (JIT) compiler which generates assembly code. As such, to test the code outputted by the JIT compiler, the simulation environment is a hybrid execution environment: the generated runtime, *i.e.* the machine code compiled methods generated by the JIT compiler, is executed using a machine code simulator.

We extended the simulation environment with a testing infrastructure that allows us to have fine-grained control of testing scenarios. This makes tests small, fast, reproducible, and cross-ISA The VM development environment is illustrated in Figure 4.1. Our testing infrastructure lies within the simulation environment and has access to the same infrastructure as the full simulation. Eventually, the VM is generated to C and compiled for a given ISA/architecture.

Listing 4.1 shows an example of test case testing that the primitive #primitiveAdd fails in case the results of the addition do not fit in a SmallInteger. This is achieved by pushing the maximum value for a SmallInteger on the stack (line 2), pushing another SmallInteger on the stack (line 3), and trying to add them by using the dedicated primitive (line 5). The assertion then checks that the interpreter reports that the primitive failed (line 9).

```
1 InterpreterPrimitiveTest >> testPrimitiveAddWithOverflowFails
2   interpreter push: (memory integerObjectOf: memory maxSmallInteger).
3   interpreter push: (memory integerObjectOf: 42).
4
5   interpreter primitiveAdd.
6
7   self assert: interpreter failed.
```

Listing 4.1: Example of test case that is executed on the simulated Pharo VM. This test case tests that the primitive #primitiveAdd fails in case the result of the addition does not fit in a SmallInteger.

Based on this infrastructure we defined a hybrid testing methodology that mixes three different execution modes: unit testing, full-system simulation, and full-system real-hardware execution. Our methodology takes advantage of the strengths of unit testing, full-system simulation, and real-hardware execution where they perform better.

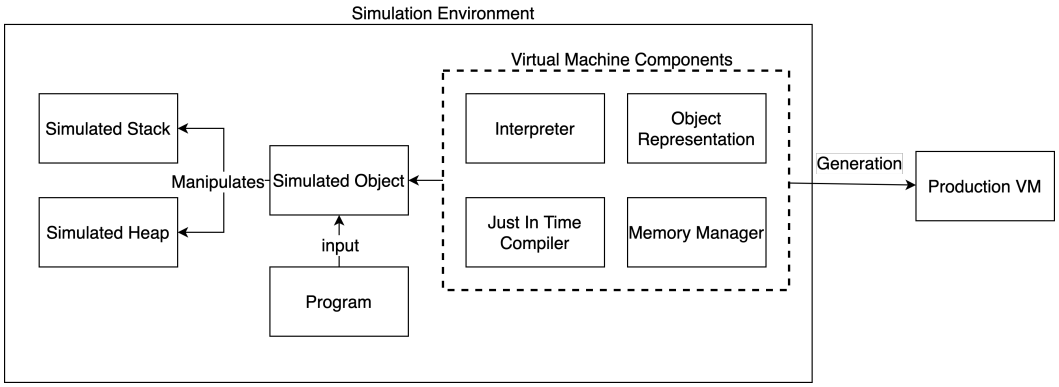

Figure 4.1: **Development environment of the Pharo VM**. The VM is executed as Pharo code in the simulation environment and generated to C to produce the production artifact. The simulation environment has its own heap and stacks. This new testing infrastructure extends and makes use of the existing simulation environment.

4.2 An Iterative Test-Based Methodology

4.2.1 Motivation

Both the simulated and generated VM execution modes have different benefits and constraints, as shown in Table 4.1. Real hardware is not always readily *available* or *easy to debug*, but its execution is the most *precise* in comparison with simulated hardware. At the same time, testing changes in real hardware is *expensive to compile and run*, leading to slow develop-compile-test feedback cycles. On the other side of the spectrum, unit tests are a handy way to express *reproducible* and *representative* scenarios and are capable of capturing regressions (Listing 4.1). However, since they are based on a simulation environment, they suffer from execution *imprecisions* because the simulation machinery is not 100% representative of real executions.

4.2.2 VM Testing: An Agile Perspective

Based on these observations, we characterise the three different execution modes as per Table 4.1, using the following criterion:

- **Feedback-Cycle Speed.** How fast is the develop-compile-test cycle for a single test scenario? Unit tests support fast development cycles because they work on small and precise scenarios and do not require a full system initialization. Full-System simulations have slower development cycles because they require a full-system initialization plus the time to arrive at the interesting execution

Table 4.1: Characterisation of different execution modes.

	Simulated VM Unit Testing	Full-System Simulation	Full-System Real Execution
Feedback-cycle Speed	+	-	-
Availability	+	+	-
Reproducibility	+	-	-
Precision	-	-	+
Debuggability	+	+	-

spot. Full-system real hardware execution has on average even slower feedback cycles than the full-system simulation because they require also a full VM compilation, which in our case includes generation to C and C compilation.

- **Availability.** Is the testing scenario readily available to execute? On the one hand, simulation-based solutions (Unit tests, full-system simulations) are highly available although tied to the availability of a machine code simulator, which are in general software-based portable solutions. On the other hand, hardware-based solutions (real hardware) are much less available, as they require target hardware access.
- **Reproducibility.** What are the chances of reproducing a single test scenario? Full-System executions (simulation, real hardware) have low reproducibility by default because millions of instructions may need to be executed before hitting an actual problem, and non-determinism may worsen the problem. In contrast, unit tests are by construction repeatable.
- **Precision.** How precise is the scenario execution? Simulation-based solutions (Unit tests, full-system simulations) have generally lower precision than real hardware because a simulator introduces a distance between the executions. For example, Slang's type system uses low-level typing that is not simulated by the Pharo environment.
- **Debuggability.** Is the testing scenario easy to debug? On one hand, the simula tion-based solutions (Unit tests, full-system simulations) are often built with debugging support in mind. On the other hand, real hardware execution requires abandoning the abstractions of the simulation environment and often deal with abstractions closer to the machine.

4.2.3 A Single Methodology to Rule Them All

Based on these constraints, we developed the methodology illustrated in Figure 4.2:

1. Spend as much time as possible working on unit tests: they are cheap to write

and execute.

2. Introduce full-system simulations when we consider coverage is good enough: it represents a full-system execution without access to real hardware.

3. Introduce testing on real hardware when we consider full-system simulation reliable enough: to obtain final precise and reliable feedback.

The key of this methodology is to augment the team's velocity and to apply Test-Driven-Development (TDD) techniques from Software Engineering to VM development. In this schema, unit tests are the main development unit because of their fast feedback. In our experience, unit tests have caught the vast majority of bugs without the need to validate them in real hardware. Really few problems seemed to remain *untestable* at first because of simulation imprecisions, but most of them became testable after introducing fixes in the simulation environment.

We then use full-system executions (either simulated or on real hardware) to get feedback on failing untested functionalities. As soon as a full-system execution fails, we use that as feedback for unit testing: we build one or more failing tests for the scenario, we manually do test reduction on them, and finally fix the actual problem to make them pass. Such a mixed-mode methodology allowed us to port most of the JIT compiler to AArch64 before we had access to real hardware. As a nice side effect, we created a large set of tests that are small, fast to run, reproducible, and cross-ISA, and even unveiled old bugs in the pre-existing backends.

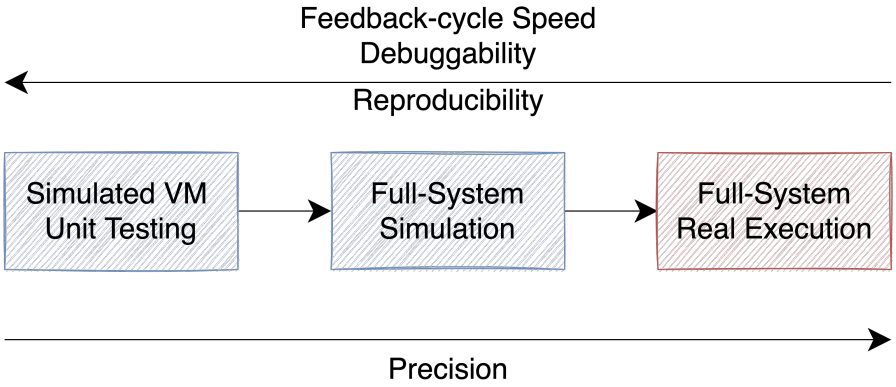

Figure 4.2: Our methodology is based mainly on developing mostly black-box unit tests. When the number of running unit tests gives us enough confidence we introduce full-system simulations, and when we are confident enough about the full-system simulations we spent time on real hardware. At each step, we reduce the problems we find to a bare minimum test case that reproduces it. We then introduce it as a regression test.

4.2.4 The Pharo VM Testing Infrastructure

All unit test cases and the testing infrastructure reported in this dissertation are available in the package VMMakerTests, in the branch PharoX of our Git public repository[1]. Our testing suite defines as-per-today 1367 written unit tests[2], Table 4.2 details the processor-independent test cases.

The most complex part of writing a VM-specific unit test is defining the test fixture/setUp. Indeed, executing a short method doing push a, push b, send +, return requires the initialization of the heap, the execution stack, and internal VM data structures, among others. Because of these, the largest effort of our infrastructure was put into creating the specific initializations and testing primitives. The core of our infrastructure lies within four abstract Test classes that developers extend to define their new testing scenarios:

- **Heap Initialized Test Case.** The root of our testing hierarchy. All tests that inherit from this class run configured with a heap. This class also provides facilities to create/load classes and instantiate objects.
- **Interpreter Test Case.** All tests that inherit from this class run configured with a heap and an execution stack. They have access to a call-stack builder to create well-formed stack frames.
- **Compiler Test Case.** All tests that inherit from this class run configured with a heap, an execution stack, and a code cache. They have access to a simplified compiler interface that compiles either entire methods, compiler intrinsics, and individual bytecodes. Moreover, all tests extending this class are by default executed on all supported ISAs.

4.3 VM Testing Guidelines

In our journey of writing tests for the Pharo VM, we developed a large test suite that covers different VM concerns such as the interpreter, the compiler, the object format, and the garbage collector. In this section, we report on our experience designing VM-specific unit tests in the form of several guidelines. These VM-specific guidelines emerged from applying general well-known testing principles such as:

Fast. A test should be as instantaneous as possible to ensure we have a fast feedback cycle.

Reproducible. Test executions should be deterministic.

[1] https://github.com/pharo-project/opensmalltalk-vm

[2] https://github.com/pharo-project/opensmalltalk-vm/commit/8f3028057c4f98afa38c5645223d19ecfbfb2bf3

Repeatable. Several test case executions should be independent of each other.

Unitary. Each test should test a single concern. Particularly, if a given functionality has many aspects to test such as several border cases, a test exists for each of them.

Validating. A test must have at least one assertion, and ideally only one.

4.3.1 Black Box Testing

We use black-box testing in the vast majority of our tests: we test externally observable behaviour. For example, most of our memory management tests are word-size independent. Another example is compiler tests: we avoid as much as possible exposing the internals of the compiler such as the Intermediate Representation (IR) or the generated machine code within the test code. Indeed, most of our compiler-related tests work with the granularity of a single bytecode. This design gives our tests two main properties: (1) resistance to changes in the VM and compiler implementation, and (2) architecture/ISA independence.

Listing 4.2 shows one of the simpler tests in our compiler test suite. The test uses our compiler interface to compile a single bytecode, execute the generated machine code and test that the constant zero was pushed to the operand stack. This test is written once and runs automatically in all supported compiler backends (AArch64, AArch32, x86, and x86-64), both in 32bits and 64bits machines.

```
1 testPushConstantZeroBytecodePushesASmallIntegerZero
2   self compile: [ compiler genPushConstantZeroBytecode ].
3   self runGeneratedCode.
4   self assert: self popAddress equals: (memory integerObjectOf: 0)
```

Listing 4.2: Test compilation of push constant zero.

To automatically execute the test cases in multiple configurations, we use configuration matrices.

4.3.2 Word Size Independent Testing Using Test Parameterisation

Pharo test infrastructure makes use of parameterized test cases to automatically run a single test on multiple configurations. Matrix configurations direct how the test cases are to be tested. Particularly, every test is executed for 32 and 64 bits. Listing 4.3 shows the configuration of our interpreter and memory testing matrix which runs the test cases automatically in 32bits and 64bits. The processor-dependent test cases, the JIT test cases, are executed on every processor architecture. The 1367 unit test cases represent a total of 3603 run tests when the different matrix

configurations are taken into account. A full run of Pharo's test suite amounts to a total time of ~2.5 minutes in our continuous integration server.

```
testParameters
  ^ ParametrizedTestMatrix new
      addCase: { #wordSize -> 8};
      addCase: { #wordSize -> 4};
      yourself
```

Listing 4.3: Testing matrix for memory and interpreter tests.

4.3.3 Grow Slowly in Complexity

A simple heuristic we use to develop tests is to always start with the simplest test we can write. If a test cannot be easily written with the testing infrastructure as-is this is an indicator that either the infrastructure is missing support for some feature, or that a simpler test needs to be developed before.

A second heuristic is to treat test code as any other code. In other words, test code can be subject to refactorings, cleaning, and the extraction of other reusable components. In the course of testing the VM, we created multiple setup methods for the different initialization phases *e.g.* initialized memory or initialized interpreter, and multiple helpers particularly to ease object creation.

4.3.4 Dealing with Platform Specific Constraints

The memory and interpreter test cases presented in this chapter have little to no platform-specific constraints. However, since the JIT compiler generates assembly code, not every JIT compiler test case is valid for every architecture.

The guidelines above, therefore, make test cases that are generic by default, but generic tests do not deal correctly with platform-specific constraints. Platform-specific constraints arise from different VM configurations, target operating systems, or target processors. For example, AArch64 differs from Intel-based processors in many ways: multiplication overflow does not set the overflow flag, subtraction presents an inverted carry flag, and the stack pointer has alignment restrictions. Some operating systems impose exclusive write-executable permissions. VM-specific differences appear between 32bits and 64bits memory models.

In cases like the ones exemplified above, our testing infrastructure does not forbid testing but requires more fine-grained control and *narrowing* the scope of the test. We achieve fine-grained control in platform-specific tests by allowing test cases to manually feed Intermediate Representation instructions to the compiler. Also, when narrowing test scenarios for a particular platform, we observed the emergence of two main cases in our test suite:

Generic tests with exceptions. Some tests are valid for all but one matrix configuration. In such cases, the specific configuration is explicitly skipped for that test.

Platform Specific Tests. Some tests are valid only for a single platform. In such cases, a separate test class with a specialized test matrix is written to host them. An example of this is the ARM stack alignment tests, in class VMARMStackAlignmentTest, with a test matrix using a single case, as shown in Listing 4.4.

```
testParameters
  ^ ParametrizedTestMatrix new
      addCase: { #wordSize -> 4. #wordSize -> 8};
      yourself
```

Listing 4.4: Testing matrix for a platform specific test.

4.4 VM Simulation-Based Testing Limitations

As stated earlier in this chapter, both the simulated and generated VM execution modes have different benefits and constraints. However, in a simulation environment, they suffer from execution *imprecisions* because the simulation machinery is not 100% representative of real executions. These imprecisions are called **semantic gaps** with the generated VMs [Besnard 2017]. More semantic gaps also appear because a generator may imprecisely map code from one form to the other [Terekhov 2000]. This might be because of a bug in the generator but this may also be a hypothesis that the developer must be aware of. Moreover, the test cases are only executed in the simulation environment, but those test cases do not ensure the correctness of the generated VM. The generated VM is only tested by full hardware execution. This section presents the semantic gaps problem of simulation-based VM generators and the limits of simulation-based testing driven by Slang VM generator examples.

4.4.1 Pharo VM Semantic Gaps by Example

The Slang VM generator takes as input a VM definition written in Pharo itself and generates C code [Misse-Chanabier 2019]. 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 mapped to C code or rejected [Ingalls 1997].

```
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 4.5: 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 4.5 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: (lines 4 and 6). It also provides VM developers with a way to specify environment-specific code (lines 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 even though the simulation ignores them (line 2). This means that wrong type annotations do not impact the simulated VM, it only impacts the generated VM by producing possible execution and generation differences. Notice that the semantic gaps are aggravated by additional Pharo features *e.g.* inheritance, polymorphism, method redefinition, super message sends, block closures, managed memory, and absence of stack allocation which require more complex code transformations.

4.4.2 Semantic Gaps Related Bugs

As we described, we use test cases in the simulation environment. Let's consider for example the VM test case depicted in Listing 4.6. This test case is executed as plain Pharo code during the 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 [RPythonCommunity 2016].

```
1  testPrimitiveNaturalLogarithmShouldFailForNegativeNumber
2    | aFloat |
3    aFloat := self newFloatFromInt: -1.
4    interpreter push: aFloat.
```

```
5     interpreter primitiveNaturalLogarithm.
6     self assert: interpreter failed.
```

Listing 4.6: Excerpt of a test case validating that the logarithm of a negative number sets the interpreter in a failed state. This test case passes in simulation but does not capture potential typing failures of the generated code.

Multiple other semantic gaps are not caught by the simulation test cases. For example, we found that the generator may generate C code containing C undefined behavior that the simulation test cases miss.

These issues come from the fact that the simulation is not a perfect replica of the generated environment. This is worsened by the fact that the VM generator may also add semantic gaps. The generated VM may therefore exhibit behaviour that is not the same as the simulated VM. To solve this issue, we propose in the next chapters to generate the test cases to test the generated VM.

4.5 Conclusion

In this Chapter, we have presented the hybrid approach we use to test the Pharo VM. This methodology mixes three different execution modes: unit testing, full-system simulation, and full-system real-hardware execution. This methodology takes advantage of the strength of each execution mode.

Moreover, we have shown that the simulation is not able to detect some bugs, due to semantic gaps. Particularly, as the explicit types are ignored during the simulation, the simulation test cases miss explicit type-related bugs. To solve this issue, we propose to generate the test cases to test the generated VM.

Table 4.2: **Processor Independent Tests (32bits / 64bits):** These test cases focus mainly on the memory representation and its management, as well as the Interpreter and its primitives. They are simulated for 32 and 64 bits machines. They do not use Unicorn's machine simulator.

VM Component	Operation	Independent Test Cases	Total Executions
Test Infrastructure	Method Builder	10	20
	Stack Builder	18	36
	Total	*28*	*56*
Object Memory	Stack Reification	7	14
	Context / Stack Mapping	13	26
	GC Data Structures	13	26
	Unmovable Objects	9	18
	Old Object Garbage Collection	63	126
	Young Objects Garbage Collection	38	76
	Weak Object Garbage Collection	9	18
	Ephemeron Object Garbage Collection	19	38
	Old Objects FreeSpace Management	85	170
	Memory Structure Preconditions	30	60
	Total	*286*	*572*
Interpreter	Bytecode Tests	43	86
	Method Lookup	15	30
	Object Representation	17	34
	Primitives	62	124
	Total	*137*	*274*
	Total	**451**	**902**

CHAPTER 5

Test Transmutation

Contents

This chapter introduces Test Transmutation. Test Transmutation proposes to test the generated VM by co-generating the test cases with the VM and executing the generated test cases. We first describe the properties of generated test cases and compare them to the properties of other execution modes presented in Chapter 4. We describe the oracle that we use to validate our test cases and the mutated test cases. Our oracle uses *functional equivalence* to compare the *test case results*. As the test case results come from different environments, we call this differential testing strategy a *Cross-Environment Differential Testing* strategy. We explain further how we apply non-semantic-preserving mutations on test cases. The validation of Test Transmutation is described in the next chapter.

5.1 Introduction

We propose Test Transmutation (Figure 5.1) to address semantic gaps between the simulated and generated VM by generating the simulated VM's test cases and executing them on the generated VM (Section 5.2). We execute both simulated and generated test cases and compare their results with differential testing [McKeeman 1998] to detect execution differences (Section 5.3). 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 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 always pass by design, making them poor inputs for the differential testing process.

To address these challenges, we extend our approach with automatic non-semantic-preserving mutations [DeMillo 1979] (Section 5.4.2). 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 fail in the generated VM. A non-equivalence of test results shows that there is a bug in the VM or in the VM generator.

We present the validation of Test Transmutation in the next chapter (Chapter 6)

5.2 Co-Generation of VM and Test Cases

5.2.1 Generating Simulation Test Cases

To test the generated VM in spite of the semantic gaps, we propose to generate the simulation test cases.

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, (ii) the generated source code does not compile, (iii) or an assertion check fails, or (iv) an unexpected runtime error happens.

One of the biggest advantages of using simulation-based VM generator frameworks is the ability to use the simulation during development and to use the VM generator to generate the production version. This results in a unique code base

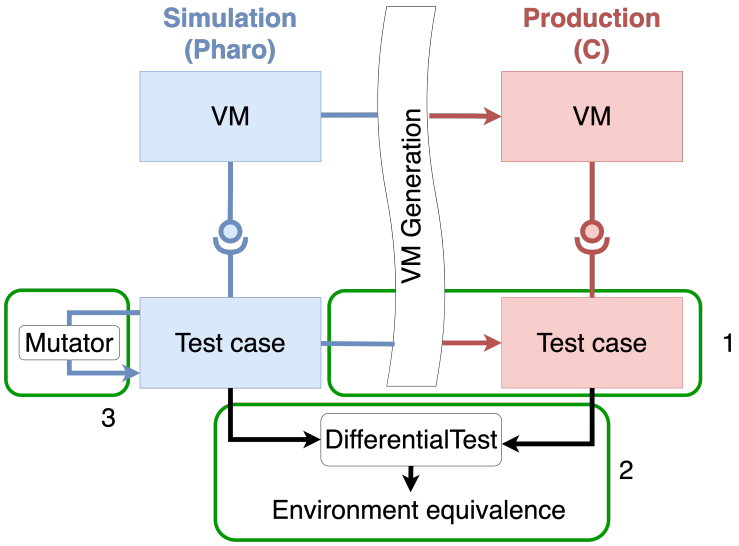

Figure 5.1: Test Transmutation overview. We generate test cases with the VM (1) and compare their results with differential testing (2). Mutations increase the number and variety of test cases (3) and are validated by applying differential testing (2).

that is used in both simulation and production environments. Co-generating the test cases allows developers to have a unique test code base in a similar manner.

5.2.2 Generated Test Cases properties

We expand the table initially presented in the previous chapter to incorporate Generated Unit Testing (Table 5.1).

Table 5.1: Characterisation of different execution modes with the characterisation of generated VM unit testing.

	Simulated VM Unit Testing	Full-System Simulation	Full-System Real Execution	Generated VM Unit Testing
Feedback-cycle Speed	+	-	-	~
Availability	+	+	-	-
Reproducibility	+	-	-	+
Precision	-	-	+	+
Debuggability	+	+	-	~

Feedback-cycle Speed. The VM under test is the generated VM. Therefore it requires to be compiled at least once. Plus, the compiled VM binary is invalidated

every time the source code is modified and requires a recompilation. However, the unit test cases generation and compilation are fast. Therefore, as long as the compiled VM is valid, developers are able to execute any number of test cases on the generated VM.

Availability. Similarly to real hardware execution, real hardware unit testing requires to have access to the hardware.

Reproducibility. Unit tests are highly repeatable by constructions. This remains the case regardless of the execution environment.

Precision. Real hardware unit testing is more precise than the simulated VM because the distance between the real hardware execution and the test case execution is as small as it may possibly be. For example, Slang's type system uses low-level typing that is not simulated by the Pharo environment. Simulation test cases, therefore, miss low-level typing bugs, whereas generated test catch such bugs.

Debuggability. As developers work in the simulation, most of the tooling they create remains in the simulation. Therefore developers have to rely on general-purpose low-level debuggers when debugging the compiled VM. The generated test cases provide a significantly easier program to debug than real hardware execution. Moreover, they provide additional debugging information by regenerating the test case without recompiling the full VM.

5.3 Generated Test Cases Oracle

The **oracle** problem is the problem of automatically determining whether a given input matches the expected semantics [Chen 2020]. Our oracle is based on differential testing [McKeeman 1998]. Differential testing oracles are *comparing* some *properties* of two program executions with a comparison function. Our oracle uses *functional equivalence* (Section 5.3.1) to compare the *test case results*. The resulting oracle uses a *Cross-Environment Differential Testing* Strategy (Section 5.3.3).

5.3.1 Comparison Function: Functional Equivalence

Informally, two pieces of code producing the same results for all inputs are functionally equivalent. In other words, two pieces of code that use different algorithms to achieve the same behavior are functionally equivalent. Listing 5.1 shows an example of two methods that are functionally equivalent. Unfortunately, detecting whether two pieces of code are functionally equivalent or not is an undecidable

problem [Person 2008]. Therefore we use an heuristics based on test case results which we describe in the following.

```
1 "aMethod1 is functionally equivalent to aMethod2"
2 AClass >> #aMethod1: aBoolean
3   aBoolean ifTrue: [ ^ 1 ]
4
5 "aMethod2 is functionally equivalent to aMethod1"
6 AClass >> #aMethod2: aBoolean
7   aBoolean not ifFalse: [ ^ 1 ]
```

Listing 5.1: Example of functionally equivalent methods. Both code executions return the same result. The second boolean is the negation of the first one, and the #ifTrue: message send is replaced with #ifFalse: message send.

5.3.2 Compared Property: Test Case Results

Test cases are always self-validating and have a deterministic and discrete result: they either succeed or they fail. We, therefore, differentiate test case results in a binary way. We consider that a test case passes in the well-known usage of the word: It runs all its instructions without errors. An error is either: a compilation failure, a failed assertion, or a run-time exception. In case of any error, the test case fails.

The simulation test cases fail because an assertion check fails or a runtime error is signaled during the test case execution. For example, simulation test case executions are time constrained and the runtime may signal a TimeOut error.

Generated test cases may fail due to other runtime errors kinds such as segmentation faults. But although the runtime error kinds differ the generated test cases fail for the same reasons. However, they also fail if the simulated test case code is rejected by the VM generator or if the generated test case code is rejected by the C compiler.

5.3.3 Cross-Environment Differential Testing Strategy

We leverage the fact that developing a VM in the simulation environment of a simulation-based VM generator framework by definition creates at least one generated VM. Having two executable VM enables us to use differential testing to create an oracle (Listing 5.2). Our oracle uses *functional equivalence* to compare the *test case results*. As the test case results come from different environments, we call this differential testing strategy a *Cross-Environment Differential Testing* strategy. We propose in this thesis to compare test case results of the execution of a test case in two different environments, *i.e.* the simulation and production environment. There-

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

Table 5.2: Truth table for the interpretation of the differential testing function.

fore we call such a differential testing strategy a **cross-environment differential testing** strategy.

```
1 differentialTestingFor: aTestCase
2   simulationResult := aTestCase executeOnSimulatedVM.
3   generationResult := aTestCase executeOnGeneratedVM.
4
5   self assert: simulationResult equals: generationResult.
```

Listing 5.2: Our oracle uses *functional equivalence* to compare the *test case results*. As the test case results come from different environments, we call this differential testing strategy a *Cross-Environment Differential Testing* strategy.

Table 5.2 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 in the VM or the VM generator.

5.4 Test Case Mutation

We generate test cases from existing simulation test cases. Therefore test cases suffer from similar semantic gaps as the VMs *e.g.* typing semantics. 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 test cases used as input (Section 5.4.1). Mutations automate the creation of more inputs allowing one to gain confidence in the correctness of the generation process and the generated test cases (Section 5.4.2). Moreover, we use a coverage-directed approach to mutation (Section 5.4.3).

5.4.1 Mutation by Example

Mutating a piece of code consists in adding, deleting, or replacing parts of the code. Mutations are either preserving the semantics of the initial code or not. Mutations

are either:

Semantic-Preserving Mutations. Such mutations keep the semantics of the initial code. For example, it has been used by adding code in *dead code* sections. Therefore the added code does not change the semantics of the initial input and the resulting program is supposed to be equivalent [Le 2014, Le 2015, Sun 2016b].

Non-Semantic-Preserving Mutations. Such mutations do not make any guarantee about the resulting program. It either behaves as the initial program behaves or differently.

In the following, we use non-semantics-preserving code.

Code is mutable in many ways. For example, Listing 5.3 uses a plus operator. Applying the mutation operator **plus to minus** replaces the plus operator with a minus operator. (Listing 5.4). The mutated test input is then fed to the DTIO with the compilers under test.

Non-semantics-preserving mutation operators are not ensuring that the semantics of the test input change. For example, Listings 5.3 and 5.4 used as an example of mutation exhibit a code for which the non-semantics-preserving mutation operator actually does not impact the semantics. Therefore the mutant and the initial code are functionally equivalent.

```
1 AClass >> #testFivePlusFourIsSmallerThanTen
2   assert( 5 + 4 < 10 )
```

Listing 5.3: Example of original test input fed to the mutator

```
1 AClass >> #testFivePlusFourIsSmallerThanTen
2   assert: 5 – 4 < 10
```

Listing 5.4: Mutated code of test input depicted in 5.3 after applying the mutation operator plus to minus.

5.4.2 Test Case Variations with Non-Semantic-Preserving Mutations

We generate test cases from existing simulation test cases. Therefore test cases suffer from similar semantic gaps as the VMs *e.g.* typing semantics. 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.

For example, Listing 5.5 shows the application of the mutation operator *Remove Statement Mutation* which removes a statement from the method. It removes in the listing a message sent triggering the garbage collection. Therefore an object that has been allocated should remain in the memory, and the assertion should fail.

```
1 GCTests >> #testUnrefererecedObjectIsReclaimedByGC
2   self newNewSpaceObject.
3   "memory collectNewSpace. <- removed statement"
4   self assert: memory isEmpty
```

Listing 5.5: Example of an automatically mutated test case. The original test case allocates an object (Line 2), triggers a garbage collection (Line 3), and expects the memory to be freed after the garbage collection (Line 4). Applying the mutation operator RemoveStatementMutationOperator on this test case removes the garbage collection (Line 3). Therefore the allocated object is still in the memory, which means the assertion should now fail (Line 4) in both the simulated and the production VMs.

5.4.3 Coverage Directed Approach to Mutations

We look for functional equivalence of the test inputs. To validate whether a mutated test case detects a bug, we are executing test cases. A test case execution has little chance of encountering a miscompilation on code it is not executing. This kind of compiler bug exists and has been researched by Equivalence Modulo Input [Le 2014]. However, a test case has significantly higher chances to encounter a mutation installed in its execution path. Mutating blindly, therefore, serves little purpose and is unnecessarily time-consuming. By applying a coverage-directed mutation approach, only the code in the call graph of at least one test case is eligible to be mutated. Every mutation, therefore, impacts at least one test case.

Moreover, the coverage information also enables coverage-driven evaluation of the test cases. Instead of executing every single test case available, only the test cases impacted by a mutation are executed. Most of the time only a few test cases are necessary to ensure functional equivalence of the outputs for mutated test inputs. Both these coverage-directed approaches are illustrated with an example in Listing 5.6.

```
AClass >> #test1
  self usedMethod

AClass >> #test2
  self anotherMethod

AClass >> #usedMethod
  "usedMethod body"

AClass >> #unusedMethod
  "unusedMethod body"
```

Listing 5.6: Example of coverage-directed mutation and evaluation. usedMethod is in the call graph of test1 but not test2. usedMethod is eligible for mutation. When usedMethod's mutated code is installed, only test1 is executed as it does not affect test2. unusedMethod is not covered by the test cases of the program and is therefore not eligible for mutations.

5.5 Conclusion

This chapter presents Test Transmutation, a new approach to test VMs. Test Transmutation uses existing simulation test cases and generates them to execute them on the generated VM. To validate whether the generated test cases are generated correctly, we use differential testing of the test case results of the simulated and generated VMs to check whether they are functionally equivalent or not. Two test case results that differ exhibit a bug. As generated test cases are also sensible to semantic gaps, we also apply non-semantic-preserving mutations on them and reuse differential testing on the execution result of the mutated test cases.

The next chapter presents our validation of Test Transmutation. We describe both a qualitative and quantitative analysis of how Test Transmutation is relevant to test VMs.

CHAPTER 6

Test Transmutation Validation

Contents

This chapter presents the validation of Test Transmutation. This validation is two-fold. First, we present a qualitative analysis of whether test cases generated from simulated test cases are able to catch bugs in the simulated VM and in the generated VM. This analysis is achieved by manually introducing errors in the VM and executing test cases on both the simulated and the generated VM. This validation shows that this approach is able to detect bugs that VM developers typically encounter. Second, we present the results of mutating the simulated test cases. Particularly, we apply these mutations to the memory manager test cases. We also present the multiple new issues in the VM that we found. Moreover, applying these mutations shows that the test case generator is correct as well as exploring more VM execution paths.

6.1 Introduction

The previous chapter presented Test Transmutation to test VMs. However, executing test cases in the generated VM only makes sense if the generated test cases detect bugs and are generated properly. This chapter presents the evaluation of Test Transmutation.

We start by presenting a qualitative analysis of Test Transmutation (Section 6.2). To assess that test cases detect bugs, we manually introduce bugs inside the VM code and check that the test cases are able to detect them. Due to the semantic gaps, some bugs are detected only in the simulation environment VM whereas others are detected only in the generated environment. This analysis shows that the simulation test cases are not able to detect some kinds of bugs and existing simulation test cases are able to detect them in the generated VM.

Secondly, we present a quantitative analysis of applying Test Transmutation to the Pharo VM. We start by characterizing the test cases we use for this analysis. Then we present the results of applying test case mutation. It creates mutated test cases from simulation test cases. Test Transmutation creates 494 mutants which are validated with 1890 test case executions. We present the three VM bugs that we have detected. Finally, we present the threats to the validity of this study before concluding this chapter.

6.2 Test Transmutation Qualitative Analysis

To evaluate the ability of Test Transmutation to detect bugs, 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 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 (Table 6.1). We consider it 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:

Type annotations. Errors in type annotations may produce compilation errors and runtime failures.

Name Conflicts. Name conflicts create compile-time errors.

Undefined Behavior. The Slang VM generator is not entirely aware of all the target language's undefined behavior. Therefore the generated code may exhibit

undefined behavior.

Invalid Execution. Some programs are invalid at simulation time, although they can be generated and executed correctly. Some programs are valid at simulation time, but cannot be generated nor executed correctly.

Table 6.1: Bug detection by the test cases in the simulated and generated VM.

	Simulated VM	Generated VM
Type Annotation	✓	✗
Memory Simulation	✓	✗
Name Conflict	✓	✗
Literal Type Error	✗	✗
Undefined Behavior	✗	✓
Name Mangling	✗	✓

6.2.1 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.

Bug Description. Stack allocations are simulated as simple heap allocations Listing 6.1. This means that simulated stack allocations outlive 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. In C however, memory allocated on the stack by using the alloca() function is returned after the current function returns. Accessing this memory after it is not accessible anymore creates segmentation faults, preventing VM execution.

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

Listing 6.1: Example of memory management differences. The alloca() C function allocates memory on the stack. Stack-allocated memory is not accessible anymore after the function calling alloca() returns. However, it is simulated as a heap allocation. Therefore it outlives the method sending the message #alloca:. Therefore unaccessible memory is still accessible in the simulation but triggers segmentation faults in generated test cases.

Results. All simulated test cases passed, whereas all generated test cases failed with segmentation faults.

6.2.2 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 such as late binding and method lookup. On the other hand, C is statically typed. Moreover, C requires developers to specify not only the type but also the space they take in memory. Therefore, generating the VM to C requires the VM source code to have some type annotations. Moreover, even though type annotations are available in the Slang source code, they are ignored by the simulation environment.

Bug Description. Wrong or missing type annotations break the generation process or make the executable VMs produce runtime errors. The introduced bug changes a type annotation from a long int pointer to a char pointer (Listing 6.2).

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

Listing 6.2: Example of type annotation. We replace the type annotation of the variable varLoc from a long int pointer to char pointer.

Results. All simulated test cases passed because they do not consider the type annotation at all. However, all generated test cases failed with segmentation faults. This is because we changed the size in memory of the pointed value to a smaller type. Therefore the VM puts values too big for this type in this variable, which affects unexpectedly memory that shouldn't be accessed in this manner. We consider this a VM generator bug.

6.2.3 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 6.3.

```
1 weakArrayFormat
2   "^4 ### Original code"
3   ^4.0 "### Buggy replacement"
```

Listing 6.3: 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 the VM generator eagerly rejects 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.

6.2.4 Name Conflicts and Name-mangling

Background. The simulated and the generated VM 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 (Listing 6.4). The generated version uses the identifier bogon to reference a variable and two functions (Listing 6.5).

```
1 bogon
2   <inlineInC: #false>
3   ^ bogon
4
5 bogon: aNumber
6   <inlineInC: #false>
7   bogon := aNumber
```

Listing 6.4: Example of a non-inlined getter, creating a name conflict on generation. Its generation is depicted in Listing 6.4

```
1 int bogon;
2 int bogon(){ return bogon; }
3 void bogon(int aNumber){ bogon = aNumber; }
```

Listing 6.5: Code generated from the code in Listing 6.4. The identifier bogon is declared and defined multiple times, which results in a C compilation error.

Results. All these being valid Pharo idioms, simulated test cases keep 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.

6.2.5 Undefined Behavior

Background. The C programming language targeted by the Slang VM generator contains 193 sources of undefined behavior in C99 [Committee 2007]. Upon encountering an undefined behavior, C compilers are able to apply aggressive optimization. Thus, C developers must avoid using them in their programs. 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 arithmetic overflow (Listing 6.6). This behavior in the C specification is explicitly unspecified. Therefore upon encountering this behavior, the compiler is free to deal with it in any way it wants.

```
1 howManyFreeObjects
2   | counter |
3   counter := 0.
4   self allFreeObjectsDo: [ :freeObject | counter := counter + 1 ].
5   "^ counter ### Original code"
6   ^ counter + self maxCInteger + 1 " ### Buggy replacement"
```

Listing 6.6: 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 generated VM. We consider that this is 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.

6.2.6 External functions and Name-mangling

Background. The Slang VM generator allows any function or method to call any external C function such as memcpy 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 the 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 Slang name-mangling point of view (Listing 6.7). However, the simulation defines only one method to simulate memcpy. Therefore only one of the two versions works in the simulated VM.

```
1 "Original code"
2 self me: destAddress mc: sourceAddress py: bytes
3
4 "Buggy replacement"
5 self memc: destAddress p: sourceAddress y: bytes
6
7 "generated version"
8 memcpy(destAddress, sourceAddress, bytes);
```

Listing 6.7: Example of two valid ways of writing a call to the memcpy function in Slang. Both are generated properly to memcpy(). However, the simulation defines only one method to simulate memcpy. Therefore only one of the two versions works in the simulated VM.

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.

6.3 Empirical results on the Pharo VM

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

6.3.1 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. The Pharo memory

manager is a generational scavenger garbage collector that uses a copy collector for young objects in the **new space** and a mark-compact collector for older objects [Ungar 1984] in the **old space**. We categorize the test cases as follows, and we summarize them in Table 6.2:

Memory structure. Test cases checking the memory structural properties of the new space and the old space. Particularly, they check that Garbage Collection specific variables are initialized. There are 8 test cases testing the new space structure, 7 of which are passing in the generated VM. There are 7 test cases testing the old space structure, only one of them passing in the generated VM. This is because the old space test cases test simulation-specific properties that the generated VM does not respect.

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. There are 9 test cases testing the new space allocation scheme, 8 of which are also passing in the generated VM. There are 21 test cases testing the new space allocation scheme, all of which are also passing in the generated VM. There are also 127 test cases testing the multiple kinds of allocation strategies that are used in the VM.

New space garbage collection. Test cases checking the copy garbage collector used in the new space garbage collection. For example, they check that allocating two objects in a row should allocate both objects next to one another. There are respectfully 9 test cases that are testing weak objects garbage collection 19 testing the ephemeron objects which are all passing in the generated VM. Weak and ephemeron objects are objects which require a special kind of finalization process. There are also 38 test cases testing regular object garbage collection, of which 29 are passing in the generated test cases. The remaining 10 encounter type inference issues that prevent their generation at the moment of writing.

Old space garbage collection. Test cases checking the mark-compact algorithm used for old space garbage collection. For example, they check that objects referenced are marked, and not reclaimed nor moved by the garbage collection. There are 9 test cases testing regular objects garbage collection, of which 8 are passing in generated VM. There are 9 test cases testing unmovable or pinned objects, which are all passing in the generated VM. Unmovable objects are objects that should not be moved by the compactor.

6.3.2 Prototype Results

Table 6.2 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 of 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 test case is executed in both the simulated and generated VM for each mutant. There were 1890 test case executions to validate the 494 generated mutants.

We detail the results in the following, and recap them in Table 6.2:

Memory structure. Each test case eligible for mutation generates only one mutated test case and is validated by executing only itself. The 8 test cases execution did not find bugs.

Memory allocation. Memory allocation test cases were eligible for many mutations and required many test case execution for their validation. The 13 new space test case executions and the 71 old space test case executions were not able to find bugs. However, memory allocation strategies required 1464 test case executions for their validation, two of which were able to detect bugs.

New space garbage collection. Object test cases were eligible to generate 62 mutants out of the 29 initial passing test cases. Out of these test case execution, 11 were able to detect bugs. Out of the 9 test cases testing weak objects, there were 34 mutant executions, none of which detected bugs. Finally from the 19 ephemeron object test cases, 10 mutant execution found bugs out of the 101 that were executed.

Old space garbage collection. Both the 8 regular objects test cases and the 9 unmovable objects test cases resulted in the execution of 33 and 104 test cases for the validation of their mutants, however, none of them found bugs.

6.3.3 Results 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. Additionally, this approach didn't detect bugs in the test case generator. This improves our confidence in the test case generator.

Category	Sub-Category	Passing Simulated	Passing Generated	Mutant Executions	Mutants Bugs
Memory	New Space	8	7	7	0
Structure	Old Space	7	1	1	0
Memory	New Space	9	8	13	0
Allocation	Old Space	21	21	71	0
	Strategies	127	127	1464	2
New Space	Objects	38	29	62	11
GC	Weaks	9	9	34	0
	Ephemerons	19	19	101	10
Old Space	Objects	9	8	33	0
GC	Unmovables	9	9	104	0
Total		256	238	1890	23

Table 6.2: Detailed categorization of the test cases considered and the statistics related to the validation of the mutants. Only test cases passing in both the simulated and the generated VM are eligible to be used in the mutation part of Test Transmutation. Mutated test cases are validated with one or multiple test case executions. Particularly, the Memory allocation strategy required 1464 test case executions to validate all the mutated test cases generated from memory allocation strategies test cases. Finally, we present the number of test case executions that detected a bug.

Stack Allocation Issues. At first, Test Transmutation uncovered the alloca simulation issue described in Section 6.2.1 before even applying mutations. 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 apply mutations, obtain these results and find 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 performs exact integer division whereas the second returns a fraction. Moreover, #// truncates the result of the division and returns an integer. Both selectors are generating the / operator in C. Mutants changing the arithmetic operators break an implicit VM invariant the VM, creating a runtime error in the simulated VM. However, both selectors are generated to the same C operator. Therefore the C test cases pass.

Runtime Assertion Differences. Mutations creating failing assertions 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. The mutations, therefore, broke an invariant. 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. Note that the assertions are generated differently when generating the VM and the test cases. The failing assertions and the semantics difference are on the VM assertion.

6.4 Threats to Validity

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 [Ungar 1984] and a mark-compact collector for older objects their structure and their allocation schemes.

The selected subset may be considered not representative of the entire VM process. To address this issue, Section 6.2 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 6.2.1) need to be inlined in their caller, otherwise, the allocated memory is freed right on return. 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 selectively when required for a test case compilation.

6.5 Conclusion

Our validation showed multiple cases in which generating simulation test cases are able to find diverse bugs. Such bugs are typical of real-world bugs that happen during VM development. Co-generating the simulation test cases are therefore able to detect typical real-world bugs that are encountered by developers during VM development. The second part presents the results of mutating the test cases. From the existing 238 test cases, we generate a total of 494 mutants which are validated by 1890 test case executions. Finally, we presented the threats to the validity of this study. As with any kind of testing, it required some deoptimization of the VM code. However, we have reduced these threats as much as possible.

CHAPTER 7

Conclusion and Future Work

Contents

7.1 Summary

Virtual Machines (VMs) are modern programming language implementations. Testing and debugging a VM is a laborious task without the proper tooling. This is particularly true for VMs implementing garbage collection, Just in Time (JIT) compilation, and interpreter optimisations. This situation is worsen when the VM builds and runs on multiple target processor architectures.

To ease the development of VMs, simulation-based VM generator frameworks were studied. Such frameworks allow developers to work on the VM in a simulation environment and to generate the VM once it's ready. Developers leverage the simulation environment to ease not only the development process but also tooling and debugging the VM. However, it creates abstraction gaps between the simulated and generated VM. Therefore the execution of the simulated and the generated VM are not functionally equivalent.

Chapter 2 describes the problems and their solutions in the current state of the art. We highlight two well-known problems: the **oracle** problem and the **test input** problem. (1) The oracle problem, or how to automatically determine whether a

given input matches the expected semantics, is solved with two main approaches: **differential** or **metamorphic** testing. (2) The test input problem, or how to automatically generate relevant test inputs, is solved with three main approaches: **handwritten programs**, **test program generation**, and **test program mutations**. (3) Moreover, we also investigate simulation-based testing in other domains.

We concluded from other simulation-based approaches that testing only the simulation was not enough. We started by writing handwritten test cases on the simulated VM. We have decided that the best approach for our research was differential testing with a cross-environment strategy and reusing handwritten test inputs and generating them to C. Moreover, the handwritten test inputs are also mutated to test the test case generator.

Chapter 3 presents our experimental context: Pharo. Particularly we have presented Pharo itself, its syntax, and its relevance for this thesis. More importantly, we have presented the Pharo VM. The Pharo VM is written in Slang, a subset of the Pharo language. The Pharo VM is generated to C and compiled by a C compiler for production purposes.

Chapter 4 presents the first step in our research: how to test the simulated VM. We present the methodology we devised to test the VM, as well as the different execution modes we leverage: unit testing, full system simulation, and real hardware execution. Basically, the aim is to use each execution mode where it performs best. Particularly, we leverage the execution speed and the feedback-cycle speed of unit test cases to execute them as much as possible. Similarly, we leverage the precision of the execution of the real hardware execution. However, as it is significantly slower, it is executed less often.

Chapter 5 introduces Test Transmutation. Test Transmutation proposes to test the generated VM by co-generating the test cases with the VM and executing the generated test cases. We first describe the properties of generated test cases and compare them to the properties of other execution modes presented in Chapter 4. We describe the oracle that we use to validate our test cases and the mutated test cases. Our oracle uses *functional equivalence* to compare the *test case results*. As the test case results come from different environments, we call this differential testing strategy a *Cross-Environment Differential Testing* strategy. We explain further how we apply non-semantic-preserving mutations on test cases.

Chapter 6 presents the validation of Test Transmutation. This validation is twofold. First, we present a qualitative analysis of whether test cases generated from

simulated test cases are able to catch bugs in the simulated VM and in the generated VM. This analysis is achieved by manually introducing errors in the VM and executing test cases on both the simulated and the generated VM. This validation shows that this approach is able to detect bugs that VM developers typically encounter. Second, we present the results of mutating the simulated test cases. Particularly, we apply these mutations to the memory manager test cases. We also present the multiple new issues in the VM that we found. Moreover, applying these mutations shows that the test case generator is correct as well as exploring more VM execution paths.

7.2 Answering Research Questions

As a conclusion to this thesis, we answer the questions formulated in the introduction.

RQ1: How can VM developed in simulation-based VM generator be tested and verified in spite of semantic gaps? We present a methodology to test VMs developed in simulation-based VM generator frameworks (Chapter 4). It leverages unit testing and full-system simulation to test the simulated VM. It then leverages real hardware execution for its precision to confirm that the production VM indeed works in spite of semantic gaps. However, real hardware execution suffers from low debuggability. We improve on this methodology with Test Transmutation, which applies unit testing on the generated VM as well (Chapter 5).

RQ2: Can existing test cases from the simulation environment be re-used to test the generated VMs? To evaluate whether test cases written to test the simulated VM are able to test the generated VMs, we present an evaluation of manually injecting faults (Chapter 6). We present each fault reasoning, bug description, and whether test cases were able to find the bug introduced. Moreover, we also show that applying mutations on the simulated test case and validating with differential testing discovers bugs by exploring more execution paths. We show that both the simulated and generated test cases are able to find different kinds of bugs. Therefore existing test cases from the simulation can be re-used to test the generated VMs.

7.3 Contributions

The main contribution of this thesis is an approach to test VMs developed in simulation-based VM generator frameworks. It includes:

1. A simulation-based testing methodology to test a VM developed in a simulation-based VM generator framework (Chapter 4).

2. Test Transmutation, an approach extending the simulation-based test methodology to apply unit testing on the generated VM based on test co-generated with the VM test case mutation (Chapters 5 and 6).

7.4 Future Work

We finish this thesis by presenting multiple future works that could take place on VMs testing, starting from our research.

7.4.1 JIT Real Hardware Testing

In this thesis, we have created the required bricks to test the generated VM more precisely, with unit test cases. However, we did not use the JIT test cases. The test infrastructure for the JIT test cases is more complicated because of the nature of the test cases. Particularly, the test cases rely on a machine code simulator. Moreover, simulating the JIT requires tricks to stop the execution at some particular points. One such trick has been to use illegal addresses to get feedback on the execution [Miranda 2011, Ingalls 2020]. The co-generation of these test cases will have unique challenges that were not covered in this thesis. Moreover, testing the JIT with unit test cases per platform could find bugs significantly faster than the current process.

7.4.2 Extend Existing Techniques to the Generated VM

Most VMs implement at their core behavior duplication to support different kinds of code execution. Particularly, the interpreter provides one implementation of a primitive, while the JIT compiler provides another implementation at the machine code level. This enables differential testing to detect bugs between the different implementations.

Recently, Polito et al. proposed a concolic testing approach to apply differential testing of the Pharo VM interpreter and JIT compiler [Polito 2022a]. This approach tests the simulated VM by generating meaningful inputs. However, this approach is purposefully focused on the testing of the interpreter and JIT compiler because they have duplicated semantics with different implementations. Such an approach cannot test components that do not have multiple implementations with differential testing.

This thesis focused on the creation of an oracle to test the generated VM, therefore this concolic approach could be extended to use also test the generated VM.

More widely, differential testing is now available to test any components. Therefore multiple approaches such as the ones described in our state of the art could now be applied to the VM generators.

7.4.3 Back to the Start: Improving VM Generators

At its root, this research project was not about testing itself but about improving VM generators. However, once we started experimenting with the VM, it became quickly clear that VM generators were missing a key feature: the validation of the generated artifact. This lead us to test the simulated VM, to enable us to test the generated VM. However, the initial concern about improving the VM generators is still very relevant and in dire need of validation of the generated VM. For instance, Polito et al. proposed an optimization that automatically localizes a few critical VM variables to the interpret function automatically [Polito 2022b]. Now that the validation of the generated VM is available, more experiments are able to take place at the level of the VM generator.

7.4.4 Improving the Oracle with Cross-Architecture Differential Testing

Developers using simulation-based VM generator frameworks mostly work on their VM in the simulation. As such, the most important part of the oracle was not only to validate the generated VM but also to validate that the generated VM was equivalent to the simulated VM. Moreover, it allows developers to keep the whole code base and tooling infrastructure in the simulation.

However, Simulation-based VM generator frameworks often generate more VMs that are compilable to multiple processor architectures. Therefore developers are able to use cross-architecture differential testing. Particularly, this would allow a developer to test the semantics of multiple back ends of JIT compilers natively.

Bibliography

[Alipour 2016] Mohammad Amin Alipour, Alex Groce, Rahul Gopinath and Arpit Christi. *Generating focused random tests using directed swarm testing*. In Proceedings of the 25th International Symposium on Software Testing and Analysis, pages 70–81, 2016.

[Alpern 2000] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J. D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan and J. Whaley. *The Jalapeño virtual machine*. IBM Systems Journal, vol. 39, no. 1, pages 211–238, 2000.

[Alsmadi 2020] Izzat Alsmadi, Anis Zarrad and Abdulrahmane Yassine. *Mutation Testing to Validate Networks Protocols*. In 2020 IEEE International Systems Conference (SysCon), pages 1–8. IEEE, 2020.

[Amodio 2017] Matthew Amodio, Swarat Chaudhuri and Thomas W Reps. *Neural attribute machines for program generation*. arXiv preprint arXiv:1705.09231, 2017.

[Ancona 2007] D. Ancona, M. Ancona, A Cuni and N. Matsakis. *RPython: a Step Towards Reconciling Dynamically and Statically Typed OO Languages*. In Proceedings of the 2007 Symposium on Dynamic Languages (DSL 07), pages 53–64. ACM, 2007.

[Austin 1991] SM Austin, DR Wilkins and Brian A Wichmann. *An Ada program test generator*. In Proceedings of the conference on TRI-Ada'91: today's accomplishments; tomorrow's expectations, pages 320–325, 1991.

[Bazzichi 1982] Franco Bazzichi and Ippolito Spadafora. *An automatic generator for compiler testing*. IEEE Transactions on Software Engineering, no. 4, pages 343–353, 1982.

[Behnel 2011] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D. S. Seljebotn and K. Smith. *Cython: The Best of Both Worlds*. Computing in Science Engineering, 2011.

[Béra 2013] Clément Béra and Marcus Denker. *Towards a flexible Pharo Compiler*. In International Workshop on Smalltalk Technologies 2013, 2013.

[Béra 2014] Clément Béra and Eliot Miranda. *A bytecode set for adaptive optimizations*. In International Workshop on Smalltalk Technologies (IWST), August 2014.

[Béra 2016] Clément Béra, Eliot Miranda, Marcus Denker and Stéphane Ducasse. *Practical Validation of Bytecode to Bytecode JIT Compiler Dynamic Deoptimization*. Journal of Object Technology, vol. 15, no. 2, pages 1:1–26, 2016.

[Berry 1983] Daniel M Berry. *A new methodology for generating test cases for a programming language compiler*. ACM Sigplan Notices, vol. 18, no. 2, pages 46–56, 1983.

[Besnard 2017] Valentin Besnard, Matthias Brun, Philippe Dhaussy, Frédéric Jouault, David Olivier and Ciprian Teodorov. *Towards one Model Interpreter for Both Design and Deployment*. In Third International Workshop on Executable Modeling (EXE 2017), September 2017.

[Black 2009] Andrew P. Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou and Marcus Denker. Pharo by example. Square Bracket Associates, Kehrsatz, Switzerland, 2009.

[Blackburn 2004] S. M. Blackburn, P. Cheng and K. S. McKinley. *Oil and water? High performance garbage collection in Java with MMTk*. In Proceedings. 26th International Conference on Software Engineering, pages 137–146, 2004.

[Bolz-Tereick 2022] Carl Friedrich Bolz-Tereick. *How is PyPy Tested?*, 2022. https://www.pypy.org/posts/2022/04/how-is-pypy-tested.html (Accessed: 2022-05-12).

[Bolz 2008] Carl Friedrich Bolz, Adrian Lienhard, Nicholas D. Matsakis, Oscar Nierstrasz, Lukas Renggli, Armin Rigo and Toon Verwaest. *Back to the future in one week — implementing a Smalltalk VM in PyPy*. In Self-Sustaining Systems, volume 5142 of *LNCS*, pages 123–139. Springer, 2008.

[Bolz 2015] Carl Friedrich Bolz and Laurence Tratt. *The impact of meta-tracing on VM design and implementation*. Science of Computer Programming, vol. 98, pages 408–421, 2015.

[Boujarwah 1999] Abdulazeez S Boujarwah, Kassem Saleh and Jehad Al-Dallal. *Testing syntax and semantic coverage of Java language compilers*. Information and Software Technology, vol. 41, no. 1, pages 15–28, 1999.

[Bouraqadi 2016] Noury Bouraqadi and Dave 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, August 2016.

[Burgess 1986] CJ Burgess. *Towards the automatic generation of executable programs to test a Pascal compiler*. Software Engineering, vol. 86, pages 304–316, 1986.

[Burgess 1996] Colin J Burgess and M Saidi. *The automatic generation of test cases for optimizing Fortran compilers*. Information and Software Technology, vol. 38, no. 2, pages 111–119, 1996.

[Callahan 1988] David Callahan, Jack J Dongarra, David Levine*et al.* Vectorizing compilers: A test suite and results. Argonne National Laboratory, 1988.

[Casey 2005] Kevin Casey, David Gregg and M. Anton Ertl. *Tiger – An Interpreter Generation Tool*. In Rastislav Bodik, editeur, Compiler Construction, pages 246–249, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[Chen 1998] Yih-Farn Chen, Emden R. Gansner and Eleftherios Koutsofios. *A C++ Data Model Supporting Reachability Analysis and Dead Code Detection*. IEEE Transactions on Software Engineering, vol. 24, no. 9, pages 682–693, September 1998.

[Chen 2016] Tse-Hsun Chen, Stephen W. Thomas and Ahmed E. Hassan. *A survey on the use of topic models when mining software repositories*. Empirical Software Engineering, vol. 21, no. 5, pages 1843–1919, 2016.

[Chen 2019a] Chunyang Chen, Zhenchang Xing, Yang Liu and Kent Long Xiong Ong. *Mining likely analogical APIs across third-party libraries via large-scale unsupervised API semantics embedding*. IEEE Transactions on Software Engineering, 2019.

[Chen 2019b] Junjie Chen, Guancheng Wang, Dan Hao, Yingfei Xiong, Hongyu Zhang and Lu Zhang. *History-guided configuration diversification for compiler test-program generation*. In 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), pages 305–316. IEEE, 2019.

[Chen 2019c] Yuting Chen, Ting Su and Zhendong Su. *Deep Differential Testing of JVM Implementations*. In International Conference on Software Engineering (ICSE'19), pages 1257–1268. IEEE Press, 2019.

[Chen 2020] Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao and Lu Zhang. *A Survey of Compiler Testing*. ACM Computing Surveys, pages 1–36, May 2020.

[Ching 1993] Wai-Mee Ching and Alex Katz. *The testing of an APL compiler*. ACM SIGAPL APL Quote Quad, vol. 24, no. 1, pages 55–62, 1993.

[Committee 2007] C Standardization Committee. *C99 specification*, 2007. shorturl.at/goyJQ (Accessed: 2021-10-19).

[DeMillo 1979] Richard A DeMillo, Richard J Lipton and Frederick G Sayward. *Program mutation: A new approach to program testing*. Infotech State of the Art Report, Software Testing, vol. 2, no. 1979, pages 107–126, 1979.

[Dewey 2015] Kyle Dewey, Jared Roesch and Ben Hardekopf. *Fuzzing the Rust Typechecker Using CLP*. In Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering, ASE '15, page 482–493. IEEE Press, 2015.

[Donaldson 2016] Alastair F Donaldson and Andrei Lascu. *Metamorphic testing for (graphics) compilers*. In Proceedings of the 1st international workshop on metamorphic testing, pages 44–47, 2016.

[Donaldson 2017] Alastair F Donaldson, Hugues Evrard, Andrei Lascu and Paul Thomson. *Automated testing of graphics shader compilers*. Proceedings of the ACM on Programming Languages, vol. 1, no. OOPSLA, pages 1–29, 2017.

[Dongarra 1991] Jack Dongarra, Mark Furtney, Steve Reinhardt and Jerry Russell. *Parallel Loops—A test suite for parallelizing compilers: Description and example results*. Parallel Computing, vol. 17, no. 10-11, pages 1247–1255, 1991.

[Duncan 1981] Arthur G Duncan and John S Hutchison. *Using attributed grammars to test designs and implementations*. In Proceedings of the 5th international conference on Software engineering, pages 170–178, 1981.

[ECMAScript] ECMAScript. *ECMAScript test suites*. https://github.com/tc39/test262 (Accessed: 2022-08-06).

[Eide 2008] Eric Eide and John Regehr. *Volatiles are miscompiled, and what to do about it*. In Proceedings of the 8th ACM international conference on Embedded software, pages 255–264, 2008.

[Ertl 2003a] M. Ertl and David Gregg. *The Structure and Performance of Efficient Interpreters.* J. Instruction-Level Parallelism, vol. 5, November 2003.

[Ertl 2003b] M Anton Ertl and David 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.

[Felgentreff 2016] Tim Felgentreff, Tobias Pape, Patrick Rein and Robert Hirschfeld. *How to build a high-performance vm for squeak/smalltalk in your spare time: An experience report of using the rpython toolchain*. In Proceedings of the 11th edition of the International Workshop on Smalltalk Technologies, pages 1–10, 2016.

[GCC] GCC. *GCC test suites*. https://gcc.gnu.org/onlinedocs/gccint/Testsuites.html#Testsuites (Accessed: 2022-08-06).

[Godefroid 2005] Patrice Godefroid, Nils Klarlund and Koushik Sen. *DART: directed automated random testing*. In Proceedings of Programming Language Design and Implementation (PLDI'05), pages 213–223. ACM, 2005.

[Goldberg 1983] Adele Goldberg and David Robson. Smalltalk 80: the language and its implementation. Addison Wesley, Reading, Mass., May 1983.

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

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

[Hein 2010] Andreas M Hein. *Identification and bridging of semantic gaps in the context of multi-domain engineering*. In Forum on Philosophy, Engineering & Technology, pages 58–57, 2010.

[Holler 2012] Christian Holler, Kim Herzig and Andreas Zeller. *Fuzzing with code fragments*. In 21st USENIX Security Symposium (USENIX Security 12), pages 445–458, 2012.

[Hölzle 1991] Urs Hölzle, Craig Chambers and David Ungar. *Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches*. In European Conference on Object-Oriented Programming (ECOOP'91), 1991.

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

[Ingalls 1997] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace and Alan Kay. *Back to the Future: The Story of Squeak, a Practical Smalltalk Written in Itself*. In Proceedings of Object-Oriented Programming, Systems, Languages, and Applications conference (OOPSLA'97), pages 318–326. ACM Press, November 1997.

[Ingalls 2020] Daniel Ingalls, Eliot Miranda, Clément Béra and Elisa Gonzalez Boix. *Two decades of live coding and debugging of virtual machines through simulation*. Software: Practice and Experience, vol. 50, no. 9, pages 1629–1650, 2020.

[Kalinov 2003a] Alexey Kalinov, Alexander Kossatchev, Alexandre Petrenko, Mikhail Posypkin and Vladimir Shishkov. *Coverage-driven automated compiler test suite generation*. Electronic Notes in Theoretical Computer Science, vol. 82, no. 3, pages 500–514, 2003.

[Kalinov 2003b] Alexey Kalinov, Alexandre Kossatchev, Alexander Petrenko, Mikhail Posypkin and Vladimir Shishko. *Using ASM specifications for compiler testing*. In International Workshop on Abstract State Machines, pages 415–415. Springer, 2003.

[Knuth 1968] Donald E Knuth. *Semantics of context-free languages*. Mathematical systems theory, vol. 2, no. 2, pages 127–145, 1968.

[Koster 1991] Cornelis HA Koster. *Affix grammars for programming languages*. In International Summer School on Attribute Grammars, Applications, and Systems, pages 358–373. Springer, 1991.

[Kotselidis 2017] Christos Kotselidis, Andy Nisbet, Foivos S Zakkak and Nikos 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.

[Le 2014] Vu Le, Mehrdad Afshari and Zhendong Su. *Compiler Validation via Equivalence Modulo Inputs*. In Programming Language Design and Implementation, PLDI '14, 2014.

[Le 2015] Vu Le, Chengnian Sun and Zhendong Su. *Finding deep compiler bugs via guided stochastic program mutation*. ACM SIGPLAN Notices, vol. 50, pages 386–399, October 2015.

[Leroy 2006] Xavier Leroy. *Formal certification of a compiler back-end, or: programming a compiler with a proof assistant*. In Principles of Programming Languages, POPL'06, pages 42–54. ACM Press, 2006.

[Lidbury 2015] Christopher Lidbury, Andrei Lascu, Nathan Chong and Alastair F Donaldson. *Many-core compiler fuzzing*. ACM SIGPLAN Notices, vol. 50, no. 6, pages 65–76, 2015.

[Lima 2020] Igor Lima, Jefferson Silva, Breno Miranda, Gustavo Pinto and Marcelo d'Amorim. *Exposing Bugs in JavaScript Engines through Test Transplantation and Differential Testing*. arXiv:2012.03759 [cs], 2020.

[Lindig 2005a] Christian Lindig. *Find a Compiler Bug in 5 Minutes*. Symposium on Automated Analysis-Driven Debugging, January 2005.

[Lindig 2005b] Christian Lindig. *Random testing of C calling conventions*. In Proceedings of the sixth international symposium on Automated analysis-driven debugging, pages 3–12, 2005.

[LLVM] LLVM. *LLVM test suites*. llvm.org/docs/TestingGuide.html (Accessed: 2022-08-06).

[Mandl 1985] Robert Mandl. *Orthogonal Latin squares: an application of experiment design to compiler testing*. Communications of the ACM, vol. 28, no. 10, pages 1054–1058, 1985.

[Martignoni 2010] Lorenzo Martignoni, Roberto Paleari, Giampaolo Fresi Roglia and Danilo Bruschi. *Testing System Virtual Machines*. In International Symposium on Software Testing and Analysis, 2010.

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

[Midtgaard 2017] Jan Midtgaard, Mathias Nygaard Justesen, Patrick Kasting, Flemming Nielson and Hanne Riis Nielson. *Effect-driven QuickChecking of compilers*. Proceedings of the ACM on Programming Languages, vol. 1, no. ICFP, pages 1–23, 2017.

[Miranda 1987] Eliot Miranda. *BrouHaHa — A Portable Smalltalk Interpreter*. In Proceedings OOPSLA '87, ACM SIGPLAN Notices, volume 22, pages 354–365, December 1987.

[Miranda 2011] Eliot Miranda. *The Cog Smalltalk Virtual Machine*. In Proceedings of VMIL 2011, 2011.

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

[Misse-Chanabier 2019] Pierre Misse-Chanabier, Vincent Aranega, Guillermo Polito and Stéphane Ducasse. *Illicium A modular transpilation toolchain from Pharo to C*. In International workshop of Smalltalk Technologies (IWST'19), Köln, Germany, August 2019.

[Misse-Chanabier 2022a] Pierre Misse-Chanabier, Guillermo Polito, Noury Bouraqadi, Stéphane Ducasse, Luc Fabresse and Pablo Tesone. *Differential Testing of Simulation-Based Virtual Machine Generators*. In International Conference on Software and Software Reuse (ICSR'22), pages 103–119. Springer, 2022.

[Misse-Chanabier 2022b] Pierre Misse-Chanabier, Guillermo Polito, Stéphane Ducasse, Noury Bouraqadi, Luc Fabresse and Pablo Tesone. *Differential testing of simulation-based VM generators: automatic detection of VM generator semantic gaps between simulation and generated VMs*. In Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing (SAC'22), pages 1280–1283, 2022.

[Misse-Chanabier 2022c] Pierre Misse-Chanabier and Théo Rogliano. *Ease Virtual Machine Level Tooling with Language Level Ordinary Object Pointers*. In Proceedings of the 14th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (VMIL'22), 2022.

[Morisset 2013] Robin Morisset, Pankaj Pawan and Francesco Zappa Nardelli. *Compiler testing via a theory of sound optimisations in the C11/C++ 11 memory model*. ACM SIGPLAN Notices, vol. 48, no. 6, pages 187–196, 2013.

[Nagai 2012] Eriko Nagai, Hironobu Awazu, Nagisa Ishiura and Naoya Takeda. *Random testing of C compilers targeting arithmetic optimization*. In Workshop on Synthesis And System Integration of Mixed Information Technologies (SASIMI 2012), pages 48–53, 2012.

[Nagai 2014] Eriko Nagai, Atsushi Hashimoto and Nagisa Ishiura. *Reinforcing random testing of arithmetic optimization of C compilers by scaling up size and number of expressions*. IPSJ Transactions on System LSI Design Methodology, vol. 7, pages 91–100, 2014.

[Ofenbeck 2016] Georg Ofenbeck, Tiark Rompf and Markus P"uschel. *RandIR: differential testing for embedded compilers*. In Proceedings of the 2016 7th ACM SIGPLAN Symposium on Scala, pages 21–30, October 2016.

[OpenJDK] OpenJDK. *OpenJDK test suites*. openjdk.org/jtreg (Accessed: 2022-08-06).

[Pałka 2011] Michał H Pałka, Koen Claessen, Alejandro Russo and John Hughes. *Testing an optimising compiler by generating random lambda terms*. In Proceedings of the 6th International Workshop on Automation of Software Test, pages 91–97, 2011.

[Patra 2016] Jibesh Patra and Michael Pradel. *Learning to fuzz: Application-independent fuzz testing with probabilistic, generative models of input data*. TU Darmstadt, Department of Computer Science, Tech. Rep. TUD-CS-2016-14664, 2016.

[Person 2008] Suzette Person, Matthew B. Dwyer, Sebastian Elbaum and Corina 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, November 2008.

[PlumHall] PlumHall. *Plum Hall Validation Test Suite*. https://www.plumhall.com/stec1.html (Accessed: 2022-08-06).

[Polito 2021] Guillermo Polito, Pablo Tesone, Stéphane Ducasse, Luc Fabresse, Théo Rogliano, Pierre Misse-Chanabier and Carolina Hernandez 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, September 2021.

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

[Polito 2022b] Guillermo Polito, Pablo Tesone and Stéphane Ducasse. *Interpreter Register Autolocalisation: Improving the performance of efficient interpreters*. In More VM internatlonal Workshop, 2022.

[Prathibhan 2014] C Mano Prathibhan, A Malini, N Venkatesh and K Sundarakantham. *An automated testing framework for testing android mobile applications in the cloud*. In 2014 IEEE International Conference on Ad-

vanced Communications, Control and Computing Technologies, pages 1216–1219. IEEE, 2014.

[Purdom 1972] Paul Purdom. *A sentence generator for testing parsers*. BIT Numerical Mathematics, vol. 12, no. 3, pages 366–375, 1972.

[Qu 2011] Xiao Qu and Brian Robinson. *A case study of concolic testing tools and their limitations*. In 2011 International Symposium on Empirical Software Engineering and Measurement, pages 117–126. IEEE, 2011.

[Rigo 2006] Armin Rigo and Samuele 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.

[RPythonCommunity 2016] RPythonCommunity. *RPython documentation on test translation*, 2016. https://rpython.readthedocs.io/en/latest/faq.html?highlight=test#can-rpython-modules-for-pypy-be-translated-independently (Accessed: 2021-10-18).

[Sen 2005] Koushik Sen, Darko Marinov and Gul 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. Association for Computing Machinery.

[Sheridan 2007] Flash Sheridan. *Practical testing of a C99 compiler using output comparison*. Software: Practice and Experience, pages 1475–1488, 2007.

[Simon 2006] Doug Simon, Cristina Cifuentes, Dave Cleal, John Daniels and Derek 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.

[Sirer 1999] Emin Gün Sirer and Brian N Bershad. *Testing Java virtual machines*. In Proc. Int. Conf. on Software Testing And Review, 1999.

[Starov 2015] Oleksii Starov, Sergiy Vilkomir, Anatoliy Gorbenko and Vyacheslav Kharchenko. *Testing-as-a-service for mobile applications: state-of-the-art survey*. In Dependability Problems of Complex Information Systems, pages 55–71. Springer, 2015.

[Sun 2016a] Chengnian Sun, Vu Le and Zhendong Su. *Finding and analyzing compiler warning defects*. In 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE), pages 203–213. IEEE, 2016.

[Sun 2016b] Chengnian Sun, Vu Le and Zhendong 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, October 2016.

[Terekhov 2000] A. A. Terekhov and C. Verhoef. *The realities of language conversions*. IEEE Software, vol. 17, no. 6, pages 111–124, November 2000.

[Ungar 1984] Dave Ungar. *Generation Scavenging: A Non-Disruptive High Performance Storage Reclamation Algorithm*. ACM SIGPLAN Notices, vol. 19, no. 5, pages 157–167, 1984.

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

[Vaarala 2013] Sami Vaarala. *Duktape website*, 2013. https://duktape.org (Accessed: 2022-05-30).

[vanWijngaarden 1965] Adriaan vanWijngaarden. *Orthogonal design and description of a formal language*. Stichting Mathematisch Centrum. Rekenafdeling, no. MR 76/65, 1965.

[Vilkomir 2015] Sergiy Vilkomir, Katherine Marszalkowski, Chauncey Perry and Swetha Mahendrakar. *Effectiveness of multi-device testing mobile applications*. In 2015 2nd ACM International Conference on Mobile Software Engineering and Systems, pages 44–47. IEEE, 2015.

[Whaley 2005] John Whaley. *Joeq: A virtual machine and compiler infrastructure*. Science of Computer Programming, vol. 57, no. 3, pages 339–356, 2005.

[Wimmer 2012] Christian Wimmer, Michael Haupt, Michael L. Van De Vanter, Mick Jordan, Laurent Daynes and Douglas Simon. *Maxine: An Approachable Virtual Machine For, and In, Java*. Rapport technique 2012-0098, Oracle Labs, 2012.

[Wimmer 2013] Christian Wimmer, Michael Haupt, Michael L. Van De Vanter, Mick Jordan, Laurent Daynès and Douglas Simon. *Maxine: An approachable virtual machine for, and in, java*. ACM Transaction Architecture Code Optimization, vol. 9, no. 4, January 2013.

[Wolf 2016] Klaus-Hendrik Wolf and Mike Klimek. *A Conformance Test Suite for Arden Syntax Compilers and Interpreters.* In MIE, pages 379–383, 2016.

[Würthinger 2013] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon and Mario Wolczko. *One VM to Rule Them All*. In International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (ONWARD'13), 2013.

[Yang 2011] Xuejun Yang, Yang Chen, Eric Eide and John Regehr. *Finding and Understanding Bugs in C Compilers*. In Programming Language Design and Implementation, PLDI '11, 2011.

[Ye 2021] Guixin Ye, Zhanyong Tang, Shin Hwei Tan, Songfang Huang, Dingyi Fang, Xiaoyang Sun, Lizhong Bian, Haibo Wang and Zheng Wang. *Automated conformance testing for JavaScript engines via deep compiler fuzzing*. In Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, pages 435–450, 2021.

[Yoshikawa 2003] Takahide Yoshikawa, Kouya Shimura and Toshihiro Ozawa. *Random program generator for Java JIT compiler test system*. In Third International Conference on Quality Software, 2003. Proceedings., pages 20–23. IEEE, 2003.

[Zarrad 2017] Anis Zarrad and Izzat Alsmadi. *Evaluating network test scenarios for network simulators systems*. International Journal of Distributed Sensor Networks, vol. 13, no. 10, page 1550147717738216, 2017.

[Zelenov 2003] Sergey V Zelenov, Sophia A Zelenova, Alexander S Kossatchev and Alexander K Petrenko. *Test generation for compilers and other formal text processors*. Programming and Computer Software, vol. 29, no. 2, pages 104–111, 2003.

[Zelenov 2007] Sergey Zelenov and Sophia Zelenova. *Model-based testing of optimizing compilers*. In Testing of Software and Communicating Systems, pages 365–377. Springer, 2007.

[Zhang 2017] Qirun Zhang, Chengnian Sun and Zhendong Su. *Skeletal program enumeration for rigorous compiler testing*. In Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, pages 347–361, 2017.

[Zhao 2009] Chen Zhao, Yunzhi Xue, Qiuming Tao, Liang Guo and Zhaohui Wang. *Automated test program generation for an industrial optimizing compiler*. In 2009 ICSE Workshop on Automation of Software Test, pages 36–43. IEEE, 2009.